

Python para Ciencia y Tecnología

Fundamentos y aplicaciones



$$\int_0^{\infty} f'(x) = \sqrt{a - \frac{1}{2}} dx = \frac{x}{s} \quad \frac{1}{s} = \sqrt{\pi} = \int \frac{dx}{dx} = \log_c \sqrt{x} = \frac{\pi}{C}$$

$$s = \frac{1}{e^2} - \frac{1}{\pi} \int_0^{\infty} \frac{1}{t} = \hat{f}(x) = \sin(x), \frac{\pi}{e} = \int_0^{\infty}$$

$$a - \frac{1}{r} = \int_3^{\infty} \dim \frac{x}{x-1} c + \frac{1}{x_0} t c = \frac{i}{Re^{\infty}} \int \lim_{x \rightarrow \infty} \frac{\Delta x}{e^2} - \frac{c}{2} =$$

$$= \frac{\sqrt{1-1}}{c^2} = \sum_{n=0}^{\infty} \frac{1}{(1-\frac{1}{2})^n} dx = \frac{C}{x} \quad \int_0^{\infty} s'x dx = \frac{\pi}{2} \left(\frac{\sqrt{1-tx}}{x} \right)$$

$$\frac{C}{O}(R) = \frac{i}{2} \int dx = C \quad \frac{1}{x} \int dx = \frac{1}{s_i} \log(x) = \frac{\rho}{C}$$

$$\prod_n t(x) \quad \hat{r} = \frac{b}{C} i \sqrt{O} = \frac{R}{i_n}$$

$$\frac{\pi}{\sqrt{5}} \frac{\pi}{s} \quad \int_0^{\infty} i \frac{in}{\sqrt{1}} \frac{\pi}{tr}$$

$$\frac{6}{10} \frac{x}{\sqrt{1-\frac{12}{c}}} \quad \frac{\pi}{e^{\infty}} = \frac{\sqrt{}}{n}$$

$$\frac{\pi x}{2} \frac{C}{\sqrt{n}} = \quad \log \frac{\hat{r}}{s} = \frac{\pi}{s^2}$$

$$\int_0^{\infty} \frac{\ln t}{t \sqrt{1-t^2}} \frac{\pi}{C} \Delta x \quad \frac{s}{r} = \frac{a+b}{l} - \frac{t}{v}$$

$$[i = \frac{S(t)}{t^2} \sin(x), \frac{\pi}{e} = \int_0^{\infty} \frac{1}{x} \int dx = \frac{1}{s_i} \log(x) = \frac{\rho}{C} \quad \frac{1}{x} = a$$

Facundo Batista y Manuel Carlevaro

versión 1.0.2

Python para Ciencia y Tecnología

Facundo Batista y Manuel Carlevaro

14 de abril de 2026

Título: Python para Ciencia y Tecnología
Autores: Facundo Batista y Manuel Carlevaro
ISBN: 978-631-00-9794-7
© Facundo Batista y Manuel Carlevaro
Versión 1.0.1
Escrito con X_YLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 ([CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/))
Lugar: Florida Oeste y La Plata, Buenos Aires, Argentina
Año: 2020 - 2025
Web: <https://libropython.science/>

*Dedicado a todas las personas que hacen ciencia y
desarrollan tecnología en nuestra querida Latinoamérica,
remando a contracorriente frente a intereses que nos
prefieren dependientes y desinformados.*
– Facundo Batista

*A Marisa, Florencia e Ignacio.
A Facundo, Norma, y Carlos.*
– Manuel Carlevaro

Agradecimientos

Agradecemos a quienes participaron en el curso de posgrado “*Herramientas Computacionales para Científicos*”, tanto docentes como estudiantes. Ese espacio compartido durante varios años sentó las bases que hicieron posible la creación de este libro. También queremos reconocer a quienes colaboraron en su redacción con comentarios y correcciones, en particular a Ramiro Irastorza, Diego Mascialino y Ana Julia Velez Rueda.

Nuestro reconocimiento se extiende también a todos los que nos precedieron en la construcción del conocimiento colectivo. La escritura de este libro nos brindó incontables horas de lectura, aprendizajes y descubrimientos.

Facundo Batista quiere agradecer, ante todo, a su familia, por todas las horas robadas día a día para poder dedicarle tiempo a este material. También a Manuel, por haberse subido a esta aventura medio quijotesca, por su constancia con la evolución del libro durante todos estos años, y principalmente por haberle permitido acercarse un poco al mundo de la ciencia y la investigación en Argentina, con el que había perdido contacto desde que finalizó la Universidad. Finalmente, a Guido van Rossum, no solo por haber creado Python como un lenguaje libre y abierto, sino por haber sembrado desde el comienzo las bases de una comunidad abierta, diversa e inclusiva.

Manuel Carlevaro desea expresar su gratitud a Facundo, quien lo guió con maestría por el laberinto de bibliotecas y artes pythónicas, convirtiendo la escritura del libro en una excusa perfecta para posponer otras responsabilidades; y principalmente a su familia, por esa forma de compañía silenciosa y constante que solo quienes nos conocen profundamente saben ofrecer. Les debe tiempo —esa sustancia de la que estamos hechos, según un viejo argumento—, y sabe que una parte de ese tiempo ha sido entregada de manera irreversible a estas páginas. El saldo no ha sido equitativo; confía en poder restituir, en actos y presencia, algo de lo ausente. Deja este libro, al menos, como testimonio de su afecto y como una modesta forma de permanencia.

Cómo leer este libro

Este libro está estructurado en tres partes. La primera está dedicada completamente al lenguaje propiamente dicho, abordando los temas fundamentales para la comprensión de Python y los elementos básicos que permiten escribir programas.

En la segunda parte tratamos temas que constituyen herramientas básicas en la práctica de la programación en el ámbito científico y tecnológico. Principalmente describimos el módulo NumPy, que provee estructuras y algoritmos fundamentales para el cálculo numérico, analizamos con detalle la aritmética de punto flotante, introducimos el módulo SymPy que permite el cálculo simbólico, describimos herramientas para organizar el desarrollo de proyectos mediante el versionado de código, y por último mostramos algunas formas para manipular archivos de datos.

Finalmente, la tercera parte comprende capítulos en los que abordamos algunas tareas de uso frecuente entre los miembros de la comunidad académica y tecnológica: elementos de estadística y modelado estadístico, solución de ecuaciones algebraicas, integración numérica, solución de ecuaciones diferenciales ordinarias y en derivadas parciales, optimización, aprendizaje automático y procesamiento en placas gráficas.

Nos permitimos sugerir algunos recorridos de este libro, en función de los conocimientos previos y necesidades de quienes accedan a él.

Para quienes estén realizando sus primeras experiencias con Python, resulta imprescindible la lectura de los capítulos 1 al 4, en donde abordamos las características centrales del lenguaje, los tipos de datos, el control del flujo de un programa y el encapsulamiento del código por medio de funciones, clases y módulos.

Los siguientes capítulos de la primera parte (5 al 10) son apropiados para quienes completaron la primera parte, o bien ya tienen un conocimiento básico de Python y quieran profundizar en aspectos más avanzados. Allí se abordan temas que incluyen generadores, administradores de contexto, programación orientada a objetos, entornos de ejecución, *profiling*, procesamiento en paralelo e interfaces gráficas.

En la segunda parte del libro, el capítulo 11 es de central importancia para quienes desarrollen código de uso científico o aplicaciones tecnológicas, ya que se describe con detalle el módulo NumPy, los arrays y diversos métodos para su manipulación. También sugerimos enfáticamente la lectura del capítulo 12, que trata sobre la representación de punto flotante binario y su aritmética, aspecto ineludible para el tratamiento numérico de cálculo mediante computadoras digitales.

Por último, los capítulos agrupados en la Parte 3 resultan de lectura opcional, según las necesidades de tratamiento computacional de diversos temas. Tratamos de exponer allí una selección (necesariamente incompleta) de temas usuales en el ámbito científico-tecnológico, sección que nos ilusiona ampliar en sucesivas ediciones de este libro.

Siempre intentamos proveer enlaces a la documentación de Python (o de bibliotecas específicas) que estén en español. No obstante, a veces no está actualizada por lo que recomendamos acceder a la versión original en inglés ante cualquier duda.

Sobre el código incluido en este libro

Este libro contiene numerosos fragmentos de código en Python. Algunos están presentados como *scripts* que pueden ejecutarse directamente desde una consola; otros forman parte de notebooks interactivos escritos en Jupyter.

Para reproducir estos notebooks, el lector deberá contar con una instalación funcional de Python (recomendamos la versión 3.9 o superior) y disponer del entorno Jupyter Notebook o JupyterLab, además de las bibliotecas necesarias mencionadas en cada capítulo.

Los ejemplos están pensados para ser explorados activamente: modificarlos, ejecutarlos y extenderlos es parte fundamental del recorrido que proponemos.

Por último, todos los códigos utilizados en este libro están disponibles *online* en:

<https://github.com/facundobatista/libro-pyciencia>

En particular, al inicio de cada capítulo se indica la ubicación específica en el repositorio de los códigos que contiene, y mencionamos además los módulos que deben estar instalados para poder ejecutar los programas. Recomendamos el uso de un entorno virtual para instalar dichos módulos, tal como describimos en el Capítulo 7.

Autores



Facundo Batista es desarrollador de software, divulgador técnico y uno de los principales referentes de la comunidad Python en América Latina. Es cofundador de PyAr (Python Argentina), una comunidad fundamental en el crecimiento y la difusión del lenguaje Python en la región. Desde sus comienzos en el mundo del software libre, Facundo ha sido un impulsor del uso abierto, colaborativo y responsable de la tecnología.

Es core developer de Python y fellow de la Python Software Foundation. Además, ha participado activamente en el desarrollo de herramientas de software, en la organización de eventos como PyCon Argentina y en la creación de espacios de formación y discusión técnica. Su compromiso con la comunidad se extiende también al trabajo como mentor y formador de nuevas generaciones de programadores. Estudió Ingeniería en Electrónica en la Universidad Nacional de La Matanza (Argentina) y obtuvo un máster en Ingeniería de la Innovación en la Università di Bologna (Italia).

Además de su trabajo técnico, Facundo es un entusiasta de la escritura y la comunicación clara de ideas complejas. Es autor de numerosos artículos y charlas, tanto en español como en inglés, en los que combina precisión técnica con una visión humanista de la tecnología. Su estilo cercano, generoso y pedagógico lo convierte en una figura muy valorada en comunidades de software libre y programación.

<https://facundo.taniquetil.com.ar/>

Manuel Carlevaro es físico e Investigador Independiente del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) en el Instituto de Física de Líquidos y Sistemas Biológicos. Además, es Profesor Titular en la Facultad Regional La Plata de la Universidad Tecnológica Nacional, donde dirige el Grupo de Materiales Granulares (GMG) y tiene a su cargo la materia Cálculo Avanzado en la carrera de Ingeniería Mecánica.

Formado como Licenciado en Física por la Universidad Nacional de Rosario (1995) y Doctor en Ciencias Exactas por la UNLP (2002), Manuel también tuvo roles en la Universidad de Buenos Aires y en la Autoridad Regulatoria Nuclear antes de incorporarse al CONICET. Sus actividades de docencia abarcan cursos de grado y posgrado en herramientas computacionales, métodos de modelado, física de materiales granulares y sistemas dinámicos, tomando parte activa en la formación de científicos e ingenieros en la región.

En el campo de la investigación, su trabajo se centra en la física de materiales blandos, biofísica, fluidos confinados y sistemas complejos. Su enfoque combina modelado matemático, simulación computacional y análisis teórico con una fuerte vocación aplicada.

<https://carlevaro.ar/>



Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

Este libro nace con la idea primaria de acercar Python al mundo científico, pensado para quienes se desempeñan en el ámbito científico o tecnológico, a partir de nuestra experiencia en el dictado del curso “Herramientas Computacionales para Científicos” que ofrecemos en la Universidad Nacional de La Plata y en la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas (índice, bibliografía, etc.), el libro tiene tres grandes partes. Sin entrar en el detalle de cómo abordar la lectura (que pueden encontrar en [Cómo leer este libro](#)), la primera parte habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar algunos problemas usuales en la ciencia o en la tecnología utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquivo la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de quienes dan los primeros pasos en el lenguaje y abordan la búsqueda de soluciones de algunos problemas básicos (pero no por eso menos importantes) mediante técnicas numéricas y, en algunos casos, también analíticas.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional ([CC BY-NC-SA 4.0](#)), salvo que se especifique puntualmente lo contrario.

*Facundo Batista y Manuel Carlevaro
Florida Oeste y La Plata
Buenos Aires, Argentina*

Prólogo

Desde muy pequeño siempre quise ser físico teórico. Es difícil de explicar si no lo has sentido, pero hay algo casi espiritual en el acto de descifrar cómo funciona el universo. No es solo curiosidad: es una especie de reverencia, una sensación de estar tocando algo inmenso. Siempre me gustaron las matemáticas, por supuesto. Son el lenguaje de la belleza, del orden, del arte puro. Pero aunque las matemáticas son perfectas, también son arbitrarias. Uno decide las reglas, crea un universo propio y explora lo que ocurre dentro de él. Es un acto casi divino: eres el demiurgo de un pequeño cosmos de símbolos, y tu tarea es descubrir las consecuencias de tus propias leyes. La física, en cambio, no ofrece esa libertad. También tiene su belleza, pero no es un mundo inventado: es el mundo. Es la realidad misma. Y el mundo real rara vez se comporta como queríamos. Es complejo, contradictorio, muchas veces oscuro. Hay ecuaciones tan hermosas que parecen poesía, pero la realidad que describen suele ser infinitamente más extraña, más sutil, más difícil de capturar.

Dentro de la física, la relatividad general fue mi gran fascinación. Era la unión perfecta entre la abstracción matemática más profunda y uno de los campos más desconcertantes de la ciencia, solo comparable a la mecánica cuántica. Estudiarla se siente como escalar una de las montañas más altas del conocimiento humano. Cada paso exige apoyarse en conocimientos previos: el álgebra y el análisis te llevan a la topología, la topología te abre las puertas a la geometría diferencial, y una vez que entras allí, ya no hay regreso. Es una montaña sin cima visible: cuando crees haber llegado, descubres otra pendiente más empinada esperándote. Pero en el esfuerzo hay una recompensa, una belleza austera que solo se revela a quienes insisten.

Recuerdo que una de mis primeras grandes metas era entender cómo la luz se curva en presencia de un agujero negro. Mucho antes de que Interstellar hiciera famosas sus imágenes generadas por las ecuaciones de Einstein, ya me obsesionaba ese fenómeno. Era una idea que tenía algo de revelación mística. Desde niños nos enseñan que la luz viaja en línea recta; lo comprobamos encendiendo un láser y viendo cómo su haz avanza recto, implacable. Pero la realidad, esa maestra caprichosa, nos dice lo contrario. La luz no viaja en línea recta: sigue el tejido curvado del espacio-tiempo. No es que la luz se doble; es el espacio mismo el que se dobla, y la luz, obediente, simplemente lo recorre. Esa idea, tan sencilla en palabras y tan profunda en significado, me resultaba irresistible. Quería entenderla, no solo aceptarla.

Para hacerlo, hay que aprender a describir superficies curvas, a medir distancias en espacios donde la noción misma de “recto” deja de tener sentido. Hay que redefinir conceptos tan básicos como los vectores o las derivadas, reaprender cómo se mueve algo sobre una superficie que no se puede ver ni tocar. Y eso exige abandonar casi toda intuición visual. Como dijo Michael Atiyah:

“El álgebra es la oferta hecha por el diablo al matemático. El diablo dijo: ‘Te daré esta poderosa máquina, que responderá cualquier pregunta, pero a cambio debes abandonar la geometría.’”

Cuando uno asciende por la montaña de la abstracción, sacrifica la visión por la razón. Pero el premio parece valer la pena: entender cómo la gravedad dobla la luz. Llegar a resolver las ecuaciones de Einstein para el caso más simple —una simetría esférica— ya es una hazaña. Incluso Einstein pensaba que sería imposible encontrar una solución exacta a sus propias ecuaciones. Las

había formulado en 1915 y eran tan complejas que parecían indomables. Sin embargo, apenas unas semanas después, desde una trinchera del frente oriental de la Primera Guerra Mundial, un físico alemán llamado Karl Schwarzschild le escribió. Entre el barro, el frío y el estruendo de los cañones, había hallado una solución exacta para el caso más simple, una masa esférica y estática. Aquel descubrimiento, hecho entre explosiones y muerte, revelaba cómo el espacio y el tiempo se deforman alrededor de una estrella. Era una especie de milagro científico nacido en el infierno. Lo que Schwarzschild encontró fue más que una solución: fue una ventana hacia lo imposible. Sus ecuaciones describían un punto donde el espacio-tiempo se curvaba tanto que ni la luz podía escapar, un lugar donde el concepto mismo de “afuera” dejaba de tener sentido. Einstein pensó que aquello era solo una rareza matemática, una curiosidad sin reflejo en la realidad. Décadas más tarde, ese punto imposible recibiría un nombre: **agujero negro**. Pero Schwarzschild no llegó a saberlo. Murió poco después de enviar su hallazgo, dejando tras de sí la ecuación que cambiaría para siempre nuestra manera de mirar el universo: una fórmula nacida de la guerra que revelaba la geometría de la oscuridad. Con ella, por fin, podíamos describir cómo el espacio y el tiempo se deforman alrededor de un agujero negro ideal. Y al resolver las ecuaciones de las geodésicas —los caminos que siguen las partículas y los rayos de luz— uno obtiene una ecuación elegante, pero implacable. Hermosa, pero distante. Puedes analizarla, deducir de ella cómo se mueve la luz, predecir órbitas imposibles... pero no puedes **verla**. Es como llegar a la cima de la montaña y descubrir que la vista está cubierta por una niebla densa. Has alcanzado el conocimiento, pero no la comprensión sensible. Esa fue la primera vez que la física me rompió un poco el corazón.

El agujero negro de Schwarzschild, además, no es realista. Es una idealización tan perfecta que roza lo absurdo. En su versión más pura, la singularidad —ese punto donde la densidad se vuelve infinita— no está “allí”, en el espacio, sino “mañana”, en el tiempo. Los físicos decimos, medio en broma, que la singularidad “no está aquí, sino que ocurrirá”. Es una especie de muerte inevitable en el futuro, no un punto que uno pueda señalar. Pero los agujeros negros reales giran. Y cuando el agujero negro rota, se convierte en un objeto descrito por la métrica de Kerr, y el universo se vuelve todavía más extraño. En ese espacio-tiempo, el giro arrastra la realidad misma: el tiempo y el espacio se entrelazan, las trayectorias se deforman, y aparecen regiones donde nada puede permanecer quieto. Analizar la luz en ese contexto es una odisea matemática. Las ecuaciones son tan complejas que uno apenas puede entenderlas, mucho menos visualizarlas. Y allí, en ese abismo de símbolos, mi amor por la física se volvió a tambalear. Yo quería ver. Quería entender. Quería mirar al universo y decir: “así es como se curva la luz”. Pero solo tenía ecuaciones. Fue entonces cuando encontré a Python.

La computación numérica ya era, desde hace décadas, una herramienta esencial para los científicos. Pero hacerlo en FORTRAN, C o C++ era como construir una catedral con piedra y martillo: poderoso, sí, pero agotador. Cada experimento requería horas de preparación; cada error podía destruirlo todo. Había que compilar, depurar, sufrir. El conocimiento venía a un precio: el de la paciencia infinita. Python cambió todo eso. De pronto, el pensamiento y la acción se fundieron en un mismo flujo. No había que esperar a que la máquina aceptara tu ofrenda; simplemente escribías, ejecutabas y veías el resultado. Era un lenguaje que respiraba contigo. No te hablaba en bits ni en registros, sino en ideas. Con Python, el científico podía volver a ser explorador. Cada línea de código era un paso hacia lo desconocido, pero un paso ligero, ágil, casi lúdico.

Y lo mejor fue descubrir que no estaba solo. Miles de personas también habían sentido esa chispa, y habían construido bibliotecas que multiplicaban las posibilidades: *NumPy*, *SciPy*, *matplotlib*, *SymPy*, *Astropy*. Con ellas, la ciencia dejó de ser una lucha contra la máquina y se convirtió en un diálogo. Por primera vez podía ver. Esas ecuaciones imposibles ahora se volvían

imágenes. Podía dibujar cómo la luz se doblaba alrededor de un agujero negro, cómo los rayos se entrelazaban formando anillos de fuego. Podía experimentar, probar, cambiar parámetros y ver en segundos lo que antes requería días. Era como si, después de tanto escalar, de repente uno descubriera que podía volar.

Y cuando algo es fácil y además es hermoso, enamorarse es inevitable. Este libro nace de ese amor: de dos personas que no solo aman la ciencia y a Python, sino que sienten la necesidad casi física de compartirlo. Porque cuando el entusiasmo desborda, cuando el corazón no puede contener tanta admiración, solo queda una salida: contagiarla. Este libro no quiere solo enseñarte a programar o a entender la ciencia; quiere invitarte a mirar el universo con asombro, a tocarlo con tus propias manos, a verlo curvarse y transformarse bajo tu propio código. Y, con un poco de suerte, mientras avances, sentirás cómo el peso desaparece, cómo tus pies comienzan a separarse del suelo, y cómo poco a poco empiezas a volar. Ese es el poder de Python. Ese es el poder del amor por la ciencia y el conocimiento.

Vuela con nosotros.

Pablo Galindo Salgado

$$\nabla_{\mu}\xi_{\nu} + \nabla_{\nu}\xi_{\mu} = 0$$

Índice general

Agradecimientos	III
Cómo leer este libro	IV
Autores	VI
Prefacio	VII
Prólogo	VIII
I Python	1
1. Introducción a Python	2
1.1. ¿Qué es Python?	2
1.1.1. Propiedades del lenguaje	5
1.1.2. Biblioteca estándar y módulos externos	6
1.2. Editando, corriendo, e interpretando	8
1.2.1. Editando y ejecutando Python	8
1.2.2. Editando Python	9
1.2.3. Usando módulos	10
1.2.4. El intérprete interactivo	10
1.2.5. Jupyter Notebook	12
1.2.6. Explorando	12
1.3. Cómo pedir ayuda	13
1.4. Zen de Python	14
2. Tipos de datos	16
2.1. Números	16
2.2. Cadenas	21
2.3. Listas	29
2.4. Tuplas	33
2.5. Pensando como un pythonista	37
2.6. Conjuntos	44
2.7. Diccionarios	47
2.8. Iteradores	51

ÍNDICE GENERAL

3. Controles de flujo	55
3.1. If, elif, else	55
3.2. While	58
3.3. For	60
3.4. Excepciones	65
3.4.1. ¿Qué son las excepciones?	65
3.4.2. Capturando y levantando excepciones	68
4. Encapsulando código	75
4.1. Funciones	75
4.1.1. Espacios de nombres	83
4.1.2. Generadores	85
4.2. Clases	88
4.3. Módulos	92
5. Python más avanzado	95
5.1. Generadores	95
5.2. Administradores de contexto	100
5.3. Acercándonos a la programación funcional	104
5.3.1. Funciones lambda	105
5.3.2. map, filter y reduce	105
5.3.3. Alejándonos de la programación funcional	108
5.4. Pruebas de unidad	109
5.5. Decoradores	116
5.6. Coincidencia de patrones	120
6. Clases	125
6.1. De instancia o de clase	125
6.2. Heredando o no	130
6.3. Usando propiedades	134
6.4. Creando tipos de datos	139
7. Entornos de ejecución de Python	144
7.1. Por qué tener distintos entornos	145
7.2. Entornos virtuales	146
7.2.1. En detalle	146
7.2.2. Herramientas	150
7.3. Contenedores	156
7.3.1. ¿Imágenes o contenedores?	157

ÍNDICE GENERAL

7.3.2.	Creando imágenes	158
7.3.3.	Ejecutando contenedores	159
7.3.4.	Valor extra de los contenedores como entornos aislados	161
7.3.5.	Compartiendo imagenes	163
8.	Sobre la velocidad de procesamiento	165
8.1.	Profiling	166
8.1.1.	Encontrando dónde un programa es lento	168
8.1.2.	Midiendo pequeñas partes de código	178
8.1.3.	Usando Python de forma más eficiente	180
8.2.	OK, sigue siendo lento, ¿entonces?	185
8.2.1.	Un ejemplo usando Python	186
8.2.2.	Usando Mypyc	188
8.2.3.	Trabajando con Cython	190
8.2.4.	Explorando Numba	192
8.2.5.	Conclusiones de los distintos procesos	193
8.3.	Usando directamente código compilado	194
8.3.1.	Extendiendo Python con C o C++	194
8.3.2.	Usando directamente código previamente compilado	198
9.	Procesamiento en paralelo	201
9.1.	¿Qué es la concurrencia?	202
9.2.	Threading	205
9.2.1.	Usando hilos	207
9.2.2.	Modificando estructuras en sistemas multithreading	208
9.3.	Async	214
9.3.1.	Usando async	216
9.4.	Procesamiento en múltiples procesadores	224
9.4.1.	Introducción	224
9.4.2.	Trabajando exclusivamente con números	230
9.4.3.	Ejemplo práctico	232
10.	Interfaces gráficas	240
10.1.	Qt	241
10.1.1.	Unas palabras sobre la documentación	242
10.2.	Una aplicación mínima	243
10.3.	Una aplicación real	247

ÍNDICE GENERAL

II Herramientas fundamentales	265
11. Numpy	266
11.1. Forma de trabajo	268
11.2. Multidimensionalidad	270
11.3. Indización avanzada	275
11.4. Broadcasting	278
11.5. Vectores y matrices	282
12. Aritmética de punto flotante	287
12.1. Punto flotante binario	289
13. Matemática simbólica	295
13.1. Conceptos y operaciones básicas	296
13.2. Simplificación de expresiones	303
13.2.1. Simplificación de funciones polinómicas y racionales	304
13.2.2. Simplificaciones trigonométricas	306
13.2.3. Simplificación de potencias	307
13.2.4. Simplificación de exponenciales y logaritmos	309
13.2.5. Funciones especiales	311
13.3. Solución de ecuaciones	312
13.4. Funciones y cálculo diferencial e integral	317
13.4.1. Funciones	317
13.4.2. Límites	320
13.4.3. Derivadas	322
13.4.4. Integrales	325
13.4.5. Sumas, productos y series	327
13.5. Evaluaciones numéricas y gráficos	331
14. Versionado de código	334
14.1. Ramas	335
14.2. Git	336
14.3. Comenzando con Git	338
15. Manipulación de datos	344
15.1. Leyendo archivos	345
15.2. Trabajando con datos en formato textual	347
15.2.1. Tres formatos textuales muy comunes	349
15.3. Trabajando con datos binarios	361
15.3.1. Un formato binario muy usado: HDF5	367

ÍNDICE GENERAL

15.4. Pandas	370
III Métodos numéricos y técnicas computacionales	377
16. Elementos de estadística	378
16.1. Introducción	378
16.2. Números aleatorios	379
16.3. Variables aleatorias y distribuciones	383
16.3.1. Medidas de centralidad	385
16.3.2. Medidas de dispersión	388
16.4. Test de hipótesis	395
16.5. Estimación no paramétrica de la FDP	403
16.6. Lecturas recomendadas	408
17. Modelado estadístico	409
17.1. Introducción	409
17.2. Coeficiente de correlación	410
17.3. Definición de modelos estadísticos con <i>patsy</i>	415
17.4. Regresión lineal	418
17.5. Regresión discreta	426
17.6. Series temporales	435
17.7. Lecturas recomendadas	446
18. Ecuaciones algebraicas	447
18.1. Sistema de ecuaciones lineales	448
18.1.1. Condicionamiento	449
18.1.2. Descomposición LU	451
18.1.3. Métodos iterativos	456
18.2. Problema de autovalores	463
18.3. Ecuaciones no lineales	467
18.3.1. Ecuaciones no lineales unidimensionales	468
18.3.2. Sistema de ecuaciones no lineales	472
18.4. Lecturas recomendadas	476
19. Integración numérica	478
19.1. Integración simbólica	479
19.1.1. Transformaciones integrales (Laplace y Fourier)	481
19.2. Integración numérica en una dimensión	484
19.2.1. Métodos de Newton-Cotes	485

ÍNDICE GENERAL

19.2.2. Fórmulas de Newton-Cotes compuestas	488
19.2.3. Cuadraturas gaussianas	489
19.2.4. Integración de Monte Carlo	492
19.3. Integración numérica con SciPy	495
19.3.1. Fórmulas de Newton-Cotes	495
19.3.2. Cuadraturas con funciones evaluables	497
19.4. Integración múltiple	498
19.4.1. Integración en dos, tres y n dimensiones con SciPy	499
19.4.2. Integración Monte Carlo	502
19.5. Lecturas recomendadas	504
20. Ecuaciones diferenciales ordinarias	505
20.1. Ecuaciones diferenciales ordinarias	506
20.2. Solución analítica de EDOs	507
20.3. Métodos numéricos para resolver EDOs	510
20.4. Solución de EDOs con valores iniciales	514
20.5. Lecturas recomendadas	517
21. Ecuaciones en derivadas parciales	518
21.1. Método de las diferencias finitas	520
21.1.1. Ecuación de conducción de calor en 1D con un método explícito	523
21.1.2. Ecuación de conducción de calor en 1D con un método implícito	528
21.1.3. Implementación con matriz rara	531
21.2. Método de elementos finitos	534
21.2.1. Ecuación de Poisson	535
21.2.2. Formulación variacional	536
21.2.3. Implementación en FEniCS	538
21.3. Lecturas recomendadas	542
22. Optimización	543
22.1. Optimización unidimensional	545
22.2. Optimización multidimensional sin restricciones	547
22.3. Optimización con restricciones	554
22.3.1. Restricciones en las variables de decisión	555
22.3.2. Programación lineal	557
22.3.3. Programación no lineal	559
22.4. Algoritmo genético	561
22.5. Lecturas recomendadas	569

ÍNDICE GENERAL

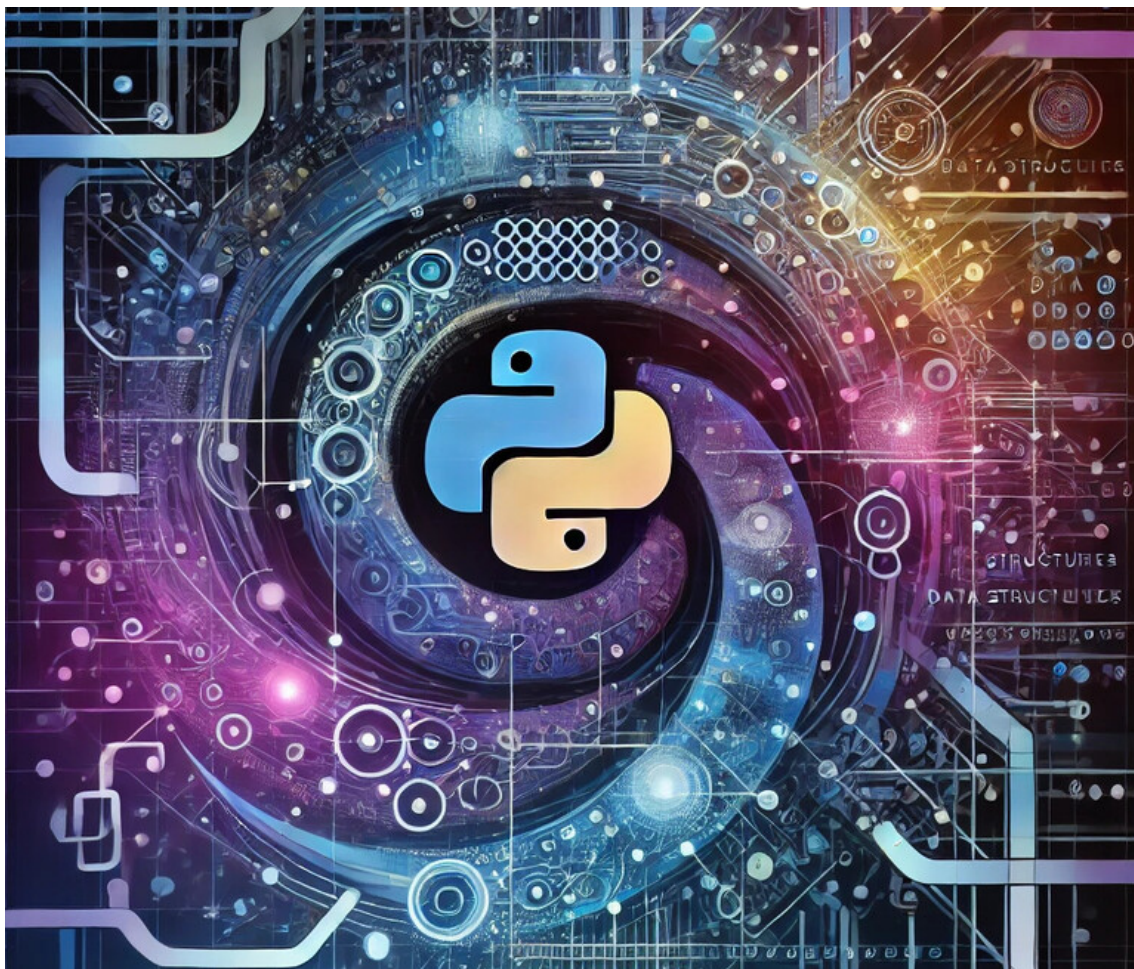
23. Aprendizaje automático	570
23.1. Nodos, pesos y funciones de activación	571
23.2. Aprendizaje	574
23.3. Ejemplo de red usando NumPy	580
23.4. Ejemplo usando Keras	590
23.5. Lecturas recomendadas	597
24. Procesamiento en GPU	598
24.1. Arquitectura de una GPU	600
24.2. Kernels y threads en CUDA	601
24.3. Organización de <i>threads</i>	603
24.4. Producto de matrices en GPU con PyCUDA	608
24.5. Producto de matrices con PyOpenCL	613
24.6. Lecturas recomendadas	616
Bibliografía	617

Parte I

Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.



1 | Introducción a Python

En este capítulo mostraremos las bases de Python.

Haremos un recorrido por las distintas características del lenguaje, y luego mostraremos cómo ejecutar programas y usar el intérprete en modo interactivo. Además haremos un repaso de los tipos de datos más usados, los controles de flujo que posee el programa (incluidas las excepciones, para manejo de error), y las distintas formas de encapsular código. Finalmente indicaremos las mejores formas de pedir ayuda.

[Código disponible](#)

Este contenido es imprescindible para entender lo suficiente de Python como para poder leer el resto del libro. Por supuesto, no es todo lo que se puede aprender de Python, solamente son las estructuras iniciales que permitirá arrancar con el lenguaje, utilizarlo para los primeros programas, y de allí escalar todo lo que se desee.

1.1. ¿Qué es Python?

Python es un lenguaje de programación de muy alto nivel y multiparadigma.

Es un lenguaje maduro, usado por muchos años en todos los ámbitos posibles (desde electrodomésticos hasta en el espacio). Su desarrollo comenzó a fines de 1989 por Guido Van Rossum, en un instituto de investigación estatal de los Países Bajos (CWI, algo así como “Centro para las Matemáticas y Ciencias de la Computación”). La motivación principal de van Rossum era tener un lenguaje de más alto nivel que los disponibles para otras investigaciones que se llevaban en el instituto y que al mismo tiempo sirviera de puente sencillo entre la aplicación y el sistema... y el resto es historia.

Al mismo tiempo, es un lenguaje que nunca se estancó, sino que está en constante evolución a través de mejoras permanentes que realiza la Comunidad. Porque es la Comunidad la responsable del desarrollo y progreso del lenguaje y sus herramientas relacionadas, así como de llevar adelante conferencias y eventos alrededor del lenguaje. Hay grupos de desarrolladores y gente que usa Python alrededor de todo el mundo, cuyo principal medio de comunicación y coordinación es online.

La Comunidad es uno de los puntos fuertes de Python, ya que no sólo ofrece la capacidad técnica de mantener y evolucionar el lenguaje, o administrativa de realizar eventos, sino que provee una estructura social que cobija a quienes comienzan a utilizar el lenguaje. Y esto no sucede solamente al principio, sino que es una red que facilita el aprendizaje y el perfeccionamiento a lo largo de toda la vida.



Guido van Rossum - Informático neerlandés (1956-), conocido por ser el autor y creador original del lenguaje de programación Python. Recibió el *FSF Award for the Advancement of Free Software* como reconocimiento por su trabajo.



¿Qué es una P.E.P.?

Es una Propuesta de Mejora de Python (*Python Enhancement Proposal*, en inglés), un documento que reúne las razones, ideas, discusiones de la Comunidad, conclusiones, y todos los detalles relevantes con respecto a cambios grandes en el lenguaje, la biblioteca estándar, o el proyecto en general. El punto de entrada para explorarlas es la [PEP 0](#).

Es la misma Comunidad la que garantiza tanto soporte como ayuda, ya que Python no depende de una empresa, y por tanto no corre el riesgo de estancarse por prioridades de esa empresa, conflictos legales por derechos de autor, etc. A este respecto cabe mencionar a la [Python Software Foundation](#), una organización sin fines de lucro encargada de proteger el copyright de Python, garantizando su libertad de uso y disponibilidad de forma gratuita.

Volviendo a Python como lenguaje propiamente dicho, es muy fácil de aprender ya que por diseño posee una sintaxis muy sencilla. Todo el lenguaje está basado en estructuras conceptuales que se repiten continuamente, bajando muchísimo la curva de adopción del mismo. En parte, es por esto que Python está tan recomendado para enseñar informática, porque todo el esfuerzo pedagógico se puede enfocar en algoritmos y lo que se quiera transmitir, sin tener que aprender primero el lenguaje en sí.

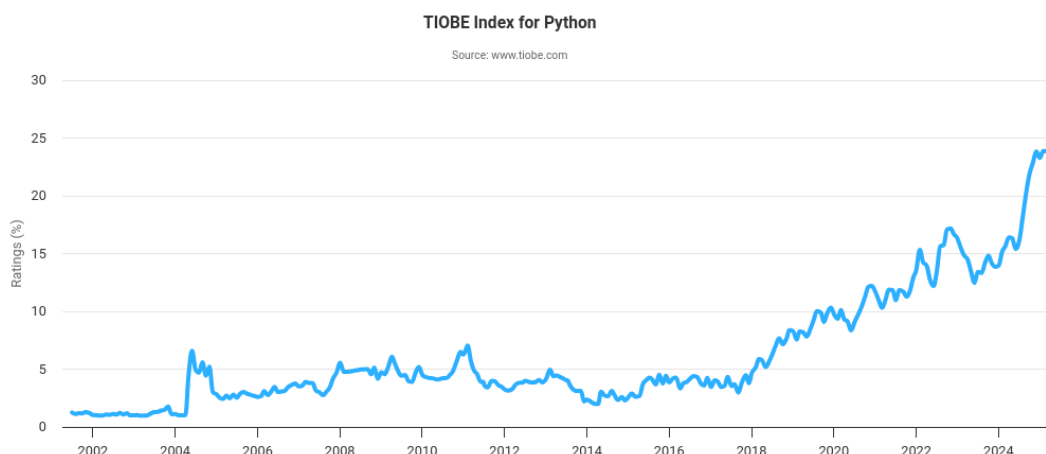


FIGURA 1.1: Evolución de la popularidad de Python en el mundo.

Esto mismo vemos reflejado en el mundo de la Ciencia, donde el uso de Python creció de forma vertiginosa en los últimos años, ya que es un lenguaje que permite enfocarse en el problema a resolver y no es una traba en sí mismo. Una medida de la popularidad de Python la ofrece el [índice TIOBE](#), que indica la proporción de consultas en línea de un lenguaje respecto del total de consultas de todos los lenguajes considerados en el índice. La Figura 1.1 muestra la evolución de la popularidad de Python en los últimos años, en la que se puede apreciar el rápido crecimiento desde 2018. Actualmente¹ Python se mantiene como el lenguaje más popular con un *rating* del 23.85 %, seguido por C++ (11.8 %) y Java (10.36 %).

Esta simplicidad se transmite en la cultura de Python, en ideas como la de “debería haber una, y preferiblemente solo una, manera obvia de hacerlo”. Esta y otras frases que forman el núcleo de la filosofía de Python son parte del [Zen de Python](#) (que incluimos traducido en 1.4), una serie de indicaciones generales para pensar sobre nuestros desarrollos y para enfocar el intercambio de ideas alrededor de un pedazo de código, no reglas duras que haya que cumplir a rajatabla.

El término “pitónico” (o pythónico, o *pythonic* en inglés) es un neologismo común en la Comunidad de Python, que expresa varios conceptos aplicados al estilo del programa. Decir que un código es pitónico es decir que usa correctamente los idiomas de Python, que se adapta a la filosofía minimalista de Python, y hace énfasis en su legibilidad, entre otros aspectos. Por el contrario, si el código es difícil de entender, trabado, o aplica idiomas traídos de otros lenguajes (con estructuras que normalmente se pueden resolver de forma más sencilla en Python), se lo denomina “no pitónico” (*unpythonic*).



El nombre Python proviene del grupo comediante británico Monty Python que tuvo su auge en los años 60 y 70, no por la serpiente, aunque esta se utilice en tantas imágenes y logos.

Por último, y no menos importante, es gratis, libre, y de código abierto. Tiene una licencia muy relajada, por lo cual podemos usar Python de forma segura tanto en ámbitos estatales como

¹ Valores del 3 de marzo de 2025.

privados, sin necesitar consultar con ningún abogado primero :). La única restricción que tiene a este respecto es si quisiéramos distribuir Python nosotros mismos, nada más.

1.1.1. Propiedades del lenguaje

Python es al mismo tiempo de tipado dinámico y de tipado fuerte. Dinámico, porque no hace falta declarar los nombres antes de utilizarlos. Fuerte, porque se respeta el tipo de los objetos y no se realizan conversiones implícitas. Habiendo dicho eso, tenemos que tener en cuenta que las definiciones más estrictas sobre “tipado” son sobre lenguajes que utilizan “variables”, y Python no posee variables como tales, sino que son todos objetos y nombres para referenciar esos objetos (más detalle sobre este tema en la sección 2.5).

Posee una administración dinámica de la memoria, usando una combinación de conteo de referencias y *garbage collector* (“recolector de basura”, en castellano, pero se acostumbra mencionarlo en inglés) para referencias cíclicas. Por lo tanto, aunque hay mecanismos para tener mayor o menor control de lo que sucede en memoria, en general no nos tenemos que preocupar de la misma.

Como mencionamos en la introducción, Python es multi-paradigma. Esto implica que aunque en Python todos son objetos, nosotros podemos programar completamente de forma estructurada, e incluso utilizar muchas características y módulos que vienen de la programación funcional u orientada a objetos. Resaltamos esta versatilidad en su aspecto pedagógico, porque normalmente se aprende programación estructurada primero y el pasaje a la programación orientada a objetos no es para nada trivial, entonces poder utilizar un lenguaje que permite empezar estructurado, e incluir eventualmente la creación de clases propias, mientras estamos expuestos a objetos todo el tiempo, hace que la curva de aprendizaje de la programación orientada a objetos sea gradual.

También en la introducción mencionamos que es un lenguaje de alto nivel. Esto es porque posee muchas estructuras modernas que permiten expresar la resolución de un algoritmo o la construcción de una solución de manera más cercana a la idea misma que formamos en nuestro cerebro, sin necesitar lidiar con el detalle que se necesitaría al trabajar en un lenguaje más cercano al procesador de la máquina. Resaltamos entre estas estructuras a los conjuntos y diccionarios como tipos de datos integrados, funciones y clases como ciudadanos de primer orden (o sea, son simplemente objetos), módulos y paquetes para estructurar código, iteradores y generadores muy bien integrados al lenguaje, etc. Además posee un manejo moderno de errores a través de excepciones.

Aunque solemos decir simplemente que Python es un lenguaje interpretado, realmente tiene un paso de compilación interno. En detalle, la ejecución de un programa en Python se realiza en dos pasos a partir del código fuente que le pasamos: primero Python compila ese código a una serie de instrucciones de la máquina virtual de Python, y luego procede a ejecutar esas instrucciones (que comúnmente denominamos *bytecode*). Esto es similar a otros lenguajes (como Java, VisualBasic, o COBOL, por ejemplo) con la diferencia fundamental que el proceso es automático: no hay intervención humana entre el primer paso de compilación y el segundo de ejecución en la máquina virtual.



El [sitio oficial](#) es el punto de entrada para todo lo que es Python: documentación, comunidades, descargas, etc.

Un factor clave para la velocidad de desarrollo en Python es el intérprete interactivo, una herramienta que nos permite ejecutar pequeñas muestras de código sin tener que incluirlas en un programa propiamente dicho, facilitando al mismo tiempo una exploración de los objetos con los que estemos interactuando. Realizando esta exploración y experimentación en el intérprete interactivo, de forma rápida y sencilla, se logra incorporar al código esos algoritmos o líneas de código ya probadas y estables, reduciendo notablemente el ciclo de prueba y error.

Las distintas funcionalidades y herramientas de Python no están integradas todas en su núcleo, sólo algunas forman parte de Python en sí, y el resto están disponibles a través de módulos y paquetes. Una gran cantidad de módulos están disponibles siempre que tengamos Python instalado en el sistema: son aquellos que forman parte de la llamada Biblioteca Estándar. Esta biblioteca es tan variada y extensa que generó la frase “Python viene con las pilas incluidas”, ya que gran parte del desarrollo día a día se puede realizar sin echar mano a módulos externos.

Y también es portable, ya que corre en muchas plataformas. Y no estamos hablando solamente de las tres principales (Linux, Windows y MacOS) sino también las derivadas de Unix (como FreeBSD, o Solaris) y algunas más inusuales como Cygwin o AIX.

1.1.2. Biblioteca estándar y módulos externos

Como decíamos anteriormente, parte de la funcionalidad está integrada en Python como lenguaje, y luego tenemos la biblioteca estándar con gran cantidad de módulos y paquetes. Pero también existen módulos y paquetes de terceros. En esta sección vamos a formalizar esos conceptos, y a echar luz sobre cómo aprovechar estas funcionalidades.

Podemos pensar que tenemos tres zonas. La más interior, donde podemos utilizar directamente lo que allí se encuentra, una intermedia donde para usar la funcionalidad tenemos que importar un módulo, y la más externa, donde tenemos también que importar el módulo pero además debemos instalarlo.



1.1.2.1. Integrada en Python

También denominada *builtin*, es la funcionalidad que está adentro de Python mismo como intérprete del lenguaje. Podemos separarla por un lado en aquella que tenemos a disposición por la sintaxis y semánticas mismas del lenguaje, y por el otro en la gran cantidad de objetos que podemos utilizar directamente.

La sintaxis y semántica del lenguaje en sí se encuentra detalladas en la [Referencia del Lenguaje](#), una sección de la documentación pensada para entender cómo funciona realmente el lenguaje y sus estructuras internas. Si estamos arrancando con Python no hace falta que nos involucremos con estos documentos ahora, pero es un paso obligado cuando el conocimiento del lenguaje se profundiza.



Python realmente no es un lenguaje, es la especificación de un lenguaje. La implementación de esa especificación más conocida es la que está escrita en C (a la que llamamos cPython cuando queremos marcar esa diferencia). Otras implementaciones son Jython (un Python escrito en Java), IronPython (en C#), MicroPython (que corre nativo en algunos microcontroladores), y notablemente PyPy, un Python escrito... ¡en Python!

Un ejemplo de funcionalidad parte del lenguaje mismo es la capacidad de poder llamar a una función con una cantidad variable de argumentos.

Por otro lado, los objetos que tenemos disponibles para utilizar directamente por estar integrados en el lenguaje están documentados al principio de la [Referencia de la Biblioteca Estándar](#), agrupados en funciones, algunas constantes, tipos de datos y excepciones.

Como ejemplos de esto último podemos mencionar la función `len`, que nos permite saber el largo de una estructura, o el tipo de dato `int`, para representar enteros.

1.1.2.2. Parte de la Biblioteca Estándar

El resto de la [Referencia de la Biblioteca Estándar](#) versa sobre aquella funcionalidad externa al intérprete de Python en sí mismo, pero disponible en cualquier instalación de Python.

Está agrupada en distintos módulos, y para acceder a ella debemos importar esos módulos (más detalle sobre módulos y paquetes en [4.3](#)).

Por ejemplo, para calcular el factorial de un número, podemos usar la función `factorial` del módulo `math`, que es el que debemos importar primero.

1.1.2.3. El resto del mundo

Si queremos funcionalidad de terceros, que no está integrada a Python o en la biblioteca estándar, siempre podemos instalar módulos disponibles en Internet.

La forma más sencilla y directa de hacer esto es utilizar el programa `pip`, que automáticamente descarga lo que indiquemos del Índice de Paquetes de Python (en inglés Python Package Index, al que normalmente conocemos como PyPI) y luego lo instala en nuestro sistema o algún entorno en particular.



Hay una convención para nombrar PyPI y separarlo de PyPy (el Python hecho en Python): PyPI lo nombramos “pai-pi-ai”, y a PyPy “pai-pai”.

Por supuesto, también podemos utilizar el administrador de paquetes de nuestro sistema (como apt en Debian/Ubuntu), utilizar pip para instalar paquetes provenientes de otros repositorios (como Github) o directamente podemos descargar los paquetes necesarios a mano e instalarlos.

A lo largo del libro iremos utilizando varios paquetes que se necesitan instalar de esta forma, los cuales estarán detallados al principio de cada capítulo. En función de eso recomendamos repasar el capítulo donde mostramos más en detalle las distintas alternativas para instalar paquetes [7](#).

1.2. Editando, corriendo, e interpretando

En esta sección veremos cómo hacer nuestro primer programa en Python. Algo sencillo, pero que nos permitirá explorar distintas formas de editarlo y ejecutarlo, y otros conceptos aledaños.

1.2.1. Editando y ejecutando Python

Mencionamos arriba que podemos considerar que Python es interpretado, pero realmente es compilado. A fines prácticos, especialmente al arrancar, podemos considerar efectivamente que es interpretado: nosotros le pasamos el código fuente, y Python *lo ejecuta*.

¡Probemos eso! Agarremos cualquier editor (más adelante profundizamos ahí) y escribamos un programita muy simple, y lo grabamos en un archivo `hola.py`:

```
1 print('¡Hola mundo!')
```

Luego, en una terminal ejecutamos nuestro programa...

```
$ python3 hola.py
¡Hola mundo!
```

Es así de simple. Grabamos el archivo, lo ejecutamos, tenemos el resultado. Si queremos realizar modificaciones al archivo, volvemos a grabar, volvemos a ejecutar. Nada más. Este “ciclo corto” hace que iterar sobre la construcción de un programa sea muy eficiente en Python.

Por otro lado, ¿qué es eso de ir a la terminal para ejecutar un programa? Bueno, tenemos principalmente dos tipos de programas, con y sin interfaz gráfica. El que mostramos arriba no posee una interfaz gráfica, entonces lo usamos desde la terminal.

Si nuestro programa fuese de interfaz gráfica, podríamos ir y hacerle doble-click al programa y que se ejecute. En realidad también lo podemos hacer en un programa sin interfaz gráfica, pero todo lo que veríamos es una terminal que nuestro sistema abriría para ejecutar el programa, que se cerraría cuando este termine (y si el programa termina rápidamente, como el ejemplo de arriba, veríamos sólo un parpadeo).

Más adelante nos encargaremos de hacer programas con interfaz gráfica (ver Capítulo 10), pero por ahora enfoquémonos en saber al menos cómo ejecutar cualquier tipo de programa: desde la terminal.

Obviamente para ejecutar nuestro programa con Python necesitamos a Python instalado en nuestro sistema. Lo vamos a encontrar ya instalado en cualquier Linux o MacOS, pero vamos a necesitar instalarlo en Windows. En cualquier caso, todas las versiones para todos los sistemas están en la [página oficial de descargas](#).

Como mostramos arriba, siempre podemos pasarle nuestro programa al intérprete de Python para que lo ejecute. Si queremos poder ejecutarlo directamente, quizás tengamos que realizar una acción extra, dependiendo de nuestro sistema. En Windows, el instalador mismo de Python asocia la extensión `.py` al intérprete de Python, entonces no necesitamos hacer nada.

En Linux y MacOS, por otro lado, tenemos que hacer que nuestro programa sea ejecutable e indicar en el mismo programa que debe ejecutarse con Python. Veamos esto en detalle. Primero modificamos el programa, agregando una línea muy particular al principio y luego una línea en blanco:

```
1 #!/usr/bin/python3
2
3 print('¡Hola mundo!')
```

Esa línea particular del principio tiene una estructura muy específica: arranca con estos dos caracteres `'#!'` (llamados *shebang*) que indican que a continuación está el programa que va a interpretar las líneas del archivo, `python3` en nuestro caso.

Luego hacemos ejecutable al programa, y lo corremos directamente:

```
$ chmod +x hola.py
$ ./hola.py
¡Hola mundo!
```

1.2.2. Editando Python

Python tiene una sintaxis tan sencilla y limpia que no se necesita demasiada ayuda del editor que estemos usando para escribir código.

Entonces, cualquier editor de texto que sea útil para escribir programas (NO Microsoft Word, por ejemplo) nos es suficiente. Ejemplos de este tipo son Vim, Emacs, Kate, Textmate o Notepad++.

Habiendo dicho eso, igualmente mucha gente prefiere desarrollar utilizando Entornos de Desarrollo Integrados (en inglés, *Integrated Development Environment*, lo que forma la sigla “IDE” que es la que se usa normalmente también en castellano). Esto es porque un IDE integra el editor (simple, como decíamos) con un navegador de archivos, herramientas de *debugging*, un intérprete interactivo, y muchas funcionalidades más. Ejemplos de IDEs son PyCharm, VisualStudio, o Spyder.

Esas funcionalidades extras que proveen los IDEs son tan útiles que incluso se han ido montando herramientas sobre los editores simples para obtener algunas de ellas. Quizás el ejemplo más famoso de eso es la [configuración de Fisa para Vim](#).

En cualquier caso, usar un editor simple más bien pelado o un IDE súper completo (o cualquier intermedio entre esos dos extremos) no deja de ser una elección personal. Nuestra recomendación es que las personas nuevas a la programación no se traben demasiado en arrancar con la herramienta ideal, que elijan una más o menos rápido y se pongan a programar, la elección del editor o IDE ideal para esa persona irá decantando con el tiempo.

Para elegir con qué arrancar siempre es bueno revisar la página de Python Argentina dedicada a [editores e IDEs](#).

1.2.3. Usando módulos

Un concepto básico que necesitamos para arrancar es cómo utilizar los módulos que Python trae en la Biblioteca Estándar. Porque el programa que hicimos arriba usa la función integrada `print`, pero enseguida vamos a tener que empezar a utilizar otras funciones no integradas.

Para usar un módulo primero tenemos que importarlo. La forma más directa de hacer esto es a través de la declaración `import`, y luego podremos acceder a los contenidos del módulo a través de la “notación punto” (este es uno de esos conceptos genéricos del lenguaje: siempre que queramos acceder a algo que está adentro de un objeto, podemos usar el punto: `'.'`).

Veamos un simple ejemplo.

```
1 import math
2
3 print('Raíz cuadrada de dos:', math.sqrt(2))
```

En la línea 1 vemos que importamos el módulo `math` y luego llamamos a la función que nos calcula la raíz cuadrada usando el “punto”: `math.sqrt(2)`.

Pueden encontrar más detalles sobre módulos y distintas formas de importarlos y usarlos en [4.3](#).

1.2.4. El intérprete interactivo

En realidad no tenemos que escribir todo un programa para probar algo en Python. Una de las mejores características del lenguaje es que trae incorporado un intérprete interactivo. Llamamos “interactivo” a este modo del intérprete, porque Python compilará y ejecutará cada línea que escribamos, sin necesidad de otros pasos.

Si en la terminal ejecutamos² `python3` sin pasarle ningún programa, Python automáticamente abrirá el intérprete interactivo, mostrándonos un *prompt* y esperando que escribamos algo:

```
$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

²En Windows, donde el uso de la terminal no está tan extendido, la instalación de Python incluye a IDLE, un entorno integrado y de aprendizaje que muestra un intérprete interactivo.

Obviamente las primeras líneas pueden variar en función de la versión de Python que tengamos instalada, pero lo importante es ese `>>>` que nos habilita a empezar a escribir.

A lo largo de este libro veremos muchísimos ejemplos de código que son realizados directamente en el intérprete interactivo, y justamente nos podemos dar cuenta de eso por el prompt. Por ejemplo, reveamos como importar y usar un módulo, pero sin tener que escribir un programa para ello:

```
1 >>> import math
2 >>> print(math.sqrt(2))
3 1.4142135623730951
```

En la línea 1 del ejemplo tenemos un `import` que no devuelve nada, por eso en la línea 2 volvemos a tener directamente el prompt, donde ahora escribimos el `print` que nos mostrará la raíz cuadrada de dos en la línea 3 (que no tiene el prompt, porque no es una línea para que escribamos nosotros algo).

En realidad no es necesario hacer un `print` para ver un resultado en el intérprete interactivo, ya que este luego de ejecutar cada línea nos mostrará abajo el objeto resultante de esa línea, a menos que sea `None`.



`None` es el equivalente de Python al más conocido NULL en otros sistemas. No es ni verdadero ni falso (aunque a nivel booleano evalúa a falso), no es vacío, ni cero. Es *la nada*. La ausencia de algo. Es `None`.

Esto nos permite usar el intérprete de forma muy sencilla para explorar distintos comportamientos:

```
1 >>> 2 + 3
2 5
3 >>> len("hola")
4 4
```

Hay una diferencia significativa, sin embargo, entre la salida de `print` y lo que nos muestra el intérprete interactivo en cada resultado. Para construir la salida (convertir el objeto resultado a una representación textual a mostrar en la terminal) el `print` utiliza la función integrada `str`, que nos deja la representación más “simple y humana” del objeto, mientras que el intérprete interactivo usa la función integrada `repr`, que apunta a lograr la representación más “exacta” del objeto.

Vemos en el siguiente ejemplo como con el `print` corremos el riesgo de confundir el tipo de dato del resultado (¡parece un número!), mientras que con el `repr` es evidente que es una cadena de texto:

```
1 >>> print("23")
2 23
```

```
3 >>> "23"  
4 '23'
```

En realidad el intérprete interactivo procesa la línea que acabamos de escribir cuando esa línea termina, lo que nos lleva a marcar la diferencia entre líneas lógicas y líneas reales: el intérprete interactivo se da cuenta cuando la línea todavía no terminó (aunque hayamos apretado ENTER) porque falta cerrarla lógicamente. Veamos un ejemplo de eso, donde la línea continúa porque falta cerrar el paréntesis del `print`, y veamos como el intérprete nos marca esa “línea continuación” con tres puntos abajo del prompt:

```
1 >>> print("Hola",  
2 ... 123)  
3 Hola 123
```

1.2.5. Jupyter Notebook

El intérprete interactivo que trae Python por default no es el único que existe, sino que tenemos a disposición múltiples alternativas, especializadas para distintos fines.

Quizás el intérprete alternativo más popularizado es [IPython](#), un shell interactivo que añade funcionalidades extras al intérprete interactivo incluido en Python, como resaltado de líneas y errores mediante colores, una sintaxis adicional para el shell, etc.

La funcionalidad núcleo de IPython fue evolucionando en el tiempo y se convirtió en una aplicación web llamada Jupyter Notebook. Por su baja barrera de entrada para interactuar con el mismo, soporte para integrar textos y gráficos entre las celdas de ejecución, y facilidad para la distribución del contenido, Jupyter Notebook se ha vuelto el intérprete interactivo de facto en el ámbito científico.

En las próximas secciones utilizaremos casi exclusivamente Jupyter Notebooks para los ejemplos mostrados en el libro (y en todos los casos haremos referencia al notebook real, accesible en internet, para que puedan descargarlo, modificarlo y jugar con los ejemplos, que es una muy buena manera de aprender).

1.2.6. Explorando

Tanto el intérprete interactivo como Jupyter Notebook nos permite explorar el lenguaje y sus comportamientos. Esto es especialmente evidente si utilizamos las funciones integradas `dir` y `help`, que nos permiten ver los interiores de un objeto y directamente pedir ayuda en la terminal y ver su documentación.

En el siguiente ejemplo vemos como podemos descubrir qué atributos tiene el tipo de dato `list` y pedir ayuda sobre uno de ellos.

```
1 >>> dir([])  
2 ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',  
3 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',  
4 '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
```

```

5 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
6 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
7 '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
8 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
9 'remove', 'reverse', 'sort']
10 >>> help([].count)
11
12 Help on built-in function count:
13
14 count(value, /) method of builtins.list instance
15 Return number of occurrences of value.

```

Siempre que miremos dentro de los distintos objetos vamos a encontrar muchos atributos que empiezan y terminan con doble guión bajo. Estos son métodos especiales que permiten a los objetos integrarse y utilizar la sintaxis y semántica del lenguaje en sí (ya sean estos objetos integrados en Python, creados por nosotros o por terceros. Mientras estemos aprendiendo Python podemos ignorarlos tranquilamente, dejándolos para cuando avancemos con el lenguaje. Habiendo dicho eso, durante el libro mencionaremos y utilizaremos algunos, que explicaremos oportunamente.



Es difícil y cansador decir, por ejemplo para referirse al `__len__`, “doble guión bajo len doble guión bajo”, o su equivalente en inglés *double underscore len double underscore*, por eso se creó una forma especial para nombrarlos: usando *dunder* (que es una especie de abreviatura del double underscore en inglés), con lo que para el ejemplo nos quedaría *dunder len*.

1.3. Cómo pedir ayuda

Una de las grandes cosas buenas de Python es su Comunidad, alrededor del mundo y en infinidad de idiomas.

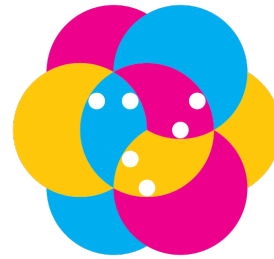
Siempre vamos a encontrar un foro, una lista de correo, o algún recurso gratuito puesto a disposición por gente tratando de ayudar. Y la misma Comunidad se autorregula para seguir siendo sana, tanto informalmente como formalmente, por ejemplo creando la Python Software Foundation en Estados Unidos, o la [Asociación Civil Python Argentina](#), organizaciones que aprueban y alientan la participación de todos. Nuestra Comunidad está basada en respeto mutuo, tolerancia y fomento, y estamos trabajando para ayudar a cada uno y a cada una a vivir a la altura de estos principios. Queremos [que la Comunidad sea más diversa](#): quien quieras que seas, y cualquiera sean tus orígenes, te recibiremos.

Los canales de comunicación virtuales son el principal medio de comunicación e integración de la Comunidad, y es un buen punto de entrada para cuando tenemos consultas y preguntas. Aprender a utilizar estos mecanismos es un activo tan importante en un programador o una desarrolladora de software como saber tal función, tal estructura del lenguaje, o tal detalle de implementación.

Es por esto que queremos hacer mucho énfasis en que busquen como interactuar con la Comunidad, cuales grupos de usuarias y usuarios tienen cerca, qué lista de correo o foro les apetece más, y empiecen a participar allí. Si ya tienen consultas pueden hacerlas, pero también es muy valioso leer las preguntas de otras personas y las respuestas que se ofrecen, y también podemos responder. Esta es una de las mejores formas de profundizar nuestro conocimiento en Python.

Te recomendamos explorar el sitio de [Python Argentina](#), o entrar directamente en su [grupo de Telegram](#) o en [el foro](#). Pero no dejes de revisar si hay alguna [Comunidad local](#) más cerca de donde estés.

Y siempre está la Comunidad global de Python, con muchísimos recursos en [su sitio](#), en particular las [listas de correo](#) y [el foro](#), pero claro, esto es todo en inglés.



En el [sitio de Python Argentina](#) podrán encontrar muchos recursos en castellano: foros de consulta, cartelera laboral, etc.

1.4. Zen de Python

El Zen de Python [1] es una lista de frases breves que describen cómo debería ser un buen código Python. No son reglas estrictas, sino principios para tomar decisiones cuando hay más de una forma de hacer algo.



Una sorpresa oculta en Python (lo que en inglés llamamos *easter egg*) es que si en una consola de Python escribimos `import this` el resultado serán las frases del Zen.

Lo escribió Tim Peters, un desarrollador muy influyente en la Comunidad Python, allá por 2004, como mezcla de guía práctica y valores estéticos que refleja muchas de las ideas que Guido van Rossum promovía desde los comienzos del lenguaje.

En efecto, Tim Peters logró una colección de principios que resumen la filosofía de diseño de Python, lo cual es importante porque ayuda a los programadores a escribir código limpio y mantenible, e incluso ha influido no solo en Python, sino también en cómo se enseña y se piensa la programación.

Incluimos entonces aquí las frases traducidas:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.

- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!



2 | Tipos de datos

En este capítulo mostraremos los tipos de datos más utilizados en Python.

No incluiremos *todos* los tipos de datos integrados en el lenguaje (ni mucho menos todos los presentes en la Biblioteca Estándar), sólo haremos foco en aquellos más utilizados y que les permitirá comenzar y realizar gran parte de todo lo que harán con Python.

[Código disponible](#)

2.1. Números

Comenzamos con el tipo de datos más básico en todos los lenguajes: los números enteros, cuyo tipo de datos es `int`.

Las operaciones básicas son las mismas que en cualquier otro lenguaje de programación y no tenemos ninguna sorpresa. Los únicos detalles a destacar son que la división de números enteros da como resultado un `float` (el punto flotante binario, que veremos a continuación), y que el operador para la exponenciación es `**`.

CELDA 1 DE TIPOS-NUMS.IPYNB
<code>3 + 5</code>
8

CELDA 02
<code>2 - 8</code>
-6

CELDA 03
<code>4 * -8</code>
-32

CELDA 04
<code>32 / 5</code>
6.4

CELDA 05
<code>5 ** 2</code>
25

Complementando la división, Python incluye un operador para la división con el resultado truncado (celda 6), el módulo (celda 7), e incluso una función integrada que da ambos resultados al mismo tiempo (celda 8):

CELDA 06
<code>32 // 5</code>
6

CELDA 07
<code>32 % 5</code>
2

CELDA 08
<code>divmod(32, 5)</code>
(6, 2)

Más allá de estas operaciones particulares, el rasgo más relevante a destacar en los enteros de Python es que no tienen límite (porque no están atados a ninguna representación en particular), entonces no hace falta que nos preocupemos por ese detalle:

CELDA 09
2529 ** 15 * 1834
2030634551567076694888541641414492972028354488232975466

Los números enteros en Python también funcionan como secuencias de bits, y más allá que usemos la representación decimal para expresarlos, o en binario, o en hexadecimal, simplemente son números enteros. Escribirlos en otras bases es sencillo, y para verlos en esas representaciones tenemos funciones integradas:

CELDA 10
0b1001010
74

CELDA 11
0x3f
63

CELDA 12
hex(63)
'0x3f'

CELDA 13
bin(74)
'0b1001010'

Python tiene operadores para trabajar con los enteros como secuencias de bits: el “and” (&), el “or” (|), el “xor” (^), y los “shift” a izquierda y derecha (<< y >>).

CELDA 14
bin(0b1001 & 0b0001)
'0b1'

CELDA 15
bin(0b1001 0b0001)
'0b1001'

CELDA 16
bin(0b1001 ^ 0b0001)
'0b1000'

	CELDA 17
<code>bin(0b1100 << 2)</code>	
	<code>'0b110000'</code>

	CELDA 18
<code>bin(0b1100 >> 2)</code>	
	<code>'0b11'</code>

El punto flotante binario (`float`) de Python, por otro lado, es exactamente el punto flotante ejecutado en el procesador, y tiene el mismo comportamiento que en los otros lenguajes:

	CELDA 19
<code>2.35 * 28</code>	
	<code>65.8</code>

	CELDA 20
<code>15 + 2.3</code>	
	<code>17.3</code>

	CELDA 21
<code>95 / 17</code>	
	<code>5.588235294117647</code>

	CELDA 22
<code>2.3 ** 15</code>	
	<code>266635.23546439095</code>

Así y todo siendo uno de los tipos de datos más utilizados en todos los lenguajes, hay ciertas trampas del punto flotante binario que es imperioso conocer, y a los que les dedicamos un capítulo separado en [12](#).

Uno de esos detalles es que no debemos utilizar `float` para manejar dinero. De esa limitación nació la implementación del punto flotante decimal, que podemos encontrar en el módulo `decimal` de la Biblioteca Estándar.

	CELDA 23
<code>from decimal import Decimal</code>	
<code>Decimal(23) * 53</code>	
	<code>Decimal('1219')</code>

	CELDA 24
Decimal(15.4) + 87	
Decimal('102.400000000000003552713679')	

	CELDA 25
<code>Decimal('15.4') + 87</code>	
<code>Decimal('102.4')</code>	



La forma `from ... import ...` nos permite importar el módulo y usar sus objetos directamente, sin tener que luego repetir el nombre del módulo todas las veces. Más detalles en [4.3](#).

Como vemos en el ejemplo, no es lo mismo construir el Decimal a partir de la representación textual de un número “con coma” que de un `float`; volvemos a hacer referencia al capítulo 12 donde nos adentramos en estos detalles.

Relacionado, tenemos la posibilidad de directamente utilizar fracciones si así fuese necesario, a través del módulo `fractions`:

CELDA 26

```
from fractions import Fraction

Fraction(1, 3) * 6

Fraction(2, 1)
```

		CELDA 27
Fraction(27, 5) ** 2		
Fraction(729, 25)		

	CELDA 28
Fraction(2, 3) + Fraction(6, 9)	

Fraction(4, 3)	

Finalmente, debemos mencionar el tipo de datos `complex`, integrado en el lenguaje, para manejar números complejos. En verdad tenemos soporte en la sintaxis para números imaginarios (agregándoles una `j` al final), y con ello armamos los complejos directamente:

CELDA 29	
3j	
3j	

CELDA 30
$3j + 2$ $(2+3j)$
CELDA 31
$(3j + 2) * (7 + 2.5j)$ $(6.5+26j)$
CELDA 32
$1j ** 2$ $(-1+0j)$
CELDA 33
<pre>import math x = 1.5 math.cos(x) + 1j * math.sin(x) == math.e ** (x * 1j)</pre> <pre>True</pre>

Las clases de cada tipo de dato funcionan como conversores entre un tipo de dato y otro, con un resultado específico en cada caso si las llamamos sin pasarles un valor.

CELDA 34
<pre>int()</pre> <pre>0</pre>
CELDA 35
<pre>int(9.2)</pre> <pre>9</pre>
CELDA 36
<pre>float(-7)</pre> <pre>-7.0</pre>

2.2. Cadenas

Las cadenas constituyen otro de los tipos de datos básicos que están presentes en infinidad de lenguajes. En Python no tienen nada de especial a primera vista, pero vamos a ir explorando algunas particularidades más adelante.

Por lo pronto hagamos la definición formal: las cadenas en Python son secuencias de caracteres Unicode delimitadas por comillas dobles o simples indistintamente.

En el siguiente ejemplo vemos casos de uno y otro delimitador, especialmente para el caso donde necesitamos usar uno de ellos como carácter dentro de la cadena:

CELDA 1 DE TIPOS-CADS.IPYNB
"Hola mundo"
'Hola mundo'

CELDA 02
'Python :)'
'Python :)'

CELDA 03
"Let's rock!"
"Let's rock!"

También como en tantos otros lenguajes, la barra invertida tiene el propósito de escapar ciertos caracteres especiales (y es un último recurso para usar los delimitadores dentro de la cadena). Hay que tener en cuenta siempre que la barra invertida sirve para escaparse a sí misma, y que los caracteres especiales suelen mostrarse distinto en el modo `str` (representación más humana) que en el modo `repr` (representación más exacta):

CELDA 04
"Hola\nmundo"
'Hola\nmundo'

CELDA 05
<code>print("Hola\nmundo")</code>
Hola mundo

CELDA 06
"foo \" bar"
'foo " bar'

CELDA 07
"foo \" bar ' baz"
'foo " bar \' baz'

CELDA 08
<pre>'foo \'\' " bar'</pre>
<pre>'foo \'\' " bar'</pre>

CELDA 09
<pre>print('foo \'\' " bar')</pre>
<pre>foo \'\' " bar</pre>

Python tiene otro delimitador para cadenas: la triple comilla (ya sea doble o simple), que nos permite escribir una cadena a través de múltiples líneas:

CELDA 10
<pre>mensaje = """ Hola mundo :) """ print(mensaje)</pre>
<pre>Hola mundo :)</pre>

CELDA 11
<pre>mensaje</pre>
<pre>'\nHola\n mundo\n :)\n'</pre>

Las operaciones básicas a realizar sobre las cadenas están integradas en el lenguaje: saber el largo, concatenar y repetir:

CELDA 12
<pre>len("Hola")</pre>
<pre>4</pre>

CELDA 13
<pre>"Ho" + 'la'</pre>
<pre>'Hola'</pre>

CELDA 14
<pre>'Na' * 5 + " Batman!"</pre>
<pre>'NaNaNaNaNa Batman!'</pre>

Las cadenas tienen también muchos métodos para trabajar con las mismas, veamos algunos

de ellos (sólo a modo de ejemplo, les recomendamos revisar todos los métodos en la sección correspondiente de la [Referencia de la Biblioteca](#), ya que son muy útiles en el día a día):

CELDA 15
<pre># transformamos la cadena a mayúsculas "Moño".upper()</pre>
<pre>'MOÑO'</pre>
CELDA 16
<pre># hacemos que la cadena sea un "título" "python is a great language".title()</pre>
<pre>'Python Is A Great Language'</pre>
CELDA 17
<pre># buscamos la posición de "great" en la cadena "python is a great language".index("great")</pre>
<pre>12</pre>
CELDA 18
<pre># preguntamos si la cadena está formada exclusivamente por dígitos "12345".isdigit()</pre>
<pre>True</pre>
CELDA 19
<pre># sacamos los "blancos" (espacios, tabs, enters) del principio y final "\t\t corrido \n".strip()</pre>
<pre>'corrido'</pre>

El método `join` merece una mención especial. Arma una cadena a partir de los elementos que encuentra en otro objeto. Todavía no vimos el concepto de “iterable”, pero mencionemos que cuando recorremos una cadena vamos encontrando cada uno de sus caracteres. Entonces, el `join` construirá una nueva cadena separando los caracteres recibidos con la cadena dada.

CELDA 20
<pre>"-".join("hola")</pre>
<pre>'h-o-l-a'</pre>
CELDA 21
<pre>". . ".join("CIENCIA")</pre>
<pre>'C. .I. .E. .N. .C. .I. .A'</pre>

El `join` acepta otros iterables, por supuesto, y podemos pasarle listas, tuplas, y otros tipos de datos que veremos más adelante, siempre y cuando sus elementos sean cadenas.

Un método en particular de gran utilidad es el `format`, que permite armar cadenas reemplazando valores, lo cual es mucho más legible y ofrece más control que andar concatenando cadenas individuales. Más allá de estos ejemplos puntuales que mencionamos a continuación, el sistema de formateo en cuestión es muy poderoso, y vale la pena al menos sobrevolar [su documentación](#).

CELDA 22
<pre>"Tiempo estimado: {:.2f}s".format(1.34844)</pre>
<pre>'Tiempo estimado: 1.35s'</pre>

CELDA 23
<pre>"Valor para muestra {:05d}: {:>13s} ".format(7, "failure")</pre>
<pre>'Valor para muestra 00007: failure '</pre>

Notablemente, no mencionamos ningún método para acceder a un carácter de la cadena, o a una subcadena de la misma. Esto se debe a que dicha funcionalidad está integrada en el lenguaje y es la misma para todas las secuencias.

La forma de acceder a un carácter de la secuencia es utilizando los corchetes, escribiendo directamente la posición entre ellos:

CELDA 24
<pre>cad = "Hola mundo" cad[0]</pre>
<pre>'H'</pre>

CELDA 25
<pre>cad[2]</pre>
<pre>'l'</pre>

CELDA 26
<pre>cad[-2]</pre>
<pre>'d'</pre>

En el ejemplo vemos que la primer posición de la secuencia es la número 0, y que si utilizamos números negativos se empieza a contar desde el final (lo cual es muy práctico, porque no hay que calcular la posición que queremos restándole algo al largo de la cadena).

Para obtener subcadenas también podemos usar corchetes, pero entre ellos vamos a escribir dos valores, “desde” y “hasta”. Si no incorporamos alguno de esos números se usará un valor por defecto (“desde el principio” o “hasta el final” respectivamente), y también pueden ser valores negativos:

	CELDA 27
<pre>cad = "Hola mundo" cad[2:6]</pre>	
	'la m'

	CELDA 28
<pre>cad[:6]</pre>	
	'Ho la m '

	CELDA 29
<pre>cad[2:]</pre>	
	'la mundo'

	CELDA 30
<pre>cad[-3:]</pre>	
	' ndo '

Esta operación de tomar la subcadena de una cadena en inglés se llama *slicing*, y en general usamos ese término también en castellano (porque “rebanar” cadenas nunca terminó prendiendo).

Entender qué carácter se accede por posición es sencillo (sólo tenemos que acordarnos contar desde 0), pero en el caso de los slices es más difícil, hasta que nos acostumbramos. Una buena regla mnemotécnica es pensar que también arranca desde 0, pero lo que se cuentan son las “separaciones entre las letras”. Más allá de cómo nos acordemos, la regla que tiene Python para numerar en los slices tiene dos propiedades muy útiles: por un lado si hacemos `cad[x:y]` con valores positivos el largo de la subcadena va a ser $y - x$, y por el otro es sencillo separar una cadena en dos, porque se usa el mismo número, o sea que `cad[:x] + cad[x:]` nos termina dando la misma cad.

También como todas las secuencias, si queremos tomar elementos de la misma pero con un determinado paso, podemos utilizar un tercer valor entre los corchetes:

	CELDA 31
<pre>cad = "Hola mundo, chau mundo\n" cad[:-1:2]</pre>	
	'Hl ud,ca ud'

En el ejemplo usamos varias propiedades al mismo tiempo. Por un lado el “desde” lo dejamos en blanco, para que tome el inicio por default, luego el “hasta” lo usamos negativo porque nos interesaba obviar el *newline* del extremo derecho, y finalmente el tercer valor, el paso, en dos, para ir agarrando cada dos caracteres.

Como con los números, podemos usar el constructor `str` de la cadena para convertir desde otros tipos de datos (así como podemos también usar los constructores de números para pasar de cadenas a enteros o flotantes):

CELDA 32
<code>str(2.3)</code>
<code>'2.3'</code>

CELDA 33
<code>float("1.44")</code>
<code>1.44</code>

CELDA 34
<code>int("23")</code>
<code>23</code>

CELDA 35
<code>int("3F", base=16)</code>
<code>63</code>

Cuando mencionamos que los delimitadores de las cadenas eran la comilla doble, o simple, o sus variantes “triple”, evitamos mencionar que en todos estos casos se puede poner un prefijo para cambiar el tipo de cadena definida. Por default si no especificamos nada (o si usamos la letra “u”, por compatibilidad con versiones viejas de Python), la cadena será de tipo Unicode, lo que implica que será una secuencia de caracteres Unicode, con lo cual podemos escribir caracteres con acento, o en otros idiomas:

CELDA 36
<code>"El camión llegará mañana"</code>
<code>'El camión llegará mañana'</code>

CELDA 37
<code>"El valor de π es aproximadamente 3.14"</code>
<code>'El valor de π es aproximadamente 3.14'</code>

CELDA 38
<code>"¡Hola! привет"</code>
<code>'¡Hola! привет'</code>

Tengamos en cuenta que considerar a los caracteres como Unicode, más allá de su posible representación como bytes, nos permite trabajar con herramientas de transformación de texto sin mayor inconveniente:

CELDA 39
<pre>m = "moño" len(m)</pre>
4

CELDA 40
<pre>m.upper()</pre>
'MOÑO'

Si necesitamos trabajar con bytes directamente, podemos tener cadenas de bytes (donde no es más una secuencia de caracteres, sino una secuencia de bytes) si prefijamos la cadena con la letra “b”. En estos casos es muy útil la notación \x para ingresar el valor del byte en hexadecimal, como mostramos en el siguiente ejemplo:

CELDA 41
<pre>val = b"ab\x00\xffcd" val</pre>
b'ab\x00\xffcd'

CELDA 42
<pre>len(val)</pre>
6

CELDA 43
<pre>val[3]</pre>
255

Es importante entender esta distinción no sólo porque las cadenas de bytes eran el default en versiones viejas de Python (y nos podemos cruzar con algún código así) sino también porque eventualmente necesitaremos convertir las cadenas Unicode a cadenas de bytes, ya que es la única manera de mandar cadenas a través de la red o grabarlas en disco.

Estas conversiones las hacemos con los métodos encode y decode. Aunque en el siguiente ejemplo parece simple, el tema de convertir de un lado para el otro le trae muchos dolores de cabeza a la mayoría de los programadores (más allá del lenguaje en que programen); les recomendamos que si quieren adentrarse en el tema vean la charla [Entendiendo Unicode](#) dada por uno de los autores del libro.

CELDA 44
<pre>mu = "moño" mu</pre>
'moño'

CELDA 45
<pre>mb = mu.encode("utf8") mb</pre>
<pre>b'mo\xc3\xblo'</pre>

CELDA 46
<pre>mb.decode("utf8")</pre>
<pre>'moño'</pre>

Otro tipo de cadena usada frecuentemente es la de tipo *raw* (término que usamos en inglés, porque decir que son “sin procesar” o “crudas” es raro), donde la diferencia con las cadenas comunes es que la barra invertida NO funciona como carácter de escape (lo cual es especialmente útil al escribir expresiones regulares, tema que por respeto a quienes lean este libro NO tocaremos aquí), sino que es tratada como un carácter normal:

CELDA 47
<pre>r"\n"</pre>
<pre>'\n'</pre>

CELDA 48
<pre>len(r"\n")</pre>
<pre>2</pre>

Finalmente, en las versiones más modernas de Python tenemos un tipo de cadena que se formatea automáticamente con las variables del entorno, sin tener que llamar explícitamente a `format`, permitiendo incluso expresiones y algunos detalles más:

CELDA 49
<pre>t = 1.34844 f"Tiempo estimado: {t:.2f}s"</pre>
<pre>'Tiempo estimado: 1.35s'</pre>

CELDA 50
<pre>vals = [1, 4, 2, 1, 7] f"Proceso completo! init={vals[0]} promedio={sum(vals)/len(vals)}"</pre>
<pre>'Proceso completo! init=1 promedio=3.0'</pre>

2.3. Listas

Las listas son secuencias de objetos, se delimitan con corchetes, y cada objeto (sus elementos internos, que pueden ser cualquier cosa) se separan con comas.

Se acceden como cualquier secuencia, exactamente como vimos antes con las cadenas [2.2](#), e incluso tienen la misma forma de repetirlas o concatenarlas:

CELDA 1 DE TIPOS-LISTAS.IPYNB
<pre>lista = [1, "a", 5.6, "foo"] lista</pre>
<pre>[1, 'a', 5.6, 'foo']</pre>

CELDA 02
<pre>len(lista)</pre>
<pre>4</pre>

CELDA 03
<pre>lista[1]</pre>
<pre>'a'</pre>

CELDA 04
<pre>lista[2:]</pre>
<pre>[5.6, 'foo']</pre>

CELDA 05
<pre>lista * 2</pre>
<pre>[1, 'a', 5.6, 'foo', 1, 'a', 5.6, 'foo']</pre>

CELDA 06
<pre>lista + [1, 2]</pre>
<pre>[1, 'a', 5.6, 'foo', 1, 2]</pre>

La gran diferencia con las cadenas a nivel comportamiento (más allá que unas son secuencias de caracteres y las otras son secuencias de cualquier objeto Python) es que mientras las cadenas son “inmutables”, las listas (como muchos otros tipos de datos) son “mutables”, o sea que pueden cambiar.

El detalle fino de esto se explica un poco más adelante [2.5](#), pero veamos como para las listas efectivamente tenemos métodos que permiten modificarlas:

CELDA 07
<pre>lista = [1, 2, 3] lista</pre>
<pre>[1, 2, 3]</pre>

CELDA 08
<pre>lista.append(4) lista</pre>
<pre>[1, 2, 3, 4]</pre>

CELDA 09
<pre>otra = [5, 6] lista.extend(otra) lista</pre>
<pre>[1, 2, 3, 4, 5, 6]</pre>

¡Y no sólo agregarle elementos! También podemos reemplazar algún elemento o toda una parte:

CELDA 10
<pre>lista</pre>
<pre>[1, 2, 3, 4, 5, 6]</pre>

CELDA 11
<pre>lista[1] = 99 lista</pre>
<pre>[1, 99, 3, 4, 5, 6]</pre>

CELDA 12
<pre>lista[2:] = [0, 0] lista</pre>
<pre>[1, 99, 0, 0]</pre>

También podemos borrar elementos, no sólo con el método que nos permite especificar cual elemento en sí queremos borrar, sino también con la sentencia **del**.

CELDA 13
<pre>lista</pre>
<pre>[1, 99, 0, 0]</pre>

CELDA 14
<pre>lista.remove(99) lista</pre>
<pre>[1, 0, 0]</pre>

CELDA 15	
<pre>del lista[1] lista</pre>	

[1, 0]	



La sentencia **del** es bastante especial en Python, ya que *borra* elementos, lo cual nunca es trivial en un sistema que administra memoria automáticamente. En parte por eso no es una función (como **len**), sino una sentencia, porque está muy integrada con el funcionamiento base del lenguaje.

Como en los otros casos, el constructor de las listas nos es bastante útil para crear listas a partir de otros objetos. **list** construye una lista a partir de los elementos del iterable que reciba:

CELDA 13	
<pre>lista</pre>	

[1, 99, 0, 0]	

CELDA 14	
<pre>lista.remove(99) lista</pre>	

[1, 0, 0]	

CELDA 15	
<pre>del lista[1] lista</pre>	

[1, 0]	

CELDA 16	
<pre>list("abcd")</pre>	

['a', 'b', 'c', 'd']	



Un “iterable” es cualquier objeto capaz de devolver sus elementos internos uno por vez, lo que nos permite iterarlos en un bucle for o para construir una lista, por ejemplo; una cadena es un iterable que devuelve caracteres, una lista es un iterable que devuelve sus objetos internos, etc.

2.4. Tuplas

Las tuplas también son secuencias de objetos, y se definen separando estos elementos con coma (como en la definición de las listas), y opcionalmente rodeándolas con paréntesis, por claridad.

CELDA 1 DE TIPOS-TUPLAS.IPYNB
(1, 'a', 2.3)
(1, 'a', 2.3)
CELDA 02
1, 5, 'xx'
(1, 5, 'xx')
CELDA 03
t = (5,)
t
(5,)
CELDA 04
len(t)
1
CELDA 05
t[0]
5

Una diferencia fundamental con las listas es que las tuplas son inmutables, y en general una buena recomendación para decidir su uso en un caso u otro es entender si para los objetos internos es importante su cantidad y posición. Por ejemplo, la lista es mejor para guardar los archivos que hay en un directorio (más allá del orden en sí, el archivo no es más o menos archivo si está antes o después, y en el directorio podemos tener 0, 3, 27, 1000 archivos), en cambio la tupla es mejor para guardar coordenadas (si es en el plano vamos a tener siempre una tupla de dos elementos, ni uno ni tres, y es importante si un número está primero o segundo, ya que son ejes distintos).

En realidad la diferencia es aún más profunda: las tuplas forman parte del núcleo más central del lenguaje, ya que se usa en infinidad de detalles internos. Por ejemplo, si recordamos la función interna `divmod` que explicamos cuando vimos números, veremos que esa función devuelve dos números... en verdad devuelve una tupla, que por otra característica de Python (“desempaquetado de tuplas”, que también tendemos a denominar en inglés como *tuple unpacking*) podemos usar directamente como si fueran dos objetos:

CELDA 06
<pre>divmod(60, 7)</pre>
<pre>(8, 4)</pre>

CELDA 07
<pre>a, b = divmod(60, 7) print("a={}, b={}".format(a, b))</pre>
<pre>a=8, b=4</pre>

Obviamente necesitamos la misma cantidad de elementos en la izquierda y la derecha del igual, para que el “desempaquetado” funcione, aunque tenemos la posibilidad de usar un “expansor” para que tome múltiples argumentos:

CELDA 08
<pre>a, b, c = 1, 2</pre>
<pre> ValueError Traceback (most recent call last) Input In [8], in <cell line: 1>() ----> 1 a, b, c = 1, 2 ValueError: not enough values to unpack (expected 3, got 2) </pre>

CELDA 09
<pre>a, b, c = 1, 2, 3, 4, 5, 6</pre>
<pre> ValueError Traceback (most recent call last) Input In [9], in <cell line: 1>() ----> 1 a, b, c = 1, 2, 3, 4, 5, 6 ValueError: too many values to unpack (expected 3) </pre>

CELDA 10
<pre>a, b, *c = 1, 2, 3, 4, 5, 6 print("a={}, b={}, c={}".format(a, b, c))</pre>
<pre>a=1, b=2, c=[3, 4, 5, 6]</pre>

CELDA 11
<pre>a, *b, c = 1, 2, 3, 4, 5, 6 print("a={}, b={}, c={}".format(a, b, c))</pre>
<pre>a=1, b=[2, 3, 4, 5], c=6</pre>

Una vez más, el constructor es la herramienta principal para convertir entre tipos:

CELDA 12
<pre>tuple("foobar")</pre> <pre>('f', 'o', 'o', 'b', 'a', 'r')</pre>
CELDA 13
<pre>tuple([2, None, 0.1])</pre> <pre>(2, None, 0.1)</pre>

Mencionábamos antes que en las tuplas cada elemento no es uno más sino que su posición es de importancia semántica. El siguiente paso en ese sentido es poder asignarle un nombre a cada posición, y para ello tenemos el tipo `namedtuple` (en el módulo `collections` de la biblioteca estándar).

Para usarlo necesitamos declarar la estructura, indicando los campos que vamos a tener y un nombre para esa estructura. Como ejemplo armamos una tupla que contiene un ítem de un subtítulo: desde y hasta cuando debería mostrarse en la película, y el texto.

CELDA 14
<pre>from collections import namedtuple</pre> <pre>SubItem = namedtuple("SubItem", ["time_from", "time_to", "text"])</pre>

Por simplicidad, podemos directamente nombrar los campos en una única cadena (incluso separándolos por coma), sin que cambie su significado:

CELDA 15
<pre># 100% equivalentes a la definición anterior</pre> <pre>SubItem = namedtuple("SubItem", "time_from, time_to, text")</pre> <pre>SubItem = namedtuple("SubItem", "time_from time_to text")</pre>

La podemos usar igual que una tupla normal, reemplaza a una tupla en todos sus aspectos, pero ya incluso al ver el objeto notamos que nos provee mucha más información que sólo los tres valores entre paréntesis:

CELDA 16
<pre>item1 = SubItem(2.3, 3.7, "Hola mundo")</pre> <pre>item1</pre> <pre>SubItem(time_from=2.3, time_to=3.7, text='Hola mundo')</pre>

Incluso la creación del objeto puede ser usando los nombres (obviamente sin importar el orden), con lo cual ya es imposible confundirse cuál valor corresponde a cuál posición:

CELDA 17
<pre>item2 = SubItem(text="Chau mundo", time_from=4.5, time_to=12.3)</pre> <pre>item2</pre>


```
SubItem(time_from=4.5, time_to=12.3, text='Chau mundo')
```

A los elementos los podemos acceder por posición (ya dijimos que se comporta como una tupla normal) con lo cual a priori no tenemos modificar el código que usa este elemento si reemplazamos una tupla normal por una `namedtuple`, pero la gran ventaja aquí es que podemos usar el nombre del atributo:

CELDA 18

```
print(item1[2], item1.text)
print(item2[0], item2.time_from)
```

```
Hola mundo Hola mundo
4.5 4.5
```

Noten en el segundo ítem como al acceder por posición se utiliza la definición de la estructura y no cómo se haya creado el elemento en sí.

Como cualquier nombre puede ser un atributo definido por el usuario, las `namedtuple` tienen sus propios métodos y atributos prefijados con un guión bajo, aunque no sean privados como a priori indicaría la convención.

Los más usados son para convertir la tupla en diccionario (usando los nombres como claves), listar los campos, e incluso generar una nueva tupla reemplazando algunos valores (ya que no podemos cambiar la `namedtuple` en sí, es tan inmutable como una tupla normal).

CELDA 19

```
item1._asdict()

{'time_from': 2.3, 'time_to': 3.7, 'text': 'Hola mundo'}
```

CELDA 20

```
item1._fields

('time_from', 'time_to', 'text')
```

CELDA 21

```
item1._replace(time_to=4.1)

SubItem(time_from=2.3, time_to=4.1, text='Hola mundo')
```

A los campos les podemos asignar valores por defecto, de manera que si los omitimos al crear el objeto tomarán esos valores:

CELDA 22

```
SubItem = namedtuple("SubItem", "time_from time_to text", defaults=[""])

item1 = SubItem(2, 3.5, "Un texto")
print(item1)
```

```

item2 = SubItem(2, 3.5)
print(item2)
-----
SubItem(time_from=2, time_to=3.5, text='Un texto')
SubItem(time_from=2, time_to=3.5, text='')

```

Los valores por defecto incluidos en esa lista corresponden a los últimos nombres definidos (ya que no pueden quedar argumentos sin valor por defecto luego de los que sí tienen, igual a lo que sucede con una función).

2.5. Pensando como un pythonista

Hay una diferencia fundamental en cómo Python maneja los objetos internamente en la ejecución de un programa con respecto a otros lenguajes, y es imperativo entenderla porque nos explica mucho del funcionamiento del lenguaje.

Python no tiene “variables”, y en consecuencia esas variables no tienen “valores”. Estos son términos y conceptos que provienen de otros lenguajes de más bajo nivel (es decir, más cercanos al procesador), y no se aplican en Python.

Python tiene “objetos”, y usamos “nombres” para hacer referencia a esos objetos. Es verdad que a veces por facilismo o sobresimplificación usamos las palabras “variable” y “valor” también cuando hablamos de un programa en Python o el funcionamiento del lenguaje, pero eso no nos tiene que confundir sobre el funcionamiento real.

Veamos la diferencia. Otros lenguajes, como C por ejemplo, sí poseen el concepto de “variable”, como lugar donde se guarda un valor. Ese lugar es en memoria, y el valor que se guarda son los bits en esa posición de memoria.

```

  0 1 1
  0 1 0
  0x3fc5b23a

```

En estos lenguajes es necesario declarar cómo se interpretan esos bits en esa posición de memoria. Por ejemplo, varios bytes seguidos pueden ser varias letras, o un número entero, e incluso podemos decirle al lenguaje que nos muestre esos bits de una manera u otra:

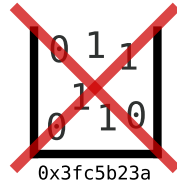
```

int a = 7303014;
printf("%i %s", a, (char *)&a);

--> 7303014 foo

```

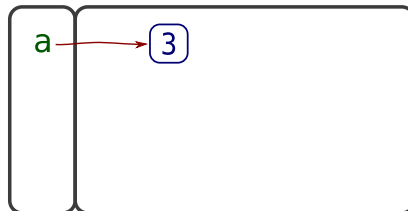
En cambio, en Python no seguimos esa filosofía.



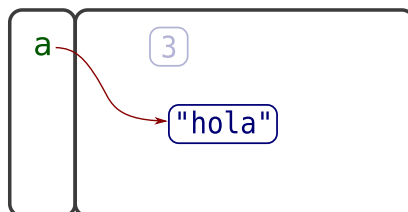
En Python manejamos directamente objetos. Estos objetos son de un tipo específico, y no pueden cambiar de tipo (“tipado fuerte”). Para referenciar esos objetos en memoria usamos nombres.



Estos nombres pueden apuntar a uno u otro objeto, y no están asociados al tipo del objeto que vinculan. Entonces, no hace falta decir que el nombre `a` va a ser un entero o una cadena, por ejemplo, porque el tipo pertenece al objeto que es nombrado. Cuando decimos que `a = 3` sólo estamos creando un objeto en memoria de tipo entero (el 3) y a ese objeto lo estamos referenciando usando el nombre `a`.



Si luego escribimos que `a = "hola"`, estamos creando otro objeto en memoria, este de tipo cadena, con el valor "hola", y estamos usando el mismo nombre `a` para referenciarlo. No es que `a` haya cambiado de tipo, sólo apunta a otro objeto.



Obviamente, como `a` apunta al segundo objeto, ya no apunta al primero. Si ese objeto no es accesible desde ningún otro lado, para todo propósito nosotros podemos considerar que ya no existe (eventualmente Python lo eliminará y liberará memoria, pero esto es algo que podemos ignorar tranquilamente ya que Python administra la memoria por nosotros).

En los diagramas mostrados arriba vemos dos áreas diferentes. El grande es el espacio de objetos (a grandes rasgos lo que llamamos “la memoria”, sin entrar en detalle cual sección de memoria en particular o cómo es manejada por Python). La columna de la izquierda, donde por ahora tenemos el nombre `a`, es llamada “espacio de nombres”, una sección en particular (de la memoria, obviamente, porque de última todo está en memoria, pero que no se nos mezcle con la otra parte genérica) donde guardamos los nombres, que son simplemente cadenas. En el espacio de nombres no podemos almacenar otra cosa que nombres, y estos nombres siempre apuntan a objetos “en la memoria” (no pueden apuntar a otros nombres en el espacio de nombres).

Python tiene un espacio de nombres “global”, que está siempre disponible durante la ejecución del programa y accesible de cualquier lado, y va creando otros espacios de nombres en diferentes momentos y con una accesibilidad limitada. Por ejemplo, cuando se ejecuta una función esta tiene su propio “espacio de nombres local”, diferente al global y diferente a los espacios de nombres de otras funciones, que es lo que permite que cada función pueda usar un mismo nombre apuntando a diferentes objetos y no entren en conflicto.

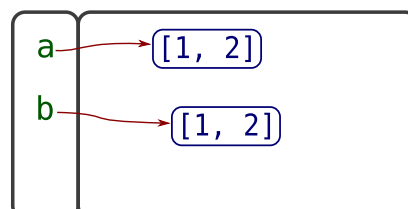
En los diagramas, además de las dos secciones, y el nombre en una y el objeto en otra, vemos una flecha que va del nombre al objeto. Esta flecha representa justamente que ese nombre referencia a dicho objeto. Por eso cuando en Python escribimos `a = "hola"` estamos realmente *vinculando* el nombre `a` con el “objeto cadena con valor `"hola"`” (en inglés el verbo es *bind*).

Retomemos el uso de estos diagramas mostrando los nombres y su vinculación a los objetos, y la diferencia fundamental entre objetos mutables e inmutables, que en conjunto son claves para entender cómo funciona Python internamente (para que eventualmente logremos “pensar como un Pythonista”).

Vayamos entonces con un ejemplo más complejo.

Ahora en los siguientes dos pasos vinculamos primero el nombre `a` a una lista con dos números, y luego `b` a otra lista con los mismos dos números.

CELDA 1 DE TIPOS-PENSANDO.IPYNB
<pre>a = [1, 2] b = [1, 2]</pre>



(El diagrama tiene una simplificación, en pos de la legibilidad: en verdad la lista no tiene a los números “adentro”, sino que los números son otros objetos en la memoria, y desde cada

posición de la lista se los referencia; vamos a ver esto mismo en los próximos diagramas cuando la complejidad lo amerite, pero en este caso solamente para los números no vale la pena.)

Vemos que en memoria tenemos dos objetos lista con el mismo contenido. Podemos adivinar que si le preguntamos a Python si ambos nombres apuntan a objetos iguales, nos dirá que sí, pero si le preguntamos si los dos nombres apuntan *al mismo objeto*, nos dirá que no.

CELDA 02
<code>a == b</code>
True

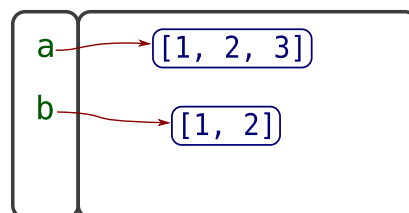
CELDA 03
<code>a is b</code>
False

El primer comparador es el de “igualdad”, mientras que el segundo es de “identidad”. Vamos a ver más comparadores luego cuando veamos el `if` en 3.1.

Modifiquemos ahora una de las listas.

CELDA 04
<code>a.append(3)</code> <code>a</code>
[1, 2, 3]

CELDA 05
<code>b</code>
[1, 2]



Vemos que la lista `a` ahora tiene un elemento más, mientras que la `b` sigue con su estado anterior, como podíamos esperar.

Intentemos algo nuevo, ahora usemos el `=` entre dos nombres. Acá es importante que recordemos que no podemos vincular un nombre a otro nombre (apuntar con la flechita de un nombre a otro nombre), sino que lo que termina sucediendo es que ese nombre nuevo queda vinculado al objeto que apuntaba el otro nombre.

CELDA 06	
<code>c = b</code> <code>c</code>	

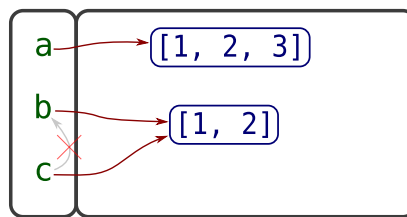
<code>[1, 2]</code>	

CELDA 07	
<code>c == b</code>	

<code>True</code>	

CELDA 08	
<code>c is b</code>	

<code>True</code>	



Es por eso que si ahora comparamos la igualdad y la identidad entre `c` y `b` Python contestará verdadero en ambos casos, porque ambos nombres apuntan al *mismo* objeto.

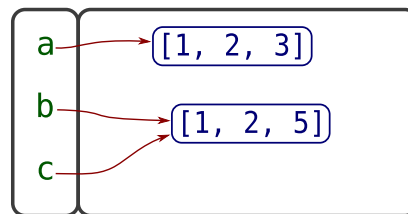
Si ahora modificamos la lista `c`, vemos que también se modifica `b`. Habiendo escrito eso, hagamos el ejercicio de entender que esa frase es inexacta, y casi tramposa. Porque `c` o `b` no son listas, son nombres que apuntan a una lista, la misma, entonces tampoco tiene sentido decir que “también se modifica”. Entonces, escribiendo esa frase correctamente, podríamos decir que “podemos modificar la lista y ver ese cambio, usando tanto un nombre como el otro”, que es lo que vemos en el siguiente paso.

CELDA 09	
<code>c.append(5)</code> <code>c</code>	

<code>[1, 2, 5]</code>	

CELDA 10	
<code>b</code>	

<code>[1, 2, 5]</code>	

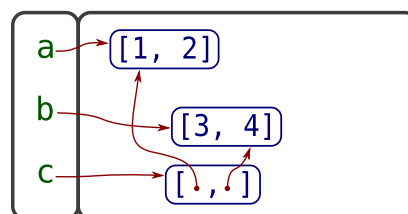


Arranquemos otro ejemplo.

Volvemos a tener dos listas con números, y ahora también armamos una tercera lista que incluye las primeras dos. En verdad las dos primeras listas no están “adentro” de la tercera, sino que son referenciadas en cada posición. Esto es exactamente lo mismo que mencionábamos arriba acerca de que dibujábamos a los números adentro de la lista pero en verdad eran objetos también en la zona de memoria, referenciados desde la lista.

CELDA 11	
<pre> a = [1, 2] b = [3, 4] c = [a, b] c </pre>	
<pre> [[1, 2], [3, 4]] </pre>	

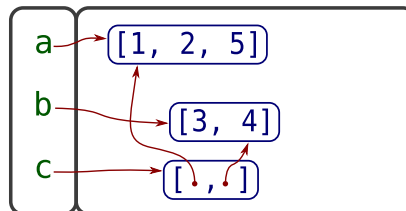
En este caso en el diagrama seguimos con la simplificación de dibujar a los números adentro de la lista, pero somos más precisos con la tercera lista apuntando a las primeras dos.



Si ahora modificamos la lista que llamamos a, obviamente vamos a ver que esa lista está cambiada, pero también vamos a ver reflejado ese cambio si miramos c, ya que estamos hablando en definitiva de la misma lista vista directamente o a través de una posición de la otra lista.

CELDA 12	
<pre> a.append(5) a </pre>	
<pre> [1, 2, 5] </pre>	

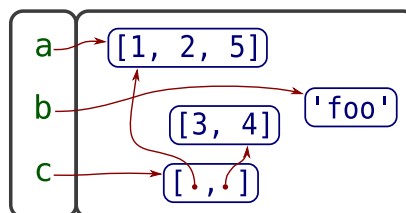
CELDA 13	
c	
[[1, 2, 5], [3, 4]]	



Por otro lado, si hacemos `b = "foo"` tenemos que entender que no estamos modificando el objeto referenciado por `b` sino que realmente estamos revinculando el nombre `b` a otro objeto, y por lo tanto el objeto anterior (el que dejamos de referenciar cuando apuntamos `b` a otro lado) queda intacto.

CELDA 14	
b = "foo"	
b	
'foo'	

CELDA 15	
c	
[[1, 2, 5], [3, 4]]	



A diferencia del ejemplo que teníamos al principio de la sección, donde el objeto “viejo” quedaba sin uso y decíamos que Python eventualmente lo iba a eliminar para liberar memoria, en este caso el objeto que antes era `b` no queda totalmente desreferenciado, sigue siendo apuntado por `c`, con lo cual permanecerá en memoria sin problemas.

E incluso podríamos volver a asignarle otro nombre, y seguir interactuando con el objeto directamente a través del nuevo nombre:

CELDA 16
<pre>d = c[1] d</pre> <pre>[3, 4]</pre>
CELDA 17
<pre>d.append(7) d</pre> <pre>[3, 4, 7]</pre>
CELDA 18
<pre>c</pre> <pre>[[1, 2, 5], [3, 4, 7]]</pre>

Como nota final, cabe destacar que el comparador de identidad **is**, con la excepción de cuando preguntamos **is None** o **is not None** (que queda bien en inglés), se usa en contadísimas ocasiones, en general necesitamos comparar por igualdad con el **==**.

2.6. Conjuntos

Los conjuntos son contenedores de objetos, se delimitan con llaves, y cada objeto (sus elementos internos) se separan con comas.

A diferencia de otros contenedores que vimos previamente, los conjuntos no son secuencias, ya que los elementos dentro del conjunto no tienen un orden en particular. Es hasta inexacto decir que “están desordenados”, porque directamente este tipo de dato no tiene el concepto de orden.

Las propiedades más interesantes de los conjuntos son las matemáticas: cada elemento puede estar solamente una vez, y además podemos realizar intersecciones, uniones y diferencias entre ellos.

CELDA 1 DE TIPOS-SETS.IPYNB
<pre>frutas = {'manzana', 'naranja', 'sandía', 'limón'} frutas</pre> <pre>{'limón', 'manzana', 'naranja', 'sandía'}</pre>
CELDA 02
<pre>vitamina_c = {'limón', 'perejil', 'fresa', 'naranja'}</pre>
CELDA 03
<pre>vitamina_c</pre> <pre>{'fresa', 'limón', 'naranja', 'perejil'}</pre>

CELDA 04
frutas & vitamina_c
{'limón', 'naranja'}

CELDA 05
frutas - vitamina_c
{'manzana', 'sandía'}

CELDA 06
frutas vitamina_c
{'fresa', 'limón', 'manzana', 'naranja', 'perejil', 'sandía'}

El conjunto es el tipo de dato dejado de lado más injustamente de todo el lenguaje, sospechamos que esto es así porque en otros lenguajes no se encuentra, entonces los programadores no se acostumbran a usarlo. Sin embargo, es muy poderoso y una vez que aprendamos a expresar nuestros algoritmos utilizando las capacidades de los conjuntos, evitaremos hacer un montón de bucles y comparaciones innecesarias en nuestros programas.

Sus elementos internos pueden ser solamente objetos inmutables (en verdad, podemos tener objetos que hagamos nosotros que siendo mutables, se les pueda calcular el `hash` en función de propiedades que no cambien, pero por simplicidad pensemos en objetos inmutables). Esto es porque para decidir si un elemento pertenece a un conjunto se usa su hash (de forma similar a los diccionarios que veremos luego). Una implicancia de esta propiedad es que es extremadamente rápido verificar si un elemento pertenece a un determinado conjunto, aunque este sea muy grande (porque se calcula su hash, y listo), en contraposición con lo que sucede con una lista o una tupla (porque en estos casos hay que comparar todos los objetos, algo potencialmente prohibitivo dependiendo del tamaño del contenedor).



En este contexto, un hash es un número entero calculado a partir de los atributos del objeto. Aquellos objetos iguales tendrán el mismo hash. Los conjuntos y diccionarios usan este hash como referencia en su funcionamiento, de ahí la restricción de que los objetos no cambien (para que sigan teniendo siempre el mismo hash).

Los conjuntos son objetos mutables, y como tales tienen métodos para agregar y eliminar elementos internos.

En el siguiente ejemplo vemos como podemos agregar objetos de a uno, o muchos simultáneamente (usando un iterable que los contiene). En ambos casos, tenemos que recordar la propiedad de los conjuntos de que no pueden tener objetos repetidos.

CELDA 07
<pre>música = {'metal', 'tango', 'rock'} música</pre>
<pre>{'metal', 'rock', 'tango'}</pre>
CELDA 08
<pre>música.add('rock') música</pre>
<pre>{'metal', 'rock', 'tango'}</pre>
CELDA 09
<pre>música.add('jazz') música</pre>
<pre>{'jazz', 'metal', 'rock', 'tango'}</pre>
CELDA 10
<pre>música.update(['tango', 'cumbia']) música</pre>
<pre>{'cumbia', 'jazz', 'metal', 'rock', 'tango'}</pre>
CELDA 11
<pre>'rock' in música</pre>
<pre>True</pre>

Tenemos varias formas de remover elementos de los conjuntos, una que saca el elemento indicado pero falla si no está, otra que saca el elemento indicado (sin fallar si no está), y otra que saca un elemento de forma arbitraria (fallando si el conjunto estaba ya vacío):

CELDA 12
<pre>música.remove('jazz') música</pre>
<pre>{'cumbia', 'metal', 'rock', 'tango'}</pre>
CELDA 13
<pre>música.remove('jazz')</pre>
<pre>KeyError Traceback (most recent call last) <ipython-input-13-533e3682df9b> in <module> ----> 1 música.remove('jazz')</pre>
<pre>KeyError: 'jazz'</pre>

CELDA 14
<pre>música.discard('metal') música</pre> <pre>{'cumbia', 'rock', 'tango'}</pre>

CELDA 15
<pre>música.discard('metal') música</pre> <pre>{'cumbia', 'rock', 'tango'}</pre>

CELDA 16
<pre>música.pop()</pre> <pre>'tango'</pre>

CELDA 17
<pre>música</pre> <pre>{'cumbia', 'rock'}</pre>

Como con el resto de los tipos, podemos usar el constructor para convertir entre ellos; en este caso el constructor toma cualquier iterable y se queda con los elementos que recibe:

CELDA 18
<pre>set(['foo', 'bar'])</pre> <pre>{'bar', 'foo'}</pre>

CELDA 19
<pre>set("son ocho los orozco")</pre> <pre>{' ', 'c', 'h', 'l', 'n', 'o', 'r', 's', 'z'}</pre>

2.7. Diccionarios

Los diccionarios también se delimitan con llaves, como los conjuntos, pero en este caso lo que se separan por comas son pares de objetos, cada par siendo una clave y un valor (separados entre sí por dos puntos).

Es que los diccionarios, a diferencia de los otros contenedores que vimos hasta ahora (listas, conjuntos, etc.) guardan objetos identificados cada uno por una determinada clave. Los valores guardados pueden ser cualquier objeto de Python, mientras que las claves pueden ser objetos que se les puede calcular el hash.

La forma más directa de acceder a los valores guardados es a través de sus respectivas claves, pero también podemos listar todas las claves, todos los valores, e incluso todos los ítems (pares

clave/valor).

CELDA 1 DE TIPOS-DICTS.IPYNB
<pre>d = {'foo': 23, 'bar': 90, 'baz': 133} d</pre>
<pre>{'foo': 23, 'bar': 90, 'baz': 133}</pre>

CELDA 02
<pre>d['foo']</pre>
<pre>23</pre>

CELDA 03
<pre>d.keys()</pre>
<pre>dict_keys(['foo', 'bar', 'baz'])</pre>

CELDA 04
<pre>d.values()</pre>
<pre>dict_values([23, 90, 133])</pre>

CELDA 05
<pre>d.items()</pre>
<pre>dict_items([('foo', 23), ('bar', 90), ('baz', 133)])</pre>

Cada clave puede estar una sola vez en el diccionario (porque se accede a través del hash de las claves, de forma similar a lo que veíamos con los conjuntos), entonces si asignamos un nuevo valor a una clave ya presente, estaremos pisando el valor anterior. Si la clave no estaba, estaremos creando un nuevo ítem en el diccionario.

CELDA 06
<pre>d</pre>
<pre>{'foo': 23, 'bar': 90, 'baz': 133}</pre>

CELDA 07
<pre>d['foo'] = 1288 d</pre>
<pre>{'foo': 1288, 'bar': 90, 'baz': 133}</pre>

CELDA 08
<pre>d['otra'] = 7 d</pre>

```
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

Tengamos en cuenta que desde Python 3.6 los diccionarios recuerdan el orden de inserción de sus claves (antes no tenían un orden en particular).

Si accedemos a una clave que no está presente en el diccionario, se genera una excepción de tipo **KeyError**. A veces es útil usar el método `get` que accede al diccionario y devuelve el valor correspondiente a la clave si es que la clave existe, pero en caso de que la clave no exista devolverá un valor que podemos especificar (o **None** si no indicamos nada).

CELDA 09

```
d
```

```
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

CELDA 10

```
d['foo']
```

```
1288
```

CELDA 11

```
d['fux']
```

```

KeyError                                Traceback (most recent call last)
<ipython-input-11-15405d883549> in <module>
----> 1 d['fux']

KeyError: 'fux'

```

CELDA 12

```
d.get('foo')
```

```
1288
```

CELDA 13

```
d.get('fux', "nada")
```

```
'nada'
```

CELDA 14

```
d.get('fux')
```

Podemos borrar elementos del diccionario con la declaración **del**, pero también a veces es útil extraer el valor correspondiente a la clave que estamos borrando, y para eso tenemos el método `pop`, al que si le pasamos un valor por default soportará no encontrar a la clave indicada.

CELDA 15
d
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}

CELDA 16
<code>del d['bar']</code>
d
{'foo': 1288, 'baz': 133, 'otra': 7}

CELDA 17
<code>d.pop('foo')</code>
1288

CELDA 18
d
{'baz': 133, 'otra': 7}

CELDA 19
<code>d.pop('foo')</code>
<pre> KeyError Traceback (most recent call last) <ipython-input-19-9d43697015c6> in <module> ----> 1 d.pop('foo') KeyError: 'foo' </pre>

CELDA 20
<code>d.pop('foo', "no estaba")</code>
'no estaba'

Podemos crear diccionarios a partir de otras estructuras, cualquier iterable en verdad, pero tendrán que ser de pares de valores, donde cada par será clave/valor. También podemos crear diccionarios tomando como fuente las claves, usando el método `fromkeys`, pero en este caso las claves tendrán siempre el mismo valor inicial. Y como detalle especial con este tipo de datos podemos usar una forma de pasar parámetros a las funciones que en este caso es particularmente útil (aunque sólo para el caso donde las claves son cadenas).

CELDA 21
<code>dict([('a', [1, 2]), ('b', [3, 4])])</code>
{'a': [1, 2], 'b': [3, 4]}

CELDA 22
<code>dict.fromkeys(['foo', 'bar'])</code>
<code>{'foo': None, 'bar': None}</code>
CELDA 23
<code>dict.fromkeys(['foo', 'bar'], 0)</code>
<code>{'foo': 0, 'bar': 0}</code>
CELDA 24
<code>dict(a=[1, 2], b=[3, 4])</code>
<code>{'a': [1, 2], 'b': [3, 4]}</code>

2.8. Iteradores

Antes de irnos de la Sección de Tipos de datos, mencionemos un concepto importante en Python: los iteradores.

A nivel de definición general, un iterador se refiere al objeto que permite al programador recorrer un contenedor. Prestemos atención a la diferencia entre el contenedor en sí (como colección de elementos) del iterador (que nos permite recorrer esos elementos).

Esta característica se expresa en muchos rincones del lenguaje. Tenemos la declaración **for**, que veremos más adelante en la Sección 3.3, que nos permite construir un bucle alrededor de iterar un objeto, pero también podemos hacerlo a mano, aunque de esta manera tenemos que pedirle al objeto iterable que nos de un iterador:

CELDA 1 DE TIPOS-ITERS.IPYNB
<code>números = [1, 2, 3]</code> <code>type(números)</code>
<code>list</code>
CELDA 02
<code>iterador = iter(números)</code> <code>type(iterador)</code>
<code>list_iterator</code>
CELDA 03
<code>next(iterador)</code>
<code>1</code>

CELDA 04
<pre>next(iterador)</pre>
2

CELDA 05
<pre>next(iterador)</pre>
3

CELDA 06
<pre>next(iterador)</pre>
StopIteration
Traceback (most recent call last)
<ipython-input-6-2389250a88e0> in <module>
----> 1 next(iterador)
StopIteration:

Esa excepción que vemos ahí es perfectamente normal, es el mecanismo que tienen los iteradores para indicar que no hay más elementos para entregar.

Una forma útil en el intérprete interactivo de iterar un objeto y ver el resultado es a través del constructor `list`. Usémoslo y veamos como en Python muchos tipos integrados en el lenguaje soportan el protocolo de iteración (decimos que son “iterables”):

CELDA 07
<pre>list([1, 2, 3]) # una lista</pre>
[1, 2, 3]

CELDA 08
<pre>list((1, 2, 3)) # una tupla</pre>
[1, 2, 3]

CELDA 09
<pre>list("abcd") # una cadena</pre>
['a', 'b', 'c', 'd']

CELDA 10
<pre>list({1, 2, 3}) # un conjunto</pre>
[1, 2, 3]

CELDA 11
<pre>list({'a': 1, 'b': 2, 'c': 3}) # un diccionario</pre>
<pre>['a', 'b', 'c']</pre>

No todos los tipos soportan este protocolo, claro. Por ejemplo, no se puede iterar un número entero, ya que no es un contenedor, no tiene elementos para entregar.

También tengamos en cuenta que normalmente cada vez que pidamos un iterador vamos a tener uno “fresco”, que arrancará desde el principio (aunque nosotros podríamos cambiar este comportamiento si hacemos nuestros propios tipos de datos). Pero si trabajamos sobre el iterador puntualmente, podemos pedirle elementos de diversas maneras sin tener que volver a comenzar.

CELDA 12
<pre>lista = [1, 2, 3, 4, 5, 6, 7] list(lista)</pre>
<pre>[1, 2, 3, 4, 5, 6, 7]</pre>

CELDA 13
<pre>iterador = iter(lista) iterador</pre>
<pre><list_iterator at 0x7f4a0cd2b700></pre>

CELDA 14
<pre>next(iterador)</pre>
<pre>1</pre>

CELDA 15
<pre>next(iterador)</pre>
<pre>2</pre>

CELDA 16
<pre>list(iterador)</pre>
<pre>[3, 4, 5, 6, 7]</pre>

Finalmente, prestemos atención cuando estamos iterando un objeto, ya que el mismo puede potencialmente entregar una cantidad muy grande de elementos, lo cual no tiene sentido tener en memoria simultáneamente. Mostremos un ejemplo de esto con la función integrada `range`, que entrega números en un determinado rango, ahí vemos como para un rango chico podemos iterarlo completamente con una lista, pero para uno muy grande, aunque podamos iterarlo a mano, realmente no tiene sentido hacerlo hasta el final.

CELDA 17
<pre>nros = range(3) list(nros)</pre>
<pre>[0, 1, 2]</pre>

CELDA 18
<pre>muchos_nros = range(1000000000000000) iter_muchos_nros = iter(muchos_nros) next(iter_muchos_nros)</pre>
<pre>0</pre>

CELDA 19
<pre>next(iter_muchos_nros)</pre>
<pre>1</pre>

Así y todo `range` en algún momento va a terminar. Pero podemos tener generadores que sean infinitos, ya veremos como construirlos cuando hablemos de Funciones [4.1](#).



3 | Controles de flujo

En este capítulo mostraremos las distintas declaraciones de Python que nos permiten controlar el flujo de ejecución de un programa, que hasta ahora veíamos como lineal.

[Código disponible](#)

Python tiene pocas estructuras en este sentido (no repitiendo innecesariamente funcionalidad, “simple es mejor que complejo”), pero las pocas que tiene son poderosas e interesantes.

En este capítulo también incluiremos *excepciones*, que aunque también se utilizan para manejar errores, en realidad es un concepto más amplio (se utilizan para situaciones excepcionales, no sólo errores), y la naturaleza de las mismas hacen que el flujo del programa se modifique.

3.1. If, elif, else

Comenzamos con el control de flujo más sencillo, el **if**. La estructura básica es muy simple:

```
if <expresión>:  
    <bloque de código>
```

Si la expresión es verdadera, se ejecuta el bloque de código; si no, no.

¿Qué es una “expresión” en este contexto? Una expresión es una combinación de valores, variables, operadores y llamadas a funciones. Cuando sea evaluada, para el **if** va a terminar siendo falsa o verdadera, no hay otra.

Más abajo seguiremos hablando de las expresiones, pero ahora centrémonos en el bloque de código que es lo que se ejecuta si la expresión es verdadera. No es más que una secuencia de líneas de código, que puede ser una o más (no cero, y sin límites prácticos en su cantidad). El comienzo y finalización del bloque está marcado por su sangría (que muchas veces llamamos “indentación”, término que aunque es usado muchísimo es solamente un anglicismo de *indentation*).

Esa sangría puede ser cualquier cantidad de espacios o tabuladores (¡pero no mezclados!, y se recomienda usar cuatro espacios), con la condición de que sea consistente. Por ejemplo, si un bloque arranca con la línea cuatro espacios a la derecha, siempre estará sangrado igual durante todo el bloque (a menos que haya bloques anidados, claro) y luego terminará volviendo a la columna original.

Ejemplo:

```
1  if foo == 42:  
2      print("foo vale 42") # arranca el bloque con 4 a la derecha  
3      print("segunda línea") # segunda línea ok  
4      print("tercera") # esta está mal, porque está demasiado a la derecha  
5      if bar == 33:  
6          print("bar es 33") # esta está bien, es un bloque nuevo adentro del otro  
7          print("muchas líneas") # ok también: cierra el bloque interior, y vuelve a la columna original  
8      print("vamos cerrando") # mal! si cerramos el bloque debemos volver a la columna original de este  
9      print("afuera") # esta está bien, ya afuera del bloque del if
```

A la estructura del **if** le podemos agregar una especie de continuación:

```
if <expresión 1>:  
    <bloque de código 1>  
elif <expresión 2>:  
    <bloque de código 2>
```

Si la expresión 1 es verdadera, se ejecutará el bloque de código 1 y se ignorará el resto. Pero si la expresión 1 es falsa, se evalúa la expresión 2: si es verdadera se ejecutará el bloque 2, sino termina.

Y finalmente podemos agregarle una especie de salida final:

```
if <expresión 1>:  
    <bloque de código 1>  
elif <expresión 2>:  
    <bloque de código 2>  
else:  
    <bloque de código 3>
```

El bloque de código 3 se ejecutará si todas las expresiones de la estructura fueron evaluadas a falso.

El **if** es obviamente obligatorio (arranca la estructura), pero el **elif** es opcional (y se pueden poner cuantos queramos, uno abajo del otro cada uno con su expresión a evaluar), y el **else** es también opcional pero no puede haber más de uno.

Veamos un ejemplo más real:

CELDA 1 DE FLUJO-IF.IPYNB
<pre>from datetime import date hoy = date.today()</pre>
CELDA 02
<pre>if hoy >= date(2030, 1, 1): print("El futuro ya llegó") elif date(2020, 1, 1) <= hoy < date(2030, 1, 1): print('La famosa "década del 20"') else: print("El lejano pasado")</pre> La famosa "década del 20"

Claro que esa estructura se puede extender mucho más. En la actualidad Python no tiene una declaración case como muchos otros lenguajes. Esto se resuelve con estructuras **if/elif** si son relativamente pocos casos, guardando funciones en un diccionario si son más, o utilizando coincidencia de patrones tal como vemos en la Sección 5.6.

Volvamos sobre algo que prudentemente esquivamos al principio de la sección: las expresiones.

Decíamos que una expresión es una combinación de valores, variables, operadores y llamadas a funciones. Esto es bastante genérico, y hay pocas cosas que no pueden ser incluidas en una expresión, como definiciones de funciones o clases, importar módulos, etc. En la práctica podemos hacer casi todo lo que deseamos, y esto nos permite ser bastante expresivos, por ejemplo como veíamos arriba al comparar un valor con el resultado de la función `date` que estamos llamando ahí mismo.

A esto hay que sumarle que todos los tipos de datos integrados en el lenguaje también son evaluables a verdadero o falso, lo cual podemos revisar sencillamente con la función `bool` (como regla general, el objeto evalúa a `False` si vale cero o está vacío, sino a `True`).

CELDA 03
<pre>bool(-1)</pre> True
CELDA 04
<pre>bool([])</pre> False

	CELDA 05
<code>bool(None)</code>	
False	

	CELDA 06
<code>bool("")</code>	
False	

	CELDA 07
<code>bool("falso")</code>	
True	

Y además tenemos los operadores de comparación:

- `==` igual a
- `!=` diferente de
- `is` es el mismo objeto que (identidad)
- `<` menor que
- `<=` menor o igual que
- `>` mayor que
- `>=` mayor o igual que

Todo esto nos permite ser muy expresivos al armar las estructuras `if`.

3.2. While

El `while` es una estructura de control de flujo que nos arma un bucle alrededor de ese bloque de código, repitiendo ese bloque en función de la evaluación de una expresión:

```
while <expresión>:  
    <bloque de código>
```

Al arrancar el bucle, Python evalúa la expresión, si es falsa sale y nunca ejecuta el bloque de código. Si es verdadera ejecuta ese bloque de código, y vuelve a evaluar la expresión, si es falsa sale, sino ejecuta el bloque, y así hasta que la expresión de falso o se interrumpa por algo (pero potencialmente durante mucho mucho tiempo, hasta que el Sol se apague, digamos).

Veamos un ejemplo:

CELDA 1 DE FLUJO-WHILE.IPYNB
<pre>a = 0 while a < 3: a += 1 print(a)</pre>

1 2 3

Ya hablamos al explicar el **if** 3.1 tanto de la expresión como del bloque de código, no hay mucho para agregar en ese aspecto. Por otro lado, con el **while** vemos que aparecen tres nuevas declaraciones, veámoslas.

El **break** nos permite interrumpir el bucle en la mitad de su ejecución. Si el código toca un **break**, entonces, el bucle se corta y sigue con lo que venía a continuación del mismo, sin terminar el bucle y sin volver a evaluar la expresión.

CELDA 02
<pre>a = 0 while a < 5: a += 1 if a == 3: break print(a)</pre>

1 2

El **continue** nos permite abortar la pasada actual del bucle, volviendo al principio del mismo, lo que incluye volver a evaluar la expresión. Vemos a continuación como el “3” no se imprime, porque al tocar el **continue** vuelve a recomenzar el bucle, sin llegar al **print** en ese caso.

CELDA 03
<pre>a = 0 while a < 5: a += 1 if a == 3: continue print(a)</pre>

1 2 4 5

Si usamos el **else** vamos a poder decidir si el bucle **while** terminó porque su expresión evaluó a falso o fue cortado con un **break**. Lo podemos pensar como en el **if**: si la expresión evalúa a verdadero se ejecuta el bucle, si evalúa a falso se ejecuta el bloque del **else**.

La combinación **while** con el **else** no es muy utilizado, en parte porque no lo vemos en otros lenguajes, pero es muy útil cuando justamente queremos saber si salimos del **while** en un caso o en el otro (lo que se resuelve en otros lenguajes utilizando otra variable como bandera).

Veamos ambos comportamientos en los siguientes ejemplos:

CELDA 04
<pre>a = 0 while a < 2: a += 1 if a == 3: break # realmente no va a llegar acá print(a) else: print("en el else") print("afuera")</pre>
<pre>1 2 en el else afuera</pre>

CELDA 05
<pre>a = 0 while a < 4: # ahora sí vamos a tocar el break a += 1 if a == 3: break print(a) else: print("en el else") print("afuera")</pre>
<pre>1 2 afuera</pre>

3.3. For

El **for** es una declaración que nos permite recorrer iterables, ejecutando un bloque de código por cada uno de esos iterables (y haciendo referencia al elemento obtenido del iterable en cada momento usando un nombre que nosotros especificamos).

Si entendemos esa definición, la estructura es casi autodescriptiva:

```
for <nombre> in <iterable>:
    <bloque de código>
```

Veamos un ejemplo sencillo:

CELDA 1 DE FLUJO-FOR.IPYNB
<pre>nros = [1, 2, 3] for n in nros: print(n)</pre>

```
1
2
3
```

En realidad en lugar del “nombre” podemos tener varios nombres, si es que los elementos del iterable que recorremos pueden desempacarse correctamente, y obtendremos el mismo efecto que en la asignación múltiple (más específicamente, tenemos toda la experiencia del desempaquetado de tuplas que explicamos antes [2.4](#)):

CELDA 02

```
cosas = [('a', 1), ('b', 2), ('c', 3)]
for letra, nro in cosas:
    print(letra, nro ** 2)
```

```
a 1
b 4
c 9
```

Vale la pena que aclaremos que el **for** es más parecido al **foreach** de otros lenguajes, y diferente al **for** de C o al **do** de Fortran, por ejemplo, que sólo cuentan números entre un principio y final (en general para indizar una estructura y obtener los elementos internos de la misma). En verdad la necesidad de trabajar con rangos de números es real, y para eso Python tiene una función integrada **range**, en el que podemos especificar el final del rango (arrancando por default en cero), o inicio y final, e incluso el paso:

CELDA 03

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

CELDA 04

```
list(range(3, 8))
```

```
[3, 4, 5, 6, 7]
```

CELDA 05

```
list(range(3, 8, 2))
```

```
[3, 5, 7]
```

CELDA 06

```
list(range(3, 12, 2))
```

```
[3, 5, 7, 9, 11]
```

Prestemos atención que el “desde” es inclusivo, mientras que el “hasta” es exclusivo; esto aunque quizás sea sorprendente tiene sentido en la foto más grande del funcionamiento general de Python. Por ejemplo, si queremos los índices para una lista de 4 elementos, haremos **range(4)**

y eso nos dará el 0, 1, 2 y 3 que son las posiciones de una lista de 4 elementos. También es muy práctica la propiedad de que si hacemos `range(M, N)` la cantidad de números que obtendremos es $N - M$.

Para el ejemplo utilizamos el `list` porque el `range` es un generador de números, entonces tenemos que consumirlo para ver esos números. Por supuesto que lo podemos iterar directamente con el `for`:

CELDA 07
<pre>for n in range(2, 5): print("El cuadrado de {} es {}".format(n, n ** 2))</pre>
El cuadrado de 2 es 4 El cuadrado de 3 es 9 El cuadrado de 4 es 16

El `for` es un bucle, y como con el otro bucle de Python (el `while`, de la Sección 3.2) podemos afectar el comportamiento utilizando el `break` (para interrumpir el bucle y salir), el `continue` (para abortar la pasada del bucle y volver al principio, obteniendo un nuevo elemento del iterable), e incluso el `else` (para discernir si el `for` terminó porque se nos acabó el iterable o porque cortamos con un `break`).

Veamos algunos ejemplos usando estas funcionalidades. Arrancamos con el `break` y el `continue`, viendo como corta en un caso y como vuelve al principio en el otro esquivando el resto del bucle:

CELDA 08
<pre>números = [1, 2, 3, 4] for n in números: if n == 3: break print(n) print("afuera")</pre>
1 2 afuera

CELDA 09
<pre>for n in números: if n == 3: continue print(n) print("afuera")</pre>
1 2 4 afuera

Y veamos el `else`, para el caso en que terminamos el `for` por consumir totalmente el iterable o porque encontramos un `break`:

CELDA 10

```
for n in números:  
    print(n)  
else:  
    print("se nos terminó el iterable")  
    print("afuera")
```

```
1  
2  
3  
4  
se nos terminó el iterable  
afuera
```

CELDA 11

```
for n in números:  
    if n == 2:  
        break  
    print(n)  
else:  
    print("se nos terminó el iterable")  
    print("afuera")
```

```
1  
afuera
```

Hay un caso de uso típico en los **for** que es arrancar con una lista, realizarle una operación, y terminar con otra lista con los resultados de esa operación. Por ejemplo, podemos tener una lista de números y queremos calcular sus cuadrados:

CELDA 12

```
números = [1, 3, -2, 2, 0, 5]  
cuadrados = []  
for n in números:  
    cuadrados.append(n ** 2)
```

```
cuadrados
```

```
[1, 9, 4, 4, 0, 25]
```

Esta construcción es tan usual que Python tiene una sintaxis especial que nos permite escribir lo mismo pero de forma más reducida, con la ventaja que hasta queda más legible; se denomina “comprensión de listas” (en inglés *list comprehension*), y se define usando corchetes para delimitar la estructura, con los elementos sintácticos del **for** adentro:

CELDA 13

```
números = [1, 3, -2, 2, 0, 5]  
cuadrados = [n ** 2 for n in números]  
cuadrados
```

```
[1, 9, 4, 4, 0, 25]
```

Podemos leer la segunda línea como “armamos una lista con ene al cuadrado para cada ene

en números”, y hace exactamente eso. Es mucho más fácil de entender que el **for** del ejemplo anterior que hace lo mismo pero a lo largo de varias líneas (entonces lo tenemos que seguir, ir y volver con la vista, entender qué hace con la lista que definimos al principio, etc.); en el caso de la “comprensión de listas” al primer vistazo (cuando reconocemos la estructura) ya sabemos que generamos una nueva lista realizando una operación con los elementos de un iterable, y nada más.

En realidad podemos complejizar apenas esa estructura, para el caso en que queramos filtrar algunos elementos del iterable fuente. Veamos los mismos ejemplos que recién pero calculando logaritmos solamente para los valores mayores a cero.

CELDA 14
<pre>import math números = [1, 3, -2, 2, 0, 5] logs1 = [] for n in números: if n > 0: logs1.append(math.log(n)) logs1</pre>
[0.0, 1.0986122886681098, 0.6931471805599453, 1.6094379124341003]

CELDA 15
<pre>logs2 = [math.log(n) for n in números if n > 0] logs2</pre>
[0.0, 1.0986122886681098, 0.6931471805599453, 1.6094379124341003]

Una generalización de esta estructura nos permite armar una “comprensión de conjuntos” (al delimitar la estructura con llaves) e incluso una “comprensión de diccionarios” (al delimitar la estructura con llaves y tener clave y valor separados por dos puntos):

CELDA 16
<pre>números = [1, 3, -2, 2, 0, 5] [n ** 2 for n in números]</pre>
[1, 9, 4, 4, 0, 25]

CELDA 17
<pre>{n ** 2 for n in números}</pre>
{0, 1, 4, 9, 25}

CELDA 18
<pre>{n: n ** 2 for n in números}</pre>
{1: 1, 3: 9, -2: 4, 2: 4, 0: 0, 5: 25}

3.4. Excepciones

El sistema de manejo de errores de Python es a través de excepciones.

3.4.1. ¿Qué son las excepciones?

Una excepción es un evento que ocurre durante la ejecución normal del programa e interrumpe el flujo normal del mismo. En general, cuando un programa en Python encuentra una situación que no puede manejar, levanta una excepción.

Entonces, a diferencia de otros lenguajes que luego de llamar a una función (por ejemplo) tenemos que revisar el resultado para ver si indica que hubo un problema, en Python el resultado será lo que tenga que devolver la función normalmente, si es que la función terminó sin inconvenientes. Pero si hubo un problema, obtendremos una excepción, que podemos manejar o dejar continuar.

Hagamos un pequeño programa para ver eso:

```
1 print("antes")
2 1 / 0
3 print("después")
```

Al ejecutar ese código, obtendremos:

```
$ python3 excep.py
antes
Traceback (most recent call last):
  File "x.py", line 2, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

Vemos que se imprime el “antes”, pero luego la ejecución se interrumpió al encontrar un error (al intentar dividir por cero). En ese punto, se levantó una excepción, y como no se capturó la terminó agarrando el intérprete de Python, el que interrumpió al programa y mostró un *traceback* por pantalla. Un traceback (término que usamos en inglés, ya que nunca se popularizó decirles “trazas de rastreo”) es información que nos da el intérprete para entender de dónde viene el problema.

Tiene tres partes, un título que nos indica el comienzo del traceback, un contenido cuyo largo dependerá de cuan profundo en la pila de llamadas a funciones haya sucedido el problema, y una última línea mostrando el tipo de excepción y un mensaje (en nuestro caso **ZeroDivisionError**, y el mensaje indicando eso). Veamos estas partes en la salida que mostramos recién:

```
$ python3 excep.py
antes
Traceback (most recent call last):
  File "x.py", line 2, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

Diagrama de anotaciones para el traceback:

- ejecución normal**: Indica la parte del código que se ejecutó correctamente antes de la excepción.
- comienzo del traceback**: Señala la línea que inicia el mensaje de error.
- el contenido, uno o más pares de líneas**: Señala la información de la pila de llamadas (archivo y línea).
- tipo y mensaje de error**: Señala la excepción y su descripción.

Para entender mejor la parte del medio veamos un ejemplo apenas más complejo (usando funciones, aunque nos adelantemos un poco a cuando las expliquemos formalmente en la Sección 4.1).

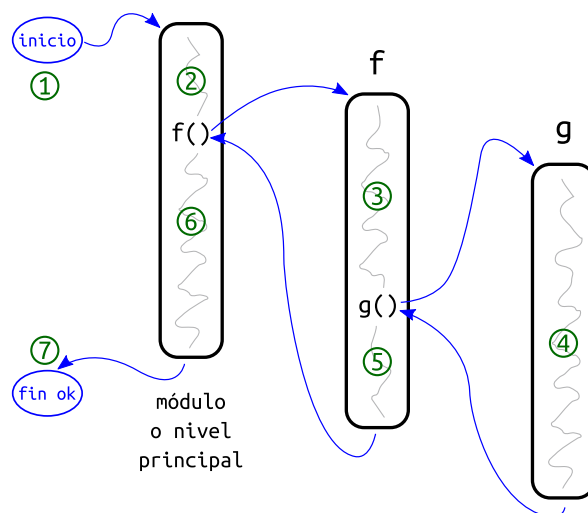
Veamos primero que pasaría si todo sale bien y no nos encontramos con ningún problema:

```
1 print("1. inicio")
2
3 def g():
4     print("4. en g")
5
6 def f():
7     print("3. antes en f")
8     g()
9     print("5. después en f")
10
11 print("2. antes en el módulo")
12 f()
13 print("6. después en el módulo")
14
15 print("7. fin")
```

Al ejecutar ese código, obtendremos:

```
$ python3 excep-stacks-ok.py
1. inicio
2. antes en el módulo
3. antes en f
4. en g
5. después en f
6. después en el módulo
7. fin
```

Los números se corresponden al siguiente diagrama donde vemos la secuencia de ejecución de las distintas funciones:



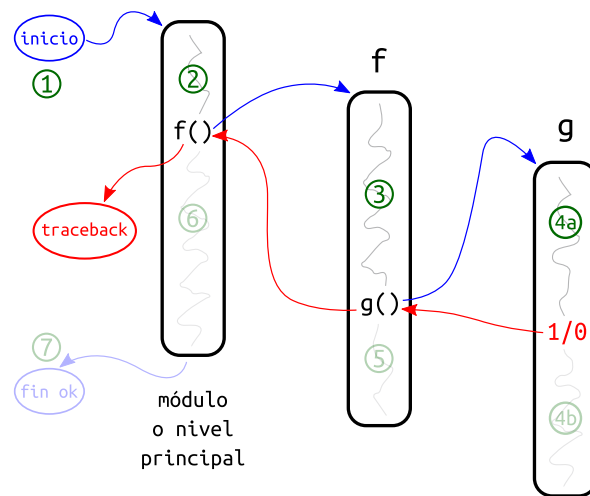
Ahora veamos qué pasaría si en la función “g” nos encontramos con un problema. Modifiquemos esa función:

```
1 print("1. inicio")
2
3 def g():
4     print("4a. antes en g")
5     print("¿uno sobre cero?", 1 / 0)
6     print("4b. después en g")
7
8 def f():
9     print("3. antes en f")
10    g()
11    print("5. después en f")
12
13 print("2. antes en el módulo")
14 f()
15 print("6. después en el módulo")
16
17 print("7. fin")
```

Vemos que tenemos los resultados de los primeros prints y luego un traceback, pero no lo que veíamos antes:

```
$ python3 excep-stacks-pumba.py
1. inicio
2. antes en el módulo
3. antes en f
4a. antes en g
Traceback (most recent call last):
  File "x.py", line 14, in <module>
    f()
  File "x.py", line 10, in f
    g()
  File "x.py", line 5, in g
    print("¿uno sobre cero?", 1 / 0)
ZeroDivisionError: division by zero
```

Podemos ver mejor el flujo de ejecución en este otro diagrama que representa lo que acabamos de experimentar:



Allí vemos que el flujo de ejecución se modifica. Cuando llegamos al punto del problema, el resto de la función “g” no se ejecuta, sino que se genera una excepción que vuela hasta el punto donde la función fue llamada (línea roja que llega a “f”). El resto de esta función tampoco se ejecuta, como la excepción no fue capturada seguirá subiendo por las funciones hasta llegar a nivel de módulo, donde Python interrumpe el proceso mostrando el traceback.

La mejor forma que tenemos para entender lo que está pasando cuando hay un error es leer cuidadosamente el traceback. Recomendamos hacerlo de abajo para arriba.

Para el ejemplo anterior, vemos que tuvimos una excepción por división por cero en la línea 5 en la función “g” (donde hicimos `1 / 0`), lo cual viene de la línea 10 en la función “f” (donde hicimos `g()`), que a su vez viene de la línea 14 a nivel del módulo (donde hicimos `f()`).

Como tenemos un stack de largo tres (el cuerpo principal del programa más las dos funciones) tenemos tres pares de líneas en el centro del traceback.

Entonces, cada par de líneas de “la parte central” del traceback corresponde a un nivel del stack, y en cada caso nos muestra dónde se sucedió el problema para ese nivel del stack (en qué archivo, en qué línea y el contexto), y luego la línea en cuestión (que podríamos ver en el archivo/posición indicado, pero así es más cómodo).

3.4.2. Capturando y levantando excepciones

Podemos capturar las excepciones que se sucedan con un bloque **try**. La construcción típica es usarlo en conjunto al **except**:

```
try:
    <bloque de código>
except:
    <bloque de código>
```

El primer bloque de código, correspondiente al **try** es el supervisado, si se sucede alguna excepción en ese bloque, el **except** entra en juego y se ejecutará el segundo bloque de código; si ninguna excepción se levanta en el bloque supervisado, el **except** será ignorado completamente.

Veamos ambas situaciones en un ejemplo mínimo:

CELDA 1 DE FLUJO-EXCEP.IPYNB
<pre>try: print("uno sobre dos:", 1 / 2) except: print("hubo un problema")</pre>
CELDA 02
<pre>try: print("uno sobre cero:", 1 / 0) except: print("hubo un problema")</pre>
hubo un problema

Hay dos reglas de oro para seguir cuando escribimos estas estructuras. La primera es que debemos supervisar el mínimo de código posible (minimizar la cantidad de código dentro del bloque del **try**), la segunda es que debemos capturar solamente las excepciones que estamos esperando que puedan suceder (especificar el **except** lo más posible, no como hasta ahora que está capturando todo).

La primer regla es fácil de entender, pero veamos un ejemplo de cómo es útil especificar el **except** lo más posible. Supongamos el siguiente código, donde obtenemos un valor (supongamos de una medición) y calculamos uno sobre eso; excepcionalmente podemos tener un cero como valor, pero eso sería que el instrumento está descalibrado, entonces lo informamos y listo:

CELDA 03
<pre>valor = 3 # todo bien try: print("uno sobre algo:", 1 / valor) except: print("instrumento descalibrado")</pre>
uno sobre algo: 0.3333333333333333
CELDA 04
<pre>valor = 0 # efectivamente descalibrado try: print("uno sobre algo:", 1 / valor) except: print("instrumento descalibrado")</pre>
instrumento descalibrado

Sin embargo, supongamos que tenemos un problema más serio, y por un error en nuestro programa terminamos teniendo otra cosa como valor, una cadena:

CELDA 05

```

valor = "error" # otro problema desconocido
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
-----
instrumento descalibrado

```

El mensaje que estamos dando es totalmente equivocado. Tengamos en cuenta que muchas veces también en el bloque del **except** se toman acciones para corregir o paliar el problema, y si tomamos las acciones equivocadas podemos estar complicando aún más la situación.

Entonces, tenemos que ser lo más específicos posibles al capturar la excepción. Para el ejemplo que estamos viendo, nosotros sabemos que podemos llegar a tener una **ZeroDivisionError**, ¡entonces capturemos sólo eso! Si le especificamos un tipo de excepción al **except** (o más de uno, entre paréntesis), capturará la excepción y ejecutará el bloque de código sólo si la excepción es de ese tipo (o de algunos de los varios tipos que pusimos entre paréntesis):

CELDA 06

```

valor = 0
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
-----
instrumento descalibrado

```

CELDA 07

```

valor = "error"
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-7-3bd734f41923> in <module>
      1 valor = "error"
      2 try:
----> 3     print("uno sobre algo:", 1 / valor)
      4 except ZeroDivisionError:
      5     print("instrumento descalibrado")

TypeError: unsupported operand type(s) for /: 'int' and 'str'

```

Entonces, en el primer caso tenemos el mensaje que esperábamos, mientras que en el segundo caso la excepción no es capturada, la termina agarrando Python y nos genera el traceback correspondiente, lo cual está perfectamente bien, porque es información útil para encontrar el error en nuestro programa.

Hay situaciones, sin embargo, en que igualmente queremos capturar todo lo que pueda llegar a suceder, lo cual se justifica en procesos que son muy largos y que no queremos que se interrumpan nunca, o siempre tienen que estar levantados, y necesitamos capturar todas las excepciones

para poder informarlas y seguir trabajando.

Pero incluso en estas situaciones no es recomendable poner el **except** “pelado”, ya que hay excepciones que son internas al funcionamiento de Python y no debemos capturarlas. En estos casos es muy útil que las excepciones en Python estén dispuestas en forma de árbol, ya que especificando un tipo de excepción realmente estamos capturando las excepciones de ese tipo y todas las de su rama (este árbol lo podemos ver [en la documentación](#)).

Entonces, podemos capturar **Exception** (que como vemos en el árbol de excepciones incluye a casi todas menos tres muy específicas), y obtendremos el efecto deseado. Es más, al especificar el tipo de excepción, podemos incluso ponerle un nombre a la excepción que capturamos y manejarla en el bloque de código.

Veamos todo esto en el ejemplo que traíamos, aprovechando la flexibilidad del **except** que nos permite especificar varios luego de un **try** (el comportamiento en estos casos es parecido a la cadena/secuencia que teníamos con los **if/elif**: se va verificando la excepción en orden en todos los **except**, si la excepción es capturada por uno de ellos se ejecuta su bloque de código y deja de verificarse en el resto.

CELDA 08

CELDA 09

En verdad, son más las herramientas que podemos usar con el **try**, no sólo tenemos el **except**, sino también el **else** y el **finally**, los cuales se pueden usar en cualquier combinación. La estructura completa sería:

```
try:
    <bloque de código>
except:
    <bloque de código>
else:
    <bloque de código>
finally:
```

<bloque de código>

Veamos un resumen mostrando las características de cada una de esas partes:

- **try**: da comienzo al manejo de posibles excepciones, supervisando un bloque de código; es obligatorio incluirlo (da comienzo a la estructura) y puede estar una sola vez.
- **except**: ejecuta un bloque de código si en el código supervisado se levantó una excepción, y esa excepción es del tipo definido en el **except** (o incluida en su rama del árbol, como veíamos arriba); puede haber muchos o ninguno, si hay varios la comprobación de la excepción se hace en orden y se ejecuta el bloque de código solamente de aquel que captura la excepción.
- **else**: ejecuta su bloque de código solamente si en el código supervisado *no* se levantó una excepción; es opcional y puede haber a lo sumo uno.
- **finally**: ejecuta su bloque de código *siempre*, no importa qué haya pasado en el código supervisado; es opcional y puede haber a lo sumo uno.

Además de todo el manejo que podemos hacer sobre excepciones que son levantadas en alguna parte del código o por alguna situación, tenemos la opción de levantar personalmente las excepciones integradas de Python, o incluso crear nuevas excepciones para nuestros programas.

Para levantar una excepción sólo tenemos que usar la declaración **raise** y la excepción y el mensaje que queremos levantar:

CELDA 10	
<pre> valor = -3 if valor < 0: raise ValueError("Valor inválido, no puede ser negativo.") </pre>	
<pre> ValueError Traceback (most recent call last) <ipython-input-10-2d18b9ea9787> in <module> 1 valor = -3 2 if valor < 0: ----> 3 raise ValueError("Valor inválido, no puede ser negativo.") ValueError: Valor inválido, no puede ser negativo. </pre>	

Vemos en el ejemplo como el tipo de excepción y el mensaje usados son los mostrados en el traceback.

También podemos usar el **raise** sin especificar una excepción, pero solamente en el contexto de manejar alguna excepción que hayamos capturado: en este caso el **raise** lo que hará es “re-levantar” la misma excepción.

Es muy útil para los casos donde capturamos una excepción pero solamente para manejar algunos casos y otros no. Por ejemplo, en las siguientes líneas capturamos un posible error que ignoramos en un caso y en otros no:

CELDA 11

```
import os
to_remove = '/tmp/somefile.txt'
try:
    os.remove(to_remove)
except FileNotFoundError as err:
    if err.filename.startswith("/tmp/"):
        print("Ignoramos no haber encontrado un temporal")
    else:
        raise
```

Ignoramos no haber encontrado un temporal

CELDA 12

```
to_remove = '/opt/somefile.txt'
try:
    os.remove(to_remove)
except FileNotFoundError as err:
    if err.filename.startswith("/tmp/"):
        print("Ignoramos no haber encontrado un temporal")
    else:
        raise
```

```
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-12-53dbdaf6d26c> in <module>
      1 to_remove = '/opt/somefile.txt'
      2 try:
----> 3     os.remove(to_remove)
      4 except FileNotFoundError as err:
      5     if err.filename.startswith("/tmp/"):

FileNotFoundError: [Errno 2] No such file or directory: '/opt/somefile.txt'
```

Definir nuestros propios tipos de excepciones es muy simple, aunque para ello necesitamos la sintaxis de clases (que veremos más adelante 4.2). El único requisito es que tenemos que “heredar” una excepción integrada de Python; de nuevo, el concepto de “herencia” todavía no lo vimos, pero para nuestro propósito es lo que ponemos entre paréntesis en la definición, veamos un ejemplo simple:

CELDA 13

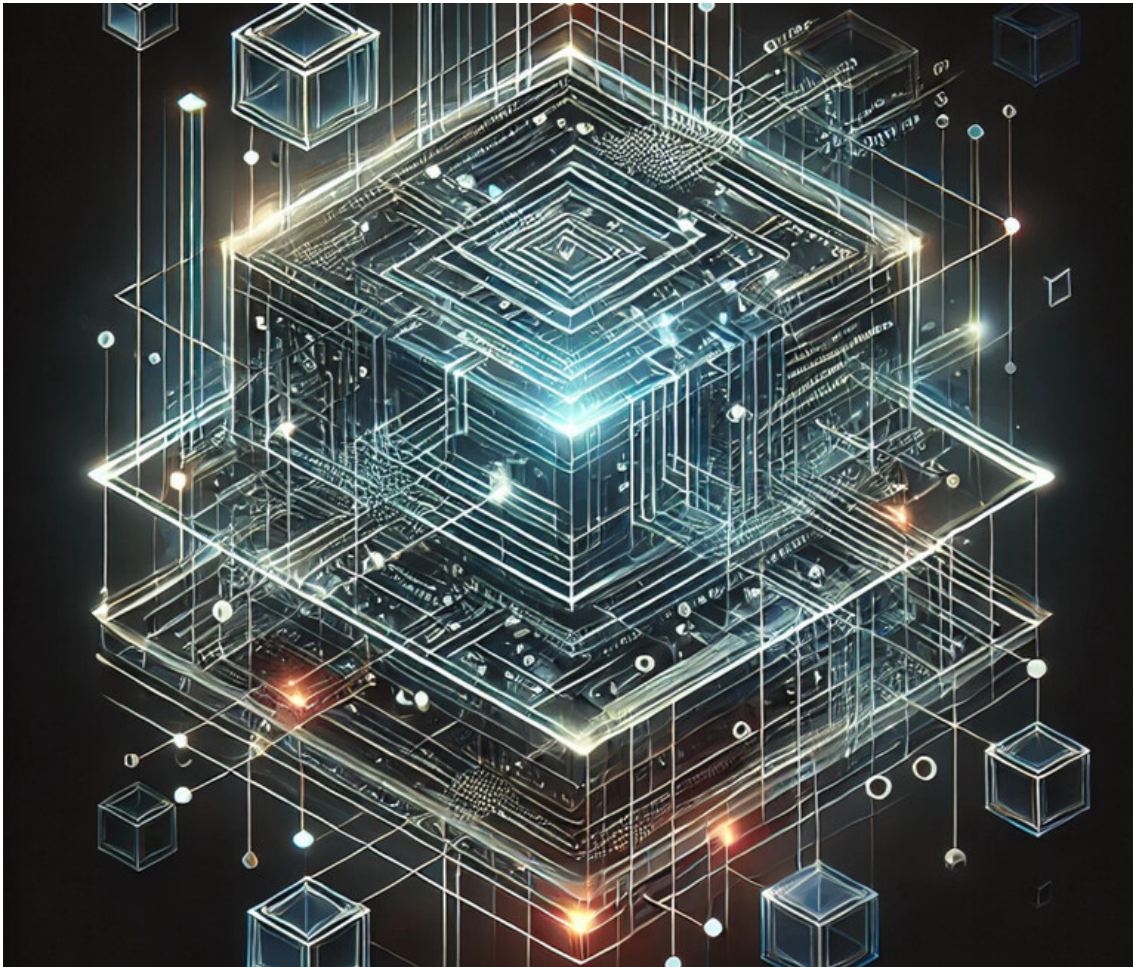
```
class FueraDeRango(Exception):
    """El valor medido está fuera del rango permitido."""
```

Es importante elegir de qué excepción heredamos porque eso es lo que termina armando “el árbol de excepciones” que mencionábamos arriba cuando decíamos que **except** captura el tipo de excepción indicado y a toda su rama.

En el caso del ejemplo estamos heredando **Exception**, pero podríamos heredar alguna otra excepción cuya semántica esté más cerca de la excepción que estamos definiendo (para ello es interesante empaparse de los tipos de [excepciones que trae Python](#)). Por ejemplo, en nuestro caso probablemente estaríamos mejor heredando de **ValueError**, ya que está relacionada con un valor obtenido.

Una vez definida, la usamos como cualquier otra excepción:

CELDA 14	
<pre>raise FueraDeRango("valor negativo")</pre>	
<pre>FueraDeRango</pre>	<pre>Traceback (most recent call last)</pre>
<pre><ipython-input-14-5161da85a699> in <module></pre>	
<pre>----> 1 raise FueraDeRango("valor negativo")</pre>	
<pre>FueraDeRango: valor negativo</pre>	



4 | Encapsulando código

Encapsular código es el acto que nos permite acomodar determinadas líneas de código en alguna estructura para poder reutilizarlas a conveniencia.

[Código disponible](#)

La forma más sencilla en Python de lograr esto son las funciones, que es lo primero que estudiaremos en este capítulo. Luego mostraremos las clases, que nos permiten encapsular no sólo el código sino también los objetos sobre los cuales trabaja ese código (abriéndonos las puertas a la Programación Orientada a Objetos), y finalmente hablaremos de módulos y paquetes, que son capas superiores que nos permiten encapsular funciones y clases para usarlas en distintos programas.

4.1. Funciones

Como mencionábamos al principio del capítulo, la función es la estructura más sencilla para encapsular código.

Nos permite escribir un bloque de código (con sus estructuras de control de flujo, con sus propios bloques de código, etc.) de forma que después podremos ejecutar ese bloque de código

“llamando” a la función desde cualquier lado.

La forma más sencilla de la estructura de una función es:

```
def <nombre>():
    <bloque de código>
```

Esa estructura, aunque funcional, no nos permite pasarle datos ni obtener un resultado. Pero nos sirve para empezar a familiarizarnos con las funciones. Tenemos entonces un “nombre” que es el que usaremos para identificar a la función, y un bloque de código (indentado, como corresponde).

Es importante entender la diferencia entre “definir” una función y “llamar a” (o “ejecutar”) una función. En el primer caso solamente hacemos que Python compile la estructura y la tenga en memoria lista para usar, mientras que en el segundo caso es realmente cuando el bloque de código de la función se termina ejecutando.

En el siguiente ejemplo podemos ver primero la definición en sí de la función, y cómo podemos referenciarla con su nombre (en el `print`, o directamente en el intérprete interactivo, y la diferencia fundamental con ejecutar esa función, al final del ejemplo, cuando escribimos el nombre de la función seguida de paréntesis (sin nada entre ellos, en este caso, porque la función no recibe parámetros).

CELDA 1 DE ENCAP-FUNCS.IPYNB	
<pre>def foo(): print(5)</pre>	
CELDA 02	
<pre>print(foo)</pre>	
<function foo at 0x7fc314631af0>	
CELDA 03	
foo	
<function __main__.foo()>	
CELDA 04	
foo()	
5	



En Python cada objeto puede especificar la mejor forma de representarse cuando se llama `str()` o `repr()` al mismo; por default Python mostrará el tipo de objeto y la posición en su memoria interna de objetos.

Tener una función como esa es válido en algunos casos, pero en realidad la mayoría de las veces estaremos pasándole valores a la función y/o recibiendo resultados de la misma.

Para recibir valores, los tenemos que especificar en la definición de la función. Esto se logra de distintas maneras, y es bastante flexible (lo veremos más abajo), pero por ahora, simplificando, digamos que escribimos los nombres con los que haremos referencia a esos valores, y los podremos acceder desde el bloque de código de la función.

En el siguiente ejemplo definimos una función que recibe dos valores (sería un error pasarle uno o tres):

CELDA 05	
<pre>def foo(a, b): print(a, b)</pre>	
CELDA 06	
<pre>foo</pre>	
<pre><function __main__.foo(a, b)></pre>	
CELDA 07	
<pre>foo(3, 4)</pre>	
<pre>3 4</pre>	

Hasta ahora la función ejecuta su bloque de código y termina. Por default, la función siempre devuelve **None** al terminar, pero tenemos control sobre eso mediante la declaración **return**.

Podemos poner cualquier cantidad de **returns** en una función. Si el flujo del código pasa por una línea con **return**, la función termina y devuelve lo que allí se indica (no importa si hay otros **returns** en otros lados de la función).

CELDA 08	
<pre>def foo(a, b): return a + b</pre>	
CELDA 09	
<pre>r = foo(3, 4) r</pre>	
<pre>7</pre>	

Prestemos atención al detalle de haber llamado a la función y realizar una asignación con el resultado de esa función, para poder trabajar luego con el mismo.

Tengamos en cuenta que podemos obtener más de un resultado cuando termina una función, para lo cual el **return** soporta que escribamos diferentes valores separados por coma y podemos acceder a esos valores con una asignación múltiple.



En realidad, a bajo nivel, lo que sucede es que el **return** está devolviendo una tupla con esos valores, y luego en la asignación del resultado entra en juego lo que llamamos *tuple unpacking* [2.4](#).

Ya con un ejemplo más complejo, armemos una función que recibe dos valores y devuelve dos resultados (¡es posible!): la multiplicación y la división de ambos números (si el segundo número recibido es cero devuelve None como segundo resultado porque la división es imposible):

CELDA 10
<pre>def foo(a, b): mul = a * b if b == 0: return mul, None div = a / b return mul, div</pre>
CELDA 11
<pre>foo(3, -2)</pre> <pre>(-6, -1.5)</pre>
CELDA 12
<pre>m, d = foo(3, -2)</pre>
CELDA 13
<pre>m</pre> <pre>-6</pre>
CELDA 14
<pre>d</pre> <pre>-1.5</pre>
CELDA 15
<pre>print("Resultados: multip={} divis={}".format(m, d))</pre> <pre>Resultados: multip=-6 divis=-1.5</pre>
CELDA 16
<pre>m, d = foo(3, 0) print("Resultados: multip={} divis={}".format(m, d))</pre> <pre>Resultados: multip=0 divis=None</pre>

No hay más para explorar por el lado de devolver valores, así que volvamos sobre el otro lado de usar funciones: pasarle parámetros.

Vayamos mostrando las distintas alternativas. El modo más básico es lo que veníamos haciendo, definir algunos parámetros y pasar valores para los mismos. Veamos cuando esto funciona bien, y también los errores que tenemos al no respetar ese “acuerdo básico”:

CELDA 17

```
def foo(a, b):  
    print(a, b)
```

CELDA 18

```
foo(5, 6)
```

```
5 6
```

CELDA 19

```
foo(5)
```

```
TypeError                                Traceback (most recent call last)  
<ipython-input-19-c5b1cea4a11b> in <module>  
----> 1 foo(5)
```

```
TypeError: foo() missing 1 required positional argument: 'b'
```

CELDA 20

```
foo(5, 6, 7)
```

```
TypeError                                Traceback (most recent call last)  
<ipython-input-20-5b675c25d8d5> in <module>  
----> 1 foo(5, 6, 7)
```

```
TypeError: foo() takes 2 positional arguments but 3 were given
```

Un detalle básico pero interesante es que hay una correspondencia ordinal entre los parámetros especificados en la definición de la función y los valores que pasamos al llamarla: el 5 va a la a y el 6 va a la b; es por esto que llamamos “posicionales” a estos argumentos (lo vemos también en el mensaje de error en el ejemplo).

Cuando definimos la función podemos especificar que algunos de esos parámetros tengan un valor por default, entonces no va a ser necesario pasarlos cuando llamemos a la función:

CELDA 21

```
def foo(a, b=1, c=3):  
    print(a, b, c)
```

CELDA 22

```
foo(9)
```

```
9 1 3
```

CELDA 23

```
foo(9, 5)
```

```
9 5 3
```

	CELDA 24
<code>foo(9, 5, 7)</code>	

9 5 7	

En el ejemplo vemos que si pasamos sólo el valor para a, c y b tienen sus valores por default. En la segunda llamada pasamos valor para a y para b (de nuevo, porque son posicionales, el primer valor al primer parámetro, etc.), pero no para c. Y finalmente, vemos que si les pasamos valores para los tres, no se consideran sus valores por default.

¿Pero cómo haríamos en el ejemplo anterior para pasar un valor a a y a c, pero no a b (y que tome su valor por default)? Para ello nos tendríamos que salir del esquema de parámetros posicionales y empezar a nombrarlos, lo cual es tan sencillo como especificar para qué parámetro queremos que vaya cada valor:

	CELDA 21
<code>def foo(a, b=1, c=3):</code> <code> print(a, b, c)</code>	

	CELDA 25
<code>foo(9, c=7)</code>	

9 1 7	

En este caso vemos que pasamos un 9 que va a a (¡posicional!) pero luego especificamos que el 7 va para c; a b no le terminamos pasando un valor, así que toma su default.

En realidad una vez que nombramos los parámetros, podemos escaparnos totalmente al orden de sus posiciones, más allá que en la definición tengan valores por default o no:

	CELDA 21
<code>def foo(a, b=1, c=3):</code> <code> print(a, b, c)</code>	

	CELDA 26
<code>foo(9, c=7, b=8)</code>	

9 8 7	

	CELDA 27
<code>foo(c=7, a=33)</code>	

33 1 7	

Hasta ahora estamos manejando cantidad de fija de parámetros. Python soporta que en la definición usemos el `*` que consumirá todos aquellos valores que pasemos por posición que no hayan sido tomados todavía:

	CELDA 28
<pre>def foo(a, b, *c): print(a, b, c)</pre>	

	CELDA 29
<pre>foo(1, 2)</pre>	
1 2 ()	

	CELDA 30
<pre>foo(1, 2, 3, 4)</pre>	
1 2 (3, 4)	

En el ejemplo vemos como en la primer llamada los dos valores que pasamos van a los primeros dos parámetros definidos, pero no quedó nada para `c` (entonces es una tupla vacía), mientras que en el segundo caso “sobraron” dos valores, entonces `c` si tiene contenido.

También Python nos ofrece el `**`, que de manera similar consumirá todos los nombrados que no hayan encontrado otro lugar:

	CELDA 31
<pre>def foo(a, b, **c): print(a, b, c)</pre>	

	CELDA 32
<pre>foo(a=1, b=2)</pre>	
1 2 {}	

	CELDA 33
<pre>foo(a=1, c=7, b=3, d=8)</pre>	
1 3 {'c': 7, 'd': 8}	

Para el caso de `*` la estructura donde Python guarda los argumentos posicionales sobrantes es una tupla, ya que sólo es importante el orden, mientras que para el `**` como tenemos valores y nombres, la estructura útil para guardar eso es el diccionario.

Obviamente se pueden combinar todos los casos que estuvimos viendo hasta recién .

	CELDA 34
<pre>def foo(a, b, *c, d=7, e=8, **f): print(a, b, c, d, e, f)</pre>	

	CELDA 35
<pre>foo(1, 2)</pre>	
1 2 () 7 8 {}	

CELDA 36
<pre>foo(1, 2, 3, 4, e=9, j=15)</pre>
<pre>1 2 (3, 4) 7 9 {'j': 15}</pre>

Todo esto funciona mientras no haya ambigüedades en la definición o en el llamado a la función; en esos casos Python mostrará un mensaje de error indicando el problema.

CELDA 37
<pre>def foo(a=7, b): print(a, b)</pre>
<pre>File "<ipython-input-37-258463965655>", line 1 def foo(a=7, b): ^ SyntaxError: non-default argument follows default argument</pre>

CELDA 38
<pre>def foo(a, b=7, **c): print(a, b, c)</pre> <pre>foo(1, 3, a=5)</pre>
<pre>TypeError Traceback (most recent call last) <ipython-input-38-cf3d3e66d1d8> in <module> 2 print(a, b, c) 3 ----> 4 foo(1, 3, a=5) TypeError: foo() got multiple values for argument 'a'</pre>

Así como podemos usar en la definición el `*` para guardar excedentes posicionales en una tupla y `**` para los excedentes nombrados en un diccionario, podemos usarlos en las llamadas a las funciones para “desarmar” una tupla con los valores o un diccionario con los nombres/valores:

CELDA 39
<pre>def foo(a, b, c): print(a, b, c)</pre>

CELDA 40
<pre>t = (1, 2, 3) foo(*t)</pre>
<pre>1 2 3</pre>

CELDA 41
<pre>d = {'b': 2, 'c': 3, 'a': 1} foo(**d)</pre>

1 2 3

Cabe acotar que no estamos mencionando todos los casos posibles, y que hay más reglas y operadores (como forzar a que los parámetros sean nombrados o posicionales), y algunos detalles más, explicados en profundidad en la [referencia del lenguaje](#).

4.1.1. Espacios de nombres

Cuando explicamos cómo funcionaba Python con sus objetos y nombres (en lugar de variables con valores, ver Sección 2.5), usamos unos diagramas donde a la derecha teníamos los objetos en memoria, y a la izquierda otra zona donde poníamos los nombres. Este espacio reservado para los nombres se llama justamente “espacio de nombres” (en inglés *namespace*), y es una zona de memoria donde justamente se guardan los nombres que referencian a los otros objetos.

Traemos esto a colación en esta subsección porque en Python no tenemos solamente un espacio de nombres, sino que pueden haber muchos, y las funciones tienen mucho que ver en eso.

Cuando arranca Python tenemos un espacio de nombre que se conserva hasta que el proceso termina y es accesible desde todos lados: el espacio de nombre “global”. Por otro lado, cada vez que ejecutamos una función, se crea otro espacio de nombres, “local” a la función, que permanecerá activo mientras la función se está ejecutando y desaparecerá cuando la misma termine.

Tenemos que tener en cuenta que lo que se destruye al terminar la función es el espacio de nombre, no los objetos referenciados por los mismos. Claro, algunos objetos quedarán sin referencia luego de que el espacio de nombre desaparezca (y de esos se encarga la administración automática de memoria de Python), pero puede ser que otros objetos estén referenciados de otros lados, y sigan vivos.

Veamos un ejemplo sencillo:

CELDA 42	
<pre>def foo(a, b): x = a * b return x</pre>	
CELDA 43	
<pre>x = 3 y = 5 foo(x, y)</pre>	
15	
CELDA 44	
x	
3	

Analicémoslo en detalle, por partes. Lo primero que tenemos es una definición de una función (que todavía no ejecutamos, claro). Luego le ponemos nombre a dos enteros ('x' e 'y'), que

usamos para llamar a la función. En ese momento se ejecuta la función, que nos devuelve 15.

Si vamos a la ejecución de la función, vemos que esos dos enteros los recibe en dos parámetros que llama 'a' y 'b', realiza un cálculo y devuelve ese valor. Como parte del procesamiento, la función también define un nombre 'x', pero este se define en el espacio de nombres *local* de la función (igual que 'a' y 'b', para el caso).

La 'x' definida en el espacio de nombres local apunta al entero 15, y luego de ejecutar la función vemos que, afuera, 'x' sigue apuntando al 3 original. Esto es porque afuera estamos usando el espacio de nombres global, no se nos mezcla con el espacio de nombres local de la función.

Es importante entender la diferencia entre “definir” un nombre en un espacio de nombres, y tener “acceso” a ese nombre (también decimos “ver” ese nombre, en inglés se usa *scope*). Veamos el siguiente ejemplo para resaltar esta diferencia:

CELDA 45
<pre>def foo(): y = 2 z = 3 print(x, y, z)</pre>
CELDA 46
<pre>x = 8 y = 9 foo()</pre> <pre>8 2 3</pre>
CELDA 47
<pre>x</pre> <pre>8</pre>
CELDA 48
<pre>y</pre> <pre>9</pre>
CELDA 49
<pre>z</pre> <pre>NameError Traceback (most recent call last) <ipython-input-49-3a710d2a84f8> in <module> ----> 1 z NameError: name 'z' is not defined</pre>

Arrancamos definiendo una función (que veremos en detalle a continuación, cuando la ejecutemos), y luego se definen en el espacio de nombres global una 'x' apuntando a un 8 y una 'y' apuntando a un 9.

Cuando ejecutamos la función, esta primero define una 'y' apuntando a un 2 y una 'z'

apuntando a un 3 (en ambos casos, en el espacio de nombres local de la función), y luego hace un print de tres nombres: para 'x' e 'y' es simple, porque está mostrando lo que encuentra en el espacio de nombres local, pero 'z' nos puede sorprender.

Es aquí donde tenemos que entender que a nivel de visibilidad, desde adentro de la función Python intenta resolver el nombre primero buscando en el espacio de nombres local, y luego si no la encuentra allí busca en el espacio de nombres global. Esta secuencia es importante, porque eso determina que el 'y' que encuentra es el que apunta al 2 (¡no al 9!), y para 'z' que no está local pero si global, igual la encuentra.

Por otro lado, desde afuera de la función no tenemos visibilidad a su espacio de nombres, por eso cuando al final queremos ver el valor de 'z' nos da error de nombre, porque 'z' no está definida en el espacio de nombres global (y es en el único en que busca).

Un detalle importante es que como desde adentro de la función tenemos visibilidad sobre los objetos del espacio de nombres global, si los objetos son mutables podremos modificarlos:

	CELDA 50
<pre>def foo(): x.append(3)</pre>	
	CELDA 51
<pre>x = [1] x</pre> <pre>[1]</pre>	
	CELDA 52
<pre>foo() x</pre> <pre>[1, 3]</pre>	
	CELDA 53
<pre>foo() x</pre> <pre>[1, 3, 3]</pre>	

Aunque podría considerarse una mala práctica de programación (porque la función nos está cambiando objetos que viven fuera de ella), en algunos casos es útil y muy ventajoso poder hacer eso. ¡Usar con precaución!

4.1.2. Generadores

Los generadores son un tipo particular de objeto que cuando los iteramos nos dan elementos, pero no los tenían de antes. Los van generando en el momento. Un ejemplo integrado en Python es el `range`. Si hacemos `range(10 ** 100)` obtenemos un objeto que si le pedimos, nos dará enteros entre 0 y $10^{100} - 1$, pero obviamente no preparó todos esos números en el momento. Los irá *generando*.

Se usan mucho en Python porque optimizan el uso de memoria y mejoran el rendimiento general. Veamos el siguiente ejemplo que aunque muy simple, muestra una optimización clara que repetida por varios rincones del lenguaje hacen una diferencia importante:

CELDA 54
<code>sum(range(10))</code>
45

El ejemplo suma los números del 0 al 9, generados por el `range`. Si `range` (en vez de funcionar como generador) armara una lista con los números del 0 al 9, el efecto sería exactamente el mismo, con el detalle que como parte del proceso, se construyó una lista con todos los números en memoria simultáneamente, que luego fue consumida por el `sum`. Esa lista no sirvió para nada, en realidad, sólo ocupó memoria y tiempo para su creación/administración.

Hay distintas formas de construir generadores en Python, pero una de las más simples es armar una “función generadora”. Parece una función normal con la excepción que dentro de su bloque de código usa la declaración `yield`, que justamente le cambia el comportamiento.

¿Recuerdan que dijimos que las funciones cuando las llamamos se ejecutan hasta que terminan, destruyendo su espacio de nombres local, y que cuando las volvemos a llamar vuelven a ejecutarse desde el principio? Bueno, justamente las funciones generadoras cambian ese comportamiento. Cuando llamamos a la función nos devuelve un generador. Cuando le vamos pidiendo elementos a ese generador lo que va a hacer es ejecutar esa función hasta que llega a un `yield`, devolviéndonos lo que allí se indica, “pausando” la ejecución de ese código. Y cuando le pidamos el próximo elemento al generador, ese código “se despertará” en el punto en que estaba y continuará su ejecución hasta que termine o encuentre otro `yield`.

Para ver esto en un ejemplo, primero hagamos una versión casera y recortada del `range` con una función clásica, para poder comparar ambos códigos.

CELDA 55
<pre>def rango(limite): nros = [] n = 0 while n < limite: nros.append(n) n += 1 return nros</pre>

CELDA 56
<code>rango(10)</code>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

CELDA 57
<code>sum(rango(10))</code>
45

En esta función vemos lo que mencionábamos arriba. La función genera una lista con los

números, luego el `sum` la consume sumando esos números y produce el 45 como resultado. No sólo la lista es innecesaria, también mantener todos los números al mismo tiempo en memoria sólo para sumarlos es un desperdicio de recursos. Esto es obviamente un factor si tenemos una lista muy grande, pero tampoco hay que desestimar la situación con estructuras pequeñas, porque cualquier necesidad de memoria puede disparar que el proceso que estamos ejecutando tenga que salir a pedirle memoria al sistema operativo, y eso siempre es caro.

Convirtamos la función de arriba en generadora.

CELDA 58
<pre>def rango(limite): n = 0 while n < limite: yield n n += 1</pre>
CELDA 59
<pre>g = rango(10) g</pre> <generator object rango at 0x7fc314591d60>
CELDA 60
<pre>next(g)</pre> 0
CELDA 61
<pre>next(g)</pre> 1
CELDA 62
<pre>sum(rango(10))</pre> 45

La estructura definida es muy similar, pero notemos que no tenemos la estructura interna `nos` (porque no estamos creando esa lista), y que aparece el famoso `yield`.

Cuando ejecutamos esta función generadora, realmente no se empieza a ejecutar el bloque de código, sino que obtenemos el objeto generador en sí, como mostramos en el ejemplo. Al hacer el primer `next` se empieza a ejecutar el código, hasta que llega al `yield`, allí devuelve el valor que teníamos en `n` (el 0) y la ejecución del código queda suspendida (ya que el control lo tenemos nosotros en el intérprete interactivo). Cuando hacemos el segundo `next`, la ejecución no arranca desde el principio, sino que continua desde donde estaba, sumándole 1 a `n`, luego vuelve a evaluar la expresión del `while` y llega nuevamente al `yield`, devolviéndonos ahora el 1.

Si siguiéramos pidiéndole números con el `next`, los seguiríamos obteniendo hasta que la expresión del `while` de falso, y en ese caso vemos que se termina la función. Como es una función generadora, cuando sale de la función en realidad se genera la excepción `StopIteration` que es la

que usa Python para indicar que no hay más ítems para iterar. Esto nos permite integrar a estos generadores en todas las estructuras normales de Python.

Finalmente entonces, usamos en el `sum` la función generadora que armamos, que vemos que nos devuelve el mismo resultado que con la estructura clásica, pero sin construir la lista intermedia.

También podemos aplicar este concepto de “generador” a las comprensiones de listas 3.3, armando directamente “comprensiones generadoras”, usando paréntesis en lugar de corchetes

CELDA 63
<pre>(x ** 2 for x in range(3))</pre>
<pre><generator object <genexpr> at 0x7fc314591e40></pre>

Incluso podemos obviar esos paréntesis cuando tenemos a la expresión generadora dentro de una llamada a función (porque los dobles paréntesis son superfluos en Python).

Veamos un simple ejemplo donde sumamos los cuadrados del 0 al 9 de dos formas distintas.

CELDA 64
<pre>sum([x ** 2 for x in range(10)])</pre>
285

CELDA 65
<pre>sum(x ** 2 for x in range(10))</pre>
285

En ambos casos arrancamos con `range` (que es generadora), pero en el primero armamos una lista intermedia con los cuadrados (notemos los corchetes que arman la compresión de listas), mientras que en el segundo tenemos una comprensión generadora (no hace falta poner los paréntesis, ya que aprovechamos los de alrededor).

Para el `sum` es exactamente lo mismo, y tenemos el mismo resultado, pero el segundo caso es más rápido y hasta más legible y conciso.

Si quieren profundizar más sobre los conceptos de generadores pueden seguir leyendo sobre este tema en su capítulo específico 5.1.

4.2. Clases

Las “clases” son una forma de encapsular código junto a los objetos que son manejados por ese código.

Aunque suena simple, realmente esto nos permite separar en distintas estructuras los distintos objetos que necesitamos manejar con el código para procesarlos, lo cual nos permite modelar eficientemente la realidad que estamos tratando de representar, abriéndonos las puertas de la Programación Orientada a Objetos (en adelante “POO”).

En este libro mostraremos el funcionamiento básico de las clases, la definición de su estruc-

tura, qué implica instanciarlas y las bases del funcionamiento de los objetos que obtenemos, pero no pretendemos enseñar POO en un capítulo, ya que es un tema para todo un libro (¡o más de uno!).

Entonces la idea es que podamos leer y entender código que usa clases, e incluso construir algunas sencillas, sin pretender desbloquear todo su potencial.

La estructura básica de una clase es extremadamente simple:

```
class <nombre>:
    <bloque de código>
```

Con eso ya armamos una clase con el nombre que queramos, y tenemos un bloque de código para continuar. Como con las funciones, las clases nos crean un espacio de nombres diferente, y allí es que empezaremos a agregar código que luego utilizaremos más adelante.

Armemos un ejemplo con más elementos, para empezar a explicarlos:

CELDA 1 DE ENCAP-CLASES.IPYNB	
<pre>class Foo: a = 3 def f(): print(8) Foo __main__.Foo</pre>	
CELDA 02	
<pre>Foo.a</pre>	
3	
CELDA 03	
<pre>Foo.f</pre>	
<function __main__.Foo.f()>	
CELDA 04	
<pre>Foo.f()</pre>	
8	

En el bloque de código definimos una variable y una función, y luego las usamos. Veamos que hicimos `Foo.a` y `Foo.f()`, porque tanto “a” como “f” no están en el espacio de nombres global, sino en el de la clase (y como con los módulos, cuando hacíamos `math.sqrt(2)` [1.2.3](#), usamos el punto para indicar que estamos usando un nombre de “adentro” de otro objeto). Entonces, es claro que tanto “a” como “f” están adentro de la clase.

Lo que es raro en ese ejemplo es como estamos usando la clase. En realidad el uso normal de una clase es para generar objetos, como es regla en la POO. Es más, como ya estamos metiéndonos en ese mundillo, adoptemos dos términos que son de uso genérico allí y que vamos a encontrar en

muchos textos. Las variables dentro de las clases y objetos las llamaremos “atributos”, mientras que las funciones que definimos allí adentro las llamaremos “métodos”. Entonces, de acuerdo con esta nueva terminología, diremos que tenemos “una clase Foo con un atributo a y un método f”.

Ahora hagamos el último salto y armemos una clase que tiene sentido ser usada para instanciar objetos.

CELDA 05
<pre>import math class TriánguloRectángulo: def __init__(self, cateto1, cateto2): self.cateto1 = cateto1 self.cateto2 = cateto2 def hipotenusa(self): return math.sqrt(self.cateto1 ** 2 + self.cateto2 ** 2)</pre>

Epa, ¡cuantas cosas nuevas! Vayamos entendiéndolas por partes, viendo cómo encaja en este uso distinto que mencionamos arriba.

Las clases, en el paradigma de POO, son las estructuras que encapsulan el código para procesar determinada información, junto a dicha información. Y funcionan como plantillas, que nos darán distintos objetos cada vez que *instanciemos* la clase. Estos objetos serán del mismo tipo (el tipo de los objetos es la clase en sí), y por lo tanto el código encapsulado será el mismo, aunque ese código procesará la distinta información que tendremos en cada objeto.

En nuestro ejemplo tenemos una clase TriánguloRectángulo, donde encapsulamos un código (cómo calcular la hipotenusa a partir de los catetos) junto a la información en sí (los catetos). Si queremos trabajar con distintos triángulos, obviamente tendremos una variedad de pares de catetos, pero en todos los casos la forma de calcular la hipotenusa es la misma; esto está en línea con la filosofía de la POO de que estos objetos modelan y representan nuestra realidad.

Armemos entonces dos triángulos, para experimentar y seguir entendiendo ese código (sin repetir aquí la definición de la clase, por brevedad):

CELDA 06
<pre>t1 = TriánguloRectángulo(4, 5) t1</pre>
<pre><__main__.TriánguloRectángulo at 0x7f18bc45f880></pre>

CELDA 07
<pre>t2 = TriánguloRectángulo(10, 1) t2</pre>
<pre><__main__.TriánguloRectángulo at 0x7f18bc45f910></pre>

CELDA 08
<pre>t1.cateto1</pre>
<pre>4</pre>

CELDA 09	
t2.cateto1	
10	

Vemos que al nombre de la clase le estamos agregando paréntesis, como hacemos con las funciones cuando las ejecutamos. Aquí es similar, pero a la clase la estamos *instanciando*, lo que nos devuelve un “objeto del tipo TriánguloRectángulo”, que guardamos en t1 (y luego hacemos lo mismo con t2).

Cuando le decimos al intérprete interactivo que nos muestre esos objetos, vemos que nos dice que son del tipo TriánguloRectángulo y nos dice que están en posiciones de memoria distintas (con lo cual podemos deducir que son dos objetos distintos). Es más, cuando nos fijamos el valor de `cateto1` de ambos objetos vemos que obtenemos distintos valores (¡son distintos objetos!), cada uno teniendo el primer valor que pasamos cuando instanciamos la clase.

¿Cómo sucedió eso? Si volvemos a la definición de la clase, vemos que allí teníamos un método con un nombre especial, `__init__`. Este método se ejecuta automáticamente cuando instanciamos la clase donde está definido. Entonces, cuando hicimos `TriánguloRectángulo(4, 5)` se instanció la clase y se ejecutó ese método de inicialización, pasándole justamente estos valores que indicamos nosotros.



Los métodos especiales son un **conjunto de métodos predefinidos**, con comportamientos específicos definidos en el lenguaje mismo, que Python utiliza para interactuar con los objetos en todo nivel, por ejemplo llamando a `__init__` para inicializar una clase, o `__iter__` cuando recorremos un objeto con el `for`.

Pero si prestamos un poco más de atención veremos que en su definición `__init__` declara que tiene que recibir 3 parámetros (`self`, `cateto1` y `cateto2`), mientras que nosotros estamos pasando solamente dos. Es que para todo lo que es métodos en las clases, Python inserta automáticamente como primer parámetro al objeto mismo que estamos manejando, al que por convención llamamos `self`.

Y allí vemos que el cuerpo del método `__init__` lo que hace es crear los nombres `cateto1` y `cateto2` *adentro del objeto*, apuntando a los objetos recibidos. Entonces, cuando instanciamos `TriánguloRectángulo` la primera vez, pasamos los valores 4 y 5 y en ese caso el `self` es el objeto que terminamos llamando t1 y guarda esos dos valores, y cuando la instanciamos por segunda vez, pasando los valores 10 y 1, `self` es el objeto que terminamos llamando t2, con estos dos otros valores en vez.

Usemos ahora el otro método que tenemos definido:

CELDA 10	
t1.hipotenusa()	
6.4031242374328485	

CELDA 11
<code>t2.hipotenusa()</code>
10.04987562112089

Vemos que lo ejecutamos desde `t1` y `t2`, y no pasamos ningún parámetro. Pero en la definición, arriba en la clase, recibe el parámetro `self`. Estamos en la misma situación que antes: como es el método de una clase, cuando lo ejecutamos desde una instancia de la clase Python insertará el objeto mismo como parámetro, que oportunamente usamos para la cuenta: cuando hacemos por ejemplo `self.cateto1 ** 2` estamos usando el nombre `cateto1` de adentro del objeto, que para `t1` apuntará a 4 y para `t2` apuntará a 10.

Al final de cuentas, lo que tenemos es un determinado código que se aplica a los valores que tiene cada instancia de esa clase. Exactamente el concepto con el que arrancamos arriba toda la explicación de clases, objetos, y POO.

Otro concepto muy útil cuando queremos modelar la realidad usando objetos es el de “herencia”, generalmente utilizado cuando tenemos algunos comportamientos que son comunes a distintos tipos de objetos. Lo normal es encontrar una clase “padre” y muchas clases “hijas”, pero Python soporta herencia múltiple, aunque es de uso más raro.

Cuando definimos una clase, entonces, si queremos que herede de otra incluiremos a esta última entre paréntesis en la definición de la primera. No vamos a entrar en detalle en este libro sobre cómo explotar todas las características del concepto de herencia, pero lo mencionamos para poder reconocer su uso cuando lo encontremos en algún código.

Particularmente, un caso que incluso ya mostramos en la Sección 3.4 es cuando creamos una excepción propia, que para que sea justamente una excepción utilizable por el lenguaje, la definimos heredando su comportamiento de alguna excepción integrada en el lenguaje:

CELDA 12
<pre>class NoNegativos(ValueError): """Usada al encontrar números negativos, que no se puede."""</pre>

Para profundizar más sobre la utilización de clases pueden seguir leyendo sobre este tema en el capítulo específico de clases 6.

4.3. Módulos

Como vimos hasta ahora, las formas más comunes de encapsular código son las funciones y las clases. Entonces, pondremos código adentro de esas estructuras, que usaremos desde distintos puntos de nuestro programa. El paso natural siguiente es el de agrupar algunas de esas funciones y clases de nuestro programa en “módulos”, de manera de poder importar esos módulos de distintos lugares y tener acceso a las funciones y clases (y cualquier otra estructura) que pongamos allí.

Los módulos no son más que archivos de Python, sin tener casi ninguna otra restricción. Mostremos un ejemplo para ver lo sencillo que es crear y usar un nuevo módulo de Python.

Pongamos el siguiente código en un archivo, al que llamaremos `perimetros.py`:

```
1 import math
2
3 dos_pi = 2 * math.pi
4
5 def cuadrado(lado):
6     """Calcula el perímetro del cuadrado."""
7     return 4 * lado
8
9 def círculo(radio):
10    """Calcula el perímetro del círculo."""
11    return dos_pi * radio
```

Vemos en el ejemplo que además de la definición de esas funciones tenemos otras líneas de código a “nivel de módulo”, como el `import` o el cálculo para tener “dos π ” a mano. Todo el código del módulo se ejecutará cuando lo importemos; se importará `math`, se definirá `dos_pi`, y también se definirán las dos funciones que usaremos luego.

Para usar ese módulo, como es el caso con cualquier otro módulo, sólo tenemos que importarlo. Entonces, en el mismo directorio que grabamos `perimetros.py`, abramos un intérprete interactivo y hagamos:

```
1 >>> import perimetros
2 >>> perimetros.círculo(12)
3 75.39822368615503
```

Necesitamos que el módulo esté en el mismo directorio donde abrimos el intérprete interactivo porque cuando hacemos el `import` Python va a buscar el nombre que indicamos en una serie de directorios, entre ellos el actual del proceso. Claro, podríamos poner nuestro módulo en algunos de los otros directorios donde Python busca, pero ello ya implicaría “instalar” nuestro módulo.

La otra opción para facilitar que podamos encontrar nuestro módulo es directamente agregar el directorio que necesitamos en la lista de lugares donde Python busca. Esto lo podemos hacer a través de la variable de entorno `PYTHONPATH` del sistema operativo, o incluso desde dentro de Python modificando `sys.path`.

Volviendo al ejemplo donde usamos nuestro módulo, vemos que lo importamos usando su nombre, y luego podemos acceder a su contenido usando el `.`, como ya vimos en otros casos. Esta no es la única manera de importar el módulo, también podemos utilizar otra forma en la que en vez de quedarnos con el nombre del módulo para trabajar, nos traemos directamente los nombres de las estructuras internas:

```
1 >>> from perimetros import círculo
2 >>> círculo(12)
3 75.39822368615503
```

Tengamos en cuenta que no cambia nada a la hora de importar el módulo en sí, no es más rápido, ni usa menos memoria, ni ejecuta menos código: la única diferencia es con qué nombres nos quedamos para trabajar.

Bien, ya sabemos agrupar nuestro código en módulos. El próximo paso es agrupar esos módulos en un próximo nivel.

La estructura para agrupar módulos se llama “paquete”, que no son más que directorios.

Para probar esto, creemos un directorio `geom` y pongamos nuestro archivo `perimetros.py` allí. Eso es todo; ahora para importar ese módulo tenemos que especificar el paquete. Veamos las distintas formas de terminar ejecutando la función `círculo` en esta nueva situación: indicando el paquete y el módulo, trayendo el módulo del paquete, y trayendo directamente la función (de nuevo, en los tres casos el módulo se importa exactamente igual, sólo cambia con qué nos quedamos para trabajar).

```
1 >>> import geom.perimetros
2 >>> geom.perimetros.círculo(12)
3 75.39822368615503
4 >>> from geom import perimetros
5 >>> perimetros.círculo(12)
6 75.39822368615503
7 >>> from geom.perimetros import círculo
8 >>> círculo(12)
9 75.39822368615503
```

Si en un directorio/paquete ponemos un archivo con el nombre especial `__init__.py`, este archivo se ejecutará cuando importemos el paquete o cualquier módulo de ese paquete, lo cual es muy práctico para cualquier tipo de inicialización que necesitemos realizar.



5 | Python más avanzado

Este capítulo está dedicado a temas breves avanzados de Python (o no tan introductorios) que son de utilidad en la programación en general y científica en particular.

Cada tema es independiente de la otro, entonces se puede evitar una lectura secuencial y consultar directamente la sección correspondiente al tema buscado.

[Código disponible](#)

5.1. Generadores

Llamamos “generador” a una forma especial de función que al ser ejecutada devuelve un objeto que cuando lo iteremos, irá “generando y devolviendo valores”. Si la miramos es muy parecida a una función común excepto que contiene una o más expresiones **yield**.

El **yield** es el que cambia el juego. A diferencia de una función común que cuando la ejecutamos siempre arranca desde el principio y sigue hasta que se acaba el bloque de código o llega a un **return**, el generador ejecuta hasta que llega al **yield**, devolviendo el valor indicado, y cuando volvemos a pedirle un valor, al iterarlo, continúa “desde donde estaba”.

Pueden ver una introducción al tema más detallada y con más comparaciones contra funciones normales en el capítulo de Encapsulando código 4.1.2, pero veamos lo que describíamos en un ejemplo:

CELDA 1 DE GENERAD.IPYNB
<pre>def supergen(): print("dentro", 1) yield "a" print("dentro", 2) yield "b" print("dentro", 3) for val in supergen(): print("val", val)</pre>
<pre>dentro 1 val a dentro 2 val b dentro 3</pre>

Ejecutamos la función e iteramos lo que nos genera usando un **for**. En realidad acá no se llega a ver el efecto de “ir y volver” al generador, así que mejor mostrarlo si iteramos “a mano” usando la función integrada **next**.

El primer paso es ejecutar la función, que como tiene algún **yield** dentro directamente devuelve el generador pero sin avanzar en la ejecución del código (¡no muestra nada!):

CELDA 02
<pre>gen = supergen() gen</pre>
<pre><generator object supergen at 0x7f252983c890></pre>

Cuando le pedimos el primer valor, vemos que empieza a ejecutar el código, todas las líneas hasta que llega al primer **yield**, donde nos devuelve ese valor:

CELDA 03
<pre>val = next(gen)</pre>
<pre>dentro 1</pre>

CELDA 04
<pre>val</pre>
<pre>'a'</pre>

En este momento el generador está “suspendido” en el **yield**. Cuando le volvemos a pedir un valor **no** va a arrancar desde cero como cualquier función sino que va a continuar a partir del **yield**; en el caso de nuestro ejemplo efectivamente imprimiendo el 2 y luego devolviéndonos la letra b:

CELDA 05
<pre>val = next(gen)</pre>
dentro 2

CELDA 06
<pre>val</pre>
<pre>'b'</pre>

Si ahora le volvemos a pedir un valor va a continuar desde el segundo **yield**: imprimirá el 3 y luego generará una excepción en particular para denotar que se terminaron las posibles iteraciones, el `StopIteration`.

CELDA 07
<pre>next(gen)</pre>
dentro 3
<pre>StopIteration Traceback (most recent call last) Input In [7], in <cell line: 1>() ----> 1 next(gen) StopIteration:</pre>

Tengamos en cuenta que esta excepción siempre sucede pero raramente la vemos; el **for**, por ejemplo, itera al generador hasta recibir esta excepción, y en ese momento lo único que hace es salir de su bucle.

La ventaja más directa de un generador es que si necesitamos una secuencia de valores, en lugar de construir una lista para guardar todos esos valores, nos irá devolviendo uno por uno.

Supongamos que queremos mostrarle al usuario algunos números de la secuencia de Fibonacci; necesitamos algo que nos genere esa secuencia, y desde afuera decidimos qué números mostrar y cómo. Usando una función normal nos quedaría:

CELDA 08
<pre>def fibonacci(limit): # función clásica numbers = [1] a, b = 0, 1 while True: a, b = b, a + b if b > limit: break numbers.append(b) return numbers for n in fibonacci(10): print(n)</pre>

```
1
1
2
3
5
8
```

En este ejemplo realmente no es necesario construir toda la lista para luego mostrarla ni tener todos los números al mismo tiempo, con tener los valores al momento de mostrarlos alcanza. Esto no sólo mejora la utilización de memoria de nuestro programa sino también la performance, ya que evitamos pedirle memoria al sistema operativo una y otra vez para ir creciendo la lista.

CELDA 09

```
def fibonacci(limit): # generador
    yield 1
    a, b = 0, 1
    while True:
        a, b = b, a + b
        if b > limit:
            break
        yield b

for n in fibonacci(10):
    print(n)
```

```
1
1
2
3
5
8
```

Otra mejora con respecto a las funciones es que podemos tener generadores “infinitos” lo cual es útil cuando a priori no tenemos un límite. Adaptando el ejemplo anterior, tendríamos:

CELDA 10

```
def fibonacci(): # generador infinito
    yield 1
    a, b = 0, 1
    while True:
        a, b = b, a + b
        yield b

for n in fibonacci():
    if n > 10:
        break
    print(n)
```

```
1
1
2
3
5
```


8

Aquí vemos como `fibonacci` nos irá dando números para siempre, y es desde afuera que controlamos cuando cortar y por qué; en el ejemplo es luego de pasar un límite, pero podría ser cualquier situación (el usuario dejó de hacer click, etc.) y no tenemos que incluir esa lógica en la función generadora en sí.

Aunque el `yield` convierta a la función en un generador también podemos tener `return` en su bloque de código. En el contexto de un generador, el `return` generará una excepción `StopIteration`, como si hubiese terminado. Veamos el siguiente ejemplo:

CELDA 11

```
def foo(simple):  
    yield 1  
    if simple:  
        return  
    yield 2
```

CELDA 12

```
list(foo(simple=True))
```

```
[1]
```

CELDA 13

```
list(foo(simple=False))
```

```
[1, 2]
```

Si el `return` especifica un objeto, este será usado para instanciar `StopIteration` (tenemos en cuenta que muchas veces no se muestra esta excepción, como cuando usamos el `for`, así que el objeto especificado no siempre será útil).



Como mencionamos en la sección sobre Listas 2.3, el constructor `list` construye una lista con los elementos que obtiene al iterar sobre el objeto que recibe. Esto es muy conveniente para *consumir* un generador y visualizar lo que produjo, como vemos en los ejemplos.

Los generadores tienen una funcionalidad que no es tan conocida: podemos influir desde afuera sobre sus estados. Por ejemplo podemos pasarle valores al `yield`, como en el siguiente generador que primero entrega un 5 y al ser continuado recibe un valor, que lo muestra y lo usa para devolverlo multiplicado por 3:

CELDA 14

```
def foo():  
    a = yield 5  
    print("en el generador:", a)  
    yield a * 3
```


Entonces creamos el generador y lo avanzamos hasta el primer **yield**, nos devuelve el valor esperado, lo mostramos. El próximo paso es usando un método nuevo, el **send**, para *enviarle* un valor al generador; este valor lo recibirá como devuelto por el **yield**, será mostrado por el generador y nos devolverá su múltiplo:

	CELDA 15
<pre> g = foo() val = next(g) print("afuera:", val) </pre>	
afuera: 5	
	CELDA 16
<pre> val = g.send(3) print("afuera:", val) </pre>	
en el generador: 3 afuera: 9	

También podemos indicarle al generador que genere una excepción (con el método **throw**) o directamente cerrarlo (con el método **close**) de manera que su próxima iteración genere **StopIteration**.

Todas estas capacidades juntas de los generadores, no sólo la de quedar suspendidos y luego reaunarse con el estado anterior, sino también la de poderse enviar “mensajes” entre ellos hacen que los generadores sean los predecesores naturales de las “corrutinas” en el mundo asincrónico de Python, del que hablamos en [9.3](#).

5.2. Administradores de contexto

Una de las formas más comunes de encapsular código son las funciones. Nos permiten proveer una determinada funcionalidad que será usada en distintas partes de un programa. Una de las limitaciones de las funciones es que proveen un sólo punto de entrada, entonces se quedan cortas cuando la funcionalidad que queremos proveer debe ejecutarse en parte *antes* de un determinado código, y también *después* de ese código.

Veamos eso en un pequeño pseudocódigo. Supongamos que necesitamos preparar un recurso antes de la ejecución de determinado código, y luego de esa misma ejecución cerrarlo o darlo por terminado; tendríamos algo como lo siguiente:

```

1  (... debemos preparar el recurso ...)
2  arranca el código que usa el recurso
3  ...
4  termina el código que usa el recurso
5  (... debemos terminar o cerrar ese recurso ...)

```

Ya dijimos que no podemos utilizar una función para toda esa funcionalidad, ¡pero podemos usar dos! Una que prepare el recurso, y otra que lo termine.

```
1 resource_setup()
2 arranca el código que usa el recurso
3 ...
4 termina el código que usa el recurso
5 resource_ending()
```

Incluso la primer función podría devolver el recurso a utilizar. Y para asegurar que el recurso se finalice apropiadamente siempre deberíamos soportar el caso en que el código que usa el recurso genere una excepción. Nos va quedando algo como lo siguiente:

```
1 resource = resource_setup()
2 try:
3     # arranca el código que usa el recurso
4     ...
5     # termina el código que usa el recurso
6 finally:
7     resource_ending()
```

Los administradores de contexto vienen a formalizar toda esta idea y al mismo tiempo simplificar nuestro código cuando los usamos. Encapsula las dos funciones que necesitamos (de preparación y de terminación) dentro de un objeto, y marca de forma explícita el bloque de código que usa el recurso (con la sangría luego de usar el **with**):

```
1 with resource_manager() as resource:
2     # arranca el código que usa el recurso
3     ...
4     # termina el código que usa el recurso
```

Ese `resource_manager` que ejecutamos en nuestro código ejemplo es el que devuelve una instancia del administrador de contexto. Este administrador de contexto tiene dos métodos, uno que se ejecuta antes de entrar al bloque de código, y otro que se ejecuta al salir del bloque de código (más allá de que se terminó porque se ejecutaron todas las líneas u ocurrió una excepción). Estos dos métodos son los equivalentes a las dos funciones que usábamos arriba.

Veamos dos ejemplos reales. En el primer caso usamos la función integrada `open` que devuelve un objeto que funciona como administrador de contexto, y nos asegura que el archivo va a estar abierto correctamente antes de arrancar el bloque de código, y se va a cerrar cuando este termine.

```
1 with open("/tmp/prueba.txt") as fh:
2     print(fh.read())
```

Notemos que en este caso de uso el administrador de contexto nos devuelve como recurso a utilizar el mismo *file handler* que usamos para leer del archivo. Es distinto al siguiente caso real donde tomamos un lock antes de entrar al bloque de código y lo liberamos luego, pero donde no necesitamos hacer referencia al recurso desde el bloque de código manejado:

```
1 with threading.Lock():
2     do_something()
3     # etc
```

Para construir nuestro propio administrador de contextos sólo tenemos que entender cómo Python va a llamar a las dos funciones que ya sabemos que necesitamos proveer, la de entrada al bloque de código para preparar el recurso, y la de salida para ocuparnos del mismo. En realidad esas dos funciones son métodos del objeto que proveemos, y tienen que tener un nombre fijo.

La primera es `__enter__`, no recibe nada, y lo que potencialmente devuelva se vincula al objetivo del `with` (el que indicamos con el `as`). La segunda es `__exit__` y recibe tres parámetros y tiene un comportamiento especial con su resultado, dependiendo si en el bloque de código sucedió una excepción o no. En caso de tener una excepción en el bloque de código recibirá el tipo y valor de la misma, y el *traceback* correspondiente, y si la función devuelve `True` esa excepción será suprimida (en caso contrario seguirá su curso natural). Si el bloque de código se completó correctamente, los parámetros de la función estarán en `None` y no importa para nada lo que devuelva.

A modo de ejemplo armemos un administrador de contexto que nos provea un directorio temporal para trabajar dentro del mismo.

```
CELDA 1 DE CONTEXTMGR.IPYNB

import os
import shutil

class TempDir:

    def __init__(self, dirname):
        self.dirname = dirname

    def __enter__(self):
        os.makedirs(self.dirname, exist_ok=True)

    def __exit__(self, exc_type, exc_value, tb):
        shutil.rmtree(self.dirname)
```

Tenemos una clase `TempDir` que al instanciarla recibe el nombre o *path* del directorio que se hará temporal, el cual se guarda en un atributo interno. Además de `__init__` la clase provee los dos métodos necesarios para funcionar como un administrador de contexto. En el `__enter__` crea todos los directorios que correspondan (sin fallar si ya existen), y en el `__exit__` borra ese directorio (sin importar lo que haya dentro). Notemos también que aunque `__exit__` recibe los tres parámetros correspondientes a una posible excepción, no hace nada con ellos: no importa si hubo una excepción o no, el directorio se borra.

```
CELDA 02

import os

example_dir = "/tmp/testbook"
print("¿Existe previamente?", os.path.exists(example_dir))
```

```
¿Existe previamente? False
```

CELDA 03

```
with TempDir(example_dir):
    print("¿Existe durante la ejecución interna?", os.path.exists(example_dir))
    testfile = os.path.join(example_dir, "prueba.txt")
    with open(testfile, "wt") as fh:
        fh.write("prueba")
    print("¿Pudimos usar ese directorio?", os.path.exists(testfile))

print("¿Existe al salir del with?", os.path.exists(example_dir))
```

```
¿Existe durante la ejecución interna? True
¿Pudimos usar ese directorio? True
¿Existe al salir del with? False
```

Antes de usarlo verificamos que el directorio de prueba no existe, y luego para usarlo instanciamos `TempDir` con ese directorio. En el bloque de código del `with` vemos que el directorio sí existe, e incluso creamos un archivo dentro. Finalmente, luego de salir del bloque de código del `with` (como denota el sangrado) vemos que el directorio no existe más.

Hay otra forma de construir administradores de contexto que a priori parece más simple, sacrificando un poco de flexibilidad. Esta segunda forma es escribiendo un generador que, al decorarlo de forma particular, nos va a proveer el comportamiento deseado.

La sintaxis es la siguiente:

```
1  from contextlib import contextmanager
2
3  @contextmanager
4  def nuestro_contextmanager(...):
5      resource = resource_setup()
6      try:
7          yield resource
8      finally:
9          resource_ending()
```

Notemos como esta estructura de código es muy parecida a la que teníamos cuando empezamos a explicar administradores de contexto al principio de la sección. Obviamente, no hace falta llamar a las funciones de preparado y terminación del recurso, ese código podría estar directamente en nuestro generador. El truco de esta sintaxis y el decorador `contextmanager` es que nuestro generador va a proveer el recurso (que se vinculará con el `as` del `with`) en el único `yield` que debe tener.

Luego dependerá de nosotros tener una estructura `try/finally` o `try/except/finally` alrededor del `yield` para reaccionar o no a posibles excepciones del bloque de código del `with`.

Escribamos nuevamente nuestro administrador de contexto de ejemplo pero utilizando esta forma:

CELDA 04

```
from contextlib import contextmanager

@contextmanager
def temp_dir_v2(dirname):
    os.makedirs(dirname, exist_ok=True)
    try:
        yield
    finally:
        shutil.rmtree(dirname)
```

CELDA 05

```
print("¿Existe previamente?", os.path.exists(example_dir))

with temp_dir_v2 (example_dir):
    print("¿Existe durante la ejecución interna?", os.path.exists(example_dir))
    testfile = os.path.join(example_dir, "prueba.txt")
    with open(testfile, "wt") as fh:
        fh.write("prueba")
    print("¿Pudimos usar ese directorio?", os.path.exists(testfile))

print("¿Existe al salir del with?", os.path.exists(example_dir))
```

```
¿Existe previamente? False
¿Existe durante la ejecución interna? True
¿Pudimos usar ese directorio? True
¿Existe al salir del with? False
```

Quedará en la evaluación de cada caso si es lo mejor usar una forma o la otra para escribir un administrador de contexto, en función de si queremos priorizar la flexibilidad o tener un código más conciso.

5.3. Acercándonos a la programación funcional

Sabemos que Python es un lenguaje que permite una multiplicidad de estilos de programación. Podemos hacer un script puramente declarativo o sistemas cien por ciento basados en la programación orientada a objetos. Y también nos provee herramientas que nos acerca a la programación funcional.

La programación funcional descompone un problema en un conjunto de funciones. Idealmente, las funciones solo reciben entradas y producen salidas, y no tienen ningún estado interno que afecte la salida producida para una entrada dada.

En esta sección estudiaremos cuatro detalles que acercan Python a lo que esperaríamos encontrar alguien que se acerca al lenguaje desde la programación funcional: la habilidad de definir funciones dentro de una expresión (que en Python se llaman “funciones lambda”) y las tres operaciones básicas *map*, *filter* y *reduce*.

5.3.1. Funciones lambda

Una función lambda es una función como las que normalmente definimos con **def** pero son anónimas (no tienen un nombre) y su sintaxis hace que permitan ser definidas dentro de otra expresión.

Por ejemplo, supongamos que queremos ordenar una lista de palabras poniendo primero aquellas cuya longitud está más cercana a 10. Para ello usamos la función integrada **sorted** que acepta un parámetro **key** que justamente recibe una función que devuelve el valor que se toma en cuenta para realizar el orden. Escribamos esto usando una función “normal”:

CELDA 1 DE FUNCProg.IPYNB
<pre>def ordenadora(palabra): """Devuelve la diferencia entre el largo de la palabra y 10.""" return abs(10 - len(palabra))</pre>
CELDA 02
<pre>palabras = ["camión", "mono", "heladera", "televisión", "pared", "banana"] sorted(palabras, key=ordenadora)</pre> <pre>['televisión', 'heladera', 'camión', 'banana', 'pared', 'mono']</pre>

La función es tan sencilla y tan orientada a ser usada sólo dentro del **sorted** que estamos ejecutando, que es una buena oportunidad para convertirla en *lambda*:

CELDA 03
<pre>sorted(palabras, key=lambda palabra: abs(10 - len(palabra)))</pre> <pre>['televisión', 'heladera', 'camión', 'banana', 'pared', 'mono']</pre>

En esta última línea vemos que a **key** le pasamos directamente la función que necesitamos, que recibe **palabra** y devuelve la distancia calculada.

La sintaxis de la función lambda es

```
"lambda" [lista_de_parámetros] ":" expresión
```

La lista de parámetros es similar a las funciones definidas con **def**, pero la función en vez de tener todo un bloque de código acepta solamente una expresión; el resultado de calcular esa expresión será lo que devuelve la función.

Cabe acotar que las funciones lambda no son particularmente especiales en Python: no consumen menos memoria ni son más rápidas. Si su sintaxis nos termina limitando para escribir una función que necesita ser más compleja (ya que las lambda no pueden tener declaraciones ni anotaciones), pasamos a utilizar una función clásica y listo.

5.3.2. map, filter y reduce

Estas tres funciones, tan conocidas en la programación funcional, operan básicamente aplicando una función a los elementos de un iterable.

Arranquemos explicando **map**. Su funcionamiento es simple: aplica una función a los valores de un iterable, generando cada uno de los resultados.

Veamos un ejemplo donde calculamos el largo de cada palabra usando la función integrada `len`:

CELDA 04	
<code>palabras</code>	

<code>['camión', 'mono', 'heladera', 'televisión', 'pared', 'banana']</code>	

CELDA 05	
<code>map(len, palabras)</code>	

<code><map at 0x7feccdaf75b0></code>	

El resultado, quizás no esperado, es un objeto `map` que va a generar los resultados cuando lo iteremos. Podemos hacer eso con un `for`:

CELDA 06	
<code>for largo in map(len, palabras):</code> <code> print(largo)</code>	

<code>6</code> <code>4</code> <code>8</code> <code>10</code> <code>5</code> <code>6</code>	

En realidad para mostrar ejemplos o probar porciones de código muchas veces utilizamos `list` para consumir fácilmente el generador y tener una lista con los resultados (cosa que casi nunca hacemos en códigos reales, donde lo iteramos directamente al usarlo):

CELDA 07	
<code>list(map(len, palabras))</code>	

<code>[6, 4, 8, 10, 5, 6]</code>	

En ese mínimo ejemplo usamos la función integrada `len`, con lo cual el resultado son los largos de las palabras. Pero supongamos que queremos los cuadrados de algunos números. Necesitamos una función que no tenemos creada con anterioridad. La necesitamos definir nosotros, pero es tan sencilla que podemos usar una función lambda:

CELDA 08	
<code>nros = [6, 4, 8, 10, 5, 6]</code> <code>list(map(lambda n: n ** 2, nros))</code>	

<code>[36, 16, 64, 100, 25, 36]</code>	

En realidad no hace falta que sean funciones lambda, puede ser cualquier función, pero la posibilidad de definir la función en la misma línea es muy cómodo en algunos casos.

La función `filter` también recibe como parámetros una función y un iterable. Aplicará esa

función a cada elemento del iterable, e irá devolviendo cada elemento solamente si el resultado de esa función es interpretable como verdadero. Dos detalles para resaltar: no devuelve el resultado de la función (como hacía `map`) sino que usa ese resultado para saber si tiene que devolver o no el elemento del iterable, y por otro lado no hace falta que la función devuelva `True` o `False`, sino que interpreta el valor booleano de ese resultado.

CELDA 09
<pre>palabras</pre>
<pre>['camión', 'mono', 'heladera', 'televisión', 'pared', 'banana']</pre>

CELDA 10
<pre>from collections import Counter def muchas_a(palabra): return Counter(palabra)["a"] >= 2 list(filter(muchas_a, palabras))</pre>
<pre>['heladera', 'banana']</pre>

Como caso especial `filter` acepta `None` en vez de la función, cambiando levemente su comportamiento: en este caso devolverá los elementos que ellos mismos evalúen a verdadero (o sea, que `bool()` sobre el objeto devuelva `True`). Veamos un ejemplo típico donde luego de procesar una bloque de texto queremos eliminar las líneas vacías:

CELDA 11
<pre>texto = """ Arranca el texto. Parte media. Vamos cerrando. """ lineas = texto.split("\n") lineas</pre>
<pre>['', 'Arranca el texto.', '', 'Parte media.', '', 'Vamos cerrando.', '']</pre>

CELDA 12
<pre>list(filter(None, lineas))</pre>
<pre>['Arranca el texto.', 'Parte media.', 'Vamos cerrando.']</pre>

La función `reduce`, nuevamente, también recibe como parámetros una función y un iterable, pero a diferencia de las anteriores no aplica la función a cada elemento del iterable. Es más, la función no recibe un argumento sino dos. Al comenzar `reduce` llama a la función con el primero y el segundo elemento del iterable, y recibe su resultado; a partir de este momento irá llamando a la función con el resultado de la ejecución anterior y el próximo elemento del iterable.

Para visualizar mejor este proceso armemos una función que no sólo multiplica dos números

sino que también previamente los muestra.

CELDA 13
<pre> from functools import reduce def multip(a, b): print("nros:", a, b) return a * b result = reduce(multip, [1, 2, 3, 4, 5]) print("result:", result) </pre>
<pre> nros: 1 2 nros: 2 3 nros: 6 4 nros: 24 5 result: 120 </pre>

Vemos en el listado que la función se ejecutó con los números 1 y 2 (de la lista fuente), luego con 2 (el resultado de la multiplicación de los anteriores) y 3 (de la lista fuente), luego con 6 (el resultado de la multiplicación de los anteriores) y 4 (de la lista fuente), etc.

Por supuesto, en estos casos también es muy práctico usar **lambda**:

CELDA 14
<pre> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5]) </pre>
<pre> 120 </pre>

Notemos como a esta función en particular la tuvimos que importar del módulo `functools`: en el pasado lejano era una función integrada igual que `map` y `filter`, pero en la evolución a Python 3 se decidió meterla en el módulo porque no es tan usada.

5.3.3. Alejándonos de la programación funcional

Python agregó las funcionalidades que recién describimos en las primeras versiones del lenguaje, pero luego fueron siendo reemplazadas por otras construcciones más pytónicas.

En concreto, las comprensiones de listas (de las que hablamos en detalle en 3.3) se solapan en funcionalidad con el `map` y `filter`, veamos algunos ejemplos usándolas por separado y en combinación.

El primer caso reemplaza el `map` por una *list comprehension* de forma bastante directa. En el caso del `map` convertimos el resultado en lista para poder mostrarlo, no hace falta hacer eso con la comprensión de lista porque justamente devuelve ese tipo de dato; por otro lado, en un código real para lograr la eficiencia del `map` (que devuelve un generador) deberíamos convertir la *list comprehension* a *generator comprehension*.

CELDA 15
<pre> import math # el logaritmo de unos valores </pre>

```
valores = [15, 0.32, 1.01, 17]
print(list(map(math.log, valores)))
print([math.log(x) for x in valores])
```

```
[2.70805020110221, -1.1394342831883648, 0.009950330853168092, 2.833213344056216]
[2.70805020110221, -1.1394342831883648, 0.009950330853168092, 2.833213344056216]
```

El segundo caso donde filtramos los resultados sí cambia un poco, porque con la comprensión de lista no hace falta crear una función que va a ser llamada para cada elemento, sino que eso es parte de la sintaxis. Esto hace al código mucho más eficiente, porque en Python las llamadas a función, por distintas características dinámicas del lenguaje, son relativamente caras.

CELDA 16

```
# filtramos y nos quedamos con los valores mayores a 1
print(list(filter(lambda x: x > 1, valores)))
print([x for x in valores if x > 1])
```

```
[15, 1.01, 17]
[15, 1.01, 17]
```

Finalmente el ejemplo donde aplicamos tanto el logaritmo como el filtro:

CELDA 17

```
# combinadas! el logaritmo de los valores mayores a 1
print(list(map(math.log, filter(lambda x: x > 1, valores))))
print([math.log(x) for x in valores if x > 1])
```

```
[2.70805020110221, 0.009950330853168092, 2.833213344056216]
[2.70805020110221, 0.009950330853168092, 2.833213344056216]
```

Aquí es más claro la mayor legibilidad de las comprensiones de listas, porque no sólo no obliga a crear nuevas funciones, sino que tampoco sufre del inconveniente del “anidado” (habiendo dicho eso recordemos que también podemos tener comprensiones de listas ilegibles, anidando unas con otras, pero en general en ese caso se termina escribiendo un **for** “clásico”).

5.4. Pruebas de unidad

Las pruebas de unidad, en inglés *unit tests* son la forma más efectiva de verificar que un programa funciona correctamente.

Son pruebas “de unidad” porque la idea es que cada prueba valide la mínima unidad posible de código, tanto para simplificar la prueba en sí como para identificar mejor la falla encontrada si el test no termina satisfactoriamente. El otro extremo de este concepto son las pruebas “de integración” donde se prueban que las distintas partes del sistema (o los distintos sistemas) interactúen correctamente. En la práctica las pruebas de unidad no terminan siendo cien por ciento puras (sólo de unidad) y algún componente de integración tienen, lo cual es útil para simplificar las pruebas en sí.

El conjunto de pruebas de unidad de un determinado programa se ejecutan normalmente al momento de desarrollo para validar que los cambios hechos al agregar una funcionalidad o

corregir un bug no van en detrimento de otras partes de ese programa. Tener un buen conjunto de pruebas de unidad permite lograr una mejora continua de la calidad del sistema; por supuesto siempre existe la posibilidad de que al modificar un programa introduzcamos un bug en alguna sección del código mal probada o directamente sin pruebas, pero si tenemos la política de ir agregando pruebas para todos esos casos la evolución favorable del programa estará asegurada, especialmente permitiendo el aumento de su complejidad sin que se vuelva inmanejable para el equipo de desarrollo.

También se acostumbra correr estas pruebas en sistemas automáticos antes de integrar el código propuesto a la rama principal, antes de empaquetar y distribuir el código, antes de poner una nueva versión del sistema a trabajar en un servidor, o en general antes de cualquier acción que potencialmente impacte en la usabilidad del programa. Estos sistemas que corren las pruebas y luego ejecutan alguna acción determinada en su conjunto se llaman “CI/CD”, por *Continuous Integration / Continuous Delivery* (integración continua / entrega continua).

Un tema importante sobre el que hay muchas discusiones y puntos de vista es en qué momento hay que escribir el test correspondiente a un código. Un extremo es escribir primero todo el código en sí y luego todas las pruebas de unidad. Este caso nunca es recomendable, pero por supuesto si llegamos a un programa o sistema que ya está escrito y no tiene ninguna prueba, obviamente debemos empezar a construir el conjunto de pruebas de unidad en ese punto.

El otro extremo es lo que se llama *Test Driven Development* (TDD), “desarrollo dirigido por las pruebas”, que versa que no debería escribirse código alguno sin antes haber hecho el test correspondiente.

Como todo en la vida, la realidad se desarrolla en los grises intermedios. En códigos grandes y más o menos estables recomendamos realizar la prueba o pruebas correspondientes antes de modificar el código para agregar funcionalidad o solucionar un problema. Pero en programas nuevos o scripts pequeños, o incluso en sistemas estables donde estamos explorando un cambio grande, siempre es recomendable avanzar con “código de exploración” hasta tener más o menos definida la estructura general del cambio o del programa nuevo y recién ahí agregar los tests para ese código, momento a partir del cual cambiamos al modo más cercano a TDD para seguir con la evolución del sistema.

A nivel estructura de proyecto vamos a necesitar diferenciar dos aspectos de “tener pruebas de unidad”. El primero es tener los archivos donde están escritos los diferentes tests, el segundo es definir y configurar qué herramienta vamos a utilizar para encontrar esos archivos con las pruebas, correrlos, y presentarnos un reporte con los resultados.

Hay algunas reglas para seguir a la hora de armar los archivos de prueba. Algunas son convenciones para simplificar el uso de los mismos, otros son defaults de la herramienta que corre pruebas y son más o menos configurables. Es por esto que debemos tomar las siguientes recomendaciones no como reglas absolutas sino como justamente lo que son, recomendaciones, y siempre habrán casos donde no se sigan al pie de la letra...

- ¿Dónde ponemos los archivos con las pruebas? En un directorio separado en la raíz de nuestro proyecto, para que sea fácil excluirllos a la hora de empaquetar nuestro proyecto para distribuirlo. Los archivos deben comenzar su nombre con `test_`.
- ¿Cuántos archivos de pruebas? Uno por módulo o script de nuestro proyecto, ya que ese paralelismo nos permite entender fácilmente dónde ir a buscar los tests correspondientes al código que vamos a tocar. Si nuestro proyectos tiene módulos separados en varios paquetes

(directorios), imitaremos esa estructura dentro del directorio de tests.

- ¿Qué estructura interna en cada archivo? Acá hay dos mundos, influidos por las capacidades de las distintas herramientas para correr las pruebas. La herramienta que viene en la biblioteca estándar de Python, `unittest`, necesita que armemos clases que hereden de `unittest.TestCase` con métodos cuyos nombres arranquen con `test_`. El corredor de tests muy popular `pytest` (que es el que usaremos aquí en los ejemplos), aunque soporta la misma estructura de `unittest`, nos permite escribir directamente cada prueba como una función en el archivo (también arrancando sus nombres con `test_`).

Aunque hay algunos casos especiales más o menos soportados por las distintas herramientas, podemos asumir que cada prueba de unidad va a terminar en uno de los siguiente cuatro estados:

- OK: terminó bien, el código a probar se ejercitó correctamente y todas las verificaciones posteriores fueron exitosas.
- Falla (*failure*): nos indica que, aunque se ejercitó correctamente el código, alguna de las verificaciones que realizamos no fue exitosa
- Error: hubo algún problema no esperado al preparar la prueba o en las posteriores verificaciones
- Saltado (*skipped*): el test no se ejecutó por alguna condición especificada en el test mismo (por ejemplo, el test funciona sólo en un entorno Windows y el desarrollador ejecutó los tests en Linux)

Vayamos construyendo un ejemplo práctico para mostrar las distintas partes de un sistema con pruebas de unidad. Por lo mínimo del ejemplo no hace falta una estructura compleja de directorios, así que para simplificar el código en el repositorio del libro tendremos el módulo con nuestra función y el archivo con las pruebas en el mismo directorio, e iremos cambiando de directorio con cada versión de código.

Supongamos entonces que tenemos el requerimiento de escribir una función que suma dos números. El requerimiento es tan simple que podemos arrancar directamente con las pruebas. Hacemos una básica, y sólo para no tener una sola, lo mismo pero al revés:

```
1 from mod import adder # importamos la función a testear
2
3
4 def test_simple():
5     result = adder(3, 5)
6     assert result == 8
7
8
9 def test_reverse():
10    result = adder(5, 3)
11    assert result == 8
```

La estructura general de cada prueba se divide en tres partes: preparación de la prueba, ejecución del código a probar, verificaciones posteriores. La preparación de la prueba varía mucho

según el caso, puede ser inexistente como en las dos pruebas que mostramos recién o de muchas líneas si el contexto necesario para probar el código es complejo. Luego se ejecuta el código a probar, lo cual casi siempre es llamar a una función y obtener un resultado de la misma (como en nuestros ejemplos recién mostrados) o esperar que se genere una excepción durante esa ejecución (más adelante veremos un ejemplo de esto). Y finalmente las verificaciones, que también pueden ser mínimas (como en nuestro caso) o complejas y en varios pasos.

Es tiempo de escribir nuestra función. Es muy simple:

```
1 def adder(n1, n2):  
2     return n1 + n2
```

Para correr estas pruebas elegimos pytest como herramienta. Usamos fides (cuya utilización se detalla en 7.2.2.3) para que lo instale en un entorno virtual y lo ejecutamos directamente desde allí (con el parámetro -q para que no nos dé un montón de información que en algunos casos es útil pero nos molestaría aquí en el libro; les dejamos como tarea probarlo sin esa opción o incluso con -v para que sea más verboso).

```
[unittesting/v01]$ fides -d pytest -x pytest -q test_adder.py  
..  
2 passed in 0.00s [100%]
```

Dos pruebas pasadas OK.

Nos damos cuenta que podríamos estar manejando más casos en nuestras pruebas: negativos, cero, etc. Más allá que a priori nos parece que nuestra función se comportaría correctamente con estos casos, no es mala idea probar extremos o condiciones menos comunes. Por supuesto, no podemos probar con *todos* los números posibles, pero siempre es recomendable pensar con cuales casos el código podría llegar a tener algún inconveniente.

Por otro lado, si vamos a hacer una función para cada uno de esos casos quedaría todo muy repetitivo. El módulo pytest nos resuelve esta situación proveyendo un decorador que parametriza el test. Entonces ahora nuestra función valida varios casos, recibiendo los números a sumar y la respuesta esperada; en realidad pytest termina ejecutando muchos tests, lo cual nos da la ventaja de la aislación de cada uno y tener como resultado una falla puntual para el caso indicado.

```
1 import pytest  
2  
3 from mod import adder  
4  
5  
6 @pytest.mark.parametrize("a, b, c", [  
7     (3, 5, 8),  
8     (5, 3, 8),  
9     (0, 1000, 1000),  
10    (-23, 0, -23),  
11    (87, -87, 0),  
12    (-15, -15, -30),  
13 ])  
14 def test_numbers(a, b, c):  
15     result = adder(a, b)  
16     assert result == c
```

El decorador de parametrización recibe dos valores: primero los nombres de los argumentos donde la función recibirá los valores, y luego una lista de los casos (en cada caso tenemos que tener la misma cantidad de valores que la cantidad de argumentos definidos). Tenemos total libertad con los nombres de los argumentos o su significado, en este caso usamos `a` y `b` para los dos números fuente y `c` para el resultado esperado.

Corremos las pruebas, vemos que efectivamente son varias:

```
[unittesting/v02]$ fades -d pytest -x pytest -q test_adder.py
..... [100%]
6 passed in 0.01s
```

Como siempre pasa en la vida real, eventualmente nos cambian las especificaciones de la tarea dada. Ahora la función tiene que soportar también que los números vengan como cadena. Por ejemplo, soportar el "3" como si fuese un 3.

Como nuestra función ya está lo suficientemente estable, este cambio lo encaramos como recomendamos arriba: haciendo primero las pruebas. Agregamos entonces algunos tests a lo que ya teníamos de antes:

```
19 @pytest.mark.parametrize("a, b, c", [
20     ("3", 5, 8),
21     (87, "-87", 0),
22 ])
23 def test_with_letters_ok(a, b, c):
24     result = adder(a, b)
25     assert result == c
26
27
28 @pytest.mark.parametrize("a, b", [
29     ("", 5), # empty
30     (87, "foo"), # not a number
31 ])
32 def test_with_letters_bad_value(a, b, c):
33     with pytest.raises(ValueError):
34         adder(a, b)
```

En este caso no sólo agregamos los casos de éxito donde la suma que tenemos que verificar es la correcta, sino también casos donde el comportamiento esperado es que la función genere una excepción del tipo `ValueError` (cuando la cadena no representa de forma válida un número).

Al correr las pruebas nos encontramos con que tenemos una salida extensa. Primero el reporte de un carácter por prueba, luego la situación en detalle para cada prueba, y al final un resumen corto:

```
[unittesting/v03]$ fades -d pytest -x pytest -q test_adder.py
.....FFEE [100%]
===== ERRORS =====
(...)
===== FAILURES =====
(...)
===== short test summary info =====
```

```
(...)
2 failed, 6 passed, 2 errors in 0.02s
```

La última línea nos pone en palabras lo que nos iba informando pytest al correr prueba a prueba al principio: 6 casos estuvieron bien, tenemos dos fallas, y dos errores. Las fallas son esperables, porque agregamos pruebas pero todavía no adaptamos el código correspondientemente, pero los errores no deberían estar allí. Revisamos el mensaje (que omitimos en la salida de recién por brevedad) y encontramos:

```
@pytest.mark.parametrize("a, b", [
    ("", 5), # empty
    (87, "foo"), # not a number
])
def test_with_letters_bad_value(a, b, c):
    E      fixture 'c' not found
```

Esto es que definimos una parametrización para la función usando dos argumentos, pero luego en la definición de la función nos quedó que aceptaba un tercero (el error viene de copiar la función de arriba donde el tercer valor era el resultado esperado, que en este caso no corresponde). Corregimos entonces el bug en nuestro código de prueba...

```
28 @pytest.mark.parametrize("a, b", [
29     ("", 5), # empty
30     (87, "foo"), # not a number
31 ])
32 def test_with_letters_bad_value(a, b):
33     with pytest.raises(ValueError):
34         adder(a, b)
```

...y volvemos a ejecutar:

```
[unittesting/v04]$ fades -d pytest -x pytest -q test_adder.py
.....FFFFF [100%]
(...)
```

Ahora sí, todas fallas, esperables porque todavía no trabajamos el código de nuestra función. Es sencillo, sólo tenemos que convertir lo recibido a número:

```
1 def adder(n1, n2):
2     return int(n1) + int(n2)
```

```
[unittesting/v04]$ fades -d pytest -x pytest -q test_adder.py
..... [100%]
10 passed in 0.01s
```

Todo terminado. Ponemos nuestra función en producción, la incorporamos al sistema, distribuimos el código de alguna manera, o lo que sea que fuese necesario para que la gente use nuestro código. Y tiempo después sucede lo esperado: alguien encuentra un bug. Recibimos un reporte que indica que si ejecutan la función con "2.1" y "3", el resultado no es el que corresponde, (5.1), sino 5.

Analizando nuestra función encontramos que el problema es que estamos convirtiendo las cadenas a números enteros. Antes de intentar cualquier corrección, el primer paso es reproducir el bug con un caso de prueba (en realidad agregamos un caso a la parametrización de `test_with_letters_ok`):

```
19 @pytest.mark.parametrize("a, b, c", [
20     ("3", 5, 8),
21     (87, "-87", 0),
22     ("2.1", 3, 5.1),
23 ])
24 def test_with_letters_ok(a, b, c):
25     result = adder(a, b)
26     assert result == c
```

Como sólo modificamos una de las funciones, ejecutemos esas pruebas nada más, usando el parámetro `-k` de `pytest`:

```
[unittesting/v05]$ fades -d pytest -x pytest -q test_adder.py -k test_with_letters_ok
..F                                     [100%]
(...)
FAILED test_adder.py::test_with_letters_ok[2.1-3-5.1] - ValueError: invalid literal for int() with
↳ base 10: '2.1'
1 failed, 2 passed, 8 deselected in 0.01s
```

Vemos como `pytest` ejecutó tres pruebas (porque esa función es parametrizada), y nos termina indicando que falló un caso, pasaron exitosamente dos, y ocho más fueron deseccionados (son parte del conjunto de pruebas pero no fueron incluidos en el filtro que especificamos).

Es hora de mejorar el código para soportar las nuevas cadenas:

```
1 def adder(n1, n2):
2     return float(n1) + float(n2)
```

Y de paso agregamos varios casos de prueba más para puntos flotantes. Acá debemos tener la preocupación particular de los puntos flotantes binarios donde no es tan sencillo comparar por igualdad (como se muestra en 12.1). Para ello aprovechamos una funcionalidad de `pytest` para comparar si dos números son aproximadamente iguales...

```
19 @pytest.mark.parametrize("a, b, c", [
20     (0, 1.0, 1.0),
21     (-5, 3.5, -1.5),
22     (2.3, -0.1, 2.2),
23 ])
```



```
24 def test_floating_point(a, b, c):  
25     result = adder(a, b)  
26     assert result == pytest.approx(c)
```

```
[unittesting/v04]$ fades -d pytest -x pytest -q test_adder.py  
..... [100%]  
14 passed in 0.02s
```

Y de esta manera terminamos los cambios en esta iteración, fuente del bug reportado. Seguramente nuestro código recibirá cambios en el futuro, productos de corrección de errores o agregado de nuevas características. Los casos de prueba seguirán creciendo y cubriendo toda la funcionalidad que tenemos, de manera de poder asegurarnos que un cambio en el futuro no rompa alguna condición del pasado (de la cual es probable que nos hayamos olvidado), logrando de esta manera que nuestro sistema pueda evolucionar a través del tiempo con la calidad deseada.

5.5. Decoradores

Los decoradores son una *syntax sugar* de Python para reemplazar la definición de una función por otra generada dinámicamente.



Con el término *syntax sugar* (en castellano “azúcar sintáctico”) denominamos a aquella sintaxis pensada para facilitar la escritura de una determinada funcionalidad que puede también implementarse de una manera más laboriosa.

Antes de entrar en la funcionalidad propiamente dicha, presentaremos un ejemplo de cómo podría ser útil esa funcionalidad, para poder entenderla mejor.

Supongamos que tenemos un sistema con varias funciones de todo estilo y queremos imprimir por pantalla los parámetros que recibe cada función antes de ejecutarse y su resultado luego de correrse. Escribamos una función sencilla pero pensemos para el ejemplo que tendríamos muchas funciones con diversas complejidades.

```
def adder(n1, n2):  
    return n1 + n2  
  
res = adder(5, 6)
```

CELDA 1 DE DECORADORES.IPYNB

Por supuesto, podríamos modificar cada una de nuestras funciones y poner *prints* dentro de cada una, algo como lo siguiente:

CELDA 02

```
def adder(n1, n2):
    print(f"Entrada de 'adder': {n1} {n2}")
    result = n1 + n2
    print(f"Resultado: {result}")
    return result

res = adder(5, 6) # ejemplo de código que no queremos tocar
```

```
Entrada de 'adder': 5 6
Resultado: 11
```

Pero en realidad no es la mejor manera: no sólo sería mucho trabajo al tener que modificar todas las funciones una por una, sino que también tendríamos bastante código duplicado en todas las funciones. Por otro lado, esta solución laboriosa tiene la ventaja que no tenemos que modificar todo el código que usa nuestras funciones; en otras palabras, no queremos modificar la última línea de nuestro ejemplo `res = adder(5, 6)`.

Podemos realizar la tarea automáticamente con otra función.

Esta nueva función “impresora” realizará los prints necesarios: el de entrada antes de llamar a la función, y el de salida con el resultado de la misma. Para no tener que modificar todo el código que usa nuestra función original, lo que hacemos es reemplazar esa función original por otra que devuelve la función impresora.

CELDA 03

```
def impresora(func_original):

    def nueva_func(*args, **kwargs):
        print(f"Entrada de '{func_original.__name__}': {args} {kwargs}")
        result = func_original(*args, **kwargs)
        print(f"Resultado: {result}")
        return result

    return nueva_func

def adder(n1, n2): # función original, no la modificamos
    return n1 + n2

adder = impresora(adder) # reemplazamos la función original

res = adder(5, 6) # evitamos tocar el código que usa la función
```

```
Entrada de 'adder': (5, 6) {}
Resultado: 11
```

Antes de entrar en el detalle de la función “impresora” prestemos atención en cómo la usamos: la ejecutamos pasándole la función `adder` que acabamos de definir, y vinculamos su resultado con el mismo nombre que esa función. Es clave entender que en todo el resto del código (la última línea en nuestro ejemplo) cada vez que se use la función `adder` no se estará corriendo la función definida originalmente sino una función creada y devuelta por `impresora`.

La función `impresora` definida arriba recibirá entonces la función original y creará una nueva

función (que es la que devuelve). Esta nueva función...

- recibirá cualquier combinación de parámetros (a través de `*args` y `**kwargs`, como explicamos en 4.1)
- mostrará por pantalla el nombre de la función y los parámetros recibidos
- ejecutará la función original, pasando los parámetros recibidos y guardando su resultado
- mostrará por pantalla ese resultado
- devolverá finalmente ese mismo resultado

En otras palabras, la nueva función definida por `impresora` (que es la que reemplaza a `adder`) tendrá el mismo comportamiento que esa función original, con el agregado de imprimir los parámetros recibidos antes de la ejecución y el resultado después.

Dejamos como tarea para los lectores el adaptar la función `impresora` para soportar el caso de que la función original genere una excepción, ante lo cual debería imprimir por pantalla la excepción ocurrida y luego dejarla continuar.

Destaquemos que la función nueva es del todo genérica: usará la función original y hará de pasamanos de los parámetros recibidos y el resultado final. De esta manera la función `impresora` se podrá usar para reemplazar cualquier función con cualquier cantidad de parámetros. Por ejemplo:

CELDA 04
<pre>def square_root(num): return num ** 0.5 square_root = impresora(square_root) res = square_root(25)</pre> Entrada de 'square_root': (25,) {} Resultado: 5.0

El azucar sintáctico que nos agrega Python sobre esta funcionalidad es la capacidad de redefinir dinámicamente la función definida en el código de forma simple. Entonces en vez de reemplazar cada función con la llamada manual a `impresora`, se hace todo automáticamente decorando la función definida:

CELDA 05
<pre>@impresora def adder(n1, n2): res = n1 + n2 return res @impresora def square_root(num): return num ** 0.5</pre>

```
print("Arrancamos")
res1 = adder(5, 6)
res2 = square_root(25)
print(f"Los resultados son {res1} y {res2}")
```

```
Arrancamos
Entrada de 'adder': (5, 6) {}
Resultado: 11
Entrada de 'square_root': (25,) {}
Resultado: 5.0
Los resultados son 11 y 5.0
```

Notemos que al usar la sintaxis del @ lo único que tenemos que especificar es la función decoradora. Podemos extender esta sintaxis para parametrizar nuestro decorador. Para mejorar nuestro ejemplo, podríamos indicarle un prefijo a la función impresora. Entonces, en vez de decorar con `@impresora`, decoraríamos por ejemplo con `@impresora("=")`. Lo que sucede allí es que en lugar de especificar el @ directamente con la función decoradora, ejecutamos una función que tiene que devolver la función decoradora.

CELDA 06

```
def impresora(prefijo):

    def decorador(func_original):

        def nueva_func(*args, **kwargs):
            print(f"{prefijo * 10} Entrada de '{func_original.__name__}': {args} {kwargs}")
            res = func_original(*args, **kwargs)
            print(f"{prefijo * 10} Resultado: {res}")
            return res

        return nueva_func

    return decorador
```

¡Vemos que ahora tenemos tres funciones anidadas! La primera es la que usamos con el @, que devuelve el decorador en sí. Luego la función decoradora adentro define otra función como veníamos entendiendo antes, no hay sorpresas en ese punto. El único cambio ahí dentro es que los prints ahora usan el prefijo que recibió impresora.

CELDA 07

```
@impresora("=")
def adder(n1, n2): # función original, no la modificamos
    res = n1 + n2
    return res

@impresora("-")
def square_root(num):
    return num ** 0.5

print("Arrancamos")
res1 = adder(5, 6)
```

```
res2 = square_root(25)
print(f"Los resultados son {res1} y {res2}")
```

```
Arrancamos
===== Entrada de 'adder': (5, 6) {}
===== Resultado: 11
----- Entrada de 'square_root': (25,) {}
----- Resultado: 5.0
Los resultados son 11 y 5.0
```

Vemos finalmente que podemos modificar el comportamiento de nuestro decorador.

5.6. Coincidencia de patrones

La coincidencia de patrones (*pattern matching* en inglés) es una funcionalidad que Python ganó en los últimos años (desde la versión 3.10) que permite tener distintos bloques de código que se ejecutarán según determinadas condiciones.

En su forma de utilización más bien básica podría pensarse como una declaración switch (que Python no tiene y conmunmente se la reemplaza por cadenas if/elif), pero en realidad la coincidencia de patrones es mucho más poderosa que eso.

Sintácticamente se construye con una declaración match que contiene la expresión sobre la cual se tratará de encontrar una coincidencia, y una o más cláusulas case, cada una de las cuales especifica el patrón, una guarda opcional (una condición que será evaluada si el patrón coincide) y un bloque de código que se ejecutará si la cláusula case es seleccionada.

Antes de profundizar con los patrones, veamos una construcción sencilla. En la siguiente celda tenemos una función que recibe un valor y eleva 2 a ese valor según algunas condiciones:

CELDA 1 DE PATTERNMATCHING.IPYNB

```
def power_2(value, allowed_floats=False):
    match value:
        case int():
            print(2 ** value)
        case float() if allowed_floats:
            print(2 ** value)
        case _:
            print("ERROR")
```

Podemos ver que al match le especificamos el valor que recibimos, y luego tenemos varios case. En el primer caso entrará si el valor es un entero, en el segundo si el valor es un float y solamente si pasamos allowed_floats en verdadero, y luego un caso final que muestra error.

Ejercitemos cada caso de uso:

CELDA 02

```
power_2(2)
```

```
4
```

CELDA 03
<code>power_2(2.0)</code>

ERROR

CELDA 04
<code>power_2(2.0, allowed_floats=True)</code>

4.0

CELDA 05
<code>power_2(None)</code>

ERROR

La cláusula `match` compara la expresión recibida para ver si coincide con alguno de los patrones. Una vez que coincide con uno, se ejecutará ese bloque de código y dejará de comparar en los próximos. Estos patrones pueden ser simplemente el tipo del valor como vimos recién, o estructuras más complejas (una secuencia a desempaquetar, un diccionario con claves específicas, la instancia de una clase, etc.), e incluso pueden tener variables para capturar parte del objeto que estamos comparando.

Veamos dos casos ejemplos para mostrar más potencialidades de esta herramienta.

El primero es para analizar una posible respuesta a un proceso. Las reglas que implementa esa respuesta son:

- debe ser un diccionario con tres claves: `success`, `result` y `error`
- si `success` está en `True`, `error` debe ser `None` y se captura el resultado
- si `success` está en `False`, `result` debe ser `None` y se captura el error
- el error, a su vez, debe ser otro diccionario con dos claves: `message` y `code`, que deben ser una cadena y un entero respectivamente
- cualquier desvío de estas reglas implica un formato incorrecto

Todas estas reglas las implementamos en un único bloque `match`, con tres `case`, uno para la respuesta satisfactoria, otro para la respuesta en error, y luego el caso final para indicar que el formato de la respuesta no es el esperado:

CELDA 06
<pre>def proc(resp): match resp: case {"success": True, "result": resp_result, "error": None}: print("Respuesta ok:", resp_result) case {"success": False, "result": None, "error": {"message": str(msg), "code": int(code)}}: print(f"Respuesta en error: ({code}) {msg}") case _: print("ERROR: formato de respuesta incorrecto")</pre>

El primer case logra una coincidencia para el caso de respuesta satisfactoria; notemos como al usar un nombre en la estructura del case ese nombre se vincula al objeto que coincide:

CELDA 07
<code>proc({"success": True, "result": 42, "error": None})</code>

Respuesta ok: 42

Veamos el segundo case, que tiene la estructura para un respuesta en error (pero con el formato esperado); en este caso capturamos los valores del error de forma muy sencilla:

CELDA 08
<code>proc({"success": False, "result": None, "error": {"message": "ejemplo", "code": 404}})</code>

Respuesta en error: (404) ejemplo

Un detalle: notemos cómo también validamos los tipos: message debe ser un str y code un int (esta forma de escribir la validación es muy cómoda pero sólo sirve para tipos integrados en el lenguaje, la forma más genérica que funciona para todo es por ejemplo `int()` as `code`).

El último case, al que entraría si ninguno de los dos anteriores coincidieron, nos indica que el formato es incorrecto. Veamos ese caso en las siguientes dos celdas; la primera tiene nulos ambos result y error, y la segunda tiene un formato casi correcto, excepto que el código de error no es un entero:

CELDA 09
<code>proc({"success": False, "result": None, "error": None})</code>

ERROR: formato de respuesta incorrecto

CELDA 10
<code>proc({"success": False, "result": None, "error": {"message": "ejemplo", "code": "not found"}})</code>

ERROR: formato de respuesta incorrecto



Las expresiones en el case parecen construcciones de objetos, pero en realidad hay que verlas como patrones que funcionan justamente a la inversa: desestructurando la expresión para extraer distintos componentes. Es similar a cuando hacemos, por ejemplo, `[a, b] = (3, 5)`; en ese caso no se crea una lista ni se acceden a valores de `a` o `b`.

El segundo ejemplo está basado en un caso más real y práctico: armemos un programa, basado en comandos, para interactuar con archivos. La idea es poder listar los archivos que hay en el directorio actual, pedir información de un archivo, cambiarle el nombre, borrar uno o más archivos, y crear archivos vacíos.

La estructura básica del programa es un bucle infinito donde se pide la orden al usuario, se separa el texto del usuario en palabras, y luego se busca que coincidan esas palabras con

la estructura de cada comando que soporta el programa. Hay un mínimo control de error para soportar que el usuario haga Ctrl-C y salga del programa, pero el resto del código, por legibilidad, no tiene ningún manejo de errores (por ejemplo, para soportar que se pida borrar un archivo que no existe).

Veamos el programa entero y luego expliquemos cómo funciona cada case:

```

1 import os
2
3 while True:
4     try:
5         text = input("\nOrden: ")
6     except KeyboardInterrupt:
7         print("\nSaliendo...")
8         exit()
9     words = text.strip().split()
10
11     match words:
12
13         case ["quit"]:
14             print("Saliendo...")
15             exit()
16
17         case ["list"]:
18             print("\n".join(os.listdir(".")))
19
20         case ["stat", filename]:
21             print(os.stat(filename))
22
23         case ["rename", filename_from, filename_to]:
24             os.rename(filename_from, filename_to)
25             print("Done")
26
27         case ["delete", filename_first, *filenames_rest]:
28             for filename in [filename_first] + filenames_rest:
29                 os.unlink(filename)
30             print("Done")
31
32         case ["touch" | "create", filename]:
33             open(filename, "wb").close()
34             print("Done")
35
36         case _:
37             print("Error: orden incorrecta")

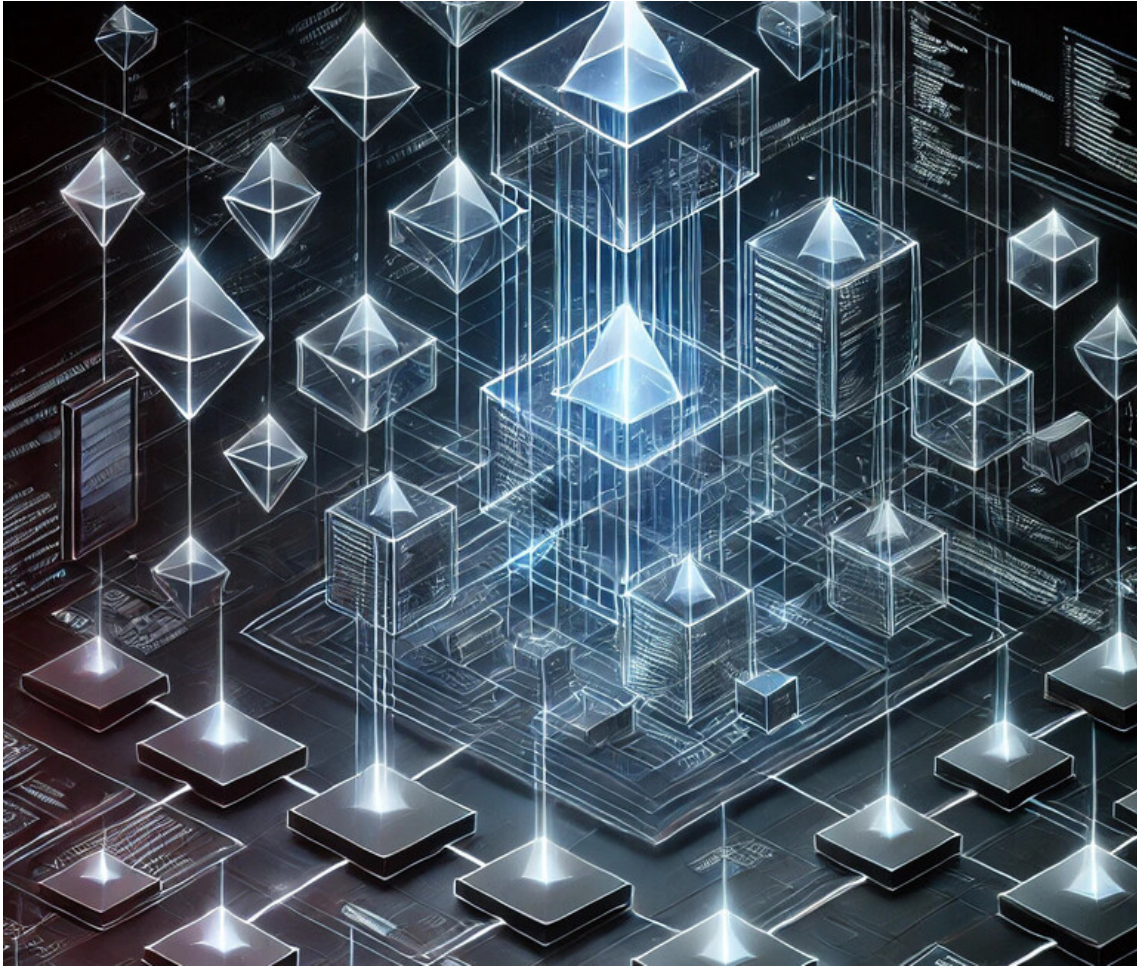
```

El match se realiza sobre la lista de palabras, así que tenemos esa estructura para buscar coincidencias:

- case ['quit'] y case ['list']: dos casos bien sencillos donde la coincidencia es por una palabra que sería el comando principal, sin parámetros, para salir del programa y listar los archivos respectivamente
- case ['stat', filename]: este caso coincide con dos palabras, siendo la primera el comando stat y la segunda queda vinculada al nombre filename, que usamos en ese bloque de código para pedir el estado del archivo

- `case ['rename', filename_from, filename_to]`: muy similar al caso anterior, pero tomando dos parámetros
- `case ['delete', filename_first, *filenames_rest]`: este comando acepta muchos nombres de archivos para borrar, entonces usamos el asterisco para coincidir con cualquier cantidad de palabras (similar al desempaquetamiento de múltiples valores), con el cuidado de tener al menos un nombre, `filename_first`, para evitar la coincidencia con el comando sin ningún parámetro
- `case ['touch' | 'create', filename]`: aquí vemos que cualquier item de coincidencia puede a su vez ser todo un nuevo patrón: también tenemos un comando y un nombre que va a vincularse a un valor, pero ese comando va a coincidir con `touch` o `create`
- `case _`: finalmente tenemos el caso que siempre va a coincidir, para usarlo como default en caso de que no haya coincidido ningún comando formado correctamente y mostrar el mensaje de error

Para seguir aprendiendo sobre coincidencia de patrones la recomendación es ir al [Tutorial de Daniel Moisset](#), que muestra más ejemplos de forma más detallada, y es un buen punto de entrada para el resto de la documentación.



6 | Clases

En este capítulo exploraremos algunos conceptos más avanzados sobre Clases, complementando la introducción que hicimos sobre este tema en [Introducción a Python 4.2](#).

[Código disponible](#)

Entraremos en detalle en distintos aspectos de la definición y comportamiento de clases e instancias. Es importante revisar este capítulo incluso si se tiene experiencia con Programación Orientada a Objetos en otros lenguajes, porque algunos términos son muy parecidos (o iguales) pero su comportamiento puede no ser exactamente el mismo.

6.1. De instancia o de clase

En Python podemos definir atributos que serán de instancia o de clase, haciéndose referencia al espacio de nombres en el que está definido dicho atributo.

Para el caso de la instancia, el atributo “vive” dentro de la instancia, y normalmente desde adentro de la instancia los accedemos usando el argumento `self` que se incluye automáticamente en los métodos de instancia:

CELDA 1 DE CLASES.IPYNB

```
class Power:
    def __init__(self, exp):
        self.exp = exp # ponemos 'exp' *en la instancia*

    def calc(self, value):
        return value ** self.exp # usamos 'exp' que vive en la instancia

pow_a = Power(2)
pow_a.calc(5)
```

25

Desde afuera podemos acceder a ese atributo trabajando con la instancia, y como está definido en el espacio de nombres de cada instancia, si lo modificamos en una, sólo lo modificamos en esa:

CELDA 02

```
pow_b = Power(3)
print(f"1. En a: {pow_a.exp}; en b: {pow_b.exp}")
pow_b.exp = 7
print(f"2. En a: {pow_a.exp}; en b: {pow_b.exp}")
```

```
1. En a: 2; en b: 3
2. En a: 2; en b: 7
```

Noten que no podemos acceder a ese atributo desde la clase, porque no tiene ninguna referencia a sus instancias:

CELDA 03

```
print("Power tiene 'exp'?", hasattr(Power, "exp"))
```

```
Power tiene 'exp'? False
```

Por otro lado, los atributos de clase “viven” dentro de la clase (están definidos en su espacio de nombre), de la misma manera que los métodos; presten atención a cómo el atributo y el método están definidos en el mismo “lugar”:

CELDA 04

```
class Power:
    exp = 2

    def calc(self, value):
        return value ** self.exp

p1 = Power()
p1.calc(5)
```

25

Más allá que esté definido en la clase, cuando se ejecuta el método el valor se puede obtener de la misma forma que antes; esto es porque en el caso de las instancias cuando se buscan en la

misma y no se encuentra, la búsqueda continúa en la clase de esa instancia.

Desde afuera podemos acceder al atributo tanto usando la instancia (por el comportamiento antes mencionado) como usando la clase (porque ahora sí está definido allí):

CELDA 05
<pre>print("Desde la instancia:", p1.exp) print("Desde la clase:", Power.exp)</pre>
Desde la instancia: 2 Desde la clase: 2

La característica de que el atributo viva en un sólo lugar (la clase) y no en todas las instancias tiene dos propiedades interesantes: por un lado nos permite ahorrar memoria si la estructura es muy pesada, y por el otro nos permite “compartirla” entre todas las instancias; en otras palabras, si modificamos el atributo desde la clase vamos a ver desde todas las instancias que se modificó:

CELDA 06
<pre>p2 = Power() print("Desde la otra instancia:", p2.exp)</pre>
Desde la otra instancia: 2

CELDA 07
<pre>Power.exp = 17 print("Desde ambas instancias:", p1.exp, p2.exp)</pre>
Desde ambas instancias: 17 17

El efecto de buscar el atributo en la clase cuando no está en la instancia se anula, justamente, cuando se crea el mismo atributo en la instancia. Entonces, vemos que si creamos el exp en la primer instancia, esa instancia siempre accederá a ese nuevo valor sin llegar al de la clase:

CELDA 08
<pre>p1.exp = 5 print("Desde esa instancia:", p1.exp) print("En la clase no cambió:", Power.exp) print("La otra instancia sigue viendo el de la clase:", p2.exp)</pre>
Desde esa instancia: 5 En la clase no cambió: 17 La otra instancia sigue viendo el de la clase: 17

CELDA 09
<pre>Power.exp = 9 print("La primer instancia ve su propio atributo:", p1.exp) print("La otra instancia sigue viendo el de la clase:", p2.exp)</pre>
La primer instancia ve su propio atributo: 5 La otra instancia sigue viendo el de la clase: 9

Pasamos de hablar de atributos a métodos. Aunque la terminología “de clase o de instancia” es similar, el comportamiento es distinto.

Es más, indicar que un método es “de instancia” es un poco confuso: los métodos siempre “viven” en la clase. En este caso, entonces, tenemos que pensar que ese término hace referencia a cómo están vinculada la función definida con la clase, y qué parámetro se incluirá automáticamente al ejecutarla.

Entonces, en el caso de un “método de instancia” cuando es invocado desde la instancia recibirá automáticamente un puntero a la instancia misma, que por convención llamamos *self* (“si mismo” en inglés) en el código:

CELDA 10
<pre>class Prueba: def met1(self): print("Método de instancia:", self) pru = Prueba() pru.met1()</pre>
Método de instancia: <__main__.Prueba object at 0x7f36445230d0>

Tengamos en cuenta que si al método “de instancia” lo llamamos desde la clase misma, no habrá ningún parámetro incluido automáticamente, es nuestra responsabilidad pasar explícitamente la instancia que recibirá el método:

CELDA 11
<pre>Prueba.met1(pru)</pre>
Método de instancia: <__main__.Prueba object at 0x7f36445230d0>

Alternativamente tenemos los métodos de clase, que recibirán automáticamente un puntero a la clase misma, y métodos estáticos, que no recibirán automáticamente nada extra. En estos casos no importa si son llamados desde la instancia o desde la clase.

Se definen usando los decoradores apropiados:

CELDA 12
<pre>class Prueba: def met1(self): print("Método de instancia:", self) @classmethod def met2(cls): print("Método de clase:", cls) @staticmethod def met3(): print("Método estático!")</pre>

```

pru = Prueba()
pru.met1()
pru.met2()
pru.met3()
Prueba.met1(pru)
Prueba.met2()
Prueba.met3()

```

```

Método de instancia: <__main__.Prueba object at 0x7f3644522560>
Método de clase: <class '__main__.Prueba'>
Método estático!
Método de instancia: <__main__.Prueba object at 0x7f3644522560>
Método de clase: <class '__main__.Prueba'>
Método estático!

```

Los métodos de clase son muy utilizados para implementar métodos alternativos para crear una instancia de la clase. Por ejemplo, en el módulo `datetime` tenemos la clase `date` que al instanciarla normalmente debe recibir el año, mes y día...

CELDA 13

```

from datetime import date

date(2022, 12, 18)

datetime.date(2022, 12, 18)

```

...y alternativamente podemos crear una instancia llamando al método de clase `fromtimestamp` desde la clase misma:

CELDA 14

```

date.fromtimestamp(1704715285)

datetime.date(2024, 1, 8)

```

Copiamos aquí el código real para dicho método (sin el *docstring* y sin el resto de la clase, por cuestiones de espacio) para ayudar a entender cómo trabaja:

```

1  class date:
2
3  @classmethod
4  def fromtimestamp(cls, t):
5      y, m, d, hh, mm, ss, weekday, jday, dst = _time.localtime(t)
6      return cls(y, m, d)

```

Vemos que el método recibe la clase como primer parámetro, automáticamente, y luego el *timestamp* que le pasamos. Separa ese valor en año, mes, día, hora, etc., y luego utiliza la misma clase recibida para construir la instancia (usando los valores correspondientes a la fecha), devolviendo justamente eso.

Por otro lado, los métodos estáticos no reciben automáticamente nada (ni cuando se los llama desde la clase ni cuando se los llama desde la instancia). No tienen diferencia con una función

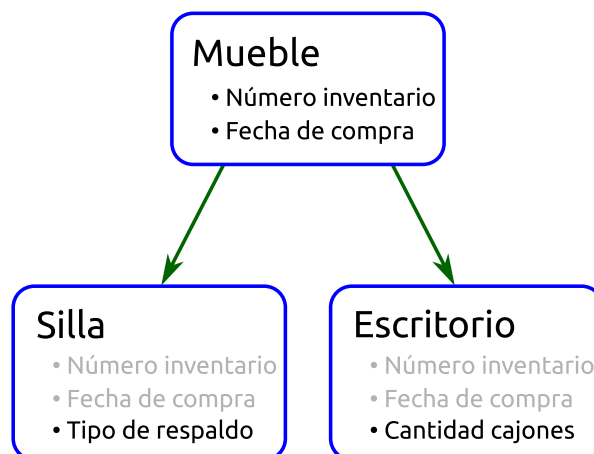
“común”. Entonces, ¿por qué definirlos como un método estático y no como una función fuera de la clase? En este caso entra en juego una de las características de la programación orientada a objetos que es la responsabilidad: es tarea de esa clase (esos objetos) realizar el cómputo implementado en ese código, entonces es mejor que quede adentro de la clase (como método estático) a que esté “suelto” como función separada.

6.2. Heredando o no

Pasemos a otro concepto que visitamos brevemente en la parte introductoria: la herencia.

Genéricamente hablando, la herencia es un mecanismo de la programación orientada a objetos para reutilizar parte de la implementación del modelado de la realidad. Como las clases hijas adquieren propiedades y comportamientos de las clases ancestras (los “heredan”), también nos referimos a que las clases hijas “especializan” a las ancestras.

Veamos un ejemplo muy sencillo de eso:



En el modelado de la realidad que hicimos para un supuesto sistema de inventario de una oficina, tenemos la clase `Mueble` que tiene atributos genéricos como el número de inventario o la fecha de compra. Elementos más específicos, como el `Escritorio` o la `Silla` tendrán cada uno sus atributos en particular, pero además heredan los que posee su clase ancestra (ya que los escritorios y las sillas son muebles).

En Python, el concepto clave a entender cuando tenemos una estructura de herencia es que los métodos y atributos no se “pisan” en tiempo de compilación sino que todo se resuelve durante la ejecución del programa.

El siguiente ejemplo nos muestra distintas características que pasaremos a explicar luego, pero básicamente tenemos una clase `Persona` que recibe la fecha de nacimiento (y la guarda como atributo) y tiene un método para calcular la edad, y una clase `Empleado` que hereda de la anterior, que además posee el sueldo y una forma de calcularlo para todo el año.

CELDA 15

```
from datetime import date

class Persona:
    def __init__(self, nacimiento):
        self.nacimiento = nacimiento

    def edad(self):
        hoy = date.today()
        return hoy.year - self.nacimiento.year

class Empleado(Persona):
    def __init__(self, nacimiento, sueldo):
        self.sueldo = sueldo
        super().__init__(nacimiento)

    def anual(self):
        return self.sueldo * 12
```

El uso de estas clases es bastante intuitivo:

CELDA 16

```
p = Persona(date(1986, 10, 4))
print("Edad p:", p.edad())
e = Empleado(date(1991, 9, 17), 15000)
print("Edad e:", e.edad())
print("Anual e:", e.anual())
```

```
Edad p: 28
Edad e: 23
Anual e: 180000
```

Notemos como usamos el método `edad` en el objeto tipo `Empleado`, ya que esa clase lo “hereda” de `Persona`. Como mencionamos recién, ese método no se “trae” o “copia” de una clase a la otra, sino que es una cuestión de cómo se “resuelve una búsqueda”. Cuando hacemos `e.edad` se trata de resolver el atributo `edad` en la instancia y en su clase, y al no encontrarse se empieza a recorrer sus “ancestras” (en este caso de forma trivial ya que hereda de una sola clase).

Por otro lado, el método `__init__` en `Empleado` “pisa” u “oculta” al método con el mismo nombre en `Persona`. Entonces cuando instanciamos `Empleado` se ejecutará el método de inicialización de `Empleado`, y no hay ningún mecanismo automático que ejecute otro método con el mismo nombre en sus ancestros; si queremos que eso suceda debemos hacerlo de forma explícita (como es el caso de nuestro ejemplo, donde inicializamos `Empleado` guardando el sueldo y luego queremos que se termine de inicializar `Persona` con la fecha de nacimiento. Por otro lado, si queremos que el método de la clase hija reemplace o substituya completamente el método de las ancestros, no hace falta realizar ninguna acción, ya que es el comportamiento por defecto.

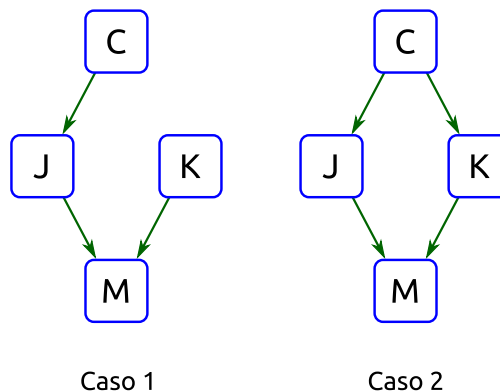
Como `Empleado` hereda de `Persona`, si quisiéramos ejecutar el método de inicialización de esta última era suficiente hacer `Persona.__init__(self, nacimiento)` (notar como debemos incluir explícitamente el `self`, puntero a la instancia, porque estamos usando el método directamente desde la clase, como ya vimos en esta sección). Sin embargo es buena costumbre no hacerlo así directamente, ya que esta estructura de dos clases podría crecer y complicarse a futuro y ya no

ser tan evidente cual es el próximo ancestro que le corresponde ejecutar el mismo método.

Entonces Python nos provee de una función integrada, `super`, que nos saca el trabajo de determinar cual es la próxima clase ancestral, más allá de como vaya cambiando la estructura (y además nos evitamos tener que pasar el `self` explícitamente). `super` (que viene del prefijo que añade los sentidos de “encima” o “por encima” a una palabra) devuelve un objeto intermediario que busca en las clases del camino de herencia, pero en realidad la determinación de cuales son esas clases no depende de `super` mismo sino que está especificado en el atributo `__mro__` de las clases (por *method resolution order*, orden de resolución de métodos).

Debemos entender que no siempre es sencillo darnos cuenta del camino correcto para ir recorriendo las clases ancestras, especialmente cuando dejamos de tener herencia “lineal” y empezamos a encontrar “rombos”.

Veamos cómo es el orden en los siguientes dos casos:



El código correspondiente para el primer caso es el siguiente:

```
CELDA 17

class C:
    pass

class J(C):
    pass

class K:
    pass

class M(J, K):
    pass

for obj in M.__mro__:
    print(obj)

<class '__main__.M'>
<class '__main__.J'>
<class '__main__.C'>
```

```
<class '__main__.K'>
<class 'object'>
```

Vemos que el contenido del atributo `__mro__` nos indica que luego de `M` tenemos a `J` (primera clase ancestra), luego `C` y `K`, y finalmente `object` (que es el tipo especial de la cual derivan todos los objetos en Python).

Para ir al segundo caso modificamos a la clase `K` (para que también herede de `C`):

CELDA 18

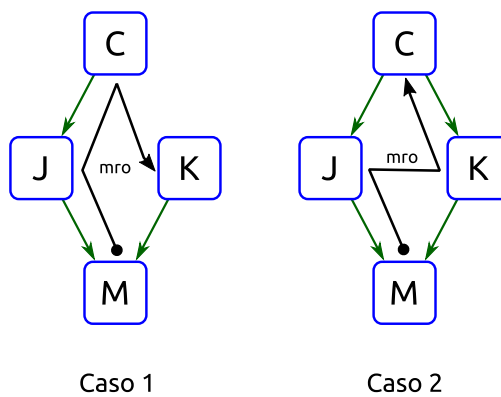
```
class K(C):
    pass

class M(J, K):
    pass

for obj in M.__mro__:
    print(obj)
```

```
<class '__main__.M'>
<class '__main__.J'>
<class '__main__.K'>
<class '__main__.C'>
<class 'object'>
```

Vemos que aunque no modificamos `M`, ¡cambió el orden de resolución de la búsqueda! Los dos caminos son:



Corolario: siempre usemos `super`.

6.3. Usando propiedades

A nivel de la interfaz que se usa en las clases a la hora de construir objetos modelando la realidad, Python difiere un poco de otros lenguajes más rígidos. Por ejemplo, en otros ámbitos es normal tener los atributos protegidos dentro de la instancia (no se los puede acceder ni modificar directamente) e implementar *getters* y *setters* para ello.

Emulando esta misma modalidad tendríamos:

CELDA 19
<pre> class Persona: def __init__(self, nombre, nacimiento): self.nombre = nombre self.nacimiento = nacimiento def get_nombre(self): return self.nombre def set_nombre(self, value): self.nombre = value def get_nacimiento(self): return self.nacimiento def set_nacimiento(self, value): self.nacimiento = value </pre>
CELDA 20
<pre> p = Persona("carlos", date(1980, 12, 15)) print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}") p.set_nombre("Carlos") print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}") </pre> Persona carlos del 1980-12-15 Persona Carlos del 1980-12-15

Si en lugar de dos atributos tenemos cinco o diez ya se pueden dar una idea de la cantidad de código “inútil” a la que nos enfrentamos.

En realidad, este código con todos los *getters* y *setters* tiene su motivo de ser. Tenemos que pensar qué pasa si luego de construir la clase y empezarla a usar en nuestros sistemas, de repente entendemos que debemos ejecutar código al leer o escribir algún atributo. Pueden haber mil razones para esto, adaptaciones de valores, validaciones de formatos, cambios de estructuras, etc. Por ejemplo, lo siguiente no queremos que sea posible, ¡debemos validar que la fecha de nacimiento esté en el pasado!

CELDA 21
<pre> p.set_nacimiento(date(2980, 12, 15)) print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}") </pre> Persona Carlos del 2980-12-15

Como tenemos los *getters* y *setters*, agregar la validación es trivial, y no tenemos que tocar nada de todo el código que usa nuestra clase:

CELDA 22

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    def get_nombre(self):
        return self.nombre

    def set_nombre(self, value):
        self.nombre = value

    def get_nacimiento(self):
        return self.nacimiento

    def set_nacimiento(self, value):
        if value > date.today():
            raise ValueError("No puede nacer en el futuro")
        self.nacimiento = value
```

CELDA 23

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
p.set_nombre("Carlos")
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
p.set_nacimiento(date(2980, 12, 15))
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
```

```
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15
```

```
ValueError                                Traceback (most recent call last)
Input In [23], in <cell line: 5>()
      3 p.set_nombre("Carlos")
      4 print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
----> 5 p.set_nacimiento(date(2980, 12, 15))
      6 print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
```

```
Input In [22], in Persona.set_nacimiento(self, value)
     16 def set_nacimiento(self, value):
     17     if value > date.today():
--> 18         raise ValueError("No puede nacer en el futuro")
     19     self.nacimiento = value
```

```
ValueError: No puede nacer en el futuro
```

La forma pythonica de armar una clase para el mismo caso es no utilizar *getters* y *setters* explícitos, sino directamente acceder a los atributos desde afuera:

CELDA 24

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento
```

CELDA 25

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nombre = "Carlos"
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nacimiento = date(2980, 12, 15)
print(f"Persona {p.nombre} del {p.nacimiento}")
```

```
-----
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15
Persona Carlos del 2980-12-15
```

Notemos como tanto la definición como el uso de esa clase son más limpios y breves. Pero claro, es evidente ahora la situación de querer agregar la validación que necesitamos (o, en forma genérica, ejecutar código específico en el momento de leer o escribir un atributo). ¿Cómo hacemos eso sin modificar todo el código que usa la clase? Para ello Python tiene las “propiedades”.

A través de `property`, integrada en el intérprete, podemos definir propiedades en las clases: declaraciones que podemos utilizar para ejecutar métodos al momento de (en vez de) leer o escribir un atributo, o incluso borrarlo.

La forma clásica de hacerlo es escribir el *getter* y el *setter* para el atributo, métodos que se ejecutarán cuando lo leamos y escribamos y que internamente van a utilizar otro nombre (normalmente el nombre original con un guión bajo al principio), y luego definir al nombre original del atributo como una propiedad indicando los métodos recién definidos:

CELDA 26

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    def _get_nacimiento(self):
        return self._nacimiento

    def _set_nacimiento(self, value):
        if value > date.today():
            raise ValueError("No puede nacer en el futuro")
        self._nacimiento = value

    nacimiento = property(_get_nacimiento, _set_nacimiento)
```

Vemos que ahora el “resto del código” queda igual que antes, usando los atributos directamente, y tenemos toda la funcionalidad que deseábamos:

CELDA 27

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nombre = "Carlos"
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nacimiento = date(2980, 12, 15)
print(f"Persona {p.nombre} del {p.nacimiento}")
```

```
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15
```

```
ValueError                                Traceback (most recent call last)
Input In [27], in <cell line: 5>()
      3 p.nombre = "Carlos"
      4 print(f"Persona {p.nombre} del {p.nacimiento}")
----> 5 p.nacimiento = date(2980, 12, 15)
      6 print(f"Persona {p.nombre} del {p.nacimiento}")
```

```
Input In [26], in Persona._set_nacimiento(self, value)
     10 def _set_nacimiento(self, value):
     11     if value > date.today():
--> 12         raise ValueError("No puede nacer en el futuro")
     13     self._nacimiento = value
```

```
ValueError: No puede nacer en el futuro
```

Mediante property también podemos definir un método a ejecutar en caso de querer borrar el atributo, e incluso una cadena de documentación (*docstring*).

Una ventaja de las propiedades es que ese método que ejecuta automáticamente también se dispara cuando se hace la asignación en el método de inicialización. De esta manera tenemos “gratis” también la validación en tiempo de instanciación:

CELDA 28

```
Persona("Juan", date(2100, 12, 15))
```

```
ValueError                                Traceback (most recent call last)
Input In [28], in <cell line: 1>()
----> 1 Persona("Juan", date(2100, 12, 15))
```

```
Input In [26], in Persona.__init__(self, nombre, nacimiento)
      3 def __init__(self, nombre, nacimiento):
      4     self.nombre = nombre
----> 5     self.nacimiento = nacimiento
```

```
Input In [26], in Persona._set_nacimiento(self, value)
     10 def _set_nacimiento(self, value):
     11     if value > date.today():
--> 12         raise ValueError("No puede nacer en el futuro")
     13     self._nacimiento = value
```

```
ValueError: No puede nacer en el futuro
```

Entonces no hace falta modificar ninguna parte del sistema que usa nuestra clase, y tenemos la funcionalidad deseada, sin tener que pagar el costo de llenar todo de *getters* y *setters* de forma

anticipada por las dudas.

Alternativamente, una forma de utilizar `property` es mediante decoradores; la funcionalidad obtenida es la misma, es sólo una cuestión de gustos. Notar la diferencia entre el decorador para la lectura del atributo, que usa `property`, y el otro para su escritura, que usa el nombre recién definido; ambos métodos usan el nombre del atributo que queremos trabajar (en el primer caso es necesario, en el segundo es por convención):

CELDA 29

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    @property
    def nacimiento(self):
        return self._nacimiento

    @nacimiento.setter
    def nacimiento(self, value):
        if value > date.today():
            raise ValueError("No puede nacer en el futuro")
        self._nacimiento = value
```

Las propiedades también pueden usarse para bloquear el acceso a un atributo. Por ejemplo en el siguiente caso al no definir un método *setter* hacemos que el atributo no se pueda escribir desde afuera:

CELDA 30

```
class Persona:

    def __init__(self, nombre):
        self._nombre = nombre

    def _get_nombre(self):
        return self._nombre

    nombre = property(_get_nombre)
```

CELDA 31

```
p = Persona("Juan")
p.nombre
```

```
'Juan'
```

CELDA 32

```
p.nombre = "Pedro"
```

```
AttributeError                                Traceback (most recent call last)
Input In [32], in <cell line: 1>()
----> 1 p.nombre = "Pedro"
```

```
AttributeError: can't set attribute 'nombre'
```

Es trivial esconderlo totalmente, al no pasarle ningún método a property.

En realidad muy pocas veces hacemos esto en Python, ya que tenemos la fuerte convención de usar nombres que comienzan con guión bajo para indicar aquellos métodos o atributos que son privados de un objeto y nunca deberían usarse “desde afuera” (con excepciones, claro, por ejemplo al realizar pruebas de unidad para esos objetos).

6.4. Creando tipos de datos

Cuando necesitamos crear nuevos tipos de datos para utilizar en nuestros programas, lo ideal es que se integren lo mejor posible al lenguaje.

Por ejemplo, supongamos que para nuestro sistema de gestión de centros de datos tenemos una estructura llamada Rack que adentro puede tener uno o más equipos informáticos, entonces podríamos hacer:

CELDA 34
<pre>rack = Rack("power-source", "disks", "cpus", "router") print("Total de componentes:", rack.length()) print("Posición 1:", rack.get_from_position(1)) print("Alimentación:", rack.power())</pre>
<pre>Total de componentes: 4 Posición 1: disks Alimentación: ['power-source']</pre>

Aunque esos métodos tienen sentido, son parte de una interfaz que hay que aprender al usar esa clase. En Python hacemos siempre foco en hacer las cosas lo más simple posible y la legibilidad importa, entonces si tenemos la funcionalidad de “obtener el largo” o de “obtener un item de una determinada posición”, ¿por qué no proveer los mecanismos para que esas acciones se realicen de la misma manera que es natural en Python?

Entonces, idealmente si nuestro tipo de datos es (por ejemplo) una “colección” de otras cosas, sería natural para los programadores hacer lo siguiente:

CELDA 36
<pre>rack = Rack("power-source", "disks", "cpus", "router") print("Total de componentes:", len(rack)) print("Posición 1:", rack[1]) print("Alimentación:", rack.power())</pre>
<pre>Total de componentes: 4 Posición 1: disks Alimentación: ['power-source']</pre>

Por supuesto, no todos los métodos tienen un equivalente en el uso de Python, como es el caso en esta clase cuando devuelve las fuentes de alimentación.

Pero en los casos donde la funcionalidad es equivalente, podemos aprovechar que Python

ofrece una especie de protocolo entre la sintaxis usada y la definición de una clase. La idea es más sencilla de lo que parece: Python resuelve las distintas estructuras sintácticas y funcionalidades específicas llamando a determinados métodos en los objetos.

Entonces podemos obtener la misma interfaz que ofrece Python usando esos métodos, llamados métodos con nombres especiales. Son muchos, y en este texto mencionaremos sólo algunos, pero pueden profundizar en [su documentación](#).

Para mostrarlo en el ejemplo, cuando nosotros hacemos `rack[1]` Python ejecuta un método especial (indicándole que buscamos el ítem en la posición 1), y similar cuando hacemos `len(rack)`.

Veamos la definición de la clase con la integración que buscamos:

CELDA 35

```
class Rack:

    def __init__(self, *components):
        self.components = components

    def __len__(self):
        return len(self.components)

    def __getitem__(self, idx):
        return self.components[idx]

    def power(self):
        return [c for c in self.components if c.startswith("power")]
```

Vemos que estos métodos especiales tienen un nombre... especial. Es por eso que hay una cierta ambivalencia con su denominación, a veces se los llama “métodos especiales”, otras “métodos de nombre especial”, e incluso “métodos mágicos” (pero no hay nada de magia en los mismos, es sólo ese protocolo de Python que antes mencionamos).



Todos estos métodos comienzan con un doble guión bajo y terminan también de esa manera. Por comodidad, se los terminó nombrando con la palabra “*dunder*” (pronunciada “dander”), por *double underscore*. Entonces es normal escuchar “dunder init” por `__init__`, “dunder len” por `__len__`, etc.

Nada evita que los usemos directamente, pero es claramente más elegante lo primero que lo segundo en el siguiente ejemplo:

CELDA 37

```
print("Posición 1:", rack[1])
print("Posición 1:", rack.__getitem__(1))
```

```
Posición 1: disks
Posición 1: disks
```

Más allá de la utilización de estos métodos cuando queremos que nuestros objetos se integren a la sintaxis de Python, también necesitamos a veces proveer funcionalidad más específica y eso

implica la utilización de más de uno de estos elementos.

Para visualizar mejor eso veamos un ejemplo donde tenemos una clase Empleado que queremos guardar luego como clave en un diccionario. Esto, a priori, lo podemos hacer:

CELDA 38

CELDA 39

Sin embargo es un poco oscuro qué elemento o valor usó el diccionario para poder guardarlo como clave... como el objeto en sí no indicaba nada, usa el identificador del objeto (lo que devuelve la función integrada `id` al aplicarla al objeto). Entonces, si luego construimos otro objeto Empleado en otra parte del código (algo que sucede todo el tiempo, por ejemplo cuando construimos los objetos al momento de necesitarlos con valores que traemos de una base de datos) encontramos que sorpresivamente parecería no estar en la estructura que definimos:

CELDA 40

Para lograr ese comportamiento nuestro objeto tiene que definir dos métodos especiales: el `__hash__` para cuando se hace hash del objeto y el `__eq__` para cuando se lo compara por igualdad (ambas acciones son realizadas por el diccionario para poder guardar el objeto como clave en su estructura interna).

Justamente ese es un punto que tenemos que pensar y analizar de nuestro objeto, ¿qué atributo queremos que sea una clave unívoca? Para el caso del ejemplo es sencillo: usaremos el DNI del empleado. Es trivial agregar entonces los dos métodos en cuestión:

CELDA 41

CELDA 42

```
e1 = Empleado("25442321", "Carla")
e2 = Empleado("32250945", "Juan")
aumentos = {e1: 70, e2: 30}

Empleado("25442321", "Carla") in aumentos
```

```
True
```

Por otro lado, si lo que queremos es que nuestro objeto se comporte completamente como uno de los tipos integrados de Python, tenemos que aprovechar el módulo `collections.abc` donde tenemos clases abstractas que funcionan como base de nuestros objetos y nos permiten, definiendo sólo algunos métodos, cumplir con todo el “protocolo estándar” de cada tipo.

Mostremos como ejemplo lo que podríamos denominar un “diccionario con claves en minúscula”. Para ello heredamos de `Mapping` (la clase abstracta para “mapas” de sólo lectura) y definimos los tres métodos mínimos necesarios:

CELDA 43

```
from collections.abc import Mapping

class Test(Mapping):

    def __init__(self, **data):
        self.data = {key.lower(): value for key, value in data.items()}

    def __getitem__(self, key):
        return self.data[key.lower()]

    def __len__(self):
        return len(self.data)

    def __iter__(self):
        return iter(self.data)
```

Debemos prestar atención en esa definición como siempre llevamos a minúscula cada clave antes de trabajar el diccionario interno que tenemos. Entonces, luego accedemos a nuestra estructura sin importar la forma de la clave:

CELDA 44

```
test = Test(Foo=3, bar=1)
print("Mayúscula:", test["F00"])
print("Un mix:   ", test["f0o"])
print("Minúscula:", test["foo"])

Mayúscula: 3
Un mix:    3
Minúscula: 3
```

Lo importante de haber heredado `Mapping` es que nuestro objeto se comporta “en general” como un diccionario de Python aunque no hayamos definido el resto de los métodos:

CELDA 45	
<pre>print("Largo:", len(test)) print("Claves:", list(test.keys()))</pre>	
<pre>Largo: 2 Claves: ['foo', 'bar']</pre>	

Como mencionamos arriba, esto sería para un diccionario de sólo lectura. Para el comportamiento completo debemos trabajar con `MutableMapping` y escribir nosotros un método más (el `__setitem__`), como indica [esta tabla resumen](#).

Recomendamos analicen [la documentación de este módulo](#) ya que es una buena forma de aprender también las distintas interfaces de los tipos de datos incluidos en Python.



7 | Entornos de ejecución de Python

El desarrollo de programas no termina cuando escribimos los mismos, sino que es responsabilidad del desarrollador el cómo se van a ejecutar esos programas. Escribir un programa que no puede ser ejecutado no tiene sentido.

[Código disponible](#)

Lo más común es que tengamos distintos entornos de ejecución de nuestros programas. El más natural es el de desarrollo mismo: empezamos a trabajar en nuestra máquina, tenemos a disposición las herramientas y bibliotecas instaladas en nuestro sistema y utilizamos eso.

Cuando el desarrollo es compartido, ya se nos presenta la primer dificultad: las distintas personas trabajando en el programa van a tener distintos entornos de ejecución. Luego, siempre hay nuevas situaciones con nuevos entornos: vamos a querer correr sistemas de integración continua¹ y esos sistemas tendrán otros entornos, etc. Y claro, finalmente queremos que nuestro programa sea realmente usado, y normalmente esto implica que se instale en otras máquinas, sean locales

¹ Una herramienta muy usada en el mundo del desarrollo de software para validar automáticamente la calidad de los programas, mediante pruebas que se ejecutan en distintos entornos (independientes del desarrollador) para cada cambio en esos programas.

o remotas.

En la descripción anterior podemos distinguir dos mundos distintos, en función del control que tenemos como desarrolladores. Por un lado están los entornos de los desarrolladores mismos y sistemas aledaños (como los de integración continua que mencionábamos); en estos normalmente tenemos el poder de definir qué y cómo se instala y está disponible para nuestro programa. Por otro lado están aquellos entornos donde tenemos poco o ningún control: el de los servidores o instalaciones en máquinas particulares, en función del tipo de programa que estamos haciendo. En este capítulo veremos distintas formas de proveer entornos controlados de ejecución pensando en el primer grupo (entornos controlados durante el desarrollo). Aunque algunas de las técnicas y herramientas que contaremos se pueden utilizar o nos pueden facilitar el despliegue de nuestros sistemas en servidores o la distribución e instalación en máquinas clientes, no es el foco de lo que expondremos.

7.1. Por qué tener distintos entornos

Hay dos grandes características que queremos obtener cuando estamos pensando en entornos de ejecución.

La primera funcionalidad es que nuestro programa se ejecute de la misma manera por un lado en nuestro entorno de desarrollo y por el otro en los entornos de otros desarrolladores y en otros sistemas donde corran las pruebas de unidad o integración, e incluso cuando lo llevamos a los entornos finales. Para que tenga el mismo comportamiento en los distintos entornos necesitamos básicamente que los entornos sean iguales, al menos en lo que refiere al contexto de nuestro programa (por ejemplo, la versión de Python utilizada y las versiones de las bibliotecas).

Esta característica que estamos buscando es “repetibilidad”. Si en las distintas computadoras usadas para desarrollar el programa se comporta de distinta manera tendremos muchos dolores de cabeza (y tiempo y esfuerzo perdidos). Entonces, tenemos que proveer entornos lo suficientemente similares para que el programa se ejecute con el mismo comportamiento.

Otra situación con la que nos encontramos es que para un determinado proyecto necesitamos una biblioteca en una versión específica pero nuestro sistema tiene una versión distinta. Si fuese más vieja, podríamos actualizar nuestro sistema, pero eso podría traernos algunos comportamientos inesperados en el mismo. Si fuese más nueva, llevar nuestro sistema a versiones anteriores podría ser incluso mas problemático. Y esto incluso podría no ser una opción, si es que para distintos proyectos necesitamos la misma biblioteca en versiones distintas, ya que no podemos estar actualizando y desactualizando nuestro sistema operativo cada vez que ejecutamos uno u otro programa.

En este caso lo que estamos buscando es “aislamiento”: que las bibliotecas que usa nuestro programa sean distintas que las que tenga el sistema (en verdad, sin importarnos qué versiones tiene el sistema, o incluso si tiene esas bibliotecas). De esta manera cada proyecto indicará cuales bibliotecas y en qué versiones las necesita, y al estar aislado del sistema base tendremos el mismo comportamiento más allá de cual sea ese sistema.

Entonces, la idea de tener distintos entornos de ejecución es, a priori, lograr estas dos grandes ventajas, repetibilidad y aislamiento. A continuación veremos dos formas de lograr estos entornos, con mayor o menor cumplimiento de estas dos ventajas y más características, que iremos explicando en cada caso.

Para hacer los ejemplos más prácticos, ejecutaremos un programa real (el `oscilador.py` que armamos en el capítulo de Ecuaciones Diferenciales Ordinarias 20) utilizando cada una de las herramientas mostradas.

7.2. Entornos virtuales

Un “entorno virtual” (que muchas veces denominamos directamente usando su nombre en inglés, *virtual environment*, o incluso acortamos con *virtualenv*) es la solución de *virtualización* de entornos nativa a Python, y por ende es la más popular. Hay muchas herramientas para trabajar con este concepto, algunas incluso en la biblioteca estándar; veremos las principales más adelante en la subsección de Herramientas 7.2.2.

El uso de un entorno virtual es la forma más sencilla y liviana de conseguir las características que presentamos en las secciones anteriores, aunque sólo nos las ofrece para las bibliotecas que utilizaremos (sólo trabaja a nivel de dependencias de Python). Estamos atados al Python instalado en el sistema para cada entorno, y tendremos que recrear ese entorno en todos aquellos sistemas donde lo necesitemos (no es trasladable).

La funcionalidad que nos ofrece es suficiente en la gran mayoría de los casos, más allá de las limitaciones indicadas, por lo que es normalmente la solución elegida sin tener que pasar a otras más complejas como la que veremos más adelante.

7.2.1. En detalle

Para mostrar la creación y uso de los entornos virtuales vamos a utilizar la herramienta integrada en la biblioteca estándar, el módulo `venv`. Lo llamamos directamente (usando la opción `-m` de Python para ejecutar un módulo como si fuese un script), indicándole el nombre del directorio donde queremos que nos cree toda la estructura.

```
[/temp]$ python3 -m venv mivenv
[/temp]$ ls -l mivenv
total 20
drwxrwxr-x 2 facundo facundo 4096 jun  8 16:02 bin
drwxrwxr-x 2 facundo facundo 4096 jun  8 16:02 include
drwxrwxr-x 3 facundo facundo 4096 jun  8 16:02 lib
lrwxrwxrwx 1 facundo facundo   3 jun  8 16:02 lib64 -> lib
-rw-rw-r-- 1 facundo facundo   69 jun  8 16:03 pyvenv.cfg
drwxrwxr-x 3 facundo facundo 4096 jun  8 16:02 share
```

Tengamos en cuenta que esta estructura puede variar según el sistema donde estemos creando el entorno, porque justamente sigue la estructura de ese sistema. Si exploramos el contenido de este directorio veremos que tiene algunos ejecutables y bibliotecas preinstaladas (notoriamente Python mismo y `pip`, la herramienta para instalar otras bibliotecas). Un detalle importante con este directorio es que no podemos moverlo a otro lugar, ya que depende de su posición al ser creado. Pero no deja de ser un directorio normal, así que si queremos “limpiar” el entorno, borrando ese directorio eliminará todo vestigio del mismo.

Cuando ejecutemos el Python de ese directorio, el mismo tendrá acceso a la biblioteca estándar del Python utilizado más las bibliotecas que instalemos en ese entorno virtual, y no podrá

acceder a las del sistema. A nivel de sistema nada se modifica cuando instalamos bibliotecas en el entorno virtual, con lo que no hay riesgo de desestabilizarlo, y un buen efecto secundario al usar los entornos virtuales es que no dependemos de tener permisos especiales de *root* o “administrador” para instalar las bibliotecas que necesitamos.

Habiendo dicho eso, hay una manera de crear el entorno virtual rompiendo un poco la aislación natural que nos provee. Si usamos el parámetro `--system-site-packages` al momento de creación, Python buscará en el sistema las bibliotecas que no encuentre en el entorno. Esto es raramente utilizado, pero muy necesario en caso de bibliotecas muy difíciles de instalar (pensemos el caso donde el proceso de instalar una biblioteca en particular lleva mucho tiempo, y no queremos pagar ese costo en cada entorno virtual que generemos). Hay otras opciones más para explorar, les recomendamos profundizar en [la documentación del módulo](#), o pedir de forma rápida la ayuda con `python3 -m venv --help`.

La forma habitual de usar los entornos virtuales es activándolos primero, lo que se hace de forma muy parecida (pero no igual) en los distintos sistemas.

Por ejemplo, en Linux y los sistemas *UNIX-like*:

```
[/temp]$ source mivenv/bin/activate  
[(mivenv) /temp]$
```

O en Windows:

```
[C:]>mivenv\Scripts\activate.bat  
[(mivenv) C:]>
```

Notar como que en cada caso, luego de activar el entorno, el *prompt* cambió para reflejar eso mismo.

Una vez dentro del entorno podremos proceder a instalar las bibliotecas que deseemos, usando la herramienta `pip`.

`pip` es el administrador de paquetes estándar de Python para manejar todas las bibliotecas (por fuera de la biblioteca estándar, claro), y no sólo instalará la biblioteca que le indiquemos, sino que automáticamente instalará sus dependencias (de ser necesario, ya que podrían estar instaladas de antes).

Aunque podemos indicar distintos repositorios o incluso indicarle que la fuente sea local, el procedimiento natural es que `pip` descargue e instale las bibliotecas desde el “Índice de Paquetes de Python”, más bien conocido como PyPI (por su sigla en inglés, *Python Package Index*).

En el siguiente ejemplo mostramos cómo instalar una biblioteca que usamos extensivamente en todo el libro:

```
[(mivenv) /temp]$ pip install numpy  
Collecting numpy  
...  
Successfully installed numpy-1.20.3
```

Una forma de validar que la biblioteca está instalada en el `virtualenv` es a través del *path* del módulo:


```
[(mivenv) /temp]$ python
Python 3.12.3 (main, Feb  4 2025, 14:48:35)
>>> import numpy
>>> numpy.__file__
'/temp/mivenv/lib/python3.12/site-packages/numpy/__init__.py'
```

Decíamos que una de las ventajas de la aislación del entorno virtual era poder tener la misma biblioteca con distintas versiones en distintos entornos. Para ello, vamos a necesitar que pip no instale la última versión disponible (como sucedió recién con NumPy), sino poder especificar puntualmente cual necesitamos.

Por ejemplo, instalemos otra biblioteca usada mucho en el libro, Matplotlib, pero no la última versión (que al momento de escribir estas líneas es la 3.9.2), sino una particular que podemos necesitar por algún motivo específico:

```
[(mivenv) /temp]$ pip install matplotlib==3.3.4
Collecting matplotlib==3.3.4
...
Successfully installed cyciler-0.10.0 kiwisolver-1.3.1 matplotlib-3.3.4 pillow-8.2.0 pyparsing-2.4.7
↪ python-dateutil-2.8.1 six-1.16.0
```

Vemos como pip no sólo nos instaló el paquete que habíamos especificado, sino también las dependencias necesarias de forma automática. Cabe mencionar también que hay que tener cuidado cuando restringimos qué versión pip instalará, ya que como podemos utilizar los comparadores `>`, `<`, `>=`, `<=` para indicar esas restricciones, podemos meternos en problemas porque esos caracteres tienen un significado especial en la línea de comandos; la solución es sencilla, con utilizar comillas es suficiente (por ejemplo `pip install "matplotlib>3"`).

Otra funcionalidad muy utilizada de pip es la de devolvernos todos los paquetes que están instalados en el entorno virtual, lo cual obviamente no solo incluye los que hayamos especificado al momento de la instalación, sino también las dependencias que se hayan instalado correspondientemente.

Vemos eso mismo en el siguiente ejemplo:

```
[(mivenv) /temp]$ pip freeze
cyciler==0.10.0
kiwisolver==1.3.1
matplotlib==3.3.4
numpy==1.20.3
Pillow==8.2.0
pyparsing==2.4.7
python-dateutil==2.8.1
six==1.16.0
```

Es normal en un proyecto tener anotadas todas las dependencias en un archivo, normalmente llamado `requirements.txt`, de manera de indicarle a pip que instale todas ellas en el entorno virtual usando el parámetro `--requirement`.

Recomandos revisar [la documentación de referencia de pip](#) para explorar otras funcionalidades que provee.

Finalmente, para salir del entorno debemos ejecutar el comando `deactivate` (que tenemos a nuestro alcance justamente porque estamos dentro del entorno):

```
[(mivenv) /temp]$ deactivate
[/temp]$
```

Realicemos todo este mismo procedimiento para ejecutar un programa ejemplo: el `oscilador.py` que mencionamos arriba, que podemos descargar [desde el repositorio del libro](#).

Primero entonces creemos el entorno virtual, activémoslo e instalemos las dependencias necesarias.

```
[/temp]$ python3 -m venv oscilador-venv
[/temp]$ source oscilador-venv/bin/activate
[(oscilador-venv) /temp]$ pip install matplotlib numpy scipy PyQt5
Collecting matplotlib
Downloading matplotlib-3.4.3-cp38-cp38-manylinux1_x86_64.whl (10.3 MB)
|████████████████████| 10.3 MB 2.4 MB/s
Collecting numpy
Downloading numpy-1.21.2-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)
|████████████████████| 15.8 MB 2.0 MB/s
Collecting scipy
Downloading scipy-1.7.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (28.4 MB)
|████████████████████| 28.4 MB 2.0 MB/s
Collecting PyQt5
Using cached PyQt5-5.15.4-cp36.cp37.cp38-cp39-abi3-manylinux2014_x86_64.whl (8.3 MB)
Collecting python-dateutil>=2.7
Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
Collecting pillow>=6.2.0
Using cached Pillow-8.3.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (3.0 MB)
Collecting cycler>=0.10
Using cached cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)
Collecting pyparsing>=2.2.1
Using cached pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)
Collecting kiwisolver>=1.0.1
Using cached kiwisolver-1.3.1-cp38-cp38-manylinux1_x86_64.whl (1.2 MB)
Collecting PyQt5-Qt5>=5.15
Using cached PyQt5_Qt5-5.15.2-py3-none-manylinux2014_x86_64.whl (59.9 MB)
Collecting PyQt5-sip<13,>=12.8
Using cached PyQt5_sip-12.9.0-cp38-cp38-manylinux1_x86_64.whl (332 kB)
Collecting six>=1.5
Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, python-dateutil, numpy, pillow, cycler, pyparsing, kiwisolver,
→ matplotlib, scipy, PyQt5-Qt5, PyQt5-sip, PyQt5
Successfully installed PyQt5-5.15.4 PyQt5-Qt5-5.15.2 PyQt5-sip-12.9.0 cycler-0.10.0
→ kiwisolver-1.3.1 matplotlib-3.4.3 numpy-1.21.2 pillow-8.3.1 pyparsing-2.4.7
→ python-dateutil-2.8.2 scipy-1.7.1 six-1.16.0
```

A veces es complicado saber cuales son las dependencias que necesita un programa. Con los proyectos más grandes, como mencionábamos antes, se acostumbra tener un archivo comúnmente llamado `requirements.txt`, pero muchas veces no lo encontramos cuando estamos lidiando con un script “suelto”. En el caso del ejemplo, una simple inspección del archivo nos indica que necesitamos `matplotlib`, `numpy` y `scipy`, pero en realidad también necesitamos `PyQt5` para que `matplotlib` pueda abrir una ventana con el resultado.

Todo lo que nos queda es ejecutar el archivo utilizando directamente python3, ya que como tenemos activado el entorno virtual estaremos ejecutando el script ahí dentro:

```
[(oscilador-venv) /temp]$ python3 oscilador.py
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
[(oscilador-venv) /temp]$ deactivate
[/temp]$
```

7.2.2. Herramientas

Hay muchas herramientas para trabajar con entornos virtuales, ya que es la metodología más establecida (e incluso soportada en la biblioteca estándar) para lograr los dos objetivos discutidos arriba.

En esta subsección no pretendemos cubrir todas las herramientas existentes, sino una selección de las mismas, algunas muy utilizadas, otras con funcionalidades específicas, con la idea de mostrar en qué casos conviene utilizar alguna u otra. Por supuesto, no hablaremos del módulo venv, ya que utilizamos este mismo arriba cuando mostramos genéricamente qué eran los entornos virtuales.

7.2.2.1. virtualenv

Esta herramienta fue durante mucho tiempo la única forma de crear entornos virtuales. Hoy en día, con parte de su funcionalidad integrada en la biblioteca estándar, deja de ser imprescindible.

Hay algunas diferencias a favor y en contra con respecto al módulo integrado venv, que pueden ser más o menos importantes dependiendo de nuestro contexto, aquí mencionamos las principales. Entonces, virtualenv...

- es varias veces más rápido en la creación del entorno: esto quizás no es importante cuando lo utilizamos manualmente, pero puede ser un factor si estamos generando muchos entornos automáticamente en algún sistema
- necesita instalarse por separado a Python, previamente a ser utilizado
- nos puede llegar a confundir con qué versión de Python está creando el entorno (en cambio cuando usamos el módulo venv nosotros explicitamos qué Python usamos)
- soporta versiones de Python en las que el módulo venv todavía no existía (aunque dicho módulo está en la biblioteca estándar desde Python 3.3)

Toda su [documentación](#).

7.2.2.2. virtualenvwrapper

Esta es una de las primeras herramientas que aparecieron alrededor de `virtualenv` y le agrega una capa de funcionalidad arriba, orientada a el manejo de los entornos virtuales como "directorios".

Por ejemplo, cuando armamos un entorno virtual nuevo, en vez de crearlo en el directorio donde estamos parados, lo creará en un lugar central (configurable con la variable de entorno `WORKON_HOME`), y permitirá listarlos, duplicarlos, y especialmente utilizarlos recordando solamente el nombre.

Entonces, cuando queremos activar un entorno virtual, en lugar de tener que recordar cómo se llamaba y dónde lo habíamos creado, sólo con tener el nombre alcanza.

La instalación no es trivial y lleva cierta configuración antes de que funcione, pero si la funcionalidad buscada es esta centralización de los entornos, se puede explorar más en [su documentación](#).

7.2.2.3. fades

Hasta ahora vimos el manejo de entornos virtuales como un proceso con beneficios, pero también con un costo de administración: debemos recordar cómo se llaman los `virtualenvs` que creamos, y (a menos que usemos `virtualenvwrapper`) donde los tenemos. Esto no es mayor problema si usamos el entorno para desarrollo de un proyecto: normalmente el lugar es en ese proyecto y el directorio tendrá un nombre conocido como `env` o `venv`. Así y todo, debemos recordar de activarlo antes de empezar a trabajar con el proyecto, y desactivarlo luego.

Ese costo de administración tiene sentido en un proyecto, pero se vuelve inmanejable cuando necesitamos entornos virtuales para simple scripts o programitas utilitarios que tengamos en nuestro sistema.

Por ejemplo, tenemos un pequeño script que nos muestra la metadata de un mp3, algo simple como lo siguiente:

```
1 #!/usr/bin/env python3
2
3 import sys
4
5 import tinytag
6
7 tag = tinytag.TinyTag.get(sys.argv[1])
8 print("{!r} por {}".format(tag.title, tag.artist))
```

Para que este script funcione, tendríamos que crear un `virtualenv` en algún lado, e instalar `tinytag` allí. Esto no es problema. La dificultad aparece cuando queremos volver a usar este script unas semanas o meses después. ¿Dónde teníamos el `virtualenv`? ¿Cómo se llamaba? O incluso, ¿este script necesitaba un `virtualenv` o lo podía ejecutar y ya?

Fades viene a solucionar este problema, justamente, ya que nos permite usar todo el poder de los entornos virtuales sin tener que preocuparnos por ellos. Fades creará automáticamente un nuevo entorno virtual e instalará las dependencias necesarias (o reusará uno creado previamente), y ejecutará el script dentro de ese entorno.

Modifiquemos levemente el script para usar esta nueva herramienta:

```
1 #!/usr/bin/env fades
2
3 import sys
4
5 import tinytag # fades
6
7 tag = tinytag.TinyTag.get(sys.argv[1])
8 print("{!r} por {}".format(tag.title, tag.artist))
```

Dos cambios. El primero es que indicamos que el intérprete del script es Fades, y el segundo es que pusimos un comentario al importar tinytag, de manera que Fades reconozca que esa es la biblioteca que tiene que proveer dentro de un entorno virtual.

Entonces, la primera vez que ejecutamos el script Fades armará el entorno virtual necesario y ejecutará el script allí. Y nos podemos olvidar. Volvemos al script luego de un tiempo, y sólo tenemos que ejecutarlo, no hace falta recordar el nombre o la ubicación del entorno virtual, ni siquiera si necesitábamos uno. Y si llevamos este script a una máquina nueva, automáticamente Fades se encargará de proveer el entorno necesario nuevamente.

Otro caso de uso muy común de Fades es probar bibliotecas en el intérprete interactivo. Para ello utilizamos Fades desde la línea de comando y especificamos la(s) dependencia(s) necesaria(s), pero no un script a ejecutar, entonces Fades creará el entorno virtual correspondiente y ejecutará el intérprete interactivo allí, con lo cual tendremos acceso a esas bibliotecas:

```
[/temp]$ fades -d tinytag --autoimport
Python 3.8.6 (default, May 27 2021, 13:28:02)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
::fades:: automatically imported 'tinytag'
>>> tinytag.__file__
'/home/facundo/.local/share/fades/.../lib/python3.8/site-packages/tinytag/__init__.py'
```

En el caso mostrado, Fades ya tenía el entorno virtual con tinytag, y lo reutilizó, pero si ejecutamos esa misma orden en una computadora diferente Fades nos informará que está creando el entorno e instalando esa dependencia.

Usemos Fades para ejecutar nuestro programa de ejemplo:

```
[/temp]$ fades -d matplotlib -d numpy -d scipy -d PyQt5 oscilador.py
*** fades *** 2021-08-21 10:41:37,143 INFO Hi! This is fades 9.0.1, automatically
↳ managing your dependencies
*** fades *** 2021-08-21 10:41:37,143 INFO Checking the availability of dependencies in
↳ PyPI. You can use '--no-precheck-availability' to avoid it.
*** fades *** 2021-08-21 10:41:41,115 INFO Installing dependency: 'PyQt5'
*** fades *** 2021-08-21 10:41:44,816 INFO Installing dependency: 'numpy'
*** fades *** 2021-08-21 10:41:47,650 INFO Installing dependency: 'scipy'
*** fades *** 2021-08-21 10:41:51,216 INFO Installing dependency: 'matplotlib'
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
[/temp]$
```

Como puede ser un incordio escribir todas las dependencias en la línea de comandos, es natural tener un archivo `requirements.txt` con las dependencias especificadas allí (el nombre de cada paquete en cada línea es suficiente si queremos la última versión disponible en cada caso, pero también podemos especificar qué versión queremos):

```
[/temp]$ cat requirements.txt
PyQt5
matplotlib
numpy
scipy

[/temp]$ fades -r requirements.txt oscilador.py
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
[/temp]$
```

Presten atención como en este segundo caso no tenemos mensajes de Fades instalando dependencias, porque al ser exactamente las mismas que antes, Fades ya encuentra un entorno virtual con esas mismas, entonces lo reutiliza.

Más allá de eso, reconozcamos que también es un incordio tener las dependencias en un archivo externo, ya que al ser nuestro ejemplo un script suelto que podemos tener en cualquier lado, la obligación de recordar cual es su archivo de dependencias también es molesto.

Volvamos a hacer entonces lo mismo que con el script sencillo del principio: especifiquemos las dependencias en el mismo script. Veamos las primeras líneas de nuestro `oscilador.py`, levemente modificado:

```
1 #!/usr/bin/env fades
2
3 """Programa que integra el sistema masa-resorte-banda elástica.
4
5 fades:
6     PyQt5 # utilizado internamente por matplotlib para abrir una ventana con el resultado
7 """
8
9 import argparse
10
11 import matplotlib.pyplot as plt # fades
12 import numpy as np # fades
13 from scipy.integrate import solve_ivp # fades
```

Allí podemos ver en la línea 1 que ahora no estamos ejecutando el script con Python sino con Fades (que llamará a Python luego de disponer el entorno virtual), y en las líneas 11 a 13 que le estamos diciendo a Fades que necesitamos las dependencias para esos imports. El caso de PyQt5 es distinto, porque no tenemos un import (lo utiliza matplotlib, no nosotros directamente), entonces le indicamos esa dependencia en el docstring del módulo, en la línea 6.

Una vez modificado el script de esta manera, ya no tenemos que recordar nada sobre las dependencias del script, archivos adjuntos, entrar a directorios en particular, o activar y desactivar entornos. Directamente ejecutamos el script:

```
[/temp]$ ./oscilador.py
Opciones del script: Namespace(archivo=None, usar_tex=False)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

Fades tiene muchas otras opciones (algunas útiles también para utilizarlo en proyectos grandes, no sólo scripts), recomendamos explorarlas con `fades --help`, revisar [la documentación](#) o mirar [algunas animaciones](#) que muestran las funcionalidades más comunes.

7.2.2.4. uv

uv es la más moderna de estas herramientas, focalizada en la velocidad de ejecución y en la intención de reemplazar todo el espectro de herramientas que de un modo u otro manejan entornos virtuales.

Por ejemplo, para crear un entorno virtual e instalar dependencias:

```
$ uv venv xcss
Using CPython 3.12.3 interpreter at: /usr/bin/python3
Creating virtual environment at: xcss
Activate with: source xcss/bin/activate
$ source xcss/bin/activate
$ (xcss) $ uv pip install requests
Using Python 3.12.3 environment at: xcss
Resolved 5 packages in 50ms
Installed 5 packages in 7ms
+ certifi==2025.4.26
+ charset-normalizer==3.4.2
+ idna==3.10
+ requests==2.32.3
+ urllib3==2.4.0
```

Es extremadamente rápido. Cabe acotar, sin embargo, que a veces no instala las mismas versiones que instalaría pip, ya que algunas opciones de instalación tiene defaults distintos de la herramienta original.

Es muy práctico porque tiene integrada la posibilidad de usar otro Python que no sea el del sistema:

```
$ uv venv xcs7 --python 3.7
Using CPython 3.7.9
Creating virtual environment at: xcs7
Activate with: source xcs7/bin/activate
```

En estos casos uv descargará el binario de Python que sea necesario y lo utilizará para armar el entorno. Hay que tener la precaución que uv utiliza binarios de Python pre-armadas por la empresa responsable de la herramienta (Astral), los cuales tienen algunas diferencias de comportamiento con el Python oficial, generalmente como una consecuencia de la portabilidad buscada.

Una de las grandes ventajas de uv es que soporta la [PEP 723](#), una formalización/generalización de la idea de Fades de declarar dentro del script qué dependencias deben instalarse para su funcionamiento.

Veamos el mismo ejemplo de antes, el oscilador, pero con esta metadata. Las primeas líneas entonces son:

```
1 #!/usr/bin/env python3
2 """Programa que integra el sistema masa-resorte-banda elástica."""
3
4 # /// script
5 # dependencies = [
6 #     "matplotlib",
7 #     "numpy",
8 #     "scipy",
9 # ]
10 # ///
11
12 import argparse
13
14 import matplotlib.pyplot as plt
15 import numpy as np
16 from scipy.integrate import solve_ivp
```

Entonces, luego:

```
$ uv run oscilador_con_pep723.py
Reading inline script metadata from `oscilador_con_pep723.py`
Installed 12 packages in 24ms
Opciones del script: Namespace(usar_tex=False, archivo=None)
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

Recomendamos revisar [la documentación correspondiente](#) para más opciones.

7.2.2.5. pipenv

Pipenv es una herramienta que apunta al manejo automático de los entornos virtuales más a nivel de proyectos. Con sólo indicarle que instale las dependencias del proyecto, creará automáticamente (de no tenerlo de antes) un entorno virtual para el proyecto en una ubicación específica, e instalará las dependencias allí.

```
[/temp/proyecto]$ pipenv install -r requirements.txt
Creating a virtualenv for this project...
Using /usr/bin/python3 (3.8.6) to create virtualenv...
[]created virtual environment CPython3.8.6.final.0-64 in 104ms
creator CPython3Posix(dest=/home/facundo/.local/share/virtualenvs/trunk-qJfLQWax, clear=False,
↪ global=False)
seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy,
↪ app_data_dir=/home/facundo/.local/share/virtualenv)
added seed packages: pip==20.1.1, pkg_resources==0.0.0, setuptools==44.0.0, wheel==0.34.2
activators
↪ BashActivator,CShellActivator,FishActivator,PowerShellActivator,PythonActivator,XonshActivator
Virtualenv location: /home/facundo/.local/share/virtualenvs/trunk-qJfLQWax
Creating a Pipfile for this project...
Requirements file provided! Importing into Pipfile...
```



```
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (00d0fa)!
Installing dependencies from Pipfile.lock (00d0fa)...
█ 4/4 — 00:00:04
To activate this project's virtualenv, run the following:
$ pipenv shell
```

Tengamos en cuenta que necesitamos recordar de activar el entorno virtual antes de trabajar con nuestro proyecto:

```
[/temp/proyecto]$ pipenv shell
Spawning environment shell (/bin/bash). Use 'exit' to leave.
. /home/facundo/.local/share/virtualenvs/trunk-qJfLQWax/bin/activate
/temp/proyecto$ . /home/facundo/.local/share/virtualenvs/trunk-qJfLQWax/bin/activate
((trunk-cLuRPqaE) /temp/proyecto)$
```

Vemos en la última línea como estamos “dentro” del entorno virtual, como veíamos al principio con otras herramientas.

Luego de ejecutar el `install` encontramos que Pipenv nos dejó dos archivos nuevos: el `Pipfile` y el `Pipfile.lock`. Estos son archivos de control que usa Pipenv mismo para proveer la funcionalidad de entornos virtuales armados determinísticamente en distintos sistemas de trabajo. La idea es que en desarrollo especificamos las dependencias necesarias en un `requirements.txt` o similar, y Pipenv armará sus dos archivos de control; luego cuando enviemos el proyecto a otros sistemas (integración continua, servidores de producción, etc.) los entornos virtuales que se armarán en esos casos serán *exactamente iguales* que los que armamos en desarrollo, evitando posibles inconvenientes (como por ejemplo que una dependencia cambió de versión entre que probamos el proyecto localmente y que lo subimos a un servidor).

Finalmente, también para esta herramienta, recomendamos revisar [la documentación correspondiente](#).

7.3. Contenedores

Un “contenedor” es una forma de empaquetar software: el código que necesitamos para ejecutar una aplicación más todas las dependencias que necesite, incluidas las que provee el sistema operativo en la que se basa el contenedor.

Estos contenedores nos proveen el aislamiento y la repetibilidad que buscamos, pero no sólo a nivel de código Python, sino a nivel de sistema, ya que está bien separado lo que se ejecuta adentro del contenedor, del sistema donde ese contenedor corre (al que llamamos “*host*” o “anfitrión”). Podemos ejecutar muchos contenedores simultáneamente en el mismo anfitrión, ya que cada uno tiene todo lo que se necesita para ejecutar una determinada aplicación. Más aún, podemos compartir fácilmente estos contenedores, de manera que otras personas pueden ejecutar el mismo contenedor en otros anfitriones, con idénticos resultados.

Hay un estándar para la creación y manejo de contenedores, la [Open Container Initiative](#), pero hay que reconocer que la herramienta pionera en este concepto es [Docker](#) (y sobre la cual

está fuertemente basada el estándar). Docker es un administrador de “contenedores”, entonces, y es la herramienta que vamos a utilizar en este texto. Tengan en cuenta que lo más probable es que Docker todavía no esté en vuestro sistema, pueden instalarlo siguiendo [las instrucciones oficiales](#).

De manera similar a lo que vimos antes con los entornos virtuales, vamos a explorar cómo armar contenedores con las dependencias que necesitamos, y cómo ejecutar una aplicación dentro de este nuevo entorno. Además, vamos a ver cómo compartir estos entornos.

Una palabra sobre Kubernetes: hay muchos textos que mezclan un poco Docker y Kubernetes. Sin entrar en mayor detalles, expliquemos un poco qué es Kubernetes, para evitar la confusión. Mientras que Docker hace foco en la creación y distribución de contenedores, Kubernetes trabaja el concepto de organizar muchos contenedores para el despliegue de un servicio complejo en algún servidor (por ejemplo, mientras que un contenedor puede incluir a una base de datos, Kubernetes organizará ese contenedor, más otros con servidores web, y otros con sistemas de autenticación, para tener un servicio web completo funcionando). Kubernetes se escapa del ámbito de este libro, donde la idea es poder compartir aplicaciones, y no desplegar servicios en la nube para consumo de terceros. Pero si les interesa explorar este mundo, pueden empezar por [la página oficial](#).

7.3.1. ¿Imágenes o contenedores?

Hasta ahora estuvimos hablando de contenedores de forma genérica, porque privilegiamos hablar sobre las ventajas de los mismos (especialmente en el contexto de este capítulo) y no mezclar el tema de las “imágenes”.

Pero entremos en este detalle ahora, antes de ir a la parte más práctica, para separar bien los conceptos y poder entender mejor las próximas subsecciones. Empecemos por las imágenes, ya que sin imágenes no vamos a poder tener contenedores.

Una imagen es un archivo binario en disco, algo estático, que contiene todo lo necesario para poder ejecutar nuestra aplicación cuando corramos el contenedor.

No entraremos mucho en el detalle de la composición interna de las imágenes, pero sí es útil saber que están compuestas por “capas” (que incluso pueden reutilizarse entre imágenes), ya que es un concepto importante para entender distintos comportamientos a la hora de construir o descargarse imágenes.

Podemos crear las imágenes desde cero (como veremos abajo, con los archivos *Dockerfile*) o pueden ser una “foto” de un contenedor que está corriendo.

Los contenedores, por otro lado, son la “ejecución” de esas imágenes. A partir de una imagen podemos crear muchos contenedores similares (no digo idénticos, porque cada ejecución es independiente).

Por ejemplo, podemos tener nuestra aplicación en una imagen de Docker, junto con todo lo necesario para que esa aplicación corra como corresponde. Luego, podemos crear 10 contenedores distintos a partir de esa imagen, y ejecutar nuestra aplicación con 10 conjuntos distintos de datos. Cada contenedor será independiente del otro, más allá que hayan sido creados a partir de la misma imagen.

7.3.2. Creando imágenes

La forma primera de crear una imagen es a partir de un documento de texto (llamado *Dockerfile*) que contiene todas las órdenes que un usuario podría ejecutar en una línea de comandos.

En este archivo especificaremos el “sistema base” sobre el cual trabajaremos, y luego distintas órdenes que irán construyendo la imagen (instalando dependencias, ajustando configuraciones, etc.).

El Dockerfile tiene su complejidad, y hay órdenes de todo tipo (pueden revisar [su referencia](#)), pero las más usadas son las siguientes, que mencionaremos brevemente para poder continuar con el resto del contenido, obviamente sin entrar en todo el detalle o potencial de cada una:

- FROM: define la imagen base para el resto de las instrucciones (el Dockerfile *debe* comenzar con esta orden); la imagen indicada puede ser cualquiera, local o remota, al principio es muy útil usar alguna de un repositorio público (como [Dockerhub](#)).
- RUN: ejecuta un comando de la misma manera que si lo estuviéramos ejecutando en el sistema que estamos armando.
- ENTRYPOINT: el comando que ejecutará Docker cuando le hagamos run a la imagen, luego de crear el contenedor en sí; la forma preferida de escribir el comando aquí es usando una lista de cadenas (como `["python3", "script.py"]`) y no directamente `python3 script.py`, ya que esto último lo interpreta el shell del contenedor y no siempre sucederá lo que esperamos. Si queremos pasarle parámetros y opciones al script que estamos ejecutando podemos ponerlo como parte del ENTRYPOINT, pero se recomienda usar otra orden para ello que es CMD ya que de esta manera podremos luego modificar esos parámetros y opciones al momento de ejecutar el contenedor.
- ENV: configura variables de entorno para todas las instrucciones que se ejecutarán a continuación.
- COPY: copia archivos y directorios desde nuestro sistema al sistema de archivos de la imagen en el destino especificado.
- WORKDIR: cambia el directorio actual para todas las órdenes que vienen a continuación.

El Dockerfile, entonces, es básicamente una secuencia de órdenes, cada una en una línea. Cada orden terminará armando una de las capas que mencionábamos antes, lo cual es importante ya que una vez creada la capa Docker la puede reutilizar en el futuro cuando rearmemos la imagen.

Docker utilizará este archivo y nos dejará como resultado una imagen según lo especificado. Veamos un ejemplo con el mismo programa que utilizamos cuando aprendimos entornos virtuales (aunque lo ejecutaremos distinto: en lugar de que nos abra una ventana con el resultado, haremos que grabe esa imagen resultado a disco ²).

Nuestro Dockerfile, para el ejemplo, será el siguiente:

² Esto es porque para abrir una ventana el proceso de adentro del container se tiene que comunicar con el servidor de ventanas del host, lo cual no sólo no es sencillo, sino que depende del sistema operativo de nuestro host, y esta configuración avanzada ya se escapa del alcance del libro.

```

1 # arrancamos con un sistema base mínimo, sólo Python 3.9
2 FROM python:3.9
3
4 # instalamos las dependencias que necesitamos directamente desde PyPI
5 RUN pip install matplotlib numpy scipy
6
7 # indicamos que vamos a trabajar en este directorio (se crea automáticamente)
8 WORKDIR /oscilador/
9
10 # descargamos el script de prueba desde el github del libro mismo
11 RUN wget https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/código/entornos/oscilador.py
12
13 # indicamos que el punto de entrada para cuando corramos el contenedor es el
14 # script, con los parámetros por default que indican que guarde la imagen
15 # resultante como un PDF
16 CMD ["--archivo=imagen.pdf"]
17 ENTRYPOINT ["python3", "oscilador.py"]

```

El siguiente paso es construir nuestra imagen. Para ello creamos un directorio nuevo y ponemos una copia de este Dockerfile, ya que a docker build le tenemos que pasar un directorio donde exista un Dockerfile (en nuestro caso será vacío, porque es un ejemplo y el script que vamos a probar lo bajamos directamente de la web, pero en la mayoría de las situaciones reales el directorio alojará a todo el proyecto que intentamos encapsular).

Ejecutamos el build, entonces, indicando el directorio actual y usando -t para indicar el nombre de la imagen creada:

```

$ docker build -t app-ejemplo .
Sending build context to Docker daemon  2.56kB
Step 1/5 : FROM python:3.9
...
Successfully built 48b2a52afd8d
Successfully tagged app-ejemplo:latest

```

Podemos ver como Docker tiene nuestra imagen recién creada:

```

$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
app-ejemplo	latest	48b2a52afd8d	About a minute ago	1.16GB

7.3.3. Ejecutando contenedores

Para probar que lo que hicimos hasta ahora funciona correctamente, ejecutaremos un contenedor a partir de la imagen recién creada.

```
$ docker run app-ejemplo
Opciones del script: Namespace(usar_tex=False, archivo='imagen.pdf')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

Docker entonces creó el contenedor a partir de nuestra imagen, y ejecutó el script tal como indicamos en el Dockerfile con ENTRYPOINT. Cuando nuestro script terminó, automáticamente el contenedor también paró.

Podemos validar que no nos quedó un contenedor corriendo usando ps.

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES
```

En otras situaciones, donde el contenedor encapsula a una aplicación que queda ejecutándose (por ejemplo, una aplicación web), veremos al contenedor en ese listado.

Podemos simular este comportamiento con una aplicación Python que tenga una demora artificial, como ejemplo. Vayamos a otro directorio y pongamos allí nuestra aplicación en un archivo demorona.py:

```
1 import time
2
3 print(time.ctime())
4 time.sleep(30)
5 print(time.ctime())
```

En ese mismo directorio creemos el Dockerfile para esta aplicación:

```
1 # arrancamos con un sistema base mínimo, sólo Python 3.9
2 FROM python:3.9
3
4 # copiamos el archivo que necesitamos desde nuestro entorno a la imagen
5 COPY demorona.py ./
6
7 # indicamos que ejecute ese script cuando corramos el contenedor; notar como ejecutamos
8 # Python con '-u', lo que indica que no use buffers para la salida (porque como vamos a
9 # imprimir sólo dos líneas, veríamos ambas al final del proceso)
10 ENTRYPOINT ["python3", "-u", "demorona.py"]
```

Y creamos la imagen con `docker build -t demorona .`; si ejecutamos entonces la imagen creada y esperamos los 30 segundos, veremos una salida similar a la siguiente:

```
$ docker run demorona
Wed Jul 18 20:17:20 2022
Wed Jul 18 20:17:50 2022
```

Pero si en vez de quedarnos esperando entre una línea y la otra, vamos a otra terminal y antes de que termine miramos los procesos corriendo, vamos a encontrar nuestro contenedor:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          ...
74f26f383fb9   demorona  "python3 -u demorona.py" 13 seconds ago  Up 12 seconds  ...
```

Volvamos a la ejecución de nuestra aplicación ejemplo (el oscilador). Como le indicamos en el Dockerfile, la opción por default al ejecutar el script es que guarde el resultado en un archivo. Obviamente queremos obtener ese archivo, ya que es el resultado del trabajo.

Esto es sencillo usando el comando `cp` de `docker`, que copia archivos y directorios desde/a contenedores. En este caso queremos traer `imagen.pdf` desde el contenedor, pero no sabemos *cual* es el contenedor en cuestión.

Veamos entonces *todos* los contenedores (incluso los parados), para encontrar el que necesitamos:

```
$ docker ps --all
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          ...
6c3589d2933d   app-ejemplo  "python3 oscilador.p..." An hour ago     Exited a minute ago ...
```

Luego, copiamos el archivo en cuestión. El comando `cp` (como otros comandos similares para copiar) necesita que le especifiquemos desde dónde y hasta dónde, con el detalle que tenemos que indicar el *ID* del contenedor separado por dos puntos:

```
$ docker cp 6c3589d2933d:/oscilador/imagen.pdf .
$ ls imagen.pdf
imagen.pdf
```

Si quisieramos dejar la imagen resultado en un archivo con otro formato, podemos aprovechar la flexibilidad que incluimos en el Dockerfile, donde el *entrypoint* está separado de los argumentos que recibe por default. Entonces, si especificamos parámetros al ejecutar el contenedor, estos reemplazarán ese default, lo cual podemos ver de forma sencilla para nuestro ejemplo porque el mismo muestra las opciones que recibió:

```
$ docker run app-ejemplo --archivo=imagen.png
Opciones del script: Namespace(usar_tex=False, archivo='imagen.png')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large'}
```

7.3.4. Valor extra de los contenedores como entornos aislados

Incluyamos en esta subsección una complejidad que estábamos evitando a propósito y que para resolverla vamos a explotar la funcionalidad clave de los contenedores con respecto a los entornos virtuales.

Si miramos la imagen que obtuvimos en la subsección anterior (o la que nos mostraban las ventanas cuando estábamos con los entornos virtuales) veremos que los nombres de los ejes están escritos con tipografía “normal”. En realidad podríamos aprovechar que `matplotlib` es capaz de

utilizar $\text{\LaTeX}$ en determinadas situaciones, y nuestro script está preparado para ello (le tenemos que pasar la opción `--usar-tex`).

La complejidad está en que `matplotlib` utiliza el $\text{\LaTeX}$ del sistema para escribir los nombres de los ejes con la tipografía elegante que queremos. Tenemos que tener instalados las utilidades correspondientes en el sistema, y esto es algo que se escapa de los entornos virtuales, que sólo manejan las dependencias a nivel de Python, y es justamente donde podemos hacer brillar a los contenedores, ya que podemos incluir en los mismos no sólo las dependencias a nivel de Python sino también las externas al mismo.

Claro, para hacer eso necesitamos instalar paquetes a nivel del sistema, lo cual dependerá de cual estamos usando como base en el contenedor. En nuestro caso estamos usando `python:3.9`, que está basado en Debian, así que tenemos que instalar los paquetes necesarios en ese sistema usando el administrador de paquetes `apt`. El `Dockerfile` nos quedaría:

```

1 # arrancamos con un sistema base mínimo, sólo Python 3.9
2 FROM python:3.9
3
4 # instalamos los paquetes necesarios para que LaTeX funcione ok
5 RUN apt update && apt install -y texlive-latex-extra cm-super dvipng
6
7 # instalamos las dependencias que necesitamos directamente desde PyPI
8 RUN pip install matplotlib numpy scipy
9
10 # indicamos que vamos a trabajar en este directorio (se crea automáticamente)
11 WORKDIR /oscilador/
12
13 # descargamos el script de prueba desde el github del libro mismo
14 RUN wget https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/código/entornos/oscilador.py
15
16 # indicamos que el punto de entrada para cuando corramos el contenedor es el
17 # script, con los parámetros por default que indican que guarde la imagen
18 # resultante como un PDF y use LaTeX
19 CMD ["--archivo=imagen.pdf", "--usar-tex"]
20 ENTRYPOINT ["python3", "oscilador.py"]

```

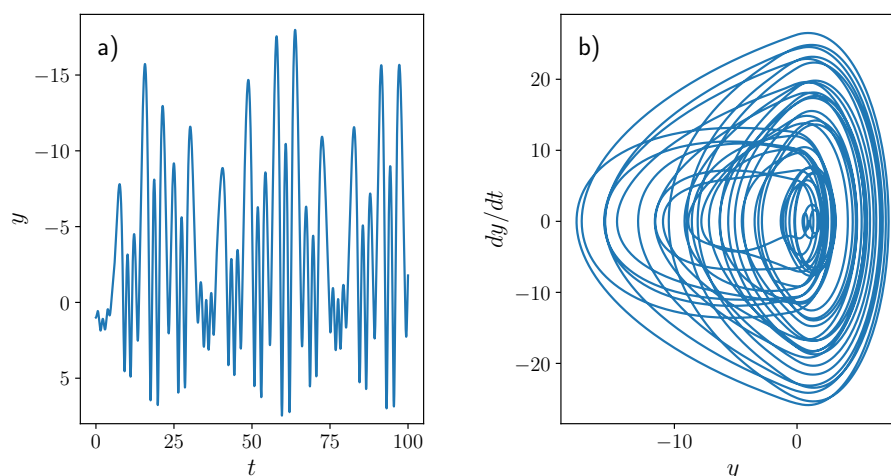
Atentos al detalle que cambiamos el argumento por default del script: ahora estamos incluyendo que use $\text{\LaTeX}$. Creamos la imagen como vimos anteriormente, y lo probamos:

```

$ docker run app-ejemplo
Opciones del script: Namespace(usar_tex=True, archivo='imagen.pdf')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large', 'text.usetex': True}

```

Y este es el resultado que nos trajimos del contenedor, donde podemos ver la elegancia obtenida:



7.3.5. Compartiendo imágenes

Una vez que tenemos nuestra imagen construida y probada, es momento de compartirla.

Para esto, como adelantamos antes, necesitamos un repositorio a donde subir la imagen y desde donde otras personas puedan bajarla. El más usado al momento de escribir este libro es [Dockerhub](#), una comunidad/biblioteca ofrecida por Docker mismo de imágenes de contenedores.

Obviamente, para utilizar ese recurso *online* primero deberemos preparar todo adecuadamente. Los pasos son sencillos, sin embargo (aunque sus detalles se escapan de el alcance de este libro): hay que crear una cuenta en dicho sitio, y luego crear un repositorio público.

Por ejemplo, para los casos que mostramos en este libro, el repositorio público usado es [facundobatista/app-ejemplo](#).

Una vez lista la parte remota, tenemos que configurar apropiadamente la parte local. La herramienta que utilizaremos para interactuar con Dockerhub es el mismo docker que veníamos usando antes. Tenemos que indicarle que se autentique:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
head over to https://hub.docker.com to create one.
Username: facundobatista
Password:
Login Succeeded
```

Nos falta un último paso, que es renombrar la imagen que armamos para que coincida con el repositorio remoto para la misma. Tengamos en cuenta que podríamos haberla nombrado apropiadamente al crearla (cuando usamos la opción `-t` en el `build`), pero tampoco es un incordio realizar ese paso a posteriori, usando `docker tag`:


```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
app-ejemplo         latest      d727e49a4b61     2 days ago      1.83GB
$ docker tag app-ejemplo facundobatista/app-ejemplo
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
facundobatista/app-ejemplo  latest      d727e49a4b61     2 days ago      1.83GB
```

Entonces sólo nos queda enviar la imagen:

```
$ docker push facundobatista/app-ejemplo
Using default tag: latest
The push refers to repository [docker.io/facundobatista/app-ejemplo]
43413aee136b: Pushed
...
latest: digest: sha256:6de9f25c02e0ae924f41d3273e01e56939366eef6d830b27e19fcc17818dde04 size: 3058
```

Ya está. A partir de este momento cualquier persona puede descargar la imagen que subimos (referenciándola de la misma manera) y ejecutarla:

```
$ docker pull facundobatista/app-ejemplo
Using default tag: latest
latest: Pulling from facundobatista/app-ejemplo
30a9a79b0d7e: Pull complete
...
Digest: sha256:6de9f25c02e0ae924f41d3273e01e56939366eef6d830b27e19fcc17818dde04
Status: Downloaded newer image for facundobatista/app-ejemplo:latest
docker.io/facundobatista/app-ejemplo:latest
$ docker run facundobatista/app-ejemplo
Opciones del script: Namespace(usar_tex=True, archivo='imagen.pdf')
Ejecutando con parámetros: a=17 b=1 λ=15.4 μ=0.75
Config de matplotlib: {'font.size': 14, 'axes.labelsize': 'large', 'text.usetex': True}
```

Esto es algo que mismo ustedes pueden probar con sólo tener docker instalado, ya que es anónimo: no hace falta sacar una cuenta en Dockerhub o autenticar docker para utilizar una imagen pública.



8 | Sobre la velocidad de procesamiento

Los programas pueden (suelen) ser lentos. Muchas veces esto no es un problema, pero frecuentemente sucede que queremos o necesitamos que un determinado programa (o parte de código dentro de un programa) sea más rápido de lo que es.

Cuando nos enfrentamos a esta situación y decidimos optimizar el programa debemos primero recordar dos reglas de oro:

- Siempre tener el programa terminado y que funcione correctamente: no tiene sentido preocuparse por si va a ser lento cuando esté terminado si todavía falta para ello (no optimizar prematuramente) y confirmar (idealmente con pruebas de unidad como ya hablamos en 5.4) su correcta operatoria, ya que es más fácil optimizar un código que funciona bien, que corregir ese código luego de que fue optimizado.
- Antes de realizar cualquier cambio en el código debemos

Módulo	Versión
Cython	3.0.11
cffi	1.17.0
line-profiler	4.1.3
mypy	1.11.2
numba	0.60.0
pandas	2.2.2
pyinstrument	4.7.3
scalene	1.5.44.1
snakeviz	2.2.0

[Código disponible](#)

descubrir en qué partes del mismo se pierde tiempo, porque no tiene sentido optimizar cualquier parte del programa. Por ejemplo, es mucho más útil mejorar en 100 milisegundos una función que se llama mil veces, que mejorar en un segundo una función que se ejecuta una sola vez.

En general es muy difícil predecir en qué partes de un código vale la pena invertir tiempo y esfuerzo para optimizar, por lo que es imprescindible medir la ejecución del programa para detectar esas zonas.

En este capítulo veremos cómo realizar esas mediciones en tiempo de ejecución, ya sea de programas enteros (*profiling*) como de pequeños pedacitos de código, haremos foco en algunos casos patológicos que son fácilmente optimizables, y finalmente veremos distintas técnicas a la hora de ir más allá de Python como lenguaje (si es realmente necesario).

8.1. Profiling

El perfilamiento (término que casi no se utiliza porque acostumbramos a mencionarlo en inglés, *profiling*) se basa en medir la ejecución de aplicaciones o programas para entender su comportamiento dinámico.

En otras palabras, no es información que se pueda obtener “inspeccionando” el código, sino que debemos ejecutarlo luego de haberlo instrumentado, obtener las distintas mediciones necesarias, y analizarlas para entender cuanto tiempo lleva la ejecución de cada parte.

Por supuesto, también podemos encontrar por inspección problemas de performance en el código (encontrando algoritmos con mal “O grande”¹ o descubriendo algunos de los casos patológicos que mencionaremos más adelante) pero revisar todo el programa en detalle lleva mucho tiempo y no es garantía de encontrar el problema. Con *profiling* podemos encontrar la zona de código culpable del comportamiento lento de nuestro programa y hacer foco ahí para realizar las optimizaciones necesarias.

A lo largo de esta sección veremos distintas herramientas para hacer *profiling*, basándonos en un programa ejemplo. Este programa fue realizado tratando de evitar cualquier optimización que salga naturalmente; casi podemos decir que a propósito cometimos varios errores inocentes, para luego poder mostrar que aunque hay varias partes del código que intuitivamente veríamos de optimizar, en realidad siempre debemos medir para luego saber donde enfocarnos (justamente, hacer *profiling*).

La idea del programa es analizar un texto grande y mostrar las 10 palabras que aparecen con mayor frecuencia. Como texto fuente usamos la obra del Don Quijote, que podemos [descargar de aquí](#) (guardarlo como `quijote.txt`).

El programa está estructurado en una función principal y tres auxiliares.

```
51 def word_count():
52     with open("quijote.txt", "rt", encoding="utf8") as fh:
53         text = fh.read()
54     all_lines = _filter_lines(text)
```

¹ la notación O grande, o *Big O* en inglés, es una herramienta para describir la complejidad en tiempo de un algoritmo, mostrando cómo se comporta ese algoritmo cuando crecen los datos

```

55     count_words, count_values = _count_words(all_lines)
56     _show_top10(count_words, count_values)

```

Esta función lee todo el texto desde el archivo y luego llama a cada una de las funciones auxiliares para filtrar las líneas útiles, contar las palabras y mostrar las 10 principales.

```

43 def _filter_lines(text):
44     all_lines = text.split("\n")
45     start_pos = all_lines.index("El ingenioso hidalgo don Quijote de la Mancha")
46     end_pos = all_lines.index("Fin")
47     all_lines = all_lines[start_pos:end_pos + 1]
48     return all_lines

```

Es necesario filtrar el contenido porque el archivo que bajamos tiene toda una sección al principio y al final que no es el texto puro del libro, que es lo que queremos procesar. Por eso la función, luego de separar el texto en líneas, toma sólo aquellas que van desde el título del libro hasta la marca de “fin”.

```

21 def _count_words(all_lines):
22     count_words = []
23     count_values = []
24     for line in all_lines:
25         for word in line.split():
26
27             for sign in SIGNS:
28                 word = word.replace(sign, "")
29                 word = word.lower()
30
31             try:
32                 word_position = count_words.index(word)
33             except ValueError:
34                 count_words.append(word)
35                 count_values.append(1)
36             else:
37                 old_value = count_values[word_position]
38                 count_values[word_position] = old_value + 1
39
40     return count_words, count_values

```

Para contar las palabras armamos dos listas para esas palabras que vamos encontrando y sus ocurrencias. Luego recorremos las líneas, separando cada una en palabras (limpiando cada una de signos de puntuación y poniéndola en minúsculas) y las contamos. El algoritmo para contar es el siguiente: buscamos la palabra en la lista de palabras que ya tenemos, si está presente (la llamada a `index` nos devuelve la posición) obtenemos el número en la posición correspondiente de la otra lista y le sumamos 1, pero si no está presente (se generó una excepción de tipo `ValueError`) la agregamos al final de la lista de palabras y agregamos el número 1 en la lista de valores de conteo.

```

4 def _show_top10(count_words, count_values):
5     print("Top 10:")

```

```
6     for selection_round in range(1, 11):
7         maxvalue = 0
8         maxpos = None
9         for pos in range(len(count_values)):
10             value = count_values[pos]
11             if value > maxvalue:
12                 maxvalue = value
13                 maxpos = pos
14
15     word = count_words[maxpos]
16     print(f"{selection_round:2d}. {maxvalue:5d} {word}")
17     del count_words[maxpos]
18     del count_values[maxpos]
```

Esta función recibe directamente las dos listas que se generaron en el proceso de conteo. Para mostrar las diez principales va a realizar el mismo procedimiento diez veces, en cada caso encuentra el valor máximo de la lista de conteos, y luego informa ese valor y la palabra de la misma posición; antes de la próxima iteración retira de la lista el valor y palabras informados, para así encontrar el próximo par.

Volvemos a recalcar lo absurdo de estos algoritmos, por favor no hagan esto en sus casas.

Veamos entonces cómo se comporta, ejecutando nuestro programa con Python, y a la vez ejecutando todo con `time`, una herramienta que luego de la ejecución del programa nos informará cuanto tiempo pasó (nos da tres valores, de los cuales nos interesa el `real`, que es el tiempo que usó el proceso):

```
$ time python3 quijote-1.py
Top 10:
1. 20626 que
2. 18214 de
3. 18188 y
4. 10363 la
5. 9823 a
6. 8241 en
7. 8210 el
8. 6335 no
9. 4748 los
10. 4690 se

real    0m5.871s
user    0m5.854s
sys     0m0.015s
```

Sospechamos fuertemente que el programa podría ser más rápido.

8.1.1. Encontrando dónde un programa es lento

Veamos entonces cómo podemos encontrar en qué zonas del programa debemos trabajar para mejorar su rendimiento. En esta subsección mostraremos tanto herramientas que vienen en la biblioteca estándar como algunas que hay que instalar previamente; excepto algunos casos puntuales donde mencionaremos una alternativa, la forma de instalación será usando `pip` (ya que están publicadas en PyPI) idealmente en un entorno virtual [7.2](#).

La biblioteca estándar de Python trae dos herramientas para hacer *profiling*, `cProfile` y `profile`. ¿Por qué dos? En realidad el que normalmente vamos a utilizar es `cProfile` ya que es el que menos carga extra le va a poner a nuestra ejecución cuando mide, mientras que `profile` se utiliza en casos donde se quiera especializar y cambiar el comportamiento de la herramienta en sí.

Es momento de comentar la diferencia entre perfiladores determinísticos y estadísticos. Un perfilador determinístico va a medir todos los pasos de ejecución del programa, generando resultados completos, pero con la desventaja de que la medición en sí afecta el tiempo total de ejecución, lo que en algunos casos es inviable. Por eso nos interesa que el perfilador en sí esté lo mas optimizado posible, y generalmente usamos `cProfile` en lugar de `profile` (ambos determinísticos). Por otro lado, un perfilador estadístico realiza mediciones sólo periódicamente, haciendo un muestreo estadístico para obtener los resultados, que en este caso pueden no ser los más precisos pero suficientes para acotar el problema; la ventaja de este tipo de perfiladores es que alteran menos el tiempo total de ejecución.

Volvamos al `cProfile`. La forma más sencilla de correr este perfilador es invocarlo como módulo en la línea de comandos indicándole el programa a ejecutar:

```
$ python3 -m cProfile quijote-1.py
Top 10:
1. 20626 que
2. 18214 de
3. 18188 y
4. 10363 la
5. 9823 a
6. 8241 en
7. 8210 el
8. 6335 no
9. 4748 los
10. 4690 se
7326537 function calls in 6.884 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1   0.000    0.000    0.000    0.000 __init__.py:43(normalize_encoding)
   1   0.000    0.000    0.000    0.000 __init__.py:71(search_function)
   1   0.000    0.000    0.000    0.000 codecs.py:260(__init__)
   1   0.000    0.000    0.000    0.000 codecs.py:309(__init__)
   1   0.000    0.000    0.003    0.003 codecs.py:319(decode)
   1   0.000    0.000    0.000    0.000 codecs.py:94(__new__)
   1   0.001    0.001    6.884    6.884 quijote-1.py:1(<module>)
   1   0.957    0.957    6.858    6.858 quijote-1.py:22(_count_words)
   1   0.000    0.000    0.005    0.005 quijote-1.py:44(_filter_lines)
   1   0.012    0.012    0.012    0.012 quijote-1.py:5(_show_top10)
   1   0.000    0.000    6.883    6.883 quijote-1.py:52(word_count)
   1   0.000    0.000    0.000    0.000 utf_8.py:33(getregentry)
   1   0.000    0.000    0.000    0.000 {built-in method __new__ of type object...
   1   0.002    0.002    0.002    0.002 {built-in method _codecs.utf_8_decode}
   1   0.000    0.000    0.000    0.000 {built-in method builtins.__import__}
   1   0.000    0.000    6.884    6.884 {built-in method builtins.exec}
   2   0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
  10   0.000    0.000    0.000    0.000 {built-in method builtins.len}
  11   0.000    0.000    0.000    0.000 {built-in method builtins.print}
   1   0.000    0.000    0.000    0.000 {built-in method io.open}
   1   0.000    0.000    0.000    0.000 {method '__exit__' of '_io._IOBase' objects}
```


45904	0.003	0.000	0.003	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
2	0.000	0.000	0.000	0.000	{method 'get' of 'dict' objects}
381219	5.445	0.000	5.445	0.000	{method 'index' of 'list' objects}
4	0.000	0.000	0.000	0.000	{method 'isalnum' of 'str' objects}
4	0.000	0.000	0.000	0.000	{method 'isascii' of 'str' objects}
1	0.000	0.000	0.000	0.000	{method 'join' of 'str' objects}
381217	0.031	0.000	0.031	0.000	{method 'lower' of 'str' objects}
1	0.004	0.004	0.007	0.007	{method 'read' of '_io.TextIOWrapper' objects}
6480689	0.401	0.000	0.401	0.000	{method 'replace' of 'str' objects}
37454	0.024	0.000	0.024	0.000	{method 'split' of 'str' objects}

Al ejecutarlo encontramos que primero se muestra la salida normal que ya habíamos visto arriba, y luego todo un reporte del cProfile: nos indica que se llamaron funciones más de siete millones de veces en casi siete segundos. Cabe destacar que el tiempo total de ejecución subió un 17 % entre la ejecución inicial y la realizada al obtener mediciones, lo cual se explica por la carga extra que mencionamos antes.

Luego tenemos una serie de columnas:

- **ncalls**: la cantidad de veces que se llamó a esa función.
- **totttime**: el tiempo total usado por esa función, sin contar el tiempo usado por llamadas a sub-funciones.
- **percall**: el tiempo por llamada (o sea, **totttime** dividido **ncalls**).
- **cumtime**: el tiempo usado por esa función ahora sí incluyendo las llamadas a sub-funciones (este número es preciso incluso para funciones recursivas)
- **percall**: **cumtime** dividido **ncalls**
- **filename:lineno(function)**: de qué función está hablando (las filas están ordenadas por esta columna)

A primera vista es difícil entender toda esta información. Mejoremos el listado ordenándolo por el tiempo acumulado de la función (y manualmente sacando aquellos con ese valor en cero, por legibilidad en el libro, así como también dejamos de mostrar el resultado del programa en sí):

```
$ python3 -m cProfile -s cumtime quijote-1.py
(...)
7326537 function calls in 6.905 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000    0.000    6.905    6.905 {built-in method builtins.exec}
      1   0.001    0.001    6.905    6.905 quijote-1.py:1(<module>)
      1   0.000    0.000    6.904    6.904 quijote-1.py:52(word_count)
      1   0.956    0.956    6.879    6.879 quijote-1.py:22(_count_words)
381219   5.466    0.000    5.466    0.000 {method 'index' of 'list' objects}
6480689   0.403    0.000    0.403    0.000 {method 'replace' of 'str' objects}
381217   0.031    0.000    0.031    0.000 {method 'lower' of 'str' objects}
```

37454	0.024	0.000	0.024	0.000 {method 'split' of 'str' objects}
1	0.012	0.012	0.012	0.012 quijote-1.py:5(_show_top10)
1	0.004	0.004	0.007	0.007 {method 'read' of '_io.TextIOWrapper' objects}
1	0.000	0.000	0.005	0.005 quijote-1.py:44(_filter_lines)
45904	0.003	0.000	0.003	0.000 {method 'append' of 'list' objects}
1	0.000	0.000	0.002	0.002 codecs.py:319(decode)
1	0.002	0.002	0.002	0.002 {built-in method _codecs.utf_8_decode}
(...)				

De arriba para abajo, vemos que el total se lo lleva el `exec` (que usa `cPython` mismo para ejecutar todo el script) y el módulo en sí al ser ejecutado. Luego ya vemos las funciones que tenemos en el código: `word_count` se lleva el tiempo total, y justo después tenemos la primera sorpresa con `_count_words`, donde se pierden 6.879 segundos, el 99.6 % del tiempo total.

Esta información es justamente la que vuelve valiosa al perfilamiento: nos podemos olvidar de las otras funciones y enfocarnos en `_count_words`.

Habiendo dicho eso, llama la atención una línea por su gran cantidad de llamadas, la correspondiente al método `replace` de las cadenas. Vemos que el único `replace` que usamos es (¡en `_count_words`!) para sacar los signos de puntuación de cada palabra. Nos damos cuenta que en lugar de llamar a ese `replace` por cada palabra, podríamos sacar los signos directamente de toda la línea... o incluso de todo el texto a la vez. Claro, no sabemos si será eficiente hacer un reemplazo de todo el texto en un sólo paso, pero esa es la idea: hacemos el cambio que nos parece correcto, y volvemos a ejecutar el perfilamiento para entender si mejoró o empeoró.

Movemos entonces estos reemplazos (y el convertirlo a minúscula) a la función `word_count`, obteniendo la segunda versión de este programa (mostrando aquí sólo las dos funciones afectadas):

```

21 def _count_words(all_lines):
22     count_words = []
23     count_values = []
24     for line in all_lines:
25         for word in line.split():
26             try:
27                 word_position = count_words.index(word)
28             except ValueError:
29                 count_words.append(word)
30                 count_values.append(1)
31             else:
32                 old_value = count_values[word_position]
33                 count_values[word_position] = old_value + 1
34
35     return count_words, count_values

```

```

46 def word_count():
47     with open("quijote.txt", "rt", encoding="utf8") as fh:
48         text = fh.read()
49
50     for sign in SIGNS:
51         text = text.replace(sign, "")
52     text = text.lower()
53

```



```

54 all_lines = _filter_lines(text)
55 count_words, count_values = _count_words(all_lines)
56 _show_top10(count_words, count_values)

```

Volvemos a perfilar:

```

$ python3 -m cProfile -s cumtime quirote-2.py
(...)
464645 function calls in 5.641 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000   0.000   5.641    5.641 {built-in method builtins.exec}
      1   0.001   0.001   5.641    5.641 quirote-2.py:1(<module>)
      1   0.000   0.000   5.640    5.640 quirote-2.py:47(word_count)
      1   0.115   0.115   5.588    5.588 quirote-2.py:22(_count_words)
 381217   5.452   0.000   5.452   0.000 {method 'index' of 'list' objects}
 37454   0.022   0.000   0.022   0.000 {method 'split' of 'str' objects}
     17   0.021   0.001   0.021   0.001 {method 'replace' of 'str' objects}
      1   0.012   0.012   0.012   0.012 quirote-2.py:5(_show_top10)
      1   0.005   0.005   0.008   0.008 {method 'read' of '_io.TextIOWrapper' objects}
      1   0.007   0.007   0.007   0.007 {method 'lower' of 'str' objects}
      1   0.000   0.000   0.005   0.005 quirote-2.py:39(_filter_lines)
      1   0.000   0.000   0.003   0.003 codecs.py:319(decode)
      1   0.003   0.003   0.003   0.003 {built-in method _codecs.utf_8_decode}
 45902   0.003   0.000   0.003   0.000 {method 'append' of 'list' objects}
(...)

```

Ahora vemos que `replace` y `lower` todavía están, pero se ejecutan muchísimo menos que en la versión anterior y ya no aportan al tiempo total. Encontramos que `_count_words` sigue siendo muy cara, y la explicación está en la gran cantidad de llamadas a `index` (el 96 % del tiempo total), que se usa en esa función. La razón subyacente es que estamos usando `index` para el conteo de palabras, y buscar en una lista es potencialmente muy costoso (porque va comparando el objeto buscado posición por posición). Pensemos una mejor forma para guardar la cantidad de veces que encontremos cada palabra: un diccionario, donde podemos tener la palabra como clave y la cantidad como valor, teniendo en cuenta que el acceso por clave a un diccionario es extremadamente rápido.

Mejoremos el código, obteniendo la versión 3.

```

4 def _show_top10(count):
5     print("Top 10:")
6     for selection_round in range(1, 11):
7         maxvalue = 0
8         maxword = None
9         for word, value in count.items():
10             if value > maxvalue:
11                 maxvalue = value
12                 maxword = word
13
14     print(f"{selection_round:2d}. {maxvalue:5d} {maxword}")
15     del count[maxword]

```

```

16
17
18 def _count_words(all_lines):
19     count = {}
20     for line in all_lines:
21         for word in line.split():
22             try:
23                 value = count[word]
24             except KeyError:
25                 value = 0
26                 count[word] = value + 1
27
28     return count

```

No sólo modificamos el conteo en sí, que ahora busca cada palabra y guarda su cantidad en un diccionario, sino que este diccionario es el que recibe `_show_top10`, donde mantuvimos el algoritmo “absurdo” original pero ahora trabajando sobre esta nueva estructura.

Veamos cómo se comporta esta nueva versión:

```

(...)
$ python3 -m cProfile -s cumtime quijote-3.py
37532 function calls in 0.124 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000   0.000   0.124    0.124 {built-in method builtins.exec}
      1   0.001   0.001   0.124    0.124 quijote-3.py:1(<module>)
      1   0.000   0.000   0.123    0.123 quijote-3.py:40(word_count)
      1   0.059   0.059   0.075    0.075 quijote-3.py:19(_count_words)
     17   0.021   0.001   0.021    0.001 {method 'replace' of 'str' objects}
  37454   0.020   0.000   0.020    0.000 {method 'split' of 'str' objects}
      1   0.008   0.008   0.008    0.008 quijote-3.py:5(_show_top10)
      1   0.005   0.005   0.008    0.008 {method 'read' of '_io.TextIOWrapper' objects}
      1   0.007   0.007   0.007    0.007 {method 'lower' of 'str' objects}
      1   0.000   0.000   0.005    0.005 quijote-3.py:32(_filter_lines)
      1   0.000   0.000   0.003    0.003 codecs.py:319(decode)
      1   0.003   0.003   0.003    0.003 {built-in method _codecs.utf_8_decode}
(...)

```

¿Podemos seguir mejorando el código? Seguramente. ¿Vale la pena? Quizás no, ya que el tiempo total de ejecución es de alrededor de 100 milisegundos.

Habiendo dicho eso, este código puntual sí merece una mejora, que es la de reemplazar la función de búsqueda propia y el algoritmo para mostrar los primeros 10, utilizando para ello el objeto `Counter` del módulo `collections`, no sólo porque es apenas más rápido, sino porque estamos reemplazando bastante código hecho por nosotros para nuestro programa por funcionalidad que ya viene en la biblioteca estándar, que está mucho más probada y optimizada. Menos código nuestro significa menos fallas ahora y menos mantenimiento en el futuro. Les dejamos como desafío luego de la lectura el adaptar la versión 3 de esta manera (pero si quieren espiar el resultado, busquen la versión 4 del código).

Volviendo al perfilamiento, el ejercicio que hicimos en el libro es válido pero sólo un ejemplo

sencillo. Si el programa fuese más complejo y largo, el reporte total puede volverse bastante confuso.

Una forma de mejorar esto es, si ya tenemos detectada la función que se lleva la mayor parte del tiempo, realizar el *profiling* solo allí. Adaptemos de esta manera la versión 1 original del código, dando lugar a una versión 5; mostramos aquí sólo el cambio, que es en la función principal:

```

54 def word_count():
55     with open("quijote.txt", "rt", encoding="utf8") as fh:
56         text = fh.read()
57     all_lines = _filter_lines(text)
58
59     profiler = cProfile.Profile()
60     count_words, count_values = profiler.runcall(_count_words, all_lines)
61     profiler.print_stats(sort=pstats.SortKey.CUMULATIVE)
62
63     _show_top10(count_words, count_values)

```

Aquí vemos que en lugar de llamar directamente a `_count_words` pasándole `all_lines`, le indicamos a un *profiler* que realice esa ejecución con ese argumento; a este *profiler* lo creamos en la línea anterior, y luego también lo usamos para mostrar los resultados, indicándole que ordene de la mejor manera que nos parezca.

Probamos esta versión, entonces:

```

$ python3 quijote-5.py
7326478 function calls in 6.841 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.950    0.950    6.841    6.841 quijote-5.py:24(_count_words)
  381217    5.437    0.000    5.437    0.000 {method 'index' of 'list' objects}
 6480689    0.400    0.000    0.400    0.000 {method 'replace' of 'str' objects}
  381217    0.030    0.000    0.030    0.000 {method 'lower' of 'str' objects}
   37453    0.020    0.000    0.020    0.000 {method 'split' of 'str' objects}
   45900    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

(...)

```

Para entender resultados complejos también podemos visualizarlos de una forma más gráfica. Los perfiladores de la biblioteca estándar permiten exportar los resultados para luego procesarlos en diferido con el módulo `pstats` (también de la biblioteca estándar) o levantarlos con otras herramientas. Las dos más conocidas para esto son `SnakeViz` (específica para los resultados del perfilador de Python) y `KCachegrind` (visualizador genérico, originalmente pensado para los resultados de la herramienta `Valgrind`).

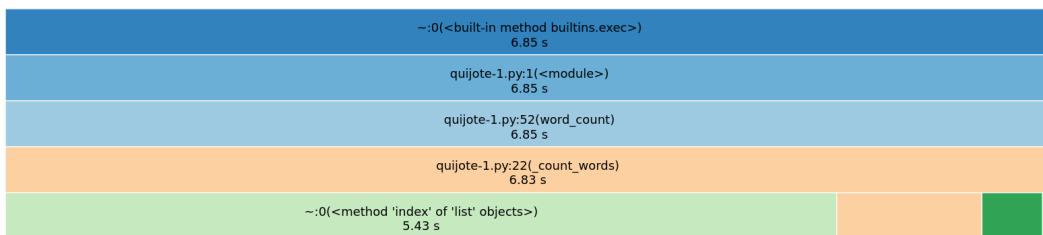
Entonces el primer paso es correr el perfilador pero guardar los resultados en lugar de que los muestre por pantalla (para simular una operatoria normal volvemos a la versión original de nuestro programa):

```
$ python3 -m cProfile -o quijote.prof quijote-1.py
(...)
```

Le indicamos a SnakeViz que procese directamente ese archivo que generamos:

```
(env) $ snakeviz quijote.prof
```

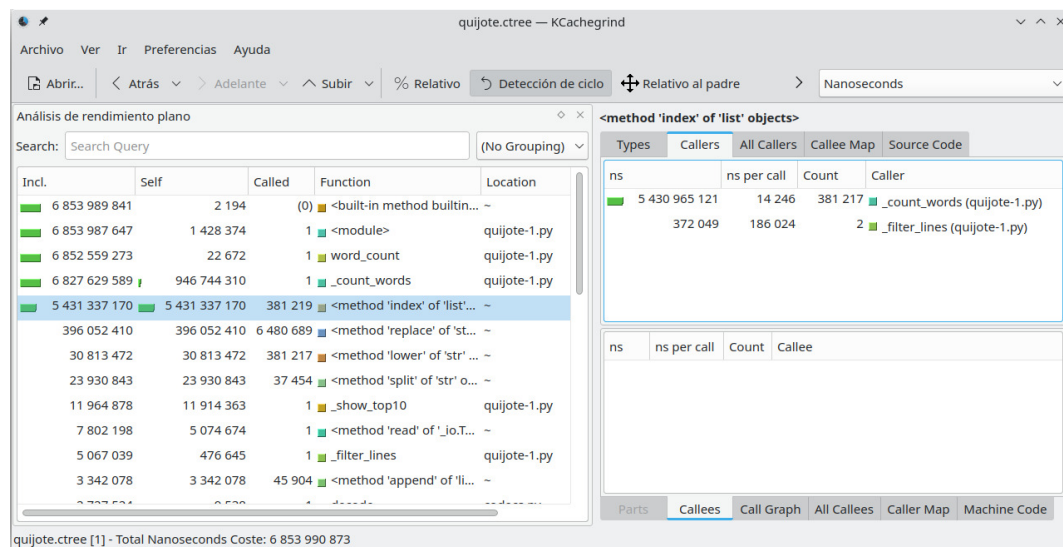
SnakeViz abrirá una página local en nuestro navegador mostrando un gráfico bastante útil porque nos permite ver donde se pierde tiempo en cada nivel de la pila de llamadas (y además nos permite controlar un poco qué vemos y cómo). Para el caso nuestro el gráfico es:



Por otro lado, para KCachegrind primero tenemos que convertir el formato de los datos fuente, lo cual hacemos con otra herramienta, `pyprof2calltree`. En Debian/Ubuntu ambas se instalan con `apt`.

```
$ pyprof2calltree -i quijote.prof -o quijote.ctree
$ kcachegrind quijote.ctree
```

En este caso se nos abrirá un programa de escritorio donde podemos interactuar bastante con las distintas mediciones y filtrarlas. Por ejemplo si a la izquierda elegimos la línea correspondiente a la función `_count_words` a la derecha veremos la información reducida para esa función (lo mismo que vimos en el caso de arriba donde sólo perfilamos dicha función), o si elegimos las llamadas a `index` veremos en dónde se realizaron esas llamadas, como se ve en la siguiente captura de pantalla:



Una forma alternativa de ver los resultados es línea por línea del código, y ese es el formato en que muestra los resultados un perfilador alternativo (si lo usamos de la forma correcta): el `line_profiler`. La utilidad que usamos es `kernprof`, pero antes de ejecutarla debemos decorar la función que queremos perfilar, agregándole un `@profile` antes de su definición (al cual no hay que importar, se encarga el perfilador mismo de proveerlo). Entonces:

```
(env) $ kernprof -lv quijote-1b.py
(...)
Wrote profile results to quijote-1.py.lprof
Timer unit: 1e-06 s

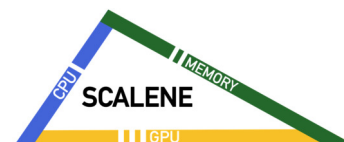
Total time: 8.36627 s
File: quijote-1b.py
Function: _count_words at line 22
```

Line #	Hits	Time	PerHit	%Time	Line Contents
22					@profile
23					def _count_words(all_lines):
24	1	1.1	1.1	0.0	count_words = []
25	1	0.3	0.3	0.0	count_values = []
26	37454	5414.8	0.1	0.1	for line in all_lines:
27	418670	99311.0	0.2	1.2	for word in line.split():
28					
29	6861906	963742.4	0.1	11.5	for sign in SIGNS:
30	6480689	1498922.1	0.2	17.9	word = word.replace(sign, "")
31	381217	87598.5	0.2	1.0	word = word.lower()
32					
33	381217	43142.2	0.1	0.5	try:
34	381217	5497893.0	4.4	65.7	word_position = count_words.index(word)
35	22950	6517.9	0.3	0.1	except ValueError:
36	22950	6013.4	0.3	0.1	count_words.append(word)
37	22950	7600.2	0.3	0.1	count_values.append(1)
38					else:
39	358267	67836.1	0.2	0.8	old_value = count_values[word_position]
40	358267	82275.2	0.2	1.0	count_values[word_position] = old_value + 1
41					
42	1	0.7	0.7	0.0	return count_words, count_values

Vemos en este formato de salida que es fácilmente detectable en qué línea(s) de nuestro programa tenemos que poner atención para mejorar el rendimiento.

Otro perfilador muy útil es `Scalene`. En este caso, antes de instalar la herramienta con `pip` debemos asegurarnos que el sistema esté configurado correctamente (lo que cambia de plataforma a plataforma, [ver detalle](#)). El diferencial de este perfilador es que no sólo analiza el uso del procesador general, sino también del consumo de memoria e incluso de la GPU si nuestro programa corre en parte en una placa de video.

La utilización es sencilla:



scalene - Profiler de alto rendimiento que proporciona información detallada sobre el uso de CPU y memoria en programas Python.

```
(env) $ scalene quijote-1.py
```

Al terminar abrirá una página web local mostrando los resultados que nos permitirá interactuar de diversas maneras. Para nuestro ejemplo tenemos:



Finalmente, veamos un perfilador que trabaja de forma estadística (lo cual no es necesario para nuestro ejemplo simple pero es vital si el tiempo de ejecución de aquello que queremos perfilar es muy alto): `pyinstrument`.

```
$ pyinstrument quijote-1.py
(...pyinstrument quijote-1.py)
_ _ _ _ _ /_ _ _ _ _ /_ Recorded: 20:54:28 Samples: 8048
/_/_/_/_ /_\ / __/_ / /__' / / Duration: 8.479 CPU time: 8.477
/_ _/_ v4.7.3

Program: pyinstrument quijote-1.py

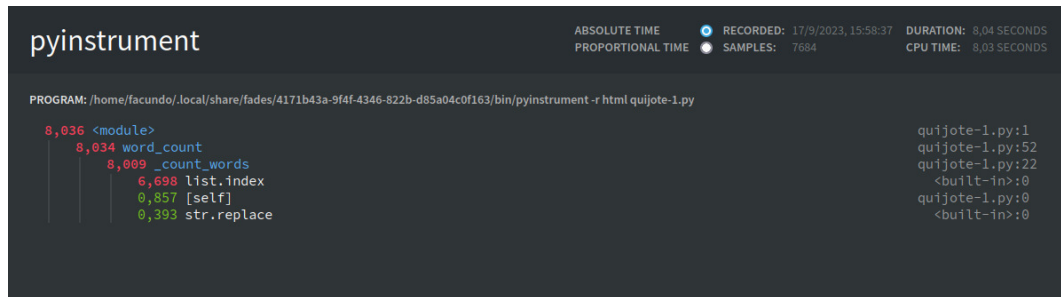
8.479 <module> quijote-1.py:1
└─ 8.478 word_count quijote-1.py:52
└─ 8.452 _count_words quijote-1.py:22
├─ 7.217 list.index <built-in>
├─ 0.816 [self] quijote-1.py
└─ 0.366 str.replace <built-in>
```

El resultado por terminal es muy interesante porque de forma concisa nos está indicando

los puntos donde debemos hacer foco. Alternativamente podemos pedirle que nos entregue el reporte en una página web:

```
$ pyinstrument -r html quijote-1.py
```

Para nuestro ejemplo, entonces:



8.1.2. Midiendo pequeñas partes de código

Cuando estamos optimizando, muchas veces sucede que encontramos que el punto a optimizar son determinadas líneas que se ejecutan muy frecuentemente, cómo usamos alguna estructura de datos, etc.

Podríamos encarar esa situación cambiando el código o la estructura y volviendo a evaluar si el programa continúa lento, con *profiling* incluido si es el caso, hasta entender si los cambios que realizamos surtieron efecto o no, de formar similar como fuimos haciendo a través de los ejemplos en el comienzo de la sección.

Pero hay una mejor manera de realizar ese proceso, y es medir directamente si el cambio que tenemos pensado efectivamente mejoraría la situación. El inconveniente es que cuanto más pequeña hagamos la medición, más se afectará por el contexto que no controlamos. En otras palabras, si evaluamos la ejecución de nuestro programa que tarda algunos minutos podemos encontrar los puntos calientes sin que variaciones temporales del sistema influyan, pero si vamos a comparar una u otra forma de escribir un algoritmo que en su totalidad tarda algunos milisegundos, cualquier imprevisto del sistema donde estamos midiendo nos puede afectar el resultado.

Para evitar caer en trampas conocidas como carga del sistema, manejo de memoria, otros procesos concurrentes, etc., Python trae una utilidad muy práctica que se ocupa de minimizar todas esas variaciones: *timeit*. Con esta herramienta podemos medir pequeños pedazos de código muy puntuales, entendiendo en qué caso tal o cual forma de escribir un algoritmo es más rápido. Por supuesto, es importante entender que estas mediciones donde ganamos milisegundos de un caso al otro sólo tienen sentido si van a ser utilizadas intensamente en el programa real. Si no, realmente será una mejora que no nos afectará el comportamiento general.

Por ejemplo, en nuestro ejemplo antes vimos que era costoso buscar en una lista, y queremos trabajar en ese punto. Se nos ocurren varias ideas, y queremos ver si es más rápido buscar un elemento que no existe en una lista, una tupla, o un diccionario de 100 elementos...

```
1 >>> import timeit
2 >>> timeit.timeit("555 in vals", setup="vals = list(range(100))")
```

```

3 0.4607653420243878
4 >>> timeit.timeit("555 in vals", setup="vals = tuple(range(100))")
5 0.4076314979849849
6 >>> timeit.timeit("555 in vals", setup="vals = dict.fromkeys(range(100))")
7 0.05606273500598036

```

Varios detalles a tener en cuenta en la secuencia mostrada. Por un lado el pedacito de código (que en inglés llamamos *snippet*) va entre comillas, para que no se ejecute antes de la llamada sino cuando esté siendo medido. Como nos interesa medir solamente la búsqueda, no creamos la lista, tupla o diccionario en el primer pedacito de código, sino que lo hacemos en otro código de `setup`. Alternativamente podríamos haberlo creado antes de llamar a `timeit` pero tendríamos que haberle pasado esos objetos en un diccionario con el parámetro `globals` para que lo encuentre como variable. Tengamos en cuenta que el código de `setup` se ejecuta una sola vez para toda la medición, mientras que el que se mide se ejecuta muchísimas veces; esto puede ser importante en la medición si el código medido modifica las estructuras armadas en `setup`. Dejamos para la subsección siguiente el porqué de la diferencia de tiempos observada.

El módulo `timeit` nos da la facilidad de usarlo directamente desde la línea de comandos. Lo anterior lo podríamos reescribir como:

```

$ python3 -m timeit -s "vals = list(range(100))" "555 in vals"
500000 loops, best of 5: 428 nsec per loop
$ python3 -m timeit -s "vals = tuple(range(100))" "555 in vals"
1000000 loops, best of 5: 391 nsec per loop
$ python3 -m timeit -s "vals = dict.fromkeys(range(100))" "555 in vals"
20000000 loops, best of 5: 18.7 nsec per loop

```

Esta forma de ejecución puede ser un poco más trabajosa porque hay que definir siempre todo el contexto (en la cadena que pasamos con `-s` para que no entre en la medición) pero tiene la ventaja que es seguro que no depende de otro contexto que podemos usar accidentalmente.

Los Jupyter Notebook nos dan la facilidad de invocar a `timeit` de forma sencilla, con el detalle que **no** tenemos que poner lo que queremos medir entre comillas sino directamente:

CELDA 1 DE PERFIMP.IPYNB
<pre> vals = list(range(100)) vals_t = tuple(vals) vals_d = dict.fromkeys(vals) %timeit 555 in vals %timeit 555 in vals_t %timeit 555 in vals_d </pre>
<pre> 439 ns ± 1.59 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each) 383 ns ± 0.971 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each) 37.2 ns ± 0.0651 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each) </pre>

Más allá de la forma de invocar al módulo, lo importante es entender que no importa la medición en sí, sino que es un accesorio que nos permite entender qué está sucediendo de forma subyacente. Lo ideal sería armar un modelo mental de cómo se está ejecutando el programa con las distintas estructuras de datos y luego confirmar o refutar el modelo que armamos utilizando la medición.

8.1.3. Usando Python de forma más eficiente

Hay distintas situaciones donde el código es lento por cómo usamos alguna estructura de datos o armamos el programa en sí, como el caso que mencionamos al final de la subsección anterior de buscar en una lista, cuya duración depende del largo de esa lista.

Es útil conocer los casos más evidentes para evitar caer en esas pequeñas trampas, entendiendo que si en nuestro código hacemos algo de esto una, dos o diez veces no es problema, pero si vamos a tener bucles grandes sí podemos tener algo para corregir.

En esta subsección presentamos varios casos, y aunque no son todos los casos posibles, sí son los más conocidos.

En lugar de usar el `%timeit` de Jupyter Notebook armamos una función para medir, principalmente para poder capturar los valores y compararlos, y de yapa nos da la ventaja de ser explícitos con respecto al contexto.

CELDA 02

```
import timeit
from quantiphy import Quantity

Quantity.set_prefs(prec=2)

def measure(case1, case2, globals=None):
    delays = []
    for case in (case1, case2):
        t = timeit.Timer(case, globals=globals)
        number, _ = t.autorange()
        total_delay = t.timeit(number=number)
        delays.append(Quantity(total_delay / number, 's'))

    d1, d2 = delays
    print(f"Primero: {d1}")
    print(f"Segundo: {d2}")
    speedup = d1 / d2
    print(f"Mejora: {speedup:.2f} X")
```

La función que armamos recibe los dos casos a comparar y opcionalmente un espacio de nombre con los objetos que necesitan esos casos. Luego, para cada uno, instancia un objeto `Timer` con el caso indicado, le pide evaluar un número de iteraciones que no sean ni pocos (bajo valor estadístico) ni muchos (tardaría demasiado) y luego realiza las mediciones, obteniendo la demora total. La demora para un caso único es ese valor dividido la cantidad de intentos, y la guardamos usando `Quantity` para que al mostrarse automáticamente esté escalado y con una unidad que tenga sentido leer. Luego mostramos los dos valores y la mejora (en “veces”) del primero al segundo.

Entonces, el primer caso que evaluamos es el que mencionábamos en la subsección anterior. Aquí comparamos buscando casi el último elemento en una estructura de un millón de largo, con lo cual la mejora es notable:

CELDA 03	
<pre>src_list = list(range(1_000_000)) src_set = set(src_list) bad = "999999 in src_list" good = "999999 in src_set" measure(bad, good, globals={"src_set": src_set, "src_list": src_list})</pre>	
Primero:	4.8 ms
Segundo:	31.4 ns
Mejora:	152841.03 X

La lista y la tupla, más allá de diferentes formas de recorrerlas, van comparando uno a uno los elementos para encontrar si el objeto está incluido, entonces si lo encuentra en la primer posición va a tardar casi nada pero si tiene que recorrer toda la estructura va a tardar en función de cuan larga sea la misma. El diccionario (como el conjunto) calcula el hash del objeto y lo busca directamente; tiene el costo de calcular el hash pero después es un acceso independiente del largo de la estructura. Y esto es muy importante porque si hubiésemos medido el buscar el número 1 en lugar de 555, o buscarlo en una lista o tupla pequeñas, la medición sería totalmente inútil.

Dejamos como ejercicio para luego de la lectura el jugar buscando elementos que estén al principio o usar estructuras más pequeñas.

Como contra-regla al caso anterior, sólo para advertir no caer en esa trampa, resaltamos que aunque buscar en un conjunto es mucho más rápido que buscar en una lista, no tiene sentido convertir esa lista a conjunto *sólo para la búsqueda*. Esto es porque la conversión a conjunto, además de calcular el hash de todos los objetos y armar la estructura necesaria, obviamente recorre toda la lista de principio a fin, que es un poco lo que queríamos evitar. Por eso vemos que el segundo caso es más lento:

CELDA 04	
<pre>src_list = list(range(1_000_000)) bad = "999999 in src_list" worse = "999999 in set(src_list)" measure(bad, worse, globals={"src_list": src_list})</pre>	
Primero:	4.84 ms
Segundo:	25.6 ms
Mejora:	0.19 X

Siguiendo el mismo carril de evitar construir objetos que no son estrictamente necesarios, tenemos que destacar que cuando sacamos una “rebanada” de una lista estamos construyendo toda una nueva estructura. Si vamos a necesitar esa lista como tal, está bien, pero si sólo queremos consumirla de alguna manera podemos usar `itertools.islice` que nos va a permitir iterar sobre esos objetos sin el costo intermedio:

CELDA 05

```
import itertools

src = list(range(1_000))

bad = "src[1:]"
good = "itertools.islice(src, 1, None)"

measure(bad, good, globals={"itertools": itertools, "src": src})
```

Primero: 1.13 us
Segundo: 118 ns
Mejora: 9.62 X

El siguiente caso nos muestra que deberíamos tratar de evitar llamar a funciones en ciclos grandes si la llamada es trivial, porque el costo de preparar y ejecutar la función en si resulta relevante. Lo mostramos comparando cómo elevar al cuadrado una lista de números, primero usando map que aplica la función a cada elemento, luego usando una comprensión de listas que hace el cálculo directamente:

CELDA 06

```
src = list(range(1_000_000))

def cuad(n):
    return n ** 2

bad = "list(map(cuad, src))"
good = "[x ** 2 for x in src]"
measure(bad, good, globals={"cuad": cuad, "src": src})
```

Primero: 221 ms
Segundo: 194 ms
Mejora: 1.14 X

Otra estructura que es costosa de preparar es la excepción. Cuando tenemos un bloque try / except el hecho de supervisar el código es extremadamente barato, pero si se sucede la excepción, ahí se incurre en un costo relevante. Es por esto que si utilizamos esa estructura dentro de un bucle debemos ver si podemos reemplazar esa funcionalidad de otra manera (validando las variables para no entrar en la situación de excepción, por ejemplo).

CELDA 07

```
def bad():
    try:
        1 / 0
    except ZeroDivisionError:
        pass

def good():
    try:
        1 / 1
    except ZeroDivisionError:
        pass
```

```
measure(bad, good)
```

```
Primero: 197 ns  
Segundo: 37.3 ns  
Mejora: 5.27 X
```

Uno de los tipos de datos más utilizados en Python es `list`, por su utilidad y versatilidad. Tenemos que tener en cuenta sin embargo que está implementado de manera que si queremos agregar valores o sacarlos del final es muy barato, pero para insertar valores al principio o en algún punto intermedio de la estructura incurre en el costo de re-acomodarla. Por eso si vamos a estar sacando o insertando valores al principio es muy recomendable usar otra estructura que tenemos en la biblioteca estándar, el deque:

CELDA 08

```
from collections import deque  
  
src_list = list(range(1000))  
src_deque = deque(range(1000))  
  
bad = "src_list.insert(0, 123)"  
good = "src_deque.appendleft(123)"  
  
measure(bad, good, globals={"src_deque": src_deque, "src_list": src_list})
```

```
Primero: 17.5 us  
Segundo: 34.3 ns  
Mejora: 509.80 X
```

Justamente las listas están, como decíamos, pensadas para agregarles elementos y esto está bien optimizado, mientras que en las cadenas no tanto. Si vamos a estar armando una cadena muy larga de a pedacitos, es mejor ir agregando esas partes en una lista y luego construir la cadena al final a partir de esta:

CELDA 09

```
def bad():  
    final = ""  
    for _ in range(10000):  
        final += "x"  
    return final  
  
def good():  
    final = []  
    for _ in range(10000):  
        final.append("x")  
    return "".join(final)
```

```
measure(bad, good)
```

```
Primero: 616 us  
Segundo: 397 us  
Mejora: 1.55 X
```

Ya vimos en el capítulo sobre NumPy 11 que si podemos delegar un procesamiento a una operación interna de esa biblioteca (en vez de iterar y realizar los cálculos de a uno en nuestro código) vamos a tener un resultado más rápido. Lo mismo sucede incluso fuera de esa biblioteca, por eso es siempre útil conocer qué herramientas tenemos en Python tanto integradas en el lenguaje como en la biblioteca estándar. Por ejemplo, veamos el caso de llamar directamente a `sum` en lugar de realizar las sumas a mano:

CELDA 10

```
def bad():
    total = 0
    for n in range(10000):
        total += n
    return total

def good():
    return sum(range(10000))

measure(bad, good)
```

Primero: 339 us
Segundo: 94 us
Mejora: 3.61 X

Debido a la naturaleza dinámica de Python, cada vez que queremos acceder a un atributo o nombre, hay ciertas reglas que se cumplen y pasos que se recorren. Por ejemplo, si accedemos a un nombre, `valor * 3`, Python lo buscará primero en el espacio de nombres local, y si no lo encuentra luego lo buscará en el espacio de nombres global, etc. O si usamos una función de un módulo, por ejemplo `datetime.now()`, Python resolverá primero la búsqueda de `datetime` de la forma que recién vimos, y luego tratará de encontrar `now` allí dentro. Si todas estas operaciones están en un punto álgido de nuestro código, podemos mejorar trayendo todos los nombres al espacio de nombre local para que Python los encuentre en el primer lugar que los busca.

En el siguiente código armamos una estructura que muestra varios de estos casos. Tenemos un factor a nivel global, y una clase que al inicializar la instancia crea una lista de datos de un millón de largo, y guarda un factor de corrección, y tiene dos métodos, el primero escrito todo directamente y el segundo con las mejoras

CELDA 11

```
import math

FACTOR = 0.555

class Foo:

    def __init__(self, correction):
        self.data = list(range(1_000_000))
        self.correction = correction

    def bad(self):
        total = 0
        for number in self.data:
            res = math.sin(number * self.correction * FACTOR)
```

```
        total += res
    return total

def good(self):
    total = 0
    correction = self.correction
    sin = math.sin
    factor = FACTOR
    for number in self.data:
        res = sin(number * correction * factor)
        total += res
    return total

foo = Foo(1.234)
measure(foo.bad, foo.good)
```

Primero: 151 ms
Segundo: 84.8 ms
Mejora: 1.78 X

El primer método recorre la estructura de datos y va sumando en un total el resultado del seno de la multiplicación de tres números: el obtenido de los datos, ajustado por la corrección y el factor global.

A la hora de aplicar las mejoras mencionadas tenemos que entender en qué caso es útil. A `self.data` lo accedemos una sola vez, así que eso está bien; el punto donde hacer foco es dentro del bucle, que se repite mucho. En la versión “rápida” del código entonces tenemos que dentro del `for` accedemos a `sin`, `correction` y `factor` siendo esos tres nombres locales que nos ocupamos de crear afuera del bucle.

Finalmente, debemos resaltar que siempre tenemos que considerar actualizar la versión misma de Python, ya que la performance en general del lenguaje es algo que siempre se intenta mejorar en cada liberación, haciéndose bastante foco en eso desde 3.9 en adelante. En particular también puede suceder que el lenguaje optimice específicamente algunos de los casos descritos anteriormente y no sean más un punto a prestar atención, de forma similar a como ha mejorado la concatenación de cadenas.

8.2. OK, sigue siendo lento, ¿entonces?

Cuando hemos optimizado Python al límite y sigue siendo el cuello de botella, es hora de dar el siguiente paso: empezar a trabajar con código compilado. En esta sección veremos distintas técnicas para compilar en mayor o menor medida nuestro código Python para acelerar las porciones de código que sean críticas.

Se puede acelerar mucho ir del mundo Python a código compilado porque justamente vamos perdiendo las ventajas de Python en el camino: su dinamismo, no tener que declarar tipos estrictos, el manejo automático de memoria, etc. El lado positivo es que como ya tenemos código funcionando y pensado bastante, en general no vamos a estar explorando alternativas con ese código, por lo que el precio de hacerlo más rígido y restrictivo no es alto. Pero es parte de la complejidad que vamos a enfrentar en este viaje.

No vamos a explicar las distintas herramientas en detalle porque se escapa del alcance de

este libro, pero sí queremos hacer un pequeño recorrido de cada una así ustedes pueden evaluar con cual trabajar y permitirles arrancar con ejemplos sencillos, para luego profundizar según sea necesario.

8.2.1. Un ejemplo usando Python

Armemos un ejemplo que nos va a servir para poder tener un código susceptible de mejoras e ir viendo los cambios. Simulemos que realizamos unas mediciones en una máquina y eso nos deja 16 mil archivos en una estructura de directorios (no todos en el mismo, por eficiencia del sistema de archivos), cada archivo con el estado del sistema en un momento dado siendo ese estado la posición en (x, y) de cada partícula y su radio ². La tarea de nuestro programa ejemplo será analizar archivo por archivo, contar cuantas partículas están dentro de un determinado círculo o tocándolo, y escribir un archivo de resultado que tenga una línea por archivo fuente: su nombre y la cantidad de partículas filtradas.

Arrancamos con la versión “inocente” del código, `particulas-1.py`. Más allá de las líneas específicas para manejar los parámetros que se le pasan al programa (que omitiremos copiar aquí, por brevedad, pero que pueden ir a investigar al archivo del programa en sí) tenemos una función principal y una específica para procesar cada archivo.

```

16 def main(basedir, outpath, center_x, center_y, radius):
17     outfh = open(outpath, "wt")
18     for dirpath, dirnames, filenames in os.walk(basedir):
19         for filename in filenames:
20             count = proc_file(os.path.join(dirpath, filename), center_x, center_y, radius)
21             outfh.write(f"{filename},{count}\n")
22     outfh.close()

```

Esta función principal recorre toda la estructura de directorios y para cada archivo encontrado llama a la función que lo procesa y devuelve el conteo, que procede a guardar (junto con el nombre del archivo procesado) en el archivo de salida indicado.

```

5 def proc_file(filepath, center_x, center_y, radius):
6     count = 0
7     with open(filepath, "rt") as fh:
8         for line in fh:
9             particle_x, particle_y, particle_radius = map(float, line.strip().split(","))
10            dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
11            if dist_centers <= (radius + particle_radius) ** 2:
12                count += 1
13    return count

```

La función que procesa el archivo lo abre y lee línea por línea, separando los tres valores y convirtiéndolos a float para poder operar. Luego calcula la distancia entre el centro de la partícula y el centro de la zona a buscar; si esta distancia es menor que la suma de los radios de la partícula y la zona a buscar significa que la partícula toca esa zona, e incrementa el contador (en

²Para simular estos datos de entrada pueden ejecutar el programa `crear-data-particulas.py` incluido en el código del libro

realidad la distancia sería la raíz cuadrada de lo calculado, pero es más eficiente elevar al cuadrado la suma de los radios, total para la comparación sirve). Al final devuelve el valor acumulado y listo.

El correr este programa sobre toda la estructura de directorios generada tarda 3 minutos 16 segundos.

```
$ time python3 particulas-1.py superdir/ results.txt 2000 2000 100

real    3m16,482s
user    3m07,887s
sys     0m3,053s
```

Para tener una referencia de todo lo rápido que podría ser hicimos un programa equivalente en C++³, y vimos que tarda sólo 2 minutos 11 segundos en realizar el mismo procesamiento.

```
$ time ./particulas superdir/ results.txt 2000 2000 100

real    2m11,621s
user    2m05,795s
sys     0m2,827s
```

Como corresponde, y de la manera antes explicada, realizamos un *profiling* de ese código, encontrando que la mayor parte del tiempo pasa en `proc_file`, particularmente procesando las cadenas del archivo para poder hacer luego los cálculos. Eso nos llevó a dos pequeñas optimizaciones: leer el archivo como bytes (para no convertir a Unicode antes de convertir a número) y ahorrarnos el `strip` ya que el `float` soporta que la cadena tenga un *newline* al final.

También, principalmente por motivos pedagógicos y relacionados con presentar la información en el libro, decidimos separar esa función en otro módulo, ya que es la que vamos a ir trabajando en el resto de las subsecciones. Entonces para las próximas iteraciones vamos a tener un sólo programa principal (`particulas-x.py`) al que le indicamos qué número de módulo usar, y los distintos módulos numerados; para el caso de las pequeñas optimizaciones que mencionábamos recién tendríamos el `particulas_mod_2.py`.

```
1 def proc_file(filepath, center_x, center_y, radius):
2     count = 0
3     with open(filepath, "rb") as fh:
4         for line in fh:
5             particle_x, particle_y, particle_radius = map(float, line.split(b","))
6             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
7             if dist_centers <= (radius + particle_radius) ** 2:
8                 count += 1
9     return count
```

Con estos cambios obtuvimos una mejora del 4%, claramente no es suficiente. El próximo cambio fue más disruptivo: en lugar de hacer los cálculos línea por línea, procedimos a cargar todo el archivo con Pandas (de la forma que ya explicamos 15.4) y hacer los cálculos a través de esa biblioteca:

³Es el `particulas.cpp`, que compilamos con `g++ -O2 -o particulas particulas.cpp`


```

1 import pandas as pd
2
3
4 def proc_file(filepath, center_x, center_y, radius):
5     df = pd.read_csv(filepath, header=None, names=["x", "y", "radius"])
6     df["included"] = (df.x - center_x) ** 2 + (df.y - center_y) ** 2 <= (df.radius + radius) ** 2
7     return df["included"].value_counts()[True]

```

```

$ time python3 particulas-x.py 3 superdir results.txt 2000 2000 100

real    1m24,140s
user    1m09,423s
sys      0m15,846s

```

Encontramos que esta versión es muchísimo más rápida, lo que nos lleva a una conclusión que también habíamos sacado en el capítulo sobre NumPy 11: si encontramos que nuestro problema ya está resuelto por una biblioteca, siempre es mejor usarla... va a estar más probada y estabilizada en el tiempo que nuestro propio código, y si abajo utiliza código compilado y optimizado será incluso más rápida que nuestro propio código en C o C++.

Programa	Tiempo	Mejora	Contra C++
C++ puro	2m11	NA	NA
Py 1: Primer intento	3m 16s	NA	+50 %
Py 2: Optimizado	3m 09s	-4 %	+44 %
Py 3: Pandas	1m 22s	-58 %	-37 %

Pero volvamos a código nuestro, porque en definitiva estamos aprendiendo cómo podemos optimizar código para el que no encontraremos bibliotecas que nos resuelven el problema.

En la secuencia que vimos llegamos al límite donde ateniéndonos puramente a Python fuimos mejorando la performance del código pero nos quedamos lejos de los tiempos de la versión en C++.

Es hora de ir un paso más allá.

8.2.2. Usando Mypyc

La herramienta Mypyc compila módulos de Python a extensiones en C.

El lenguaje compilado es una variante de Python en mayor o menor medida tipado, con lo cual gana en performance al restringir el uso de características dinámicas de Python (en la mayoría de los casos sin perder compatibilidad con el Python estándar).

Para definir el tipo de las distintas estructuras usa las sugerencias de tipado estándar de Python. La herramienta mypyc es parte de la biblioteca mypy, con lo cual aprovecha todo su poder de inferencia de tipos.

Por ejemplo, en nuestro código sólo hace falta algunas modificaciones mínimas:

```

1 def proc_file(filepath: str, center_x: float, center_y: float, radius: float) -> int:
2     count = 0
3     with open(filepath, "rb") as fh:
4         for line in fh:
5             particle_x, particle_y, particle_radius = map(float, line.split(b","))
6             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
7             if dist_centers <= (radius + particle_radius) ** 2:
8                 count += 1
9     return count

```

Vemos que sólo cambiamos la signatura de la función, indicando que el *path* del archivo es una cadena de texto y el resto de los parámetros punto flotante binario, mientras que el resultado es un entero. La biblioteca infiere que los valores leídos del archivo también son punto flotante binario (porque son el resultado de llamar a `float`), que el conteo es un entero (porque lo inicializamos en cero y siempre le sumamos un entero), etc.



mypy - Verificador de tipos estático para Python, que permite detectar errores de tipo en el código antes de su ejecución.

Es muy sencillo correr la herramienta sobre nuestro código: `mypyc particulas_mod_4.py`; vemos que mas allá de algunos mensajes de compilación el resultado es un archivo con una estructura específica: `particulas_mod_4.cpython-310-x86_64-linux-gnu.so`.

Este archivo es un módulo compilado que Python puede utilizar directamente, y la estructura de su nombre indica la posibilidad de su utilización (está compilado para CPython 3.10, en Linux, etc.). Este archivo puede convivir sin problemas con el `particulas_mod_4.py` original, Python utilizará automáticamente esta versión compilada si está presente.

```

$ time python3 particulas-x.py 4 superdir results.txt 2000 2000 100

real    2m59,327s
user    2m50,776s
sys      0m5,150s

```

Hay una ganancia marginal de tiempo, esta versión armada con `mypyc` tarda un 5 % menos que el código optimizado anterior. Podrá parecer poco, pero tengamos en cuenta que el costo de ir a esta solución es bajo, o incluso casi nulo si ya teníamos nuestro código con las sugerencias de tipado agregadas previamente.

Volviendo a la instrumentación del código, justamente, es importante entender si `Mypyc` tiene toda la información necesaria para optimizar el funcionamiento del mismo. Nuestro ejemplo es sencillo, pero así y todo hay varios objetos cuyo tipo debe inferirse, ya sea a partir de devolución de funciones o resultado de cálculos.

`Mypyc` provee una función que podemos utilizar para entender cómo está trabajando en ese sentido, `reveal_type`, que no tenemos que importar sino directamente usarla. Por ejemplo, si en la línea 9 de `particulas_mod_4.py` agregamos...

```

1     reveal_type(dist_centers)

```

... al correr `mypyc` vamos a ver esta nueva información como parte de la salida del programa:

```
particulas_mod_4.py:9: note: Revealed type is "builtins.float"
```

Alternativamente con el parámetro `--html-report DIR` podemos generar un reporte que nos indica qué porcentaje de nuestro código tiene el control de tipos; para nuestro ejemplo está cubierto al 100 %.

Para profundizar en esta herramienta les recomendamos no sólo ver [su documentación](#) sino también la del [módulo sobre tipado](#) de la biblioteca estándar.

8.2.3. Trabajando con Cython

Dando un paso más allá en el proceso de compilar parte del código nos encontramos con Cython.

Cython es un compilador tanto para Python en sí como para un lenguaje de programación propio de Cython que es parecido a Python (derivado de un viejo proyecto llamado Pyrex, de ahí la extensión que vamos a tener en nuestro archivo).

Con Cython entonces podemos ir ajustando código Python con definiciones de tipado estático que le sirven a Cython para lograr código compilado (de forma similar a lo que vimos con Mypyc) y también escribir código en Python que corra código C o C++ nativo, o integrarlo a bibliotecas y aplicaciones ya compiladas.

Esta transición resulta porque Cython es un lenguaje alrededor de Python mismo, que además soporta llamar a funciones en C y declarar tipos de C en variables y atributos de clase. Pero, al mismo tiempo, es código que no es perfectamente compatible con Python, por eso utilizamos la extensión `.pyx`.

Veamos cómo queda nuestro ejemplo:

```
1 def proc_file(str filepath, double center_x, double center_y, double radius):
2     cdef int count
3     cdef double particle_x, particle_y, particle_radius
4
5     count = 0
6     with open(filepath, "rb") as fh:
7         for line in fh:
8             particle_x, particle_y, particle_radius = map(float, line.split(b","))
9             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
10            if dist_centers <= (radius + particle_radius) ** 2:
11                count += 1
12    return count
```

Vemos que agregamos tipos tanto en la signature de la función como en las primeras dos líneas de la misma. Cuidado que esas dos primeras líneas no pueden estar en otro lado, tienen que figurar al principio de la función.

La forma más fácil de compilar Cython es utilizando un archivo de *setup*, veamos el `particulas-5-setup.py`:

```
1 from setuptools import setup
2 from Cython.Build import cythonize
```

```

3
4 setup(
5     name="Ejemplo particulas",
6     ext_modules=cythonize("particulas_mod_5.pyx", language_level="3", annotate=True),
7 )

```

La clave en esas líneas es llamar a `cythonize` como módulo externo, indicándole cual es el archivo fuente en cuestión, que es Python 3, y pidiendo con el `annotate` que genere un reporte (que veremos luego).

Le indicamos a este script que nos compile las extensiones (`python particulas-5-setup.py build_ext --inplace`) y eso nos deja un archivo muy similar al que vimos en la subsección anterior: `particulas_mod_5.cpython-310-x86_64-linux-gnu.so`.

Veamos cómo se comporta:

```

$ time python3 particulas-x.py 5 superdir results.txt 2000 2000 100

real    2m36,131s
user    2m32,264s
sys      0m2,806s

```

En este caso tenemos una mejora de performance del 17 % (comparado con el código optimizado original), pero todavía estamos lejos del código en C++ puro.

A diferencia de `Mypyc` donde los agregados al código igual nos dejaban algo que Python podía entender, con Cython no sólo salimos de esa situación sino que el código empieza a ser más verbosístico y menos legible, con lo cual lo ideal sería no declarar los tipos de las variables a menos que tengamos una buena razón. Al mismo tiempo, no obtenemos de Cython ningún aviso para aquellas variables que podrían haber sido tipadas y que le permite salir de la naturaleza tan dinámica de Python y generar código en C más simple y rápido.

La buena noticia es tenemos una herramienta que nos puede dar información para entender cómo Cython está traduciendo nuestro código a otro de más bajo nivel; el `annotate` que mencionamos arriba, cuando se compila el código, nos dejará un archivo reporte, un HTML que abrimos localmente y que nos mostrará el código original pintado y con un pequeño signo más al costado de cada línea.

El color utilizado en cada línea nos indica cuanto código en C se utilizó para reemplazar esa línea en Python; cuanto más fuerte sea el amarillo más líneas en C se utilizaron, perdiendo simpleza en el reemplazo de un lenguaje por el otro. El signo más nos permite visualizar justamente esas líneas.

En la siguiente captura de pantalla vemos el resultado para nuestro ejemplo. Noten que la línea 11 está “expandida”, y se muestra abajo el código correspondiente en C (que es trivial, por eso esa línea directamente está con fondo blanco).



Cython - Herramienta que facilita la escritura de extensiones en C para Python, permitiendo mejorar el rendimiento de los programas al compilar código Python en código C eficiente.

```

+01: def proc_file(str filepath, double center_x, double center_y, double radius):
02:     cdef int count
03:     cdef double particle_x, particle_y, particle_radius
04:
+05:     count = 0
+06:     with open(filepath, "rb") as fh:
+07:         for line in fh:
+08:             particle_x, particle_y, particle_radius = map(float, line.split(b","))
+09:             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
+10:             if dist_centers <= (radius + particle_radius) ** 2:
+11:                 count += 1
+11:                 __pyx_v_count = (__pyx_v_count + 1);
+12:     return count

```

Para terminar les recomendamos explorar [la documentación de Cython](#) para profundizar en su potencial.

8.2.4. Explorando Numba

Numba traduce funciones hechas en Python a código de máquina optimizado durante la ejecución misma.

No se necesita un paso extra de compilación ni tener un compilador de C o C++ instalado, sólo tenemos que aplicar un decorador en nuestra función y Numba hará el resto.

Numba está pensado para optimizar algoritmos numéricos, dónde puede terminar alcanzando velocidades de C o Fortran. Nuestro ejemplo se escapa un poco de ese área, y veremos que los resultados no son los mejores, pero decidimos contarles sobre esta herramienta por completitud.

La primera modificación al código optimizado original es importar `njit` (el compilador justo-a-tiempo de Numba) y aplicarlo como decorador en la función `proc_file`. Pero a diferencia de lo prometido por la documentación encontramos que teníamos que realizar cambios más profundos. Por un lado el `with` (administrador de contexto) no es soportado, ni tampoco si abrimos el archivo directamente con un `open`. Tampoco soportaba llamar a `map` ni a `float`.

En consideración de esas restricciones decidimos separar la función en dos partes, una que abre el archivo, lo lee y guarda en una lista todos los datos para operar luego, y otra función que lo único que hace es trabajar sobre esos números, a la que justamente decoramos con `njit`:

```

1 from numba import njit
2 from numba.typed import List
3
4
5 @njit
6 def _proc(data, center_x, center_y, radius):
7     count = 0
8     for particle_x, particle_y, particle_radius in data:
9         dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
10        if dist_centers <= (radius + particle_radius) ** 2:
11            count += 1
12    return count
13
14
15 def proc_file(filepath, center_x, center_y, radius):
16     with open(filepath, "rb") as fh:
17         data = List()
18         for line in fh:

```

```

19     particle_x, particle_y, particle_radius = map(float, line.split(b","))
20     data.append((particle_x, particle_y, particle_radius))
21     return _proc(data, center_x, center_y, radius)

```

Incluso habiendo ajustado el área de trabajo de Numba a la parte de nuestro código que hace exclusivamente procesamiento numérico, no tuvimos buenos resultados, encontrando que tarda el doble que nuestro código Python puro.

Evidentemente no estamos en el caso ideal de uso de Numba, pero les sugerimos leer [su documentación](#) para que exploren otras funcionalidades que no se aplican en nuestro caso.

8.2.5. Conclusiones de los distintos procesos

Luego de recorrer el uso de las distintas herramientas podemos ver que hay distintos mecanismos para optimizar código Python compilándolo todo o en parte, alejándose en mayor o menor medida del código Python original.

El resumen de los cambios de velocidad obtenidos es:

Programa	Tiempo	Mejora	Contra C++
C++ puro	2m11	NA	NA
Py 2: Optimizado	3m 09s	NA	+44 %
Py 4: Mypyc	2m 59s	-5 %	+37 %
Py 5: Cython	2m 37s	-17 %	+20 %
Py 6: Numba	6m 25s	+102 %	+193 %

Que haya diversas formas de acercarse a este tipo de solución significa que no hay una “bala de plata” que cubra todos los casos de la mejor manera posible, quedará para el público lector probar y utilizar la mejor herramienta según el proyecto que encare, algunas de las exploradas hasta ahora o incluso usando completamente código compilado (que veremos en la próxima sección).

Es de vital importancia entender que compilar el código no es una solución mágica. En general funciona porque minimizamos el dinamismo intrínseco de Python, llevando parte del procesamiento a la capa en C sin pasar por estructuras de Python más complejas, y esto muchas veces implica un recorte de la funcionalidad. Por ejemplo, cuando en Cython declaramos que nuestra variable `count` es ahora de tipo `cdef int`, estamos abandonando los enteros sin límites de Python y yendo a 32 o 64 bits dependiendo de la plataforma.

Debemos resaltar el mensaje original de hacer *profiling* antes de encarar cualquier optimización: tampoco vale la pena compilar código que no es parte de la “zona caliente” del programa.

Para cerrar esta parte, algunas palabras sobre [Nuitka](#), un compilador de Python escrito en Python que crea ejecutables que luego podrán correr sin necesitar un instalador por separado. Está más pensado para empaquetar el código junto a sus datos y el intérprete de Python en un sólo ejecutable que para acelerar la ejecución propiamente dicha. Para el caso puntual del ejemplo que manejamos en esta sección no mostró ninguna mejora en los tiempos.

8.3. Usando directamente código compilado

Hay veces que toda optimización es corta, siempre necesitamos mejor performance. O puede suceder que ya disponemos de una determinada librería escrita en C++, C, Fortran, etc., y necesitamos utilizarla pero al mismo tiempo no queremos armar *todo* nuestro programa en ese lenguaje.

En otras palabras, queremos hacer el 90 % del programa en Python aprovechando su alto nivel y facilidad de desarrollo pero también usar esa librería que ya tenemos hecha en otro lenguaje súper optimizado para la parte computacional crítica de nuestro sistema.

Es hora de conectar Python con C.

8.3.1. Extendiendo Python con C o C++

El intérprete de Python permite conectar el código hecho en ese lenguaje con código en C que maneja los objetos de Python mismo. Justamente el intérprete mismo del lenguaje hecho en C (que denominamos CPython) trabaja de esa manera.

El truco es que se expone una interfaz bien definida y muy estable en el tiempo para que se pueda trabajar desde y con otros programas hecho en estos otros lenguajes, un conjunto de funciones, macros y variables que dan acceso a la mayoría de los aspectos de Python en tiempo de ejecución. Este factor es parte del éxito de Python: históricamente siempre fue fácil utilizar desde Python otras librerías construidas con anterioridad, lo que fue clave para construir su extensa red de módulos disponibles (tanto dentro como fuera de la biblioteca estándar).

La API de Python entonces les da acceso a los programadores en C y C++ a una variedad de niveles del intérprete de Python. En nuestro caso usaremos sólo un pequeño subconjunto de toda esta funcionalidad, pero recomendamos involucrarse en [su documentación](#) para profundizar.



Aunque esta API es igualmente utilizable desde C++ por brevedad se la refiere generalmente como la Python/C API.

Hay dos formas de trabajar una unión entre Python y otros programas hechos en C.

Una es “extender” un programa hecho en Python con código en este otro lenguaje, o en otras palabras tener todo el programa en Python y utilizar alguna función o estructura hecha en C, generalmente como módulo o llamada a una biblioteca ya previamente compilada.

El otro caso es “integrar” Python en un programa o sistema más grande hecho en C. Es el caso común de programas que permiten ser utilizados programáticamente desde Python, tanto para automatizar tareas como para construir estructuras más o menos repetitivas que serían mucho trabajo de hacer a mano. Ejemplos de estos programas o sistemas son [Blender](#), una suite de creación en tres dimensiones, [Autodesk Maya](#), un software de modelado, animación y renderización orientado a los efectos visuales, o [Unreal Engine](#), utilizado para crear videojuegos y aplicaciones interactivas (en cada caso el enlace apunta justamente a la API de Python de cada programa).

En nuestro caso usaremos el mecanismo de “extensión”. Para profundizar en ambas funcionalidades el mejor punto de entrada es la [documentación oficial](#).

Volviendo a nuestro ejemplo, entonces, armaremos todo el módulo directamente en C++. Para ello reutilizaremos la función `proc_file` del programa que hicimos antes para tener una base de comparación de los tiempos. El único cambio que le haremos a esa función es renombrarla, le agregamos un guión bajo al final, ya que no la podemos usar directamente desde Python (y mantendremos el nombre original para aquella que sí).

Entonces copiamos esa función y las cabeceras correspondientes a un nuevo archivo `particulas-mod.cpp`. Pero antes de las cabeceras que copiamos, tenemos que poner:

```
1  #define PY_SSIZE_T_CLEAN
2  #include <Python.h>
```

El `Python.h` trae todos los símbolos necesarios, siempre con los prefijos `Py` o `PY`, y es importante que esté junto con esa definición bien al principio del archivo porque afectan el resto de los `include`s).

No tiene sentido entrar en el detalle de la función que trajimos del otro código pero veamos su signatura:

```
1  int _proc_file(const std::string& filepath, double center_x, double center_y, double radius) {
2      int count = 0;
3      (...)
4      return count;
5  }
```

Vemos que renombramos la función y debemos prestar atención a los tipos de datos de los argumentos que recibe y lo que devuelve, para cuando escribamos la función que la envuelve:

```
55 static PyObject *
56 proc_file(PyObject *self, PyObject *args, PyObject *kwargs)
57 {
58     const char *filepath;
59     double center_x, center_y, radius;
60     int count;
61
62     const char *kwlist[] = {"filepath", "center_x", "center_y", "radius", NULL};
63
64     if (!PyArg_ParseTupleAndKeywords(
65         args, kwargs, "sddd", const_cast<char *>(kwlist),
66         &filepath, &center_x, &center_y, &radius))
67         return NULL;
68
69     count = _proc_file(filepath, center_x, center_y, radius);
70
71     return PyLong_FromLong(count);
72 }
```

Allí tenemos la definición de `proc_file` que indica que devuelve “un objeto de Python” y recibe “argumentos” de forma genérica (tanto posicionales como nombrados... como si hiciésemos `*args`, `**kwargs` en código Python). Luego tenemos definiciones de variables y la lista de nombres de los argumentos a recibir (porque pueden ser nombrados, justamente).

Todo el procesamiento de los argumentos recibidos lo hace `PyArg_ParseTupleAndKeywords`, a la que le pasamos `args` y `kwargs` (los argumentos que efectivamente recibe la función), qué esperamos de ellos como forma de una cadena de texto (“s” para una cadena, tres “d” para tres *doubles*), la lista de nombres de los argumentos, y los punteros a las variables que definimos arriba y donde efectivamente quedarán guardados los valores recibidos para poder usarlos.

Si `PyArg_ParseTupleAndKeywords` termina correctamente tendremos efectivamente en esas variables los valores que se le pasaron a la función, y lo único que hacemos es llamar a la `_proc_file` que hace el procesamiento real. Esta función devuelve el conteo que es un `int` a nivel de C, debemos construir un objeto entero a nivel de Python (llamado `PyLong` por razones históricas) para devolverlo a esa capa.

Por otro lado, si `PyArg_ParseTupleAndKeywords` devuelve cero es que hubo un problema en la interpretación de los argumentos recibidos. En ese caso sólo debemos devolver `NULL` porque todos los objetos correspondientes a “generar una excepción” a nivel de Python ya están en su lugar.

A esta altura ya tenemos una función que podemos usar directamente desde afuera sin cambiar ese código externo. Pero todavía no tenemos un módulo propiamente dicho.

Para convertir nuestro código en C++ en un módulo utilizable por Python el punto de entrada es una función de inicialización que va a crear justamente el módulo como objeto de Python:

```

90 PyMODINIT_FUNC
91 PyInit_particulas_mod_7(void)
92 {
93     return PyModule_Create(&particulasmodule);
94 }
```

Vemos que es un paso necesario pero trivial: toda la definición del módulo está en esa estructura `particulasmodule`:

```

81 static struct PyModuleDef particulasmodule = {
82     PyModuleDef_HEAD_INIT,
83     "particulas_mod_7",
84     NULL,
85     -1,
86     ProcFileMethods
87 };
```

Esta estructura puede tener más valores pero sólo estos cinco son requeridos: el primero con un valor fijo `PyModuleDef_HEAD_INIT`, luego el nombre del módulo, el *docstring* del módulo (o `NULL` si no queremos indicar ninguno), y un valor entero que indica la memoria necesaria para contener el estado del módulo en cada subintérprete (para nuestro ejemplo alcanza con poner -1 para indicar que tiene un estado global, pero no podrá ser usado en diferentes subintérpretes), y finalmente una lista de los métodos/funciones que tenemos en el módulo, `ProcFileMethods`, que también definimos nosotros:

```

75 static PyMethodDef ProcFileMethods[] = {
76     {"proc_file", (PyCFunction)proc_file, METH_VARARGS | METH_KEYWORDS, "Process a file."},
```

```

77     {NULL, NULL, 0, NULL}
78 };

```

Aquí tenemos una lista de todas las funciones definidas en el módulo (terminadas por una guarda, al final). En nuestro caso sólo tenemos una función, y para ella indicamos el nombre con que estará expuesta la función, el puntero a la implementación, que tipo de argumentos reciben (construido como mapa de bits, METH_VARARGS para argumentos posicionales y METH_KEYWORDS para argumentos nombrados), y luego el *docstring* de esa función.

No necesitamos definir nada más para nuestro ejemplo. Obviamente para utilizar ese código debemos compilarlo. La forma más fácil es utilizar `setuptools` (de manera muy similar a lo que hicimos con Cython).

```

1 from setuptools import Extension, setup
2
3 setup(
4     ext_modules=[
5         Extension(
6             name="particulas_mod_7",
7             sources=["particulas-mod.cpp"],
8         ),
9     ]
10 )

```

Luego lo ejecutamos igual que lo que ya habíamos visto, `python particulas-7-setup.py build_ext --inplace`, lo cual nos dejará un archivo con la estructura acostumbrada: `particulas_mod_7.cpython-310-x86_64-linux-gnu.so`.

Como respetamos las interfaces con las que veníamos trabajando, podemos usarlo directamente:

```

$ time python3 particulas-x.py 7 superdir results.txt 2000 2000 100

real    2m14,961s
user    2m08,996s
sys     0m2,899s

```

Comparemos estos resultados con los anteriores:

Programa	Tiempo	Mejora	Contra C++
C++ puro	2m11	NA	NA
Py 2: Optimizado	3m 09s	NA	+44 %
Py 5: Cython	2m 37s	-17 %	+20 %
Py 7: Extendido en C	2m 15s	-29 %	+3 %

Como vemos, obtenemos una performance equivalente a haber hecho todo el programa en C++ pero sólo con la función que realmente era responsable de la mayor parte del tiempo de ejecución. En realidad sólo llevamos hasta el extremo lo que habíamos empezado con Mypyc y

Cython: ahora hicimos completamente el módulo en C++, a mano, minimizando también el costo del pasaje entre Python y C++.

Siempre va a ser menos trabajo para la persona que desarrolla un programa el hacer toda la parte de alto nivel en Python (manejo de archivos, interfaz de usuario, generación de reportes, etc.) y usar C++ para la parte crítica en velocidad, que realizar absolutamente todo el programa en C++. Y como vimos podemos terminar teniendo una performance equivalente.

Entonces, como conclusión general, recomendamos fuertemente esta forma de trabajo: arrancar con el programa en Python, si es lento hacer *profiling*, y optimizar eso. Y si sigue siendo lento utilizar algunas de las herramientas de compilado automático del código, y de última construir la parte crítica en C o C++ (a menos que esta parte ya esté construida desde antes, que es de lo que habla la siguiente subsección).

Primero hacerlo bien, luego hacerlo rápido (es mucho mejor que hacerlo rápido y después tratar de que funcione correctamente).

Y siempre, en todos estas refactorizaciones de código (como en cualquier otra que hagamos mientras programamos), lo recomendable es tener pruebas de unidad para validez que luego de cada cambio que hagamos sigamos teniendo un sistema funcionando correctamente.

8.3.2. Usando directamente código previamente compilado

Por otro lado, si lo único que necesitamos es utilizar una librería ya hecha, ¡ni siquiera hace falta crear código intermedio! Podemos usar código previamente compilado directamente desde Python.

La performance del programa no es la única razón que podemos tener para tomar este camino, también puede pasar que tengamos un código compilado pero no las fuentes (muy normal si usamos software comercial o con restricciones de copyright), es un subproducto de otro proceso, código nuestro del que tenemos las fuentes pero su proceso de compilación es complicado y lleva tiempo, etc.

En esta subsección veremos entonces dos maneras de acceder a bibliotecas previamente compiladas, que normalmente encontramos en Linux con la extensión `.so` (por *shared object*) o en Windows como `.dll` (por *dynamic link library*).

Como ejemplo usaremos la *GNU Scientific Library* (“GSL”), una biblioteca de procesamiento numérico para programadores C y C++. Posee más de mil funciones en total ([más info aquí](#)), pero para nuestro caso buscamos algo relativamente sencillo, `gsl_hypot`, una función que calcula el valor de $\sqrt{x^2 + y^2}$ en una manera que evita el *overflow*.

Obviamente tenemos que tener acceso a dicha biblioteca ya compilada; este binario lo tenemos en la mayoría de las distribuciones de Linux; al momento de escribir el libro, usando el sistema operativo Ubuntu, la encontramos en `/usr/lib/x86_64-linux-gnu/libgsl.so.27`.

La primer herramienta que veremos es el módulo `ctypes` de la biblioteca estándar de Python. Como estamos en Linux usaremos la función `cdll` (por *C dynamic library loader*), pero cuidado que en Windows se deben utilizar `windll` u `oledll`.

```
1 >>> from ctypes import cdll
2 >>> libgsl = cdll.LoadLibrary("libgsl.so.27")
3 >>> libgsl
```

```
4 <CDLL 'libgsl.so.27', handle 564375d67b50 at 0x7ff8355a44c0>
```

Con dicha función cargamos la biblioteca que vamos a usar como ejemplo, y vemos que tenemos una instancia de CDLL, que representa la biblioteca ya cargada.

Ya mencionamos que íbamos a operar con una función específica dentro de esta biblioteca:

```
1 >>> libgsl.gsl_hypot
2 <_FuncPtr object at 0x7ff83544d900>
```

Vemos que no tenemos un “objeto función” de Python, sino un puntero a una función en C; es evidente que aunque operemos desde Python estamos trabajando directamente con el código compilado. Relacionado con eso, tenemos que tener el cuidado que las funciones en CDLL usan la convención estándar de C para ser llamadas, y se asume que van a devolver enteros (int).

¿Cómo aplica eso a nuestra función ejemplo? Para saberlo tenemos que buscar su signatura; la encontramos como parte de la documentación mencionada arriba ([puntualmente acá](#)), donde vemos:

```
double gsl_hypot(const double x, const double y)
```

¿Podemos pasarle a la función objetos float de Python y recibir eso mismo como respuesta? No directamente: todos los tipos de Python (excepto enteros, cadenas y bytes) tienen que ser envueltos en su tipo específico de ctypes para que lleguen como un tipo de datos de C a la función:

```
1 >>> from ctypes import c_double
2 >>> libgsl.gsl_hypot(c_double(5.2), c_double(3.1))
3 -858993459
```

Pero cuidado, ¡ese resultado es incorrecto! Recuerden que habíamos mencionado que se asumía que la respuesta es un entero, lo cual no aplica en este caso. Debemos especificarle a la función qué tipo de dato esperamos como resultado (para que lo interprete correctamente):

```
1 >>> libgsl.gsl_hypot.restype = c_double
2 >>> libgsl.gsl_hypot(c_double(5.2), c_double(3.1))
3 6.053924347066125
```

ctypes es una herramienta poderosa, y nos permite trabajar directamente en bajo nivel con las bibliotecas compiladas, pero debemos tener cuidado de estar interpretando bien en todo momento la información ya que podemos tener resultados equivocados (o si los errores son mayores, incluso provocarle un *crash* al intérprete).



En computación el *crash* (término en inglés que también usamos en castellano) ocurre cuando un programa deja de funcionar apropiadamente y termina, la mayoría de las veces de forma no controlada y sin proveer mayor información sobre el problema sucedido.

La segunda herramienta que vamos a ver mejora mucho esa experiencia, ya que es capaz de usar la declaración de interfaz de la función y realizar automáticamente las conversiones necesarias. Se llama CFFI (por *C Foreign Function Interface*, “interfaz para funciones foráneas en C”) pero sirve también para trabajar con otros lenguajes, como Fortran.

Luego de instalar `cffi` desde PyPI, vemos que su utilización es muy similar al de `ctypes`, con la gran diferencia de que le especificamos la interfaz y luego la usamos directamente:

```

1  >>> from cffi import FFI
2  >>> ffi = FFI()
3  >>> ffi.cdef("""
4  ...     double gsl_hypot(const double x, const double y);
5  ... """)
6  >>> libgsl = ffi.dlopen("libgsl.so.27")
7  >>> libgsl.gsl_hypot(5.2, 3.1)
8  6.053924347066125

```

CFFI también nos permite ir un paso más allá y armar directamente un módulo compilado accesible desde Python, con acceso a esa biblioteca (para lo que tenemos que tener acceso a las fuentes de la biblioteca; en Debian/Ubuntu las instalamos con `sudo apt install libgsl-dev`). El procedimiento normal es tener algún pequeño script que realice la compilación, ya que luego esto se puede incorporar al mecanismo de distribución de nuestro programa; para nuestro ejemplo:

```

1  from cffi import FFI
2  ffi = FFI()
3  ffi.cdef("""
4  ...     double gsl_hypot(const double x, const double y);
5  ... """)
6
7  ffi.set_source("gsl", """
8  ...     #include "gsl/gsl_math.h"    // the C header of the library
9  ... """, libraries=['gsl'])
10
11 ffi.compile(verbose=True)

```

Al correr ese script nos deja un `gsl.cpython-310-x86_64-linux-gnu.so` similar a los que ya conocemos de las secciones anteriores. Lo usamos directamente:

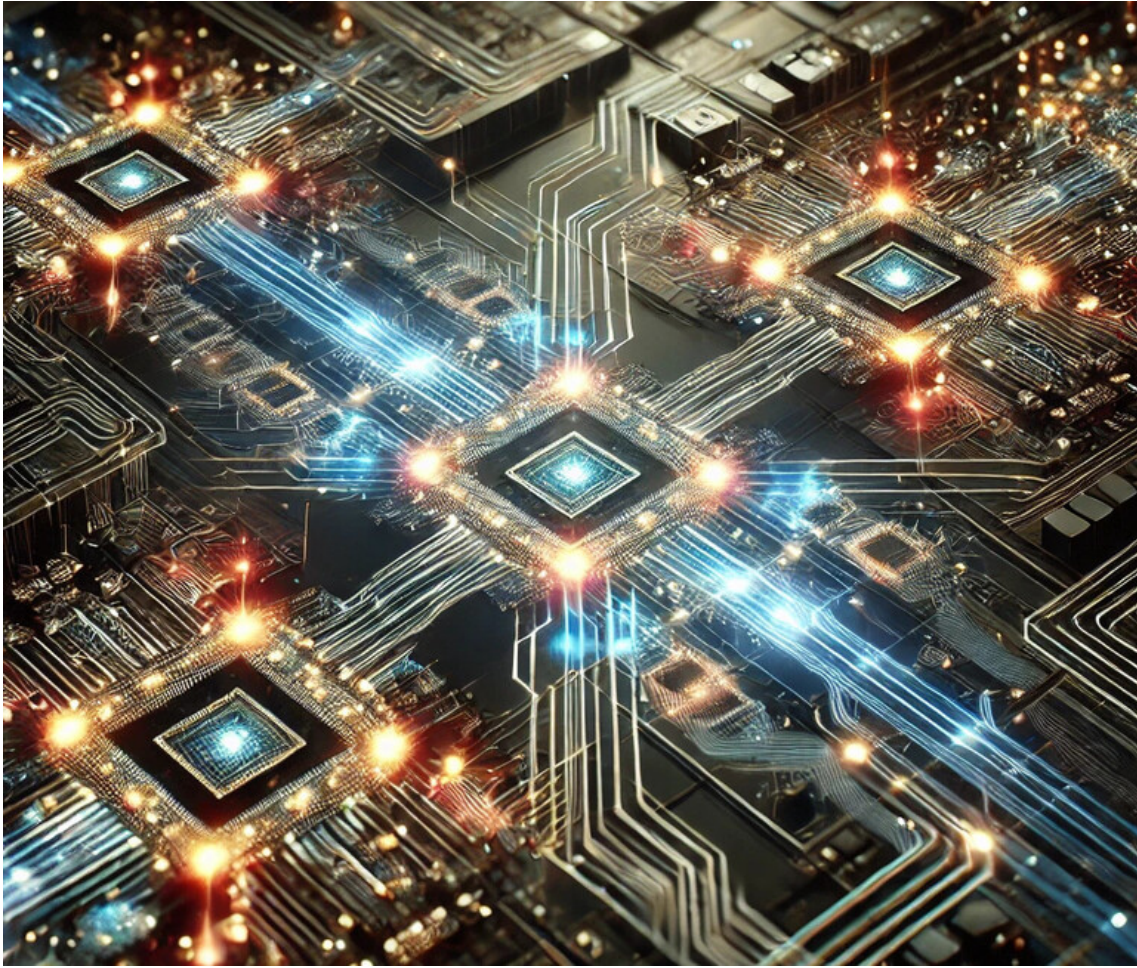
```

1  >>> import gsl
2  >>> gsl.lib.gsl_hypot
3  <built-in method gsl_hypot of _cffi_backend.Lib object at 0x7f1c04a47240>
4  >>> gsl.lib.gsl_hypot(5.2, 3.1)
5  6.053924347066125

```

Como ya habíamos declarado la interfaz antes, CFFI armó automáticamente la capa que permite a nuestro código en Python utilizar el código ya compilado, entonces el uso es directo.

Esto contrasta con `ctypes` ya que, aunque tengamos que armar ciertas estructuras para tener una interfaz compilada, luego en el uso propiamente dicho no hay ningún detalle que necesitemos considerar.



9 | Procesamiento en paralelo

Desde la invención de la computadora nos hemos acostumbrado a procesar más y más información a través de las mismas. Este procesamiento no sólo fue aumentando en cantidad, sino también en complejidad.

Hay dos formas de poder aumentar la capacidad de procesamiento, de forma “vertical” u “horizontal”, términos que son simples metáforas para expresar en el primer caso el aumento de capacidad del procesador en sí (por ejemplo, aumentando su frecuencia de reloj, o memoria *cache* interna), o en el segundo caso hacer referencia a poner varios procesadores uno al lado del otro y distribuir el procesamiento entre ellos.

A esto último lo llamamos “procesamiento en paralelo”, y puede hacer referencia tanto a distintos *cores* (o núcleos de silicio) dentro del mismo procesador, distintos procesadores en una computadora, distintas computadoras en un centro de datos, distintos centros de datos, etc.

Cuando el procesamiento de la información implica esperar eventos fuera del procesador (de forma genérica “entrada o salida”, que puede ser escribir en disco, consultar un dato a través

Módulo	Versión
aiohttp	3.10.5
matplotlib	3.9.2
numpy	1.26.4
scipy	1.14.1

[Código disponible](#)

de la red, etc.) es útil otro concepto que implica repartir el mismo *core* o procesador en distintos procesamientos, aprovechando los “tiempos muertos” de ellos. Aunque con este método logramos concurrencia, no es realmente procesamiento en paralelo.

En las siguientes secciones nos adentraremos en estos conceptos, primero definiremos más extensivamente qué es la concurrencia y mostraremos dos formas de lograrla (a través de “hilos” (o *threads*) y de procesamiento “asincrónico”. La segunda sección versa sobre el procesamiento paralelo real, mostrando distintas técnicas y formas de lograrlo.

9.1. ¿Qué es la concurrencia?

La “concurrencia” es la composición de procesos ejecutados independientemente, logrando un efecto similar a que esos procesos se sucedan al mismo tiempo sin que realmente eso suceda. La concurrencia **no es paralelismo** (que sería la ejecución simultánea real).

¿Hay un recurso que se agota? Necesitamos paralelismo. ¿Pero podemos manejar distintos recursos con un mismo controlador? En este caso podemos usar concurrencia.

Pongamos un ejemplo para diferenciar mejor un caso del otro. Supongamos una cocina de un restaurant.

Hay que lavar muchísimos platos por hora, y un lavaplatos no alcanza, aunque esté lavando todo el tiempo. No nos queda otra que poner varios lavaplatos en paralelo, muchas personas lavando platos simultáneamente, una al lado de la otra. Esto es paralelismo.

Por otro lado, tenemos un par de hornos, una freidora, varias hornallas, algún microondas; como a cada una de esas unidades hay que esperarlas, podemos tener una persona (*le chef*) manejando todo eso, ya que por ejemplo se puede poner algo en el microondas y mientras este trabaja, la persona puede retirar algo del horno. Esto es concurrencia.

Llevándolo de nuevo para el lado de la computación, para paralelismo necesitamos varias unidades de ejecución (las distintas personas lavaplatos en las distintas bachas), mientras que la concurrencia se logra administrando qué proceso entra en ejecución (horno, freidora) en el mismo procesador (chef).

En Python la única forma de usar paralelismo real es con múltiples procesos (vamos a charlar de esto más adelante en la sección de procesamiento en múltiples procesadores 9.4), y aunque tiene toda su complejidad no hay otro mecanismo conceptual que explicar. Por el contrario, para que obtengamos concurrencia, corriendo muchos procesos en el mismo procesador, alguna “magia” tiene que haber, y vamos a proceder a explicarla.



Antes de seguir, es momento de aclarar que en todo este contexto, cuando decimos procesador más bien nos referimos a “unidad de ejecución”. Los primeros procesadores tenían una sola unidad de ejecución ^a, mientras que los procesadores más modernos ofrecen distintos núcleos o *cores*. Para todo lo que estamos hablando es más fácil pensarlo como unidades de ejecución distintas, ya sean uno o varios procesadores, uno o varios cores.

^a Si vamos a ser estrictos, también hay distintas partes que ejecutan distintas cosas en los procesadores no tan modernos, como la unidad aritmético-lógica, pero la idea es hablar del procesador genéricamente.

Entonces, ¿cómo logramos concurrencia? Si tenemos un sólo procesador o unidad de ejecución, ¿quién se encarga de administrar qué proceso debe ejecutarse en cada momento?

En el modelo clásico de múltiples procesos (o hilos, que son una versión especial de procesos, como veremos más adelante 9.2), hay una combinación de funcionalidad entre el procesador y el sistema operativo específicamente diseñada para lograr este efecto. Los procesadores tienen implementados¹ mecanismos que el sistema operativo aprovecha, entonces, de manera que si un proceso necesita servicios del sistema operativo porque tiene que realizar entrada o salida, o pedir o liberar memoria, o cualquier llamada al sistema (o *syscall*), el procesador mismo dejará de ejecutar dicho proceso y pasará a ejecutar al sistema operativo, para que sirva esa llamada.

Esta no es la única manera en que un proceso puede ser “sacado” del procesador. También puede ser por cualquier interrupción del sistema que tenga que ser atendida por el sistema operativo (por ejemplo, porque llegó información por la red y debe ser leída), o incluso porque hace mucho tiempo que el proceso está ejecutándose. De cualquier manera, al proceso se lo saca del procesador *forzadamente*, sin que este haga nada para ser sacado, y en cualquier punto de ejecución del mismo. Este modelo se llama multitarea apropiativa (del inglés *preemptive multitasking*).

En ambos casos hay un costo muy importante a considerar, que es el de cambio de contexto (en inglés *context switch*), ya que se deben adecuar muchas estructuras dentro del procesador para que entre o salga un proceso cualquiera o el sistema operativo, lo cual puede implicar accesos a memoria o incluso a disco. El otro costo en este modelo es que los procesos o hilos ocupan memoria en mayor o menor medida. La suma de ambos hace que un sistema no escale en cantidad de procesos o hilos como quisiéramos: está bien cuando hay 10 trabajando, o 100, pero al crecer en cantidad en algún momento los cambios de contexto y la ocupación de memoria van a hacer que el sistema se vuelva ineficiente.

El otro modelo para lograr concurrencia, totalmente distinto al anterior, es el asincrónico. En este caso no tenemos múltiples procesos o hilos para lograr el efecto deseado, sino algo que se llama “corrutina”, que no es más que una función del programa, pero con la diferencia conceptual que sabemos que estas funciones/corrutinas van a estar ejecutándose de forma concurrente en el mismo proceso (alguna en un momento, otra luego, otra o la anterior más adelante). Todo esto está manejado por otra sección del código que se llama *reactor* o *event loop*, el cual normalmente no necesitamos implementar sino que usamos algún “framework asincrónico” que nos lo provea.

Este reactor entonces va a manejar los recursos según necesiten ser atendidos. Si una corru-

¹ No todos, claro, por ejemplo en la línea x86 de Intel esta funcionalidad recién apareció en el modelo 80386 (año 1985).

tina está esperando información de disco, el reactor ejecutará otra mientras tanto, y cuando esa información esté disponible continuará la corrutina anterior.

Todo esto sucede dentro del mismo proceso (con lo cual no existen los cambios de contexto que mencionábamos para el caso de multitarea apropiativa). Pero entonces ¿cómo es que se puede alternar entre las distintas corrutinas y el reactor, si no hay nada que “interrumpa”? De una única manera: la corrutina tiene que “soltar” el procesador cuando tiene que esperar que suceda algo externo (el *cómo* se hace eso en el código depende exclusivamente del framework asincrónico que estemos usando). Es por eso que este modelo se llama “cooperativo”, porque cada función/corrutina tiene que estar escrita de forma de cooperar con el framework asincrónico.

Tengamos en cuenta que todo lo explicado hasta ahora no es específico de Python, sino genérico a toda la informática. En las siguientes secciones profundizaremos un poco en cada concepto y sí entraremos en cómo utilizar ambos modelos desde Python.

Para facilitar el entendimiento práctico mostraremos en cada caso la versión con concurrencia del siguiente ejemplo integrador, que ejecuta la típica tarea de bajar distintos archivos de un servidor:

```
1 from urllib.request import urlopen
2
3 BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
4 TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]
5
6
7 class Adder:
8     def __init__(self):
9         self.total = 0
10
11     def add(self, value):
12         self.total += value
13
14
15 def downloader(tex_name, adder):
16     url = BASE + tex_name
17     u = urlopen(url)
18     length = len(u.read())
19     adder.add(length)
20
21
22 adder = Adder()
23 for tex_name in TEX_NAMES:
24     downloader(tex_name, adder)
25 print("Done:", adder.total)
```

En este código secuencial encontramos a la función `downloader` que baja un archivo, calcula el largo, y llama a un acumulador para ir guardando el total. Por fuera (desde la línea 22) instanciamos el acumulador, iteramos una lista de nombres de archivos `.tex`, llamamos al `downloader` por cada archivo, y finalmente mostramos el total. Este pequeño programa tiene una estructura que a priori suena excesiva, pero es para minimizar las diferencias con los códigos posteriores, donde tendremos los nuevos conceptos a aprender.

```
$ python3 concur_no.py
Done: 288998
```

9.2. Threading

Los hilos (en inglés llamados *threads*) son una especie de procesos livianos que comparten entre sí la mayor parte de las estructuras internas, particularmente la memoria a la que tienen acceso. El concepto de “hilo” a nivel de sistemas operativos es casi tan antiguo como el de “proceso”, recomendamos [este artículo de Wikipedia](#) para más información.

La gran ventaja de los hilos es que nos permite lograr concurrencia o paralelismo de forma sencilla, aunque implica tener algunos cuidados que mencionaremos más adelante. Hagamos un ejemplo sencillo para explicar mejor algunos conceptos de forma más tangible.

```
1 import random
2 import threading
3 import time
4
5 waits = []
6
7
8 def waiter():
9     wait = random.randint(0, 100)
10    time.sleep(wait / 100)
11    waits.append(wait)
12
13
14 all_threads = []
15 for _ in range(5):
16     th = threading.Thread(target=waiter)
17     th.start()
18     all_threads.append(th)
19
20 for th in all_threads:
21     th.join()
22
23 print(waits)
```

En este código tenemos una función `waiter` que calcula un número al azar entre 0 y 100, duerme esa cantidad de milisegundos, y agrega el valor a una lista global. Esta función es la que ejecutaremos en distintos hilos, de forma concurrente. Es clave en este ejemplo la diferencia entre hilos y procesos, ya que la función accede a una lista global, y eso lo puede hacer porque los hilos comparten el espacio de memoria entre sí (incluyendo al “hilo principal” que los genera); si hubiésemos ejecutado la función en distintos procesos, este modelo no serviría porque la función accedería a *copias* de `waits`, no logrando el efecto que buscamos.

Luego tenemos un bucle `for` en el que creamos un nuevo hilo (indicando que la función a ejecutar en ese hilo es `waiter`), arrancamos dicho hilo, y lo agregamos a una lista que necesitaremos luego.

Antes de terminar nuestro proceso necesitamos esperar a que los hilos generados terminen. Para eso llamamos a `.join` en cada hilo, lo cual bloqueará el hilo principal hasta que ese hilo se le una. Luego finalmente mostramos las esperas que se sucedieron.

```
$ python3 ej_hilos_simple.py
[77, 97, 6, 9, 59]
```

Si de ese código eliminamos el segundo bucle con los `.join` a cada hilo, la lista mostrada estará vacía porque no esperamos a los hilos que terminen². Les dejamos este cambio para que experimenten ustedes.

En sistemas con más de un procesador (lo cual es normal en estos días), los hilos pueden ejecutarse en distintos procesadores, según determine el sistema operativo. En lenguajes como C o C++ esto implica que los distintos hilos *podrían* ejecutarse simultáneamente, logrando efectivamente paralelismo. En Python³ esto no es así, ya que incluso cuando los hilos se ejecutan en distintos procesadores, por como trabaja internamente el lenguaje esos hilos se ejecutarán no simultáneamente.

Expliquemos eso un segundo. Python tiene determinadas estructuras internas (en especial el conteo de referencias a los objetos, parte de la administración dinámica y automática de la memoria⁴) que no pueden modificarse en cualquier momento desde cualquier hilo: para evitar que dos hilos accedan simultáneamente a esas estructuras se bloquea su uso de una forma particular (veremos estos conceptos con detalle más adelante).

Si ese bloqueo y desbloqueo sucede cada vez que se acceden a las estructuras mencionadas, su utilización sería demasiado ineficiente, especialmente en el caso en que la aplicación no esté usando hilos. Python resuelve esto teniendo un bloqueo global a todo el intérprete (*Global Interpreter Lock*, o “GIL”), que es mucho más eficiente, pero tiene como resultado colateral que los distintos hilos no se ejecutarán simultáneamente, incluso cuando estén en distintos procesadores.

Esto no es para nada un problema en programas que no usan hilos (la mayoría), o incluso en programas con muchos hilos que se basan en esperar entradas/salidas (acceso a disco, a la red, o en general a cualquier periférico o dispositivo fuera del procesador), pero sí es una limitación en programas *cpu-bound*⁵. Por eso muchas bibliotecas de procesamiento numérico escritas en lenguajes de bajo nivel (como NumPy), liberan el GIL al ejecutarse, logrando el paralelismo real que mencionábamos antes.

²En realidad, eso tampoco es 100 % determinístico, porque como mencionábamos antes, en la multitarea apropiativa no tenemos control sobre en qué momento los hilos se ejecutan o dejen de ejecutarse (a menos que usemos herramientas de sincronización como *locks* o “semáforos”), entonces puede suceder que en alguna ejecución algún hilo lanzado si haya terminado antes que el hilo principal.

³Estos conceptos de bajo nivel y otros explicados luego como el GIL son inherentes a cPython (la implementación de Python en C), otras implementaciones pueden variar.

⁴Python maneja la memoria de forma dinámica basándose en el conteo de referencias: por cada objeto creado Python lleva la cuenta de todas las referencias a ese objeto. Cuando esas referencias se pierden (el contador baja a cero), Python libera la memoria usada por ese objeto.

⁵Se denominan *cpu-bound* a aquellos programas que usan (casi) exclusivamente el procesador, sin tener ninguna entrada o salida, por ejemplo haciendo cálculos matemáticos, procesamiento de imágenes, manejo de datos científicos, etc.

9.2.1. Usando hilos

La forma más sencilla de ejecutar una función en un hilo distinto es la que mostramos arriba, donde creamos al hilo indicándole dicha función (y opcionalmente que argumentos pasarle al ejecutarla).

Hay otra forma que no es tan directa pero nos permite armar estructuras más complejas y flexibles, que es escribir una clase que hereda de `threading.Thread`.

```

1 import random
2 import threading
3 import time
4
5
6 class Waiter(threading.Thread):
7
8     waits = []
9
10    def run(self):
11        wait = random.randint(0, 100)
12        time.sleep(wait / 100)
13        self.waits.append(wait)
14
15
16 all_threads = []
17 for _ in range(5):
18     th = Waiter()
19     th.start()
20     all_threads.append(th)
21
22 for th in all_threads:
23     th.join()
24
25 print(Waiter.waits)

```

Ahora encapsulamos nuestra funcionalidad dentro de la clase, la cual debe heredar `threading.Thread` para ser un hilo. El código en si es bastante parecido a la versión más simple, porque son ejemplos sencillos, pero tenemos que darnos cuenta que pasamos de tener sólo una función a poder aprovechar todo el potencial de la programación orientada a objetos.

Cuando le hagamos `.start()` al hilo, luego de correr alguna maquinaria interna terminará ejecutando el método `.run` que es el punto de entrada a nuestro código; el hilo termina cuando este método termina. El único cuidado que tenemos que tener es que si definimos el método `__init__` en nuestra clase debemos recordar de llamar al método de inicialización de su clase padre `Thread`, caso contrario no estará listo para arrancar con el `.start`.

Volviendo a la forma más directa de usar hilos, veamos nuestro ejemplo integrador (que baja archivos y suma sus longitudes) pero adaptada a esta forma de usar hilos:

```

1 import threading
2 from urllib.request import urlopen
3
4 BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
5 TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]

```

```
6
7
8 class Adder:
9     def __init__(self):
10         self.total = 0
11
12     def add(self, value):
13         self.total += value
14
15
16 def downloader(tex_name, adder):
17     url = BASE + tex_name
18     u = urlopen(url)
19     length = len(u.read())
20     adder.add(length)
21
22
23 adder = Adder()
24 all_threads = []
25 for tex_name in TEX_NAMES:
26     th = threading.Thread(target=downloader, args=(tex_name, adder))
27     th.start()
28     all_threads.append(th)
29
30 for th in all_threads:
31     th.join()
32 print("Done:", adder.total)
```

El código es muy similar al secuencial, sólo que ahora en vez de llamar a `downloader` directamente, la ejecutamos dentro de un hilo separado (y sí, eso implica que hay que crear el hilo, arrancarlo, guardarlo para luego esperarlo... no podemos escaparnos de este pequeño costo de infraestructura).

Si lo ejecutamos, notamos que es bastante más rápido que el secuencial, y (a priori, ver luego) nos da el mismo resultado:

```
$ time python3 concur_no.py
Done: 288998

real    0m0,824s
user    0m0,108s
sys     0m0,011s
$ time python3 concur_hilos.py
Done: 288998

real    0m0,245s
user    0m0,065s
sys     0m0,024s
```

9.2.2. Modificando estructuras en sistemas multithreading

Más allá de la forma utilizada (llamando a las funciones en un hilo, o escribiendo nuestros hilos con clases), siempre debemos tener en cuenta que no tenemos control de cuando se ejecutan

cada uno de los hilos creados, y debemos modificar nuestros programas para hacerlos seguros con respecto a eso.

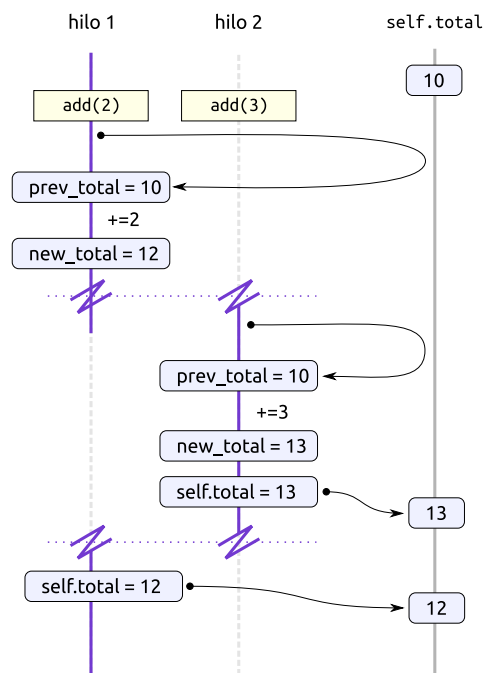
Y eso no es trivial. Aparece la gran trampa de usar hilos en nuestros programas (más allá del lenguaje utilizado): es muy difícil pensar y darse cuenta en todo el código de los potenciales problemas al no tener control de qué hilo se está ejecutando.

Analicemos en detalle, por ejemplo, el programa que mostramos antes. Tiene un punto de falla en el método `Adder.add`, donde hace `self.total += value`. Para entender mejor por qué, “abramos” esa línea en los tres pasos que realmente suceden:

```
1  def add(self, value):
2      prev_total = self.total
3      new_total = prev_total + value
4      self.total = new_total
```

Recorramos de forma detallada una secuencia que “explote” el problema. Supongamos que tenemos un valor de 10 en `self.total`, y desde dos hilos se llama a ese método con los valores 2 y 3. Si se ejecutaran en momentos distintos, el resultado final sería el correcto, 15.

Pero supongamos una ejecución particularmente problemática, mostrada en el esquema a continuación. En este caso vemos que empieza ejecutándose el primer hilo, guarda el total hasta ese momento y le suma lo que recibió, quedando `new_total` en 12. En ese momento su ejecución se interrumpe y entra el otro hilo, que lee 10 de `self.total` (porque todavía no cambió), le suma 3, y termina dejando `self.total` en 13. Entra nuevamente el primer hilo, continúa con su proceso y guarda en `self.total` el 12 que tenía en `new_total`. En este punto estamos con los dos hilos que ya ejecutaron el `Adder.add`, y el valor final de `self.total` es incorrecto.



La probabilidad de que eso suceda en ese código es baja (pero no cero), lo cual en realidad es peor aún, ya que lleva a bugs muy difíciles de reproducir y encontrar. Incluso si ejecutamos varias veces el programa con esa modificación, veremos que nos da el valor correcto:

```
$ python3 concur_hilos_explicito_1.py
Done: 288998
```

Para aumentar la probabilidad de que entremos en la secuencia problemática, hagamos que el hilo “duerma” entre esas líneas (ya que la llamada a `time.sleep` implica un cambio de hilos):

```
1  def add(self, value):
2      prev_total = self.total
3      time.sleep(.01)
4      new_total = prev_total + value
5      time.sleep(.01)
6      self.total = new_total
```

Ahora sí encontramos el inconveniente:

```
$ python3 concur_hilos_explicito_2.py
Done: 209628
```

Incluso con estos `time.sleep` arbitrarios, que fuerzan a que el hilo suelte el procesador (lo cual incrementa la chance de que entre algún otro hilo), depende mucho de los tiempos y las velocidades de procesamiento. Por ejemplo, si en lugar de dormir 10 milisegundos como mostramos arriba, lo ponemos a dormir 1 milisegundo, casi siempre se ejecuta correctamente (porque tengamos en cuenta que los otros hilos están ocupados accediendo a la red, lo cual lleva proporcionalmente bastante tiempo).

Esta situación se denomina “condición de carrera” (*race condition* en inglés), una expresión usada en electrónica y en programación para señalar cuando la salida o estado de un proceso es dependiente de una secuencia de eventos que se ejecutan en orden arbitrario, y al trabajar sobre un mismo recurso compartido se pueden producir errores cuando dichos eventos no se ejecutan en el orden esperado.

Además, tengamos en cuenta que este código era sencillo y pequeño. Extrapolemos la dificultad si tenemos miles o decenas de miles de líneas de código, ¿cuanto podemos asegurar que cada una y todas las líneas son seguras para un entorno multi-hilo?

La forma de evitar el problema es usar un bloqueo para que distintos hilos **no puedan ejecutar al mismo tiempo** la parte del código que puede ser problemática. Eso se logra usando una herramienta que tanto en inglés como en castellano denominamos *lock*.

Los *locks* se toman y se liberan. Antes de entrar a la zona problemática del código se toma el lock, y al salir se libera. Si tenemos el lock tomado y el sistema operativo deja de ejecutar nuestro hilo y entra otro, y ese otro hilo llega a la misma zona problemática, al intentar tomar el lock se bloqueará y sólo se desbloqueará cuando el primer hilo libere el lock.

Podríamos representar el uso del lock con el siguiente código:

```
1 mylock = threading.Lock()
2 mylock.acquire()
3 ... # zona problemática del código
4 mylock.release()
```

Sin embargo ese código es incorrecto y peligroso: si la zona problemática del código genera una excepción nunca se liberará el lock, y eventualmente nuestro sistema se bloqueará para siempre (porque los distintos hilos estarán esperando que se libere un lock que nunca se liberará).

Podemos evitar el problema asegurándonos que **siempre** se ejecute el release, sin importar si hubo un error o no:

```
1 mylock = threading.Lock()
2 mylock.acquire()
3 try:
4 ... # zona problemática del código
5 finally:
6 mylock.release()
```

Los locks de Python se ofrecen en modo administrador de contexto [5.2](#) que nos asegura el mejor funcionamiento, así que el código es aún más sencillo:

```
1 mylock = threading.Lock()
2 with mylock:
3 ... # zona problemática del código
```

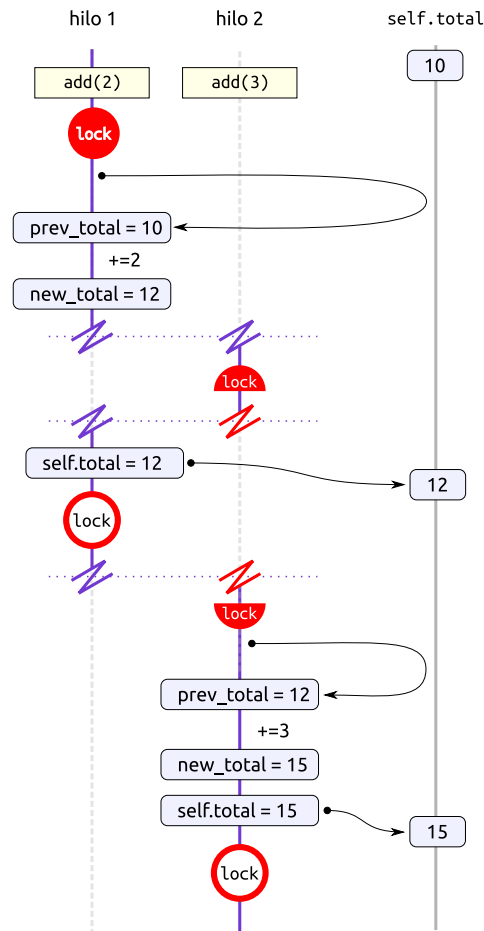
Veamos el uso del lock en el código problemático que traíamos:

```
1 import threading
2 import time
3 from urllib.request import urlopen
4
5 BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
6 TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]
7
8
9 class Adder:
10     def __init__(self):
11         self.total = 0
12         self.lock = threading.Lock()
13
14     def add(self, value):
15         with self.lock:
16             prev_total = self.total
17             time.sleep(.01)
18             new_total = prev_total + value
19             time.sleep(.01)
20             self.total = new_total
21
22
```

```
23 def downloader(tex_name, adder):
24     url = BASE + tex_name
25     u = urlopen(url)
26     length = len(u.read())
27     adder.add(length)
28
29
30 adder = Adder()
31 all_threads = []
32 for tex_name in TEX_NAMES:
33     th = threading.Thread(target=downloader, args=(tex_name, adder))
34     th.start()
35     all_threads.append(th)
36
37 for th in all_threads:
38     th.join()
39 print("Done:", adder.total)
```

```
$ python3 concur_hilos_explicito_3.py
Done: 288998
```

En el siguiente diagrama vemos el flujo de ejecución:



Aquí vemos que aunque el primer hilo es interrumpido igual que en la situación anterior (donde terminábamos con el valor incorrecto), cuando el segundo hilo quiere obtener el lock se traba porque el mismo ya está tomado. Allí el sistema operativo volverá a cambiar de hilos y entrará (eventualmente) el primero, que continuará con su proceso y liberará el lock al final. En este momento el segundo hilo pasa a estar disponible y (eventualmente) se ejecutará, terminando de tomar el lock, procesando la suma, y liberando el lock también al final.

¿Dónde tenemos que usar locks? Siempre que tengamos que proteger una secuencia de operaciones que si son interrumpidas y ejecutadas en parte por distintos hilos simultáneamente podrían generarnos problemas. Uno de los más grandes inconvenientes al analizar programas propios y ajenos es reconocer estas secuencias de operaciones.

Para el ejemplo que traíamos, la zona problemática era una sola línea que hacía `self.total += value`. Pero esta línea no es una operación “atómica” (indivisible) sino que realmente es una secuencia de operaciones. La forma de detectar esto, en Python, es analizar el bytecode generado:

```

1  >>> import dis
2  >>> def add(self, value):
3  ...     self.total += value
  
```

```

4     ...
5     >>> dis.dis(add)
6         2          0 LOAD_FAST          0 (self)
7         2 DUP_TOP
8         4 LOAD_ATTR          0 (total)
9         6 LOAD_FAST          1 (value)
10        8 INPLACE_ADD
11       10 ROT_TWO
12       12 STORE_ATTR          0 (total)
13       14 LOAD_CONST          0 (None)
14       16 RETURN_VALUE
15     >>>

```

La única línea que teníamos a nivel lenguaje Python en realidad se traduce en esa secuencia de operaciones internas de la máquina virtual de cPython, que sí son atómicas (recomendamos [la documentación del módulo `dis`](#) para profundizar sobre el bytecode de Python). Ahí vemos que los pasos son varios, parecidos a los que simulamos en los códigos ejemplos que mostramos antes.

La complejidad de tener que analizar todo el código y entender qué operaciones son atómicas y cuales no, al momento de modificar estructuras de datos en situación de hilos múltiples, está presente en todos los lenguajes (por ejemplo, en C una línea se transforma en muchas operaciones en assembler cuando compilamos). Como resultado, la mayoría de los programs multi-hilos tienen situaciones problemáticas escondidas, lo que lleva a bugs esporádicos muy difíciles de erradicar.

9.3. Async

Como mencionamos antes en la introducción, el modelo asincrónico se basa en tener varias corrutinas (en lugar de procesos) que se ejecutan de forma concurrente, siendo administradas por un reactor o loop de eventos (en lugar del sistema operativo con ayuda del procesador).

Esta administración necesita sí o sí de la ayuda de las mismas corrutinas: el reactor va a decidir qué funciones o corrutinas ejecutar teniendo en cuenta qué eventos hay para las mismas, pero no podrá interrumpirlas una vez que estén ejecutándose, y dependerá que esas corrutinas liberen el control cuando necesitan ejecutar una operación bloqueante (recordemos que todas las corrutinas y el reactor corren en el mismo proceso). Es por esto que se lo llama modelo cooperativo.

Las corrutinas entonces correrán hasta que necesiten ejecutar una operación que puede tardar un tiempo significativo en ejecutarse. En ese momento le indicarán al reactor que necesitan el resultado de esa operación, en efecto liberando el uso del procesador. Este simple código muestra el modelo descrito (explicaremos en detalle cómo se usa async en [9.3.1](#)):

```

1 import asyncio
2 import time
3
4
5 async def timings():
6     print("A:", time.time())
7     await asyncio.sleep(1)
8     print("B:", time.time())

```

```
9
10 asyncio.run(timings())
```

Esta función de Python (en realidad una corrutina, porque está definida con el `async` adelante) va a mostrar dos tiempos con un segundo de diferencia entre ellos. La corrutina no debe dejar pasar el segundo con un `time.sleep(1)` clásico porque eso bloquearía a todo el proceso, sino que en vez le indica al reactor que quiere dejar pasar ese tiempo. Luego, debe interrumpir su ejecución esperando que ese evento suceda, lo cual se realiza con el `await` que está delante.

Por supuesto, esta corrutina tiene que ser ejecutada por el reactor, lo cual se logra de forma sencilla en la última línea del código. Hay formas más complejas y versátiles de usar o crear loops de eventos; usar `asyncio.run` es la forma más simple y es suficiente por ahora.

En la llamada a `asyncio.sleep(1)` el reactor creará un evento internamente que se ejecutará luego de un segundo, y cuando eso suceda enviará el resultado a la corrutina, la cual no se ejecutará desde el principio (como si fuese una función normal llamada nuevamente) sino que continuará desde el punto que largó el procesador (un comportamiento más parecido a los generadores que vimos en 4.1).

Una simplificación casi absurda del código del reactor podría ser el siguiente:

```
1 while not self.stop:
2     events = self.get_events()
3     for event in events:
4         self.dispatch(event)
```

Por supuesto que los loops de eventos reales son más complejos que eso, pero la idea es la misma: estar todo el tiempo escuchando si suceden eventos, y cuando hay nuevos despacharlos a las corrutinas correspondientes. Los eventos pueden ser de varios tipos: un timer (como vimos recién), que un recurso del sistema operativo está listo para ser leído o escrito (un archivo, un socket de red, etc.), e incluso eventos que no son comenzados por una corrutina sino que provienen de periféricos o acciones del usuario (como el click de un botón del mouse).

El tener el reactor en el mismo proceso es una de las razones por las que el modelo asincrónico escala en cantidad de operaciones de entrada/salida concurrentes mejor que el modelo basado en hilos. También es un factor que cada elemento que está esperando un resultado es simplemente una función y no una estructura mucho más compleja como un hilo o un proceso.

El modelo asincrónico tiene varias otras ventajas. Al estar todo el sistema dentro del mismo proceso es muy fácil compartir objetos y estructuras de datos, como sucede también en el caso de los hilos; pero a diferencia de estos el acceso a esas estructuras no puede ser interrumpido en cualquier momento por un agente externo, entonces no tenemos que protegerlas con locks (y no tenemos el riesgo de no darnos cuenta que esos locks eran necesarios, lo cual resulta en programas más robustos). Otro punto de vista sobre esta característica de no ser interrumpidos en cualquier momento de ejecución es que los programas son determinísticos, lo cual los hace mucho más fáciles de testear y depurar: si estamos analizando un programa podemos estar seguros que luego de la línea N se ejecutará la línea N+1, a menos que explícitamente liberemos el procesador.

No todo son rosas, claro. El modelo tiene también sus desventajas.

El principal es que al ser un modelo cooperativo, tenemos que tener la precaución de no bloquear nunca la ejecución del proceso. Siempre utilizar una llamada preparada para trabajar

asincrónicamente para todo aquello que bloquee (usar la red, por ejemplo), o incluso abrir un hilo para esa operación (veremos más adelante esto). En algún punto estamos cambiando la “preocupación por evitar condiciones de carrera” del modelo de múltiples hilos a la “preocupación por no bloquear”, la diferencia sustancial es que aquí no estamos afectados por la aleatoriedad del orden de ejecución y dónde se interrumpen los hilos (en tiempo de ejecución), sino que el código se ejecuta de forma determinística y podemos detectar cuando estamos bloqueando incorrectamente al estudiar el código (al momento de escribir el código, no cuando se ejecuta).

Otro gran detalle es que al suceder todo dentro del mismo proceso, efectivamente este corre en el mismo procesador, por lo cual no tiene sentido para ser usado en procesos cpu-bound. En otras palabras, si lo que queremos es realizar en paralelo tareas intensivas de procesamiento, este modelo directamente no aplica.

Y no hay que desestimar la complejidad de que sea un concepto todavía no muy distribuido en el mundo del software (aunque no es un modelo nuevo). La mayoría de los desarrolladores no están acostumbrados a “pensar en modo asincrónico” y lleva un aprendizaje que sólo se logra a través del tiempo y la experiencia.

9.3.1. Usando async

La idea base del modelo asincrónico es hacer todo lo que se necesita hacer, pero nunca bloquear. Según esta idea, podemos separar todas las actividades que hacemos en nuestro programa en dos grandes secciones: lo que nos va a bloquear y lo que no.

Si necesitamos algo que sabemos que va a tardar por su naturaleza misma (hacer un request HTTP, traer algo de una base de datos, etc.), se lo pedimos a alguna librería asincrónica o directamente al reactor y esperamos el resultado. Esto es lo que hace la línea 7 del código ejemplo que mostramos al arrancar la sección. En vez de usar `time.sleep(1)`, que bloquea a todo el sistema asincrónico hasta que pase ese segundo, hace `asyncio.sleep(1)` y se pone a esperar el resultado con `await`. La corrutina se bloquea, sí, pero sólo esa corrutina: todo el sistema asincrónico sigue operando sin problemas. Y cuando haya pasado ese segundo el mismo sistema desbloqueará la corrutina, que continuará su ejecución.

Por otro lado, si necesitamos algo que sabemos que es rápido e inmediato (multiplicar dos números, formatear una cadena, instanciar una clase, etc.), lo hacemos y ya.

Claro, tenemos que poder distinguir ambos casos, y no siempre es tan fácil. Por ejemplo, si tenemos que leer un archivo de 2 kb de disco, ¿eso va a tardar o no? En realidad es siempre rápido... pero es potencialmente bloqueante: nos puede pasar que ese archivo esté en un sistema de archivos montado a través de la red y una lectura que a priori es casi instantánea tarde varios segundos, y no queremos bloquear todo el sistema durante esos segundos. Entonces tenemos que acceder a los archivos usando, por ejemplo, [la biblioteca aiofiles](#).

Empecemos a ver cómo usar el sistema asincrónico para lograr todo eso.

Ya vimos que nuestro concepto fundamental son las corrutinas (más allá del reactor o loop de eventos, que usamos pero no definimos nosotros), que son nuestras “funciones asincrónicas” y nos permiten liberar el procesador, esperando eventos sin bloquear todo, haciendo que el sistema en sí funcione correctamente.

Explorémoslas un poco más en detalle.

La definición de una corrutina es parecida a la de una función, pero se le debe agregar el

async adelante:

```
1 >>> async def corrut():
2 ...     print("Hola")
3 ...
4 >>> corrut()
5 <coroutine object corrut at 0x7f8716110140>
```

El hecho de “ejecutar la función” no pone la corrutina a correr, sólo la crea (de nuevo, muy parecido a lo que pasa con los generadores). Para que la corrutina “entre en el sistema asincrónico” y pase a ser ejecutada cuando el reactor de ese sistema lo disponga, tenemos tres mecanismos: pasársela a `asyncio.run` como punto de entrada de nuestro programa, esperarla con **await**, o crear una tarea a partir de la corrutina (a través de `asyncio.create_task`, que nos devuelve una `Task` alrededor de una corrutina con el agregado de algunos mecanismos útiles de control).

Hagamos lo primero:

```
1 >>> import asyncio
2 >>> asyncio.run(corrut())
3 Hola
```

Dentro de las corrutinas habrá código que se ejecuta inmediatamente y código que implica esperar por un resultado. Como mencionamos antes, tenemos que hacer que la corrutina espere ese resultado sin bloquear todo el sistema, y para ello tenemos el **await**. Vimos al principio de la sección un ejemplo mínimo que lo usaba, veamos ahora uno más complejo con algunos elementos que acabamos de ver:

```
1 import asyncio
2 import random
3 from datetime import datetime
4
5
6 async def sleeper(sleeper_id):
7     delay = random.random() * 5 # random float between 0 and 5
8     print(f"Arranca {sleeper_id} a las {datetime.now():%X.%f}")
9     await asyncio.sleep(delay)
10    print(f"Termina {sleeper_id} a las {datetime.now():%X.%f}")
11    return delay
12
13
14 async def main():
15     slept = await sleeper(1)
16     print(f"Durmió: {slept:.3f}s")
17
18 asyncio.run(main())
```

Vemos en el código dos corrutinas. A `main` se la pasamos como punto de entrada al sistema. Esta corrutina va a llamar/esperar a otra, `sleeper` y luego mostrará cuanto durmió, para lo cual recibe un valor en la línea del **await**. Acá vemos la otra forma de poner a correr la corrutina, que es esperándola.

Por su lado, `sleeper` calcula un valor al azar entre 0 y 5, hace un `sleep` con ese valor, mostrando la hora antes y después, y finalmente devuelve ese valor usado. El `sleep` también es asíncrono, por eso hay que esperarlo, pero no nos interesa guardar el valor que devuelve.

```
$ python3 seq_async_1.py
Arranca 1 a las 16:15:51.819157
Termina 1 a las 16:15:55.991726
Durmió: 4.168s
```

Veamos de forma más explícita la separación entre crear una corrutina y ponerla a correr:

```
14 async def main():
15     coroutine1 = sleeper(1)
16     coroutine2 = sleeper(2)
17     print("Corrutinas creadas")
18     slept = await coroutine1
19     print(f"Durmió 1: {slept:.3f}s")
20     slept = await coroutine2
21     print(f"Durmió 2: {slept:.3f}s")
```

El código crea dos corrutinas (que casi lo primero que harían sería mostrar la hora de comienzo), pero antes de “esperarlas” (que las hace disponibles para el sistema asíncrono) avisa que las corrutinas fueron creadas. Luego las espera, mostrando cuanto durmió cada una:

```
$ python3 seq_async_2.py
Corrutinas creadas
Arranca 1 a las 17:45:12.703133
Termina 1 a las 17:45:15.223655
Durmió 1: 2.519s
Arranca 2 a las 17:45:15.223749
Termina 2 a las 17:45:15.533578
Durmió 2: 0.309s
```

Como decíamos, el “Corrutinas creadas” aparece antes que cualquier mensaje de las corrutinas porque las mismas no se ponen a correr hasta que se esperan.

Otro efecto interesante que vemos aquí (relacionado con lo que decíamos) es que aunque estamos en un sistema asíncrono y usando corrutinas, las mismas se ejecutaron de forma “secuencial”. Esto es porque en el código de `main`, aunque creamos las dos corrutinas al principio, luego esperamos una (se pone a correr esa, ¡pero no la otra!), y cuando termina, luego del `print`, esperamos la segunda (y recién ahí se pone a correr).

Veamos cómo lograr efectivamente concurrencia al utilizar la tercera forma de poner a correr una corrutina: crear una tarea con ella.

```
14 async def main():
15     task1 = asyncio.create_task(sleeper(1))
16     task2 = asyncio.create_task(sleeper(2))
17     print("Corrutinas creadas")
18     slept = await task1
```

```

19 print(f"Durmió 1: {slept:.3f}s")
20 slept = await task2
21 print(f"Durmió 2: {slept:.3f}s")

```

Este tercer código es muy parecido al anterior, sólo que ahora creamos dos tareas que esperamos luego.

```

$ python3 seq_async_3.py
Corrutinas creadas
Arranca 1 a las 17:46:52.575249
Arranca 2 a las 17:46:52.575287
Termina 2 a las 17:46:53.342483
Termina 1 a las 17:46:55.811887
Durmió 1: 3.234s
Durmió 2: 0.765s

```

Aquí sí notamos que ambas se ejecutan concurrentemente, porque las corrutinas están listas para ejecutarse en el momento en que las tareas fueron creadas. Envolver corrutinas en tareas no es útil sólo por lo mencionado: también nos agrega más funcionalidad que nos permite escalar en complejidad en nuestro sistema, como por ejemplo revisar si la tarea está completada o cancelarla).

Hay dos efectos interesantes para observar. El primero es que “Corrutinas creadas” sigue apareciendo primero, porque aunque estamos creando las dos tareas en las líneas 15 y 16, realmente todavía no le pasamos el control al reactor, así que `main` sigue ejecutándose... luego viene el `print` en cuestión, y luego la línea 18 donde pasamos a esperar a `tarea1`, y es allí en ese punto donde explícitamente con el `await` le damos el control al reactor, que pone ambas tareas a correr.

El segundo efecto es que aunque por azar la segunda tarea tuvo un *delay* menor y terminó antes, el mensaje de “cuanto durmió” sigue apareciendo en segundo lugar. Esto es porque más allá de la ejecución concurrente de las corrutinas, nosotros en `main` las estamos esperando secuencialmente, primero a `tarea1` en la línea 18 y luego a `tarea2` en la 20.

Esto que en el ejemplo rompe un poco el efecto de concurrencia que estamos buscando en realidad no es algo que nos tiene que preocupar. En códigos reales pocas veces se esperan las tareas de esa manera si se tienen varias ejecutando de forma concurrente (lo cual además sería engorroso para el caso de tener decenas de tareas, sería ridículo tener decenas de líneas haciendo `await`).

Justamente para esto es que el módulo `async` provee una función `gather`:

```

14 async def main():
15     tasks = [asyncio.create_task(sleeper(n)) for n in range(3)]
16     print("Corrutinas creadas")
17     results = await asyncio.gather(*tasks)
18     print("Durmieron:")
19     for idx, slept in enumerate(results):
20         print(f"{idx:4d}: {slept:.3f}s")

```

Cambiamos un poco todo el código para esta versión, para hacerlo más genérico. Ahora en vez de crear dos corrutinas y tareas por separado, manualmente, lo hacemos dentro de una *list*

comprehension (tres en total, que podrían ser mil, pero corremos el ejemplo con pocas para que la salida que mostramos luego no quede fea en el libro).

También generalizamos la forma de mostrar los valores luego de las esperas, aprovechando una característica de las respuestas del `gather` que asegura que el orden de los resultados corresponde al de las tareas indicadas.

```
$ python3 seq_async_4.py
Corrutinas creadas
Arranca 0 a las 09:57:42.549920
Arranca 1 a las 09:57:42.549979
Arranca 2 a las 09:57:42.550000
Termina 1 a las 09:57:42.670305
Termina 0 a las 09:57:43.451375
Termina 2 a las 09:57:46.989842
Durmieron:
0: 0.900s
1: 0.120s
2: 4.438s
```

El `gather` también nos permite controlar qué pasa si una de las corrutinas genera una excepción: cancelar todas las corrutinas que todavía no terminaron y propagar la excepción para arriba, o considerar esas excepciones como “resultados” y acumularlos en la lista que nos entrega junto con otros resultados.

Como vemos, el mundo asincrónico nos ofrece muchas opciones, pero siempre volvemos al talón de Aquiles de que nada puede bloquear. ¿Cómo hacemos entonces cuando tenemos algo que sí o sí bloquea?

Veamos el siguiente código de ejemplo, que nos expone a una situación de bloqueo que no podemos evitar. Para ver qué sucede en ese caso, el programa tiene una corrutina que se pone a correr al comienzo que muestra un *spinner* en la terminal: si vemos girar al spinner es que el sistema asincrónico está corriendo sin problemas.

```
1 import asyncio
2 import itertools
3 import sys
4 import time
5
6
7 async def spinner():
8     ticks = itertools.cycle("|/-\\")
9     while True:
10         print(f"{next(ticks)}\r", end="", flush=True)
11         await asyncio.sleep(.2)
12
13
14 async def generic_work_async():
15     await asyncio.sleep(3)
16     return 5
17
18
19 def generic_work_blocking():
20     time.sleep(3)
```

```
21     return 7
22
23
24 async def main(option):
25     print("Comienzo")
26     spinner_task = asyncio.create_task(spinner())
27
28     if option == "1":
29         print("Durmiendo modo async, ¡perfecto!")
30         result = await generic_work_async()
31     elif option == "2":
32         print("Bloqueando todo, está mal :)")
33         result = generic_work_blocking()
34     else:
35         print("Bloqueando, pero en un hilo (no traba todo)")
36         result = await asyncio.to_thread(generic_work_blocking)
37
38     spinner_task.cancel()
39     print(f"Final; {result=}")
40
41 if len(sys.argv) != 2 or sys.argv[1] not in ["1", "2", "3"]:
42     print("Usar: python3 ej_async_blocking.py {1|2|3}")
43     sys.exit()
44
45 asyncio.run(main(sys.argv[1]))
```

El código tiene cuatro grandes bloques. La primera corrutina es el spinner en sí, va dibujando eternamente en la terminal algo que aparenta girar. Luego tenemos dos “tareas genéricas”, una que es asíncrona y otra bloqueante (en cada caso sólo durmiendo, porque son ejemplos). Después viene la corrutina principal que avisa del comienzo, arranca una tarea con el spinner, ejecuta alguna “tarea genérica” de alguna manera, y luego cancela al spinner y avisa del final. Para terminar tenemos un par de líneas que valida que al ejecutar el programa hayamos pasado una opción válida y ejecuta la función principal dentro de un reactor.

Los tres casos que podemos elegir al correr el programa muestran los distintos comportamientos. En todos los casos simulamos con el `sleep` lo que sería una operación que puede tardar mucho, como un request HTTP o acceder a una base de datos. Con el `asyncio.sleep` ejemplificamos el caso de hacer esa operación utilizando una biblioteca asíncrona, y con el `time.sleep` mostramos qué pasaría si utilizamos una biblioteca sincrónica, bloqueante.

El primer caso es muy directo: como utilizamos una biblioteca asíncrona, todo funciona correctamente. En la terminal deberíamos ver el “Comienzo”, el aviso del caso 1, el spinner girando durante tres segundos, y el “Final”.

Si corremos al programa eligiendo el segundo caso, directamente bloquea todo. El programa también tarda tres segundos en terminar, pero nunca vemos el spinner girar porque el sistema asíncrono está trabado.

El tercer caso es el que nos ocupa para aprender a llamar algo que inevitablemente nos va a bloquear (al ejecutar el programa deberíamos ver el spinner girando correctamente). La solución es decirle al reactor que ejecute esa función en un hilo aparte, y que nos desbloquee cuando esa función termine. De esta manera la función en sí, que es bloqueante, se va a ejecutar sin problemas, y no nos va a bloquear todo el sistema asíncrono. Lo simple de esta solución es que no tenemos que manejar realmente los hilos desde nuestro programa, pero sí tenemos que tener

el cuidado de que la función ejecutada en ese hilo no acceda recursos compartidos (porque si no volvemos a tener todo el problema de posibles condiciones de carrera que vimos para el caso de múltiples hilos).

El mundo asincrónico en general y `asyncio` tiene muchos otros conceptos, pero ya se empiezan a escapar del alcance de este libro. Los mencionamos brevemente para darles una idea de los diferentes temas, de manera que si lo desean puedan profundizar en los mismos:

- **Streams:** manejar conexiones de red (escribiendo y leyendo flujos de datos) es tan típico en los sistemas asincrónicos que hay herramientas para simplificar el proceso. En la mayoría de los sistemas asincrónicos estas herramientas son los “protocolos” y “transportes”, pero el módulo `asyncio` (además de esas herramientas) tiene el concepto de *streams*, que nos permite trabajar con conexiones de red y enviar y recibir datos de forma más simple.
- **Primitivas de sincronización:** más allá que la ejecución de cada línea es determinística en el mundo asincrónico, si tenemos código ejecutándose concurrentemente que accede a recursos compartidos, es posible que necesitemos sincronizar esos accesos, para lo cual tenemos locks, eventos, semáforos, etc., como en el mundo de múltiples hilos, pero adaptados para sistemas asincrónicos.
- **Subprocesos:** para manejar asincrónicamente, sin bloquear ni tener que mandar a otro hilo, todo lo que es ejecución de subprocesos y esperar sus resultados o ir leyendo sus salidas.
- **Colas:** con un comportamiento similar a `queue`. `Queue`, el módulo `asyncio` nos provee colas que trabajan asincrónicamente, y son la herramienta ideal para armar un modelo donde algunas corrutinas pueden ser productoras de información y otras consumidoras.

Para terminar, veamos nuestro ejemplo integrador (que baja archivos y suma sus longitudes) pero adaptado a la forma asincrónica:

```
1 import asyncio
2
3 import aiohttp
4
5 BASE = "https://raw.githubusercontent.com/facundobatista/libro-pyciencia/main/src/"
6 TEX_NAMES = ["integracion.tex", "intro.tex", "numpy.tex", "ordinarias.tex", "parciales.tex"]
7
8
9 class Adder:
10     def __init__(self):
11         self.total = 0
12
13     def add(self, value):
14         self.total += value
15
16
17 async def downloader(tex_name, adder):
18     url = BASE + tex_name
19     async with aiohttp.ClientSession() as session:
20         async with session.get(url) as resp:
21             content = await resp.read()
22     adder.add(len(content))
```

```

23
24
25 async def main():
26     adder = Adder()
27     all_coroutines = []
28     for tex_name in TEX_NAMES:
29         coroutine = downloader(tex_name, adder)
30         all_coroutines.append(coroutine)
31     await asyncio.gather(*all_coroutines)
32     print("Done:", adder.total)
33
34
35 asyncio.run(main())

```

La estructura es muy similar a la de los mismos ejemplos hechos con múltiples hilos o sin concurrencia. Antes de entrar en el código así, notemos que estamos utilizando un módulo que no es de la biblioteca standard, `aiohttp`, para hacer pedidos HTTP asíncronos; lo tenemos comentado con `fades` para que este nos arme el entorno virtual que corresponda, como vimos en el capítulo de Entornos [7.2.2.3](#).

Luego tenemos al `Adder`, que es igual a los otros en su forma simple, porque como estamos seguros que nada va a interrumpir a las corrutinas cuando suman, no necesitamos *locks*.

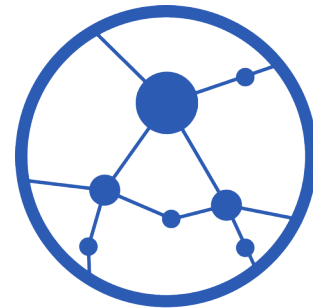
El `downloader` en sí tiene una estructura similar, pero parece más complejo a simple vista. Sí, es asíncrono, pero también aparecieron dos administradores de contexto (también asíncronos). Esto es porque interactuar con la red tiene varios pasos que son bloqueantes, y la biblioteca está diseñada para exponer eso de forma explícita.

La sesión inicializa recursos, que luego cierra automáticamente al salir del contexto. El `.get()` recibe la primer respuesta del server (y también es un administrador de contexto, porque luego hay recursos para liberar), pero esta primer respuesta tiene los headers solamente, NO trae todo el contenido de la respuesta. La acción de leer el cuerpo de la respuesta también es un paso bloqueante; en este caso el `read()` trae todo el cuerpo de la respuesta porque sabemos que es relativamente chico, pero si tuviésemos que bajar un archivo muy grande deberíamos usar la [API de streaming de aiohttp](#).

En realidad nosotros no estamos haciendo el mejor uso de la interfaz de `aiohttp`; deberíamos compartir la sesión (y no crear una en cada llama a `downloader`) para optimizar recursos. En este código ejemplo es prácticamente lo mismo (y lo dejamos así por razones pedagógicas), pero en un sistema real, si quisiéramos escalar a centenas o miles de direcciones web, lo mejor es crear la sesión una sola vez y usar siempre esa.

Finalmente agrupamos el bloque de código principal dentro de una corrutina, que es la que le indicamos al reactor para que ejecute inicialmente, pero más allá de ese detalle la estructura es muy similar a la de hilos, cambiando *threads* por *coroutines*, y aprovechando el `gather` que nos simplifica esperar varias corrutinas a la vez en lugar de tener que hacer un `.join` por cada hilo.

Cabe destacar que como en el modelo de múltiples hilos (donde a veces es difícil detectar potenciales condiciones de carrera), en el modelo asíncrono también es difícil a veces detectar potenciales situaciones bloqueantes; en el código ejemplo tenemos el caso de que todo es mane-



aiohttp - Proporciona soporte para escribir clientes y servidores HTTP asíncronos en Python.

jado asincrónicamente, excepto la resolución del nombre del servidor... o sea, la traducción de `raw.githubusercontent.com` a la dirección IP correspondiente. Para que este paso sea automáticamente asincrónico, también deberíamos instalar la biblioteca `aiodns`.

9.4. Procesamiento en múltiples procesadores

En esta sección charlaremos acerca de la ejecución de funciones o procesamiento de datos en distintos procesadores. En otras palabras, de lograr “paralelismo real”, no simular paralelismo con concurrencia como era el caso de los sistemas asincrónicos o de múltiples hilos en Python.

Tengamos en cuenta que aquí, como en las otras secciones y como explicamos en detalle al inicio de la sección sobre concurrencia 9.1, cuando decimos procesador más bien nos referimos a “unidad de ejecución”, ya que en todo lo que estamos hablando es más fácil pensarlo como unidades de ejecución distintas, ya sean uno o varios procesadores, uno o varios cores.

Exploraremos entonces algunas formas de lograr paralelismo a mano, utilizando algunas funcionalidades de Python provistas en la biblioteca estándar.

9.4.1. Introducción

A nivel de ejecutar otros procesos, Python nos ofrece las herramientas básicas sobre las que se construye todo el resto.

Entrar en el detalle de cuales son y sus alcances sería engorroso y se escapa al alcance de este libro, pero como ejemplos mencionaremos dos que están disponibles en plataformas POSIX. Por un lado tenemos al `fork`, que crea un nuevo proceso duplicando el actual; el nuevo se llama “proceso hijo”, mientras que el que ejecutó el `fork` es el “proceso padre” (los dos procesos se ejecutan en espacios de memoria separados que al momento del `fork` tienen el mismo contenido). Por otro lado tenemos la familia `exec*`, que reemplazan el proceso actual con uno nuevo, inmediatamente, y no se vuelve al original (las distintas funciones de la familia, `execl`, `execve`, `execvpe`, etc., difieren en cómo se pasan argumentos a ese nuevo proceso).

La gran ventaja de Python es que nos ofrece módulos para trabajar con multiprocesamiento que nos abstraen de esas herramientas de bajo nivel y nos permite ejecutar funciones en otros procesos de forma muy sencilla. Estos módulos se llaman `concurrent.futures` y `multiprocessing`. Arranquemos por el primero.

El `concurrent.futures` nos permite ejecutar en distintos procesadores una misma función sobre una cantidad de datos: automáticamente creará un pool de procesos e irá ejecutando la función sobre las distintas entradas a medida que se vayan desocupando del cálculo anterior.

Veamos el siguiente ejemplo donde tenemos una función que factoriza números. Más allá del algoritmo en sí que no vamos a explicar ahora porque no es el foco, la función muestra cuánto tarda en ejecutarse. Luego tenemos una serie de números que serán factorizados, y al final se termina mostrando el tiempo total y los resultados.

```
1 import time
2 from concurrent.futures import ProcessPoolExecutor
3 from itertools import chain
4
5
```

```

6 def factorizer(number):
7     tini = time.time()
8     orig_number = number
9     results = []
10    for factor in chain([2], range(3, number // 2 + 1, 2)):
11        while number % factor == 0:
12            results.append(factor)
13            number = number // factor
14        if factor > number:
15            break
16    tdelta = time.time() - tini
17    print(f"orig_number:16d: {tdelta:7.2f}s")
18    return results
19
20
21 inputs = [
22     8919372543,
23     2429192056233,
24     23286190456122,
25     51734095734,
26     25301356,
27     353061047130563,
28     537334713453,
29 ]
30 print("Ejecutando todo:")
31 tini = time.time()
32 with ProcessPoolExecutor(max_workers=4) as executor:
33     all_results = executor.map(factorizer, inputs)
34 tdelta = time.time() - tini
35 print(f"Tiempo total: {tdelta:.2f}s")
36
37 print("Resultados:")
38 for number, result in zip(inputs, all_results):
39     print(f"{number:16d}: {result}")

```

La funcionalidad interesante a explicar es el `ProcessPoolExecutor`, usado en la línea 32. Es un administrador de contexto que nos da un executor que a su vez tiene un método `map`. Este método es similar a la función `map` integrada en Python que aplica una función a todos los elementos de un iterable, pero con la gran diferencia en este caso que esa función va a ser ejecutada en otros procesos. ¿En cuantos procesos y en qué orden? El `ProcessPoolExecutor` arma una cantidad de procesos (por default la cantidad de procesadores que tengamos, para el ejemplo forzamos 4) y ejecuta en esos procesos la función; cuando termina una se le pasa un nuevo valor para que vuelva a ejecutarse, sin importar lo que suceda en otros procesos.

Los resultados, como con el `asyncio.gather` que vimos antes, estarán en el mismo orden que los valores de entrada, por eso podemos relacionarlos y mostrarlos con el `zip` al final.

Veamos cómo se ejecuta el ejemplo, considerando lo que aprendimos de cómo funciona:

```

$ python3 ej_paralelo_futures.py
Ejecutando todo:
2429192056233:    0.20s
25301356:       0.37s
353061047130563:  0.42s
23286190456122:  0.47s
537334713453:   4.36s

```

```

51734095734: 20.11s
8919372543: 154.63s
Tiempo total: 154.64s
Resultados:
8919372543: [3, 2973124181]
2429192056233: [3, 71, 2843, 4011487]
23286190456122: [2, 3, 11, 39989, 8822953]
51734095734: [2, 3, 17, 507197017]
25301356: [2, 2, 6325339]
353061047130563: [659, 73859, 7253723]
537334713453: [3, 1831, 97821721]

```

Los tiempos de ejecución que se muestran al principio son cuando termina cada función, así que no nos sorprende que estén los más rápidos primero. Los resultados, por otro lado, están en el orden original.

Veamos de forma explícita cómo sucedió el procesamiento. Especificamos que la cantidad máxima de trabajadores sea cuatro, entonces el sistema arrancó procesando los números 8919372543, 2429192056233, 23286190456122, y 51734095734. La primer función, en alguno de los procesos, termina a los 200 milisegundos, entonces el sistema pasa a ejecutar en ese proceso la función sobre el quinto número. También termina rápidamente, y sigue con los números de entrada. En paralelo, claro, algunos procesos siguen con los números difíciles de factorizar.

Eventualmente todos terminan, el administrador de contexto termina, y mostramos el tiempo total de cálculo, que es prácticamente igual al del número más “costoso”.

También tenemos la opción de crear procesos, ejecutar funciones en esos procesos, y esperar que esos procesos terminen, todos los pasos “a mano”, como mostramos antes con los hilos (y de forma muy similar a ese código). Este sería el cambio con respecto al programa que mostramos antes, solamente pasamos a usar `multiprocessing.Process` en lugar de `threading.Thread` y llamar apropiadamente a los objetos:

```

1  for i in range(n_cores):
2      proc = multiprocessing.Process(target=downloader, args=(tex_name, adder))
3      proc.start()
4      processes.append(proc)
5
6  for proc in processes:
7      proc.join()

```

Por otro lado, también podemos usar `concurrent.futures` para los hilos, en aquellas situaciones donde no nos interesa levantar algún hilo en particular para alguna tarea específica (como es bastante normal en el caso de los hilos), sino más bien ejecutar la misma función para una serie de entradas. Dejamos como tarea para los lectores el adaptar el código que usando hilos baja varios archivos y suma sus longitudes (mostrado en 9.2.1) pero usando `concurrent.futures.ThreadPoolExecutor` (que funciona igual al `ProcessPoolExecutor` que recién vimos, pero usando hilos).

Todo este modelo funciona de manera simple con dos condiciones: si el tiempo de procesamiento para una determinada entrada es relativamente grande, y si la entrada y salida para cada uno de los procesamiento es relativamente chica. Es lo que sucedía en el ejemplo, donde la entra-

da es un número y la salida una lista corta de enteros, mientras que el tiempo de procesamiento en los distintos casos fue de 200 milisegundos a los dos minutos y medio.

En cambio, si la entrada o la salida del procesamiento necesario tienen un tamaño considerable tenemos que considerar algunas alternativas o modificaciones para que el costo de enviar y recibir mucha información entre los procesos no implique un costo excesivo.

Si el corpus grande de datos a procesar puede ser manejado en paralelo, podemos particionarlo y que los distintos procesos trabajen sobre una parte de ese corpus grande. Entonces, la información de entrada que viajará a los distintos procesos serán los límites de comienzo y fin a procesar. Pero claro, de alguna manera los distintos procesos tienen que acceder al corpus grande de información. Y también, si los resultados son grandes, dejarlos a disposición del proceso central que tiene que acceder al resultado completo.

Una alternativa típica es trabajar con archivos. Podemos bajar el corpus grande de datos a disco y que los distintos procesos lean la información de allí. Otra opción, especialmente útil cuando ya tenemos la información en memoria (por cómo la obtuvimos) y luego debemos tener el resultado en memoria (para un procesamiento posterior), es compartir memoria entre el proceso principal y los otros procesos

Veamos un ejemplo donde tenemos dos listas de valores extremadamente grandes y tenemos que realizar una operación compleja y costosa entre cada elemento de esas dos listas, guardando cada resultado en una tercera. Por razones pedagógicas, sin embargo, acotemos el tamaño de las listas a 12 valores y la operación a una simple suma, para que sea más fácil de entender el resto.

Incluso veamos como primer paso un código que está conceptualmente equivocado, como excusa para aclarar algunas nociones:

```
1 from concurrent.futures import ProcessPoolExecutor
2
3 # dos secuencias de datos increíblemente grandes y complejas
4 input_a = [1] * 12
5 input_b = [3] * 12
6
7 # donde guardaremos los resultados
8 results = [None] * 12
9
10
11 def operator(limits):
12     lim_inf, lim_sup = limits
13     print(f"Operando entre {lim_inf} y {lim_sup}")
14     for i in range(lim_inf, lim_sup):
15         results[i] = input_a[i] + input_b[i]
16     print("Resultado parcial:", results[lim_inf:lim_sup])
17
18
19 all_limits = [
20     (0, 4),
21     (4, 8),
22     (8, 12),
23 ]
24 with ProcessPoolExecutor() as executor:
25     list(executor.map(operator, all_limits))
26
27 print("Resultado final:", results)
```

Primero se define las tres inmensamente grandes listas, las dos de entrada de largo 12 llenas de unos y tres respectivamente, y la de salida con 12 Nones (que indican que no se calculó nada en esa posición todavía. Luego tenemos la función que recibe los límites inferior y superior dentro de los que va a operar, y efectivamente aplica una operación compleja y costosa elemento por elemento, mostrando luego la parte del resultado sobre la cual se trabajó.

Luego tenemos los límites (escritos a mano, por simplicidad), y el `ProcessPoolExecutor` que nos va a distribuir los límites en los distintos procesos para realizar todo el trabajo. Notemos el detalle de hacer un `list()` alrededor del `executor.map`; esto es solamente para iterar sobre las operaciones, lo cual obligará al `ProcessPoolExecutor` a efectivamente esperar que todas terminen.

Finalmente, mostramos el resultado cuando todo terminó:

```
$ python3 ej_paralelo_equivocado.py
Operando entre 0 y 4
Resultado parcial: [4, 4, 4, 4]
Operando entre 8 y 12
Resultado parcial: [4, 4, 4, 4]
Operando entre 4 y 8
Resultado parcial: [4, 4, 4, 4]
Resultado final: [None, None, None, None, None, None, None, None, None, None, None, None]
```

En la salida de la ejecución vemos que la función trabajó correctamente sobre las entradas para dentro de los límites recibidos en cada caso, pero al final, la lista de resultados está intacta, lo cual es incorrecto.

Ya sabemos que el `ProcessPoolExecutor` nos va a ejecutar la función `operator` en distintos procesos, pero lo que tenemos que entender aquí es cómo se generan esos distintos procesos: son una copia del proceso original al crearse, utilizando una zona de memoria separada pero con el mismo contenido que el proceso original al momento de creación.

Entonces, cuando `operator` accede a los objetos `input_a` y `input_b`, estos tienen los valores correctos, aunque sean locales al nuevo proceso, porque son parte de lo que se copió. Y al acceder a `results` (que también existe como una lista llena de Nones), también se está accediendo a la “copia local”. Entonces, se escriben allí los resultados, que luego se muestran correctamente en la línea del “Resultado parcial”.

Pero luego los distintos sub-procesos terminan, volvemos al proceso original, y allí el `results` nunca fue modificado, por eso encontramos el problema que vemos en la ejecución.

La solución es relativamente sencilla: tenemos que tener una zona de memoria compartida. El módulo `multiprocessing` nos da herramientas para lograr esto de manera directa. Por ejemplo, si queremos compartir listas tenemos `ShareableList`, con la cual podemos compartir fácilmente algunos tipos de datos nativos de Python (`int`, `float`, `bool`, `str`, `bytes` y `None` (con algunas restricciones de tamaños para `str` y `bytes`):

```
1 from concurrent.futures import ProcessPoolExecutor
2 from multiprocessing import shared_memory
3
4 # dos secuencias de datos increíblemente grandes y complejas
5 input_a = shared_memory.ShareableList([1] * 12)
6 input_b = shared_memory.ShareableList([3] * 12)
7
```

```
8 # donde guardaremos los resultados
9 results = shared_memory.ShareableList([None] * 12)
10
11
12 def operator(limits):
13     lim_inf, lim_sup = limits
14     print(f"Operando entre {lim_inf} y {lim_sup}")
15     for i in range(lim_inf, lim_sup):
16         results[i] = input_a[i] + input_b[i]
17
18
19 all_limits = [
20     (0, 4),
21     (4, 8),
22     (8, 12),
23 ]
24 with ProcessPoolExecutor() as executor:
25     list(executor.map(operator, all_limits))
26
27 print("Resultado final:", results)
28
29 for shared in (input_a, input_b, results):
30     shared.shm.close()
31     shared.shm.unlink()
```

En este nuevo código vemos que ahora no usamos listas simples, sino que tenemos las estructuras en una zona de memoria compartida. Y no sólo para poner el resultado final, sino también las de entrada, para evitar que se copien esas estructuras (que recordemos, sin inmensamente grandes, con lo cual el costo de copiado sería algo a considerar).

Otro detalle importante es que luego de terminar con los sub-procesos que usan esa memoria compartida, tenemos que cerrar y liberar esos recursos, lo cual hacemos en las últimas tres líneas.

```
$ python3 ej_paralelo_compartida.py
Operando entre 0 y 4
Operando entre 8 y 12
Operando entre 4 y 8
Resultado final: ShareableList([4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4], name='psm_65c91d2a')
```

La estructura `ShareableList` es muy práctica, porque nos esconde la serialización y deserialización de los objetos, y aunque nos provee algunas utilidades más, tenemos que tener en cuenta que tiene sus limitaciones.

Para trabajar con memoria compartida de forma más genérica, Python nos ofrece la estructura `multiprocessing.shared_memory.SharedMemory`, que nos permite trabajar directamente sobre la memoria compartida. Claro, nos tendremos que encargar de la serialización y deserialización, pero ya no tendremos límites sobre qué estructuras o secuencias de datos podemos compartir. Veremos más adelante un ejemplo que usa esta estructura.

9.4.2. Trabajando exclusivamente con números

Para el caso puntual de tener que sólo compartir números, que es algo muy común en el ámbito científico a la hora de procesar datos, tenemos algo más eficiente en el uso de memoria: la estructura `Array`. Y si sabemos que nuestro código no estará teniendo acceso a las mismas zonas de memoria compartida desde distintos procesos, incluso podemos usar `RawArray`, que no fuerza a los procesos a estar sincronizados en el acceso a la memoria compartida, siendo por lo tanto también más rápido.

```
1 from concurrent.futures import ProcessPoolExecutor
2 from multiprocessing import RawArray
3
4 # dos secuencias de datos increíblemente grandes y complejas
5 input_a = RawArray('B', [1] * 12)
6 input_b = RawArray('B', [3] * 12)
7
8 # donde guardaremos los resultados
9 results = RawArray('B', [0] * 12)
10
11
12 def operator(limits):
13     lim_inf, lim_sup = limits
14     print(f"Operando entre {lim_inf} y {lim_sup}")
15     for i in range(lim_inf, lim_sup):
16         results[i] = input_a[i] + input_b[i]
17
18
19 all_limits = [
20     (0, 4),
21     (4, 8),
22     (8, 12),
23 ]
24 with ProcessPoolExecutor() as executor:
25     list(executor.map(operator, all_limits))
26
27 print("Resultado final:", list(results))
```

Si comparamos este último programa con el anterior veremos que son prácticamente iguales. Una diferencia importante es que a `RawArray` le tenemos que pasar el tipo de datos que va a soportar (porque arma la memoria compartida para una cantidad fija de bytes que alcanza para almacenar exactamente eso), podemos ver la lista de los distintos tipos de datos en la [documentación de array](#) (que es la misma que se usa aquí).

La otra diferencia es que ahora no tenemos que liberar los recursos al final, es automático. Esto es porque `RawArray` sólo sirve para compartir memoria entre el proceso principal y sus sub-procesos (entonces la limpieza es automática), en contraste con la memoria compartida de `ShareableList` o `SharedMemory`, a la que podemos acceder desde cualquier otro proceso (sabiendo el nombre usado al momento de creación).

```
$ python3 ej_paralelo_array.py
Operando entre 0 y 4
Operando entre 8 y 12
Operando entre 4 y 8
Resultado final: [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
```

Por otro lado, el elefante en la habitación que estamos esquivando desde que arrancamos el capítulo es que para el procesamiento de datos en general usamos NumPy y otras bibliotecas que nos permiten operar de forma extremadamente eficiente. ¿Como compaginamos esto con el multiprocesamiento?

Es muy sencillo, ya que por diseño NumPy nos expone la zona de memoria sobre la que trabaja, y eso nos permite integrarnos con las estructuras de memoria compartida de forma bastante directa.

```
1 import numpy as np
2 from concurrent.futures import ProcessPoolExecutor
3 from multiprocessing import shared_memory
4
5 # el tamaño de nuestras secuencias de datos increíblemente grandes y complejas
6 size = 12
7
8 # creamos toda la memoria compartida que vamos a usar (para los tres arreglos)
9 array_size = np.int8().size * size
10 shared_memory = shared_memory.SharedMemory(create=True, size=array_size * 3)
11
12 # los dos arreglos de entrada (con sus valores), y el de los resultados
13 input_a = np.ndarray((size,), dtype=np.int8, buffer=shared_memory.buf)
14 input_a.fill(1)
15 input_b = np.ndarray((size,), dtype=np.int8, buffer=shared_memory.buf, offset=array_size)
16 input_b.fill(3)
17 results = np.ndarray((size,), dtype=np.int8, buffer=shared_memory.buf, offset=array_size * 2)
18
19
20 def operator(limits):
21     l_inf, l_sup = limits
22     print(f"Operando entre {l_inf} y {l_sup}")
23     results[l_inf:l_sup] = input_a[l_inf:l_sup] + input_b[l_inf:l_sup]
24
25
26 all_limits = [
27     (0, 4),
28     (4, 8),
29     (8, 12),
30 ]
31 with ProcessPoolExecutor() as executor:
32     list(executor.map(operator, all_limits))
33
34 print("Resultado final:", results)
35
36 # cerramos y liberamos los recursos de memoria compartida
37 shared_memory.close()
38 shared_memory.unlink()
```

En este caso creamos sólo una zona de memoria compartida en la que ubicaremos los tres arreglos que manejamos: los dos de entrada y el de los resultados. Creamos las estructuras de NumPy usando `ndarray` que nos permite indicarle el buffer de memoria y la posición dentro del mismo.

Luego `operator` se vuelve muy simple, ya que al estar trabajando con NumPy podemos directamente sumar los arreglos, lo cual sumará uno a uno sus elementos. Por supuesto su ejecución en cada uno de los procesos sumará sólo parte del arreglo general, que es justamente el efecto que estamos buscando.

Finalmente, mostramos el resultado y liberamos los recursos de memoria compartida, lo cual siempre tenemos que hacer luego de *aprovechar* los resultados (en el caso de este ejemplo mostrándolos por pantalla), ya que una vez liberada la memoria compartida no tendremos más acceso a su contenido.

9.4.3. Ejemplo práctico

Cerremos la parte de procesamiento paralelo con un ejemplo práctico.

El desafío es hacer una transformada (rápida) de Fourier (*Fast Fourier Transform*, o FFT) sobre alguna señal obtenida de alguna máquina o sensor.



El algoritmo de la FFT básicamente toma un array de valores y deja otro; la fuente es la amplitud de la señal en cada momento de tiempo, y el resultado es una indicación de cuanta energía tiene la señal para cada frecuencia.

Para nuestro experimento o ejemplo práctico usaremos una fuente que ya tenemos en disco, en formato WAV, de manera que les sea fácil poder ejecutar los distintos scripts que usaremos a continuación sobre la entrada que deseen. Nosotros usamos los Conciertos de Brandeburgo de Johann Sebastian Bach, completos. Es más de una hora y media de audio, y el archivo pesa apenas más de un gigabyte.

El siguiente es el código secuencial (no paralelo) que logra el objetivo planteado:

```
1 import sys
2
3 import scipy.io.wavfile as wavfile
4 from matplotlib import pyplot as plt, ticker
5 from scipy.fft import fft, fftfreq
6
7 FREQ_MIN, FREQ_MAX = 20, 20000
8 TICKS = [27.5, 55, 110, 220, 440, 880, 1760, 3520, 7040, 14080]
9
10 src = sys.argv[1]
11 print("Cargando", src)
12 fs_rate, signal = wavfile.read(src)
13 print("Frecuencia de muestreo:", fs_rate)
14
15 q_channels = len(signal.shape)
16 print("Canales:", q_channels)
```

```

17
18 if q_channels == 2:
19     signal = signal.sum(axis=1) / 2
20 assert signal.ndim == 1
21 Ts = 1.0 / fs_rate
22 print("Duración [s]:", len(signal) * Ts)
23
24 FFT = abs(fft(signal))
25 half_len = len(FFT) // 2
26 FFT_side = FFT[1:half_len] # positive-frequency terms, excluding zero
27 freqs = fftfreq(signal.size, Ts)[1:half_len]
28 print("FFT terminado")
29
30 audible = (FREQ_MIN < freqs) & (freqs < FREQ_MAX)
31 FFT_side = FFT_side[audible]
32 freqs = freqs[audible]
33 assert len(FFT_side) == len(freqs)
34 print("Audible seleccionado, cantidad de puntos:", len(FFT_side))
35
36 (fig,) = plt.plot(freqs, FFT_side, linewidth=0.3)
37 plt.xscale("log")
38 plt.xticks(TICKS)
39 plt.xlim([FREQ_MIN * 0.9, FREQ_MAX * 1.1])
40 plt.xlabel('Frecuencia (Hz)')
41 fig.axes.xaxis.set_major_formatter(ticker.ScalarFormatter())
42 plt.savefig("fft-simple.pdf")
43 print("Listo")

```

Luego de los *imports* de rigor, lo primero que tenemos son unas constantes: como estaremos trabajando con audio, nos pareció interesante graficar la parte que corresponde al rango audible (entre 20 Hz y 20 kHz), y por la misma razón especificamos en qué frecuencias (las correspondiente a la nota LA en las distintas octavas) queremos los *ticks* del eje X.

Luego cargamos el archivo indicado en la línea de comandos, verificamos que sea monofónica (y si es stereo juntamos ambos canales, recuerden que estamos queriendo simular una señal simple), y calculamos la duración en segundos multiplicando la cantidad de valores por el período de la señal (la inversa de la frecuencia).

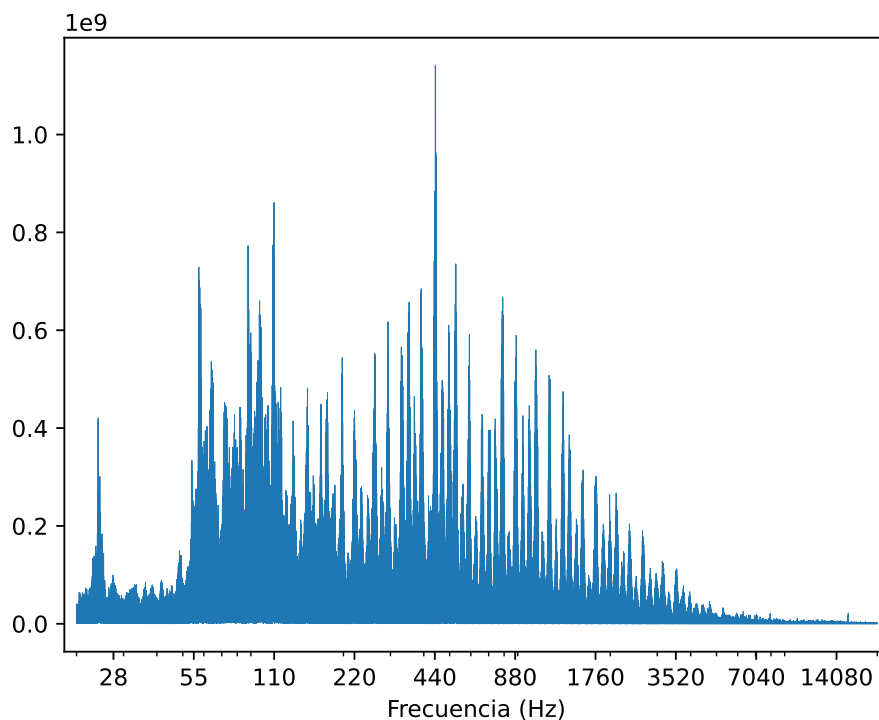
Luego sí pasamos a calcular la FFT propiamente dicha, tomamos sólo un “costado” del resultado obtenido (porque la función `fft` nos deja primero el cero, luego los valores positivos y finalmente los negativos, entonces para quedarnos con la parte positiva menos el cero tomamos desde el segundo valor hasta la mitad del array), y para terminar calculamos con `fftfreq` la frecuencia que corresponde a cada valor.

En este punto tenemos todo lo necesario para el gráfico: los valores para el eje Y (en `FFT_side`) y sus correspondientes posiciones en el eje X (en `freqs`). Pero habíamos dicho que queríamos sólo los valores correspondientes al rango audible, entonces calculamos una máscara⁶ (`audible` es un vector del largo de la señal con valores **True** y **False** según la frecuencia sea mayor que la mínima y menor que la máxima) y luego aplicamos esa máscara a los dos vectores en los que tenemos toda la información. Y como últimos pasos graficamos y guardamos la imagen resultante.

Como también vemos en el código, la ejecución nos va dando información de la señal y los pasos del procesamiento:

⁶ Pueden leer más sobre máscaras en el capítulo de Numpy¹¹.

```
$ python wav-fft-simple.py bach.wav
Cargando bach.wav
Frecuencia de muestreo: 44100
Canales: 2
Duración [s]: 6058.8
FFT terminado
Audible seleccionado, cantidad de puntos: 121054823
Listo
```



En las líneas de arriba no se nota, pero la ejecución tarda demasiado. Nos gustaría que fuese más rápido, necesitamos optimizarlo.

Una regla de oro a seguir cuando necesitamos optimizar código es que necesitamos medir para poder saber qué secciones del mismo necesitamos optimizar.

Como ya tenemos el código que va imprimiendo por pantalla los distintos pasos, podemos obtener una medición rápida de los mismos usando la utilidad `ts` (en Debian/Ubuntu puede instalarse haciendo `sudo apt install moreutils`) que nos agrega fecha y hora a cada línea de la salida de un programa.

```
$ python -u wav-fft-simple.py bach.wav | ts
2022-05-29 13:49:00.702611 Cargando bach.wav
2022-05-29 13:49:02.037465 Frecuencia de muestreo: 44100
2022-05-29 13:49:02.037580 Canales: 2
2022-05-29 13:49:09.187465 Duración [s]: 6058.8
2022-05-29 13:49:28.619166 FFT terminado
```

```
2022-05-29 13:49:29.242778 Audible seleccionado, cantidad de puntos: 121054823
2022-05-29 13:49:47.284587 Listo
```

Explicuemos brevemente cómo cambió la línea de ejecución. Ejecutamos nuestro programa con Python pero luego tenemos al *pipe* (`|`) que conecta la salida del programa de la izquierda con la entrada del de la derecha. En nuestro caso los prints de nuestro programa no van a pantalla sino que son leídos por `ts`, que les agrega la fecha y hora a la izquierda y ahí sí los muestra. Para lograr que las líneas de nuestro programa lleguen inmediatamente a `ts` (y no se queden retenidas en el buffer de entrada/salida que tienen los procesos para mejorar performance) ejecutamos a Python indicándole que no use ningún buffer (modo `unbuffered`, de allí la opción `-u`).

En la salida ahora notamos que el FFT en sí tarda bastante, pero también construir el gráfico al final. Esto es porque es demasiada la cantidad de puntos con los que estamos armando el gráfico: más de 121 millones, no hace falta tanto.

Entonces, la primer optimización es sencilla. En vez de armar el gráfico con todos ellos, usamos uno de cada cien (armando un gráfico con más de un millón de puntos, lo cual es más que suficiente), modificando sólo la línea donde pedimos el gráfico:

```
1 (fig,) = plt.plot(freqs[::100], FFT_side[::100], linewidth=0.3)
```

No hace falta mostrar todo el resto del código acá ya que no tiene cambios, pero queda almacenado en el archivo `code/wav-fft-optimiz.py`.

```
$ python -u wav-fft-optimiz.py bach.wav | ts
2022-05-29 14:02:15.010729 Cargando bach.wav
2022-05-29 14:02:15.545837 Frecuencia de muestreo: 44100
2022-05-29 14:02:15.545931 Canales: 2
2022-05-29 14:02:22.795551 Duración [s]: 6058.8
2022-05-29 14:02:40.845734 FFT terminado
2022-05-29 14:02:41.438602 Audible seleccionado, cantidad de puntos: 121054823
2022-05-29 14:02:41.905461 Listo
```

Ahora vemos como el tiempo que tarda en construirse el gráfico no influye en el total.

Es momento de paralelizar la transformada rápida de Fourier en sí, y acá nos encontramos con un gran problema. Como explicábamos arriba, la transformada rápida de Fourier nos pasa del dominio del tiempo a la frecuencia, y en su cálculo discreto debemos tener en cuenta toda la señal para calcular cada valor del vector resultante.

Es por esto que no podemos cortar la señal original “en partes” y procesar cada una de esas partes en procesadores separados. En otras palabras, si tenemos una señal de 100 segundos de largo, no podemos analizar por separado 10 bloques de 10 segundos cada uno, ya que para calcular cada valor del vector de frecuencias resultantes necesitamos siempre analizar los 100 segundos completos.



En realidad, matemáticamente, sí hay forma de paralelizar el cálculo, pero implica que implementemos FFT nosotros mismos para lograr esa paralelización, lo que ya es otra escala de complejidad y se escapa de este libro.

Si no tenemos en cuenta lo antedicho, podemos inocentemente tratar de cortar la señal en partes y luego sumar los resultados de cada análisis; por completitud implementamos esto en `code/wav-fft-paralelo-mal.py` donde analiza la señal en dos partes (el código permite extenderlo sencillamente a muchas) y termina con un resultado que ya se nota incorrecto con una simple inspección visual del gráfico generado. Dejamos a los lectores el ejercicio de ejecutar el código y compararlo con el resultado correcto.

Algo podemos hacer, sin embargo. Teniendo en cuenta que para cada valor del resultado necesitamos sí o sí recorrer toda la señal, lo que podemos hacer es re-muestrear la señal cada N elementos en cada procesador, y luego sí sumar los vectores resultados. Supongamos que paralelizamos el procesamiento en 3 partes, entonces mandaremos los valores de las posiciones 0, 3, 6... al primer procesador, los de las posiciones 1, 4, 7... al segundo, y los de las posiciones 2, 5, 8... al tercero.

Ahora, sabiendo que tenemos que procesar la señal en distintos procesos, es momento de ocuparnos de cómo va a estar disponible la señal para los distintos procesos y cómo cada uno va a dejar su resultado para el proceso principal. De forma muy similar a lo que explicamos antes, usaremos memoria compartida tanto para la señal de entrada como para el resultado.

Implementamos todo esto en el `code/wav-fft-paralelo-ok.py`, del cual comentaremos aquí sólo las partes que cambiamos, para ayudar a entenderlas.



Jean-Baptiste Joseph Fourier - Matemático y físico francés (1768-1830) conocido por sus trabajos sobre la descomposición de funciones periódicas en series trigonométricas convergentes llamadas Series de Fourier, método con el cual consiguió resolver la ecuación del calor.

```

28 signal_bytes_length = signal.itemsize * signal.size
29 signal_mem = shared_memory.SharedMemory(create=True, size=signal_bytes_length)
30 shared_signal = np.ndarray(signal.shape, dtype=signal.dtype, buffer=signal_mem.buf)
31 shared_signal[:] = signal
32 del signal
33
34 result_size = shared_signal.size // MULTPARTS // 2 - 1
35 result_bytes_length = shared_signal.itemsize * result_size
36 result_mem = shared_memory.SharedMemory(create=True, size=result_bytes_length)
37 shared_result = np.ndarray((result_size,), dtype=shared_signal.dtype, buffer=result_mem.buf)

```

El primer cambio es pasar a usar memoria compartida así cada proceso no tiene que tener una copia de las estructuras de datos de señal y resultado.

En ambos casos la idea es la misma: instanciamos `SharedMemory` con la cantidad de bytes necesarios, y luego construimos un array de NumPy con la forma y tipos de nuestra señal pero indicándole que la zona de memoria a utilizar sea la compartida.

Para el caso de la señal de entrada, luego de obtener la memoria compartida tenemos que copiar allí la señal misma, objeto que luego borramos justamente para liberar esa memoria y que no se copie a cada nuevo proceso. Para el caso del resultado es suficiente con crear el array arrancando desde la memoria compartida, ya que así tendrá todos sus valores en cero.

```

40 def calculate_partial_fft(offset):
41     global shared_result
42
43     FFT = abs(fft(shared_signal[offset:MULTPARTS]))
44     half_len = len(FFT) // 2
45     useful = FFT[1:half_len] # positive-frequency terms, excluding zero
46     with result_lock:
47         shared_result += useful
48
49
50 result_lock = Lock()
51 with ProcessPoolExecutor() as executor:
52     executor.map(calculate_partial_fft, list(range(MULTPARTS)))

```

Aquí tenemos al procesamiento en paralelo en sí. Usamos `ProcessPoolExecutor` para ejecutar nuestra función en distintos procesadores, trabajando en cada caso con un `offset` distinto.

Como guardamos el resultado final en `shared_result` sumando los valores de cada resultado parcial a esa estructura, tenemos que acceder a la variable global indicándolo a Python que justamente es global (podríamos tener una estructura más compleja para no usar una variable global, usando una clase que tenga el objeto a acceder y nuestra función como método interno, pero sacrificamos un poco de pureza en función de hacer el ejemplo más legible y conciso). También tenemos que usar un lock para no caer en el problema de que funciones en distintos procesos modifiquen esa estructura al mismo tiempo.

Debemos prestar atención al *slice* que se le hace a la señal al calcular la transformada; supongamos que al ejecutarse la función recibió el `offset` 3, eso querrá decir que esa función en ese procesador usará los valores 3, 11, 19, etc., porque arranca en 3 (el `offset`) y va tomando valores de a 8 (por `MULTPARTS`). El resto de la función toma la mitad útil del resultado (como habíamos explicado para el caso simple), para devolverlo al proceso principal.

```

71 for mem in (signal_mem, result_mem):
72     mem.close()
73     mem.unlink()

```

Y luego al final del script limpiamos los recursos de la memoria compartida.

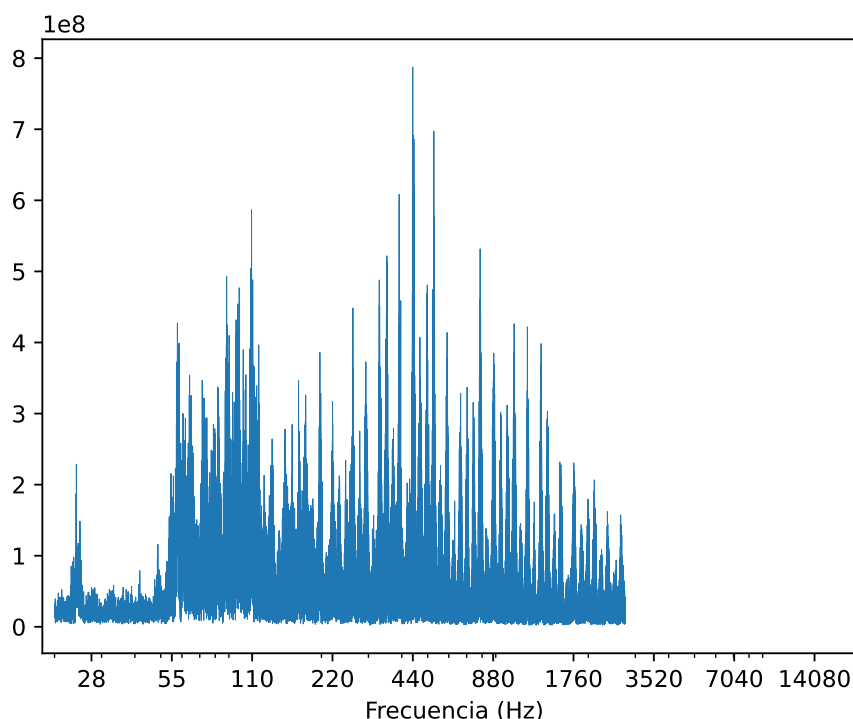
Podemos ver que la ejecución es más rápida que en el caso secuencial:

```

$ python -u wav-fft-paralelo-ok.py bach.wav | ts
2022-05-29 14:21:25.566173 Cargando bach.wav
2022-05-29 14:21:25.920670 Frecuencia de muestreo: 44100
2022-05-29 14:21:25.920731 Canales: 2
2022-05-29 14:21:31.560241 Duración [s]: 6058.8
2022-05-29 14:21:46.718089 FFT terminado
2022-05-29 14:21:46.857300 Audible seleccionado, cantidad de puntos: 16578390
2022-05-29 14:21:47.354224 Listo

```

Y el gráfico resultante es correcto... hasta cierto punto:



Vemos que entre las frecuencias 1760 y 3520 la curva se trunca. Esto se explica por el teorema de Nyquist-Shannon, que nos indica que sólo vamos a tener información sobre una señal hasta la mitad de la frecuencia de muestreo. En el diagrama original teníamos datos hasta 20 kHz sin problemas porque la frecuencia de muestreo del WAV es 44.1 kHz, pero en nuestro código paralelizado hicimos que cada uno de los 8 procesadores recorra la señal cada 8 valores, reduciendo efectivamente la frecuencia de muestreo a $44100 / 8 = 5512.5$ Hz, por eso luego tenemos datos hasta la mitad de ese valor.

El corolario de este experimento es que tenemos que entender que paralelizar el cálculo científico no sólo tiene el inconveniente de cómo implementamos esa paralelización en uno u otro lenguaje, sino que muchas veces (la mayoría) implica tener un conocimiento específico del dominio con el que estamos trabajando.

Los distintos lenguajes y sus bibliotecas estarán más o menos preparados para lograr un paralelismo del cálculo o la tarea, y nos costará más o menos trabajo lograr el resultado final, pero nunca tenemos que perder de vista que el elemento clave en la paralelización que queremos obtener es el entender las limitaciones intrínsecas de lo que queremos paralelizar (como el caso de la transformada rápida de Fourier que acabamos de ver, o por ejemplo si partes del cálculo dependen de otras partes previas) y las limitaciones al implementarlas (cuanto se puede enviar a otros procesadores o no, cuantos datos se deben compartir en memoria, etc.).

Y también es tarea del desarrollador o la científica entender y elegir cuales batallas pelear. Los códigos de arriba son genéricos con respecto al tipo de WAV que aceptan, y si reciben uno estéreo mezcla los canales para tener una sola señal con la que trabajar. Pero vemos (por ejemplo en la última ejecución) que esa operación de mezclado de los canales (el `signal = signal.sum(axis=1) / 2`) tarda casi 6 segundos, llevándose el 25 % del tiempo total de ejecución, ¡lo cual es muchí-

simo! ¿Vale la pena tratar de optimizar esto? Seguramente no podremos mejorar la suma que hace NumPy internamente para mezclar los canales. ¿E implementar nuevamente la función de lectura del WAV para dejar directamente una señal mezclada? ¿O limitar el script genérico para WAVs monoaurales? Este tipo de preguntas se tendrán que contestar analizando cada caso y cada necesidad en particular.



10 | Interfaces gráficas

Las interfaces que nos ofrecen los programas en un sistema informático se dividen principalmente entre si están pensada para otras aplicaciones o para personas. Estas últimas a su vez se dividen en varios tipos, siendo las gráficas nuestro motivo de estudio en este capítulo.

También conocidas como GUI (del inglés *graphical user interface*), las interfaces gráficas utilizan un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles. Es una herramienta muy poderosa a la hora de proporcionar un entorno visual sencillo para permitir la utilización de nuestro sistema por quienes necesiten usarlo.

No es un concepto para nada nuevo, las interfaces gráficas datan de la década de 1970, y por eso hay infinidad de variaciones. Sólo a modo de ejemplo, podemos citar algunos frameworks que son multiplataforma, como [Qt](#), [GTK+](#) o [Tk](#), y otros que son específicos a algún sistema operativo, como [Cocoa](#) para MacOS o las [Microsoft Foundation Classes \(MFC\)](#) para Windows. Y no sólo tenemos que pensar en computadoras de escritorio o laptops, por ejemplo con [Kivy](#) también podemos armar

Módulo	Versión
matplotlib	3.9.2
PyQt6	6.7.1
scipy	1.14.1

[Código disponible](#)

programas con interfaces gráficas para teléfonos.

Para este capítulo vamos a utilizar Qt, aunque la gran cantidad de conceptos sobre los que hablaremos son comunes a todos los frameworks de interfaces gráficas, más allá de sus características particulares.

10.1. Qt

Qt es un framework para construir interfaces gráficas (aunque también tiene su utilización en sistemas no gráficos) que corre en los tres principales sistemas operativos de escritorio (Linux, MacOS y Windows), algunos modelos de teléfonos y sistemas integrados, y hay planes a futuro para que funcione en Android e iOS.

Al ser una sigla vale la pena la aclaración con respecto a su pronunciación: en castellano decimos “cu-té”, mientras que en inglés se pronuncia “quiút” (no “quiú-ti”).

En realidad no vamos a usar Qt directamente, porque tendríamos que programar en C++, sino que vamos a usar PyQt, un *wrapper* alrededor del framework base. Mencionaremos luego algunas diferencias, pero para todos los efectos prácticos, seguiremos hablando de Qt o PyQt indistintamente en el resto del capítulo.

Qt, como la mayoría de los frameworks de interfaces gráficas, es un sistema asincrónico (si todavía no leyeron la sección sobre Async9.3 es un buen momento, pero no es estrictamente necesario para entender el resto de este capítulo). Por eso podemos decir que un programa de interfaz gráfica está basado en eventos: una vez que arrancó y terminó de armar todos los elementos en una ventana el sistema no estará realmente haciendo nada hasta que reciba eventos y los procese.

Estos eventos pueden venir de distintas fuentes: entrada del usuario (vía el teclado o el mouse, por ejemplo) o del sistema (alarmas luego que pasó determinado tiempo, escritura o lectura en distintas entradas/salidas, etc.). Cuando sucede el evento el reactor que maneja el sistema asincrónico (Qt mismo) ejecutará la función correspondiente para manejarlo.

Tengamos en cuenta que no todos los eventos nos interesan en una aplicación de interfaz gráfica. Aunque esos eventos son todo lo que le puede “suceder” a una ventana de nuestra aplicación, hay muchos que están manejados directamente por los componentes que ya agregamos a esa ventana. Por ejemplo, si dejamos el mouse quieto sobre uno de los iconos de la barra de herramientas de nuestra aplicación aparecerá un mensaje, pero supervisar al mouse para detectar la situación y hacer aparecer ese mensaje no es algo que tengamos que programar de nuestro lado, es una funcionalidad integrada del componente mismo que usamos para tener ese icono en esa barra de herramientas, al cual sólo le tenemos que indicar qué mensaje queremos que aparezca en ese caso.

Por otro lado, hay muchas veces que sí queremos reaccionar a los distintos eventos. Por ejemplo, ponemos un botón en nuestra aplicación y queremos ejecutar una función “nuestra” cuando se presiona ese botón. Ya veremos cómo se hace esto.

Estos componentes de la interfaz gráfica se llaman *widgets* (la traducción al castellano es



PyQt - Conjunto de enlaces de Python para el framework Qt, que permite desarrollar interfaces gráficas de usuario (GUI) sofisticadas y multiplataforma.

“artilugio” pero en realidad siempre usamos *widget*) y permiten hacer uso de la aplicación y los servicios que esta provee. Como ejemplo, mencionamos aquí algunos widgets utilizando su nombre en Qt y la funcionalidad que ofrecen (estos son widgets bien simples, en la mayoría de otras interfaces gráficas encontraremos widgets similares a estos con nombres parecidos o casi iguales):

- QLabel: muestra un texto o imagen
- QLineEdit: permite ingresar una línea de texto
- QTextEdit: permite ingresar un texto de cualquier cantidad de líneas
- QPushButton: un botón para invocar una acción al ser apretado
- QRadioButton: permite elegir una opción (y sólo una) entre muchas
- QCheckBox: permite elegir una o más opciones
- QDialog: otra ventana con una funcionalidad acotada y específica usada para recabar información desde la ventana que la genera

Si pensamos en cualquier programa de interfaz gráfica reconoceremos que usan alguna combinación de los widgets recién mencionados, entre otros. Más adelante en este capítulo mismo usaremos algunos de esos widgets y otros también.

10.1.1. Unas palabras sobre la documentación

Como mencionamos antes, en este capítulo vamos a utilizar PyQt para explicar los conceptos de interfaces gráficas y armar una aplicación ejemplo.

Si buscamos documentación online sobre PyQt encontraremos muchos tutoriales y ejemplos de cómo hacer distintas aplicaciones o estructurar distintas funcionalidades haciendo el programa en Python. Pero a la hora de buscar la documentación de referencia de los distintos widgets y otros componentes de PyQt, veremos que casi no hay nada, y siempre en algún punto terminan apuntando a la referencia escrita para Qt mismo, en C++.

Esto sucede porque PyQt es una capa arriba de la biblioteca de Qt en C++, permitiendo (más allá de proveer alguna simplicidad extra) poder usarla desde Python. Reescribir toda la documentación para hacerla en PyQt sería un esfuerzo mayúsculo, por el tamaño de la misma y por el hecho de que cambia continuamente (especialmente entre las versiones principales de la librería, de Qt 4 a Qt 5, por ejemplo). Esto, sumado a que entender la documentación en C++ (aunque usemos PyQt) es bastante sencillo, hace que nunca se encare ese esfuerzo.

Un buen punto de entrada a la documentación de referencia es [la lista de todas las clases disponibles](#), porque en general nos va a suceder que sabemos que tenemos que usar un determinado widget, pero no sabemos cómo explotarlo al máximo o queremos aprender de sus distintas funcionalidades. Desde allí podremos ir a cualquiera de las clases que usaremos cuando programemos en (Py)Qt.

Entonces la idea es transmitirles que intenten usar la documentación escrita en C++, que a priori parecerá muy críptica pero es cuestión de acostumbrarse a leerla para poder aprovecharla. Para facilitarles este proceso les recomendamos en hacer foco en las siguientes secciones cuando vayan a la documentación de referencia de alguno de los widgets:

- *Public Functions y Reimplemented Public Functions*: aquellos métodos del widget que podemos llamar para explotar todas sus posibilidades.
- *Signals*: aquellas señales a las que les podemos conectar una función nuestra para reaccionar a eventos en el widget (o directamente conectarles un *slot*... hablaremos luego más en profundidad sobre *signals* y *slots*)
- *Detailed Description*: una explicación a fondo de para qué sirve el widget, donde normalmente se mencionan distintas opciones y capacidades funcionales, y es un gran punto de entrada no sólo para aprender sobre el widget sino para entender los contextos en los que se lo puede utilizar.

Cabe destacar que todas estas clases tienen una estructura de herencia, y también tenemos que acostumbrarnos a eso cuando busquemos información. Por ejemplo, si vamos a la documentación del [QPushButton](#) (el botón “normal” de cualquier interfaz gráfica) veremos que no tiene ninguna señal listada... pero encontramos que hereda de [QAbstractButton](#) (la clase de la que derivan los distintos tipos de botones) y ahí están las señales que estábamos buscando, como `clicked`, que se emite cuando el botón es activado (o sea, apretado y soltado mientras el cursor del mouse está dentro del botón).

Luego podrán explorar más a fondo, pero ya con esta información podrán acceder a la documentación de la mayor parte de la funcionalidad que van a necesitar.

10.2. Una aplicación mínima

Para empezar a entender las distintas funcionalidades que podemos encontrarnos en una aplicación gráfica, y cómo usar esas funcionalidades desde PyQt, armemos una aplicación muy básica.

Este mínimo de código que presentaremos en esta sección nos permitirá charlar sobre distintos conceptos fundamentales, y luego podremos encarar una aplicación más real.

Arranquemos con la “cáscara” mínima que necesitaremos siempre (el código en `01.empty.py`).

```
1 import sys
2
3 from PyQt6.QtWidgets import QApplication, QWidget
4 from PyQt6.QtGui import QIcon
5
6
7 class Window(QWidget):
8     def __init__(self):
9         super().__init__()
10        self.resize(250, 150)
11        self.setWindowTitle("Primer ejemplo")
12        self.setWindowIcon(QIcon("icon.png"))
13
14
15 app = QApplication(sys.argv)
16 window = Window()
17 window.show()
18 sys.exit(app.exec())
```



Para ejecutar este ejemplo (y todos los de este capítulo) sigan las instrucciones del README que hay en el directorio de código correspondiente.

Aquí ya vemos las tres secciones que se repetirán en todos los programas. Primero, una secuencia de *imports* (líneas 1 a 4), que irá creciendo a medida que vayamos necesitando más widgets y funcionalidades. Luego, todas las clases que compondrán la estructura de nuestra aplicación; en este caso tenemos sólo una (líneas 7 a 12), que explicaremos luego. Y finalmente una serie de instrucciones para que arranque el sistema (líneas 15 a 18).

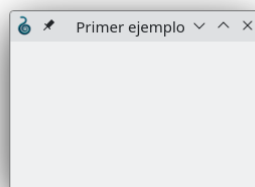
Estas últimas líneas no cambiarán para nada en todo nuestro ejemplo, y en aplicaciones más reales sólo se verán afectadas según alguna necesidad de procesamiento de opciones de línea de comando (que en aplicaciones de interfaz gráfica no es normal tener). Lo más importante de esta secuencia es la inicialización de *QApplication*, la cual maneja la configuración a nivel QT y el flujo de control de la aplicación; contiene el *event loop* principal que procesará y despachará todos los eventos, tanto los generados por los elementos de las ventanas como de otras fuentes.

Luego de inicializar *QApplication* (línea 15) instanciamos nuestra ventana y la mostramos (líneas 16 y 17), lo cual no podemos hacer antes (necesitan tener el sistema ya inicializado). Pero si terminamos ahí, nuestro programa no hará nada. Es que recién al ejecutar el `app.exec()` estamos dándole el control a *event loop* de Qt, que se quedará manejando el sistema hasta que nuestra aplicación termine (devolviendo un código final que lo pasamos directamente a `sys.exit()` para que sea el código de retorno del proceso).

Enfoquémonos entonces en la parte del medio de nuestro programa.

La clase *Window* hereda de *QWidget*, y lo único que hacemos es ajustar tres detalles en tiempo de inicialización (luego de primero inicializar la clase padre en la línea 9: recordemos que cuando redefinimos un método que ya posee la clase de la que derivamos, al llamarse al método sólo se ejecutará el de la clase hija, y si queremos que también se ejecute el método de la clase ancestral debemos llamarlo explícitamente). Las tres líneas donde configuramos el *widget* son bastante autoexplicativas: cambiamos el tamaño de la ventana, le ponemos un título, y le configuramos un icono.

En general construiremos toda nuestra aplicación gráfica heredando de determinados widgets y agregándole la funcionalidad particular que necesitamos en cada caso. Esta estructura no es obligatoria, pero al ser PyQt un framework fuertemente estructurado alrededor de un árbol de clases, si continuamos con esa metodología nos quedará un código prolijo y fácil de entender.



En realidad una ventana vacía tampoco hace a una aplicación, agreguemos algo simple (un

texto) y una forma de interacción (un botón que haga algo).

Pero antes, ahora que vamos a agregar widgets en nuestra ventana, unas palabras sobre dónde y cómo agregarlos.

En los comienzos de las aplicaciones gráficas, los elementos dentro de una ventana tenían una posición fija (incluso hoy en día se construyen aplicaciones así), donde cada widget se ubicaba en una coordenada específica dentro de la ventana. Esto siempre termina en una pobre experiencia en el uso, ya que si cambia el largo de los textos (por traducciones), o el tamaño de la ventana, o el tamaño de la tipografía (por configuraciones de accesibilidad del sistema), todo queda amontonado solapándose los elementos de la ventana o demasiado disperso.

Hay una forma superadora de ubicar elementos dentro de una ventana que es a través de *layouts* (“disposiciones”, en castellano, pero se usa el término en inglés). Por ejemplo, si queremos poner tres widgets uno al costado del otro usaremos un *layout* horizontal y agregaremos los tres widgets al layout. Tanto el tamaño del layout como de los widgets dentro se ajustarán dinámicamente en función del espacio mínimo que cada uno tiene que ocupar (por su contenido) y el tamaño máximo manejado por el usuario (por ejemplo al agrandar la ventana). Estos layouts tienen varias configuraciones que permiten controlar su comportamiento y ofrecer una solución elegante, como espacio entre los widgets, qué widget crecería con prioridad si se agranda la ventana, etc.

Obviamente hay layouts verticales y horizontales, y a través de su composición podemos lograr cualquier interfaz, pero también existen algunos más específicos, como un layout de grilla para aquellas disposiciones que son claramente una cuadrícula.

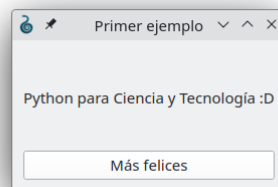
El segundo ejemplo lo tenemos en `02.content.py`, del cual mostraremos aquí sólo la clase del widget, ni la sección donde importamos los módulos necesarios ni las líneas para iniciar el programa. Para nuestra ventana ejemplo usaremos sólo un layout vertical, el cual instanciamos y configuramos como layout del widget en las líneas 15 y 16.

```
8 class Window(QWidget):
9     def __init__(self):
10         super().__init__()
11         self.resize(250, 150)
12         self.setWindowTitle("Primer ejemplo")
13         self.setWindowIcon(QIcon("icon.png"))
14
15         layout = QVBoxLayout()
16         self.setLayout(layout)
17
18         self.message = "Python para Ciencia y Tecnología"
19         self.label = QLabel(self.message)
20         self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
21         layout.addWidget(self.label)
22
23         button = QPushButton("Más felices")
24         button.clicked.connect(self.happier)
25         layout.addWidget(button)
26
27     def happier(self):
28         self.message += " :D"
29         self.label.setText(self.message)
```

Los dos bloques de código siguientes crean y agregan al layout el texto y el botón.

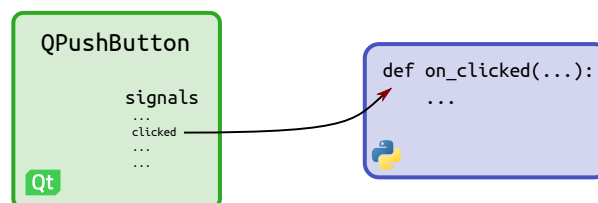
El texto lo agregamos a través de un `QLabel`, al cual le indicamos que lo alinee al centro (atención a la forma de indicar esto, no se usan cadenas como `'center'` sino flags, ya predefinidas en el framework), y finalmente agregamos el widget al layout (líneas 18 a 25). Notar que tanto el mensaje como el widget en sí los creamos en la instancia de la ventana: esto es sólo necesario porque luego usaremos ambos desde un método de la instancia.

Al botón lo creamos directamente con el texto que queremos que tenga, luego conectamos su señal `clicked` a un método que creamos abajo, y agregamos el widget al layout (líneas 23 a 25). El método `happier` entonces se ejecutará cada vez que presionemos el botón: modificará el mensaje y le indicará al `QLabel` que tenemos en la ventana que use este nuevo texto.

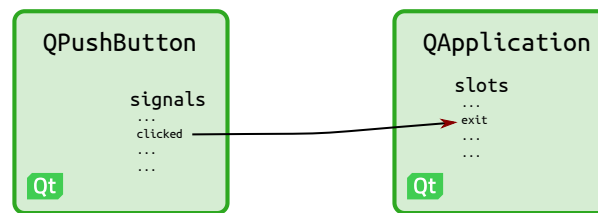


El botón que usamos, como la mayoría de los widgets, está diseñado para emitir señales en respuesta a uno o más eventos. Nosotros utilizamos `clicked`, pero tiene otras (en realidad, como ya mencionamos antes, las hereda de `QAbstractButton`). Incluso podemos crear nosotros nuestras propias señales (y lo haremos casi al final de este capítulo)

Las señales, en general, pueden conectarse a cualquier función de Python (o más de una), las que pueden estar definidas para no recibir nada (como en nuestro caso) o el evento que generó la señal (cuando ese evento nos trae información que necesitamos). Es que las señales pueden generarse por distintas situaciones: muchas son iniciadas por una acción de la persona que usa la interfaz pero eso no es una regla, lo que dispara la señal dependerá de cual sea la señal. Tengamos en cuenta que la señal en sí misma no hace nada, cuando se dispara sólo terminará ejecutando una o más funciones que hayamos conectado manualmente o conectadas por el framework mismo.



Por otro lado, cuando las conectamos no siempre tenemos que conectarlas a funciones de Python. También podemos conectarlas directamente a *slots* que los widgets tienen. De esta manera podemos relacionar widgets entre sí y que unos reaccionen a eventos de otros. Aunque las opciones son limitadas, porque dependemos de los slots que tenga definido cada widget, es interesante estar atentos a la posibilidad ya que es una solución extremadamente eficiente (porque todo se resuelve dentro del framework compilado sin salir a la capa de Python).



Para terminar la sección y antes de arrancar con una aplicación más real, nos permitimos un comentario sobre por qué diseñamos la interfaz gráfica de la aplicación usando código y no un diseñador gráfico como *Qt Designer*. Esta aplicación (no es la única, cada framework tiene algo similar) permite armar la ventana *visualmente*. Esta forma de construir ventanas era de gran ayuda en la época en que los widgets se acomodaban a mano en posiciones específicas, pero al usar layouts esa ventaja se pierde. Construir nuestros widgets especializados en código nos permite lograr una mejor reutilización del código que armamos (por ejemplo, podemos tener un layout con una separación específica entre widgets y reutilizarlo todo el tiempo), y también es mucho más fácil buscar ayuda sobre cómo realizar determinadas cosas.

10.3. Una aplicación real

En esta sección vamos a armar una aplicación completa con una doble intención: por un lado mostraremos varios widgets y técnicas necesarias cuando queremos dar una buena usabilidad sobre nuestro programa, y la otra es que ustedes tengan una aplicación “base” para construir las suyas, un código que puedan modificar y adaptar para cumplir vuestras necesidades.

Es por esto último que pensamos en una aplicación que bien podría ser típica en un entorno científico: permitir cargar un archivo de datos y mostrar un gráfico generado a partir de esos datos, según alguna opción. En nuestro caso el archivo está en formato CSV y los datos son coordenadas en el plano, el cálculo que hacemos es ajustar una función polinómica a esos puntos (dejando que se elija en la interfaz el grado de la función) y luego graficar los puntos en sí y la función ajustada.

Por razones pedagógicas partiremos del ejemplo mínimo que traemos de la sección anterior e iremos agregando gradualmente nuevas funcionalidades, explicándolas en cada caso. Pero si no les importa el *spoiler*, pueden arrancar por el final y ejecutar el código `09.complete.py` para ver la aplicación terminada.

Pero, decíamos, vayamos por partes. El primer gran cambio con respecto a lo que teníamos antes es que no queremos una ventana así no más, sino una ventana “principal” a la que le podamos agregar menú, barra de tareas, e incluso una barra de estado. Tenemos todo esto implementado en `03.app.py`, del que mostramos sólo la clase principal:

```

14 class Window(QMainWindow):
15     """Ventana principal."""
16
17     def __init__(self):
18         super().__init__()
19         self.resize(350, 200)

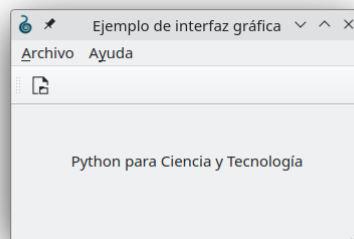
```

```
20     self.setWindowTitle("Ejemplo de interfaz gráfica")
21     self.setWindowIcon(QIcon("icon.png"))
22
23     # menú y barra de herramientas
24     toolbar = QToolBar("main-toolbar")
25     self.addToolBar(toolbar)
26     self._status_bar = QStatusBar(self)
27     self.setStatusBar(self._status_bar)
28     menubar = self.menuBar()
29
30     # las diferentes acciones tanto para el menu como para la barra de herramientas
31     open_action = QAction(QIcon.fromTheme("document-open"), "Abrir archivo", self)
32     open_action.setToolTip("Abrir archivo de datos")
33     open_action.triggered.connect(self.on_open)
34     open_action.setShortcut('Ctrl+A')
35     quit_action = QAction(QIcon(), "Salir", self)
36     quit_action.triggered.connect(app.exit)
37     about_action = QAction(QIcon(), "Acerca de...", self)
38     about_action.triggered.connect(self.on_about)
39
40     # configuramos la barra de herramientas
41     toolbar.addAction(open_action)
42
43     # configuramos el menú
44     menu = menubar.addMenu("&Archivo")
45     menu.addAction(open_action)
46     menu.addAction(quit_action)
47     menu = menubar.addMenu("A&yuda")
48     menu.addAction(about_action)
49
50     self._status_bar.showMessage("Comenzando", 3000)
51
52     label = QLabel("Python para Ciencia y Tecnología")
53     label.setAlignment(Qt.AlignmentFlag.AlignCenter)
54     self.setCentralWidget(label)
55
56     def on_open(self):
57         pass
58
59     def on_about(self):
60         pass
```

En este código vemos las distintas secciones donde armamos la funcionalidad recién indicada, pero prestemos atención a que ya no tenemos un widget genérico sino que heredamos de `QMainWindow`. Entonces, además de ese detalle importante y las primeras líneas donde fijamos el tamaño de la ventana y ponemos el título y el icono...

- creamos la barra de herramientas y la barra de estado, las agregamos a la ventana principal, y le pedimos la barra de menú (líneas 24 a 28)
- creamos distintas acciones como paso separado, ya que luego usaremos esas acciones tanto para el menú como para la barra de tareas (líneas 31 a 38); mencionaremos abajo algunos detalles de estas líneas para explicarlas mejor
- agregamos una sola acción a la barra de tareas, para abrir el archivo (línea 41)

- agregamos todas las acciones al menú, bajo dos secciones principales: Archivo y Ayuda (líneas 44 a 48); notar que el nombre de cada sección en el menú tiene un & antes de alguna letra: esa letra estará subrayada al mostrarse y permitirá acceder a la sección con la tecla *Alt* más esa letra
- mostramos un mensaje en la barra de estado que desaparecerá luego de 3 segundos (línea 50)
- y finalmente agregamos un texto como todo widget central en la ventana (que luego más adelante reemplazaremos por contenido más avanzado)



Volvamos a la parte de creación de las acciones para resaltar algunos detalles.

La primer acción, para abrir el archivo de datos, es especial porque es la que usaremos también en la barra de tareas. Por eso es que nos interesa que tenga un icono con sentido (que no cargamos de un archivo, sino que lo tomamos del stock de imágenes del sistema operativo) y también un *tooltip* (que es el texto que aparece cuando dejamos unos segundos el mouse quieto arriba del icono de la barra de tareas).

Por otro lado, las acciones de abrir el archivo y de mostrar el “acerca de” de la aplicación tienen su señal triggered conectada a métodos de nuestra clase (definidos en las líneas 56 a 60, pero todavía sin implementar). Notar que a la acción para salir del programa la conectamos directamente al slot exit de la aplicación ya que no hace falta que hagamos nada específico en el medio.

El próximo paso en la aplicación es permitir abrir un archivo y cargar los datos correspondientes. Es hora de implementar el método `on_open`, el cual tenemos en `04.load.py`:

```

59 def on_open(self):
60     """Abre un nuevo archivo de datos y refresca el widget principal."""
61     self._status_bar.showMessage("Abriendo archivo con datos", 3000)
62
63     # abrimos un diálogo para que se seleccione un archivo
64     filepath, filters = QFileDialog.getOpenFileName(self, "Abrir archivo")
65     if not filepath:
66         self._status_bar.showMessage("Ningún archivo seleccionado", 3000)
67         return
68     filepath = pathlib.Path(filepath)
69
70     # abrimos el archivo y levantamos los datos

```

```

71     try:
72         with open(filepath) as csvfile:
73             reader = csv.reader(csvfile)
74             raw_data = list(reader)
75     except Exception as exc:
76         self._status_bar.showMessage(f"No se pudo abrir el archivo correctamente: {exc}", 3000)
77         return
78
79     # convertimos los datos a punto flotante, validando largos y descartando las listas vacías
80     data = []
81     for idx, row in enumerate(raw_data, 1):
82         if not row:
83             continue
84         if len(row) != 2:
85             self._status_bar.showMessage(
86                 f"La línea {idx} del archivo no tiene exactamente dos columnas", 3000)
87             return
88         try:
89             converted = [float(datum) for datum in row]
90         except ValueError:
91             self._status_bar.showMessage(
92                 f"La línea {idx} del archivo tiene datos inválidos", 3000)
93             return
94         data.append(converted)
95
96     # tenemos los datos ok!
97     self._status_bar.showMessage("Archivo abierto correctamente", 3000)
98
99     label = QLabel(f"Registros cargados: {len(data)}")
100    label.setAlignment(Qt.AlignmentFlag.AlignCenter)
101    self.setCentralWidget(label)

```

Podemos separar este código en estas partes conceptuales:

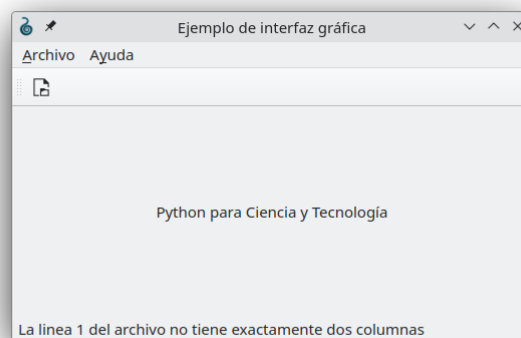
- mostramos un mensaje en la barra de estado informando que estamos abriendo el archivo de datos (línea 61)
- abrimos un “diálogo” para dejar elegir un archivo (hablaremos más sobre diálogos a continuación) en la línea 64, el cual nos puede dar un resultado vacío (en ese caso informamos que no hay archivo seleccionado), o el path a un archivo
- si tenemos un path válido lo abrimos, leemos, y validamos que tenga dos columnas de números, saliendo por error ante cualquier problema (líneas 70 a 94)
- si procesamos todo bien llegamos a la línea 97 donde informamos que se abrió el archivo correctamente
- finalmente cambiamos el texto central con un mensaje que muestra la cantidad de registros cargados (líneas 99 a 101)

Esa gran cantidad de líneas, de la 71 a la 93, es donde procesamos el archivo de datos, validando en cada uno de los pasos que la estructura y el contenido estén como esperamos (reportando un error y abortando la carga si no). Primero abrimos el archivo y lo leemos completamente con el módulo csv, para abortar si no se pudo abrir o está corrupto a nivel estructura (líneas 70 a

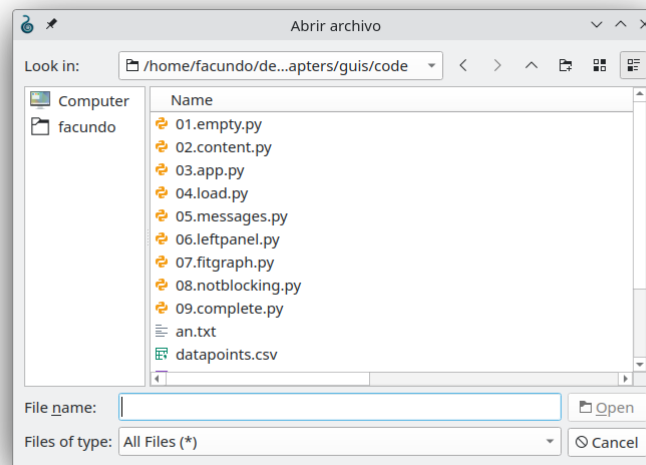
77). Luego procesamos cada línea validando que tenga dos columnas, y convirtiendo a número cada uno de los valores con `float` (líneas 79 a 94). Al final tenemos en `data` a toda la información correctamente convertida.

Este procesamiento puede parecer demasiado detallado, pero es importante considerar todos los casos de error e informar correctamente cuando algo sale mal: es la diferencia entre un programa que es fácil de usar y nos guía ante cualquier problema, y un programa que falla por “razones misteriosas”. Es verdad que la forma de mostrar cada error parece repetitiva y un poco oscura (sólo un mensaje en la barra de estado), pero vamos a mejorar esto en próximas iteraciones del código.

Vemos en la imagen cómo quedaría el error informado cuando abrimos un archivo malformado:



Antes de continuar hablemos sobre el widget que usamos para abrir el archivo. Un “diálogo” o “ventana diálogo” (en inglés *dialog window*) es una ventana que aparece arriba de la que la genera, usada en general para tareas cortas o comunicaciones breves con las personas que usan el programa. En el caso de nuestro ejemplo usamos `QFileDialog`, un diálogo con la función específica de permitir elegir un elemento del sistema de archivos. Particularmente, usamos su método de clase `getOpenFileName` que permite elegir un archivo que ya existe, pero el diálogo tiene otros métodos que permiten también seleccionar muchos archivos, elegir un directorio, elegir un path que quizás no exista todavía (útil para grabar un archivo nuevo), etc. Qt provee [varios diálogos pre-armados](#), pero siempre se puede heredar `QDialog` para armar uno específico para lo que se necesite.



Como regla general, los diálogos son modales. Una ventana es modal cuando bloquea la utilización de otras ventanas de la misma aplicación hasta que esa ventana se cierre, lo cual es muy útil para los diálogos, justamente, como en el caso de nuestro ejemplo donde no tiene sentido continuar a menos que se elija un archivo o se cancele el diálogo porque no se quiere elegir uno. Por otro lado, es normal tener ventanas no modales cuando la aplicación es de ventanas múltiples, donde se tienen simultáneamente varias ventanas abiertas de la misma aplicación y se puede usar cualquiera indistintamente.

En la próxima iteración de nuestro código, a continuación (en `05.messages.py`), nos ocuparemos de los mensajes.

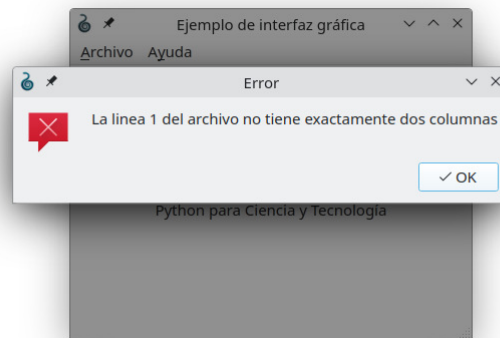
Para simplificar el código de toda la aplicación creamos dos métodos, uno que muestre un mensaje en la barra de estado (y reemplazamos todas las llamadas a `self._status_bar.showMessage` en las que había que poner siempre la misma duración), y otro para informar errores de forma genérica (al que llamamos en todos los casos de error al abrir y procesar el archivo fuente de datos).

```

61 def set_status(self, message):
62     """Muestra un mensaje efímero en la barra de estado."""
63     self._status_bar.showMessage(message, 3000)
64
65 def show_error(self, message):
66     """Muestra un mensaje de error via un diálogo y en la barra de estado."""
67     self.set_status(f"Error: {message}")
68     QMessageBox.critical(self, "Error", message)

```

Este último método no sólo muestra el mensaje en la barra de estado (que era lo que hacían las distintas llamadas originales) sino que también levanta un diálogo informativo (con un icono correspondiente al estado `critical`, con el título “Error” y el mensaje necesario).



Vemos en la imagen que tenemos una ventana modal (notar que la ventana “padre” está griseada porque no la podemos usar) con el mensaje de error, y ese mensaje aparece también en la barra de estado.

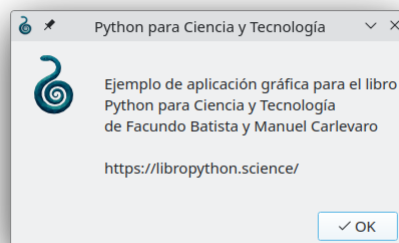
También nos ocupamos del método `on_about` que ya teníamos vinculado a la acción del menú.

```

111
112 def on_about(self):
113     """Muestra el diálogo de Acerca de."""
114     title = "Python para Ciencia y Tecnología"
115     text = textwrap.dedent("""
116         Ejemplo de aplicación gráfica para el libro
117         Python para Ciencia y Tecnología
118         de Facundo Batista y Manuel Carlevaro
119
120         https://libropython.science/
121     """)
122     QMessageBox.about(self, title, text)

```

Aquí volvemos a usar `QMessageBox` pero ahora para mostrar un dialogo con la estructura específica del *about*: un título, un mensaje largo para el cuerpo, y un icono que no hace falta especificárselo si es el mismo que ya le configuramos a la aplicación.



Es hora de ocuparnos del panel principal de la aplicación, en `06.leftpanel.py`.

Como es en ese panel donde tendremos toda la complejidad de mostrar los datos, es buena idea separar esa funcionalidad toda en otra clase, `MainPanel`, que explicaremos luego. En el código que teníamos hasta ahora lo único que haremos es eliminar esos textos temporales que teníamos (el inicial al abrir la aplicación y el que indicaba la cantidad de registros al abrir un archivo) y justamente integraremos el panel principal a la aplicación:

```

186     # armamos el área principal dentro de la ventana
187     if self.main_panel is not None:
188         self.main_panel.close()
189     self.main_panel = MainPanel(self, filepath, data)
190     self.setCentralWidget(self.main_panel)

```

Este código es el que se ejecutará luego de abrir exitosamente un archivo: instanciamos `MainPanel` (pasando una referencia a la ventana principal, el path del archivo y los datos cargados), y ponemos esta instancia como widget central de la aplicación.

Notar como antes de instanciar un nuevo `MainPanel` validamos no tener uno de antes, y en tal caso lo cerramos. Esto es para soportar abrir distintos archivos todas las veces que sea necesario: cada vez que se abra un nuevo archivo se descartará el panel anterior y se colocará uno nuevo.

En `MainPanel`, decíamos, tendremos toda la funcionalidad de la aplicación: mostraremos los datos, dejaremos elegir con qué tipo de función se ajustarán esos datos, y mostraremos el gráfico resultante. Por ahora, sin embargo, ocupémonos de todo eso menos el gráfico.

```

61 class MainPanel(QWidget):
62     """Panel principal."""
63
64     def __init__(self, main_window, filepath, data):
65         super().__init__()
66         self.main_window = main_window
67         self.data = data
68
69         # layout principal con dos layouts verticales a izquierda y derecha
70         main_layout = QHBoxLayout()
71         self.left_layout = QVBoxLayout()
72         main_layout.addLayout(self.left_layout)
73         self.right_layout = QVBoxLayout()
74         self.right_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
75         main_layout.addLayout(self.right_layout, stretch=1)
76         self.setLayout(main_layout)
77
78         # la etiqueta con el nombre del archivo que estamos usando
79         label = QLabel(filepath.name)
80         label.setAlignment(Qt.AlignmentFlag.AlignCenter)
81         self.left_layout.addWidget(label)
82
83         # armamos una tabla, le asociamos un modelo, y configuramos su estética
84         table = QTableView()
85         table.setModel(TableModel(data))
86         table.verticalHeader().setVisible(False)
87         horizontal_header = table.horizontalHeader()
88         horizontal_header.setStretchLastSection(True)
89         horizontal_header.setSectionResizeMode(1, QHeaderView.ResizeMode.Interactive)

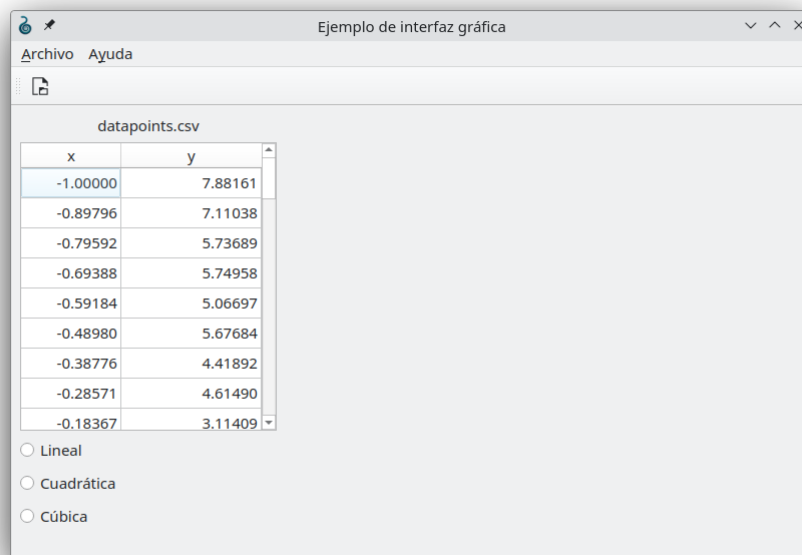
```

```
90     self.left_layout.addWidget(table)
91
92     # radiobuttons para la selección de qué función utilizar
93     options = QVBoxLayout()
94     self.left_layout.addLayout(options)
95     for degree, name in enumerate(["Lineal", "Cuadrática", "Cúbica"], 1):
96         radio = QRadioButton(name, self)
97         options.addWidget(radio)
```

Lo primero que hacemos al inicializar `MainPanel` es inicializar a su vez `QWidget` del cual heredamos y guardarnos en la instancia a esa ventana y los datos, que los vamos a usar más adelante.

Luego armamos el layout principal, horizontal, el cual tendrá dos layouts verticales en lo que llamaremos “izquierda” y “derecha”, y lo ponemos como layout del widget. Notar que al layout de la derecha (que tendrá el gráfico o un símbolo de espera si el cálculo tarda mucho), le indicamos que tome todo el espacio disponible con `stretch=1` al agregarlo al layout principal y le configuramos su alineación para que el contenido salga centrado. Esto no lo hacemos con el layout de la izquierda porque queremos que tenga el tamaño ajustado a su contenido.

Del layout de la izquierda, justamente, nos ocupamos ahora: le agregamos el nombre del archivo que abrimos, una tabla con los datos, y una serie de *radiobuttons* para permitir elegir la función de ajuste. Poner el nombre del archivo es sencillo (líneas 78 a 81); ya usamos `QLabel` anteriormente, el único detalle es que le seteamos un flag para que el texto quede centrado. Poner los radiobuttons es apenas más complicado (líneas 92 a 97): agregamos los tres que necesitamos directamente en un layout vertical, sin necesitar hacer más nada porque como tenemos un sólo grupo de radiobuttons podemos usar el default (recordemos que los radiobuttons trabajan en grupo porque al habilitar uno se deshabilita el resto); notemos que por ahora no conectamos estos botones a ninguna acción.



Para la tabla con datos necesitamos un poco más de explicación. Vemos en el código que

hay un `QTableView` y un `TableModel`. Esta estructura con distintas partes existe porque las tablas ofrecen toda una funcionalidad que en realidad no estamos usando en nuestro ejemplo (piensen que “una planilla de cálculos” también termina siendo una tabla de datos compleja con una gran cantidad de detalles e interacciones).

Así y todo no es tan complicado, pero el concepto fundamental que tenemos que entender es que nos tenemos que encargar de esas dos partes: la “vista” de los datos (que se ocupa de cómo mostrar la información y cómo interactuar con la misma) y el “modelo” de los datos, que se encarga de recuperar el valor de cada celda (que puede ser de una tabla fija como tenemos nosotros o de una base de datos), guardar también cambios en esos valores (si la tabla fuese editable), y en general proveer toda la información con respecto a los datos (como cuantas filas o columnas hay).

En el panel principal entonces nos encargamos directamente de la vista de la tabla (líneas 83 a 90). Creamos una `QTableView`, le ponemos un modelo (una clase propia que veremos a continuación), y antes de agregarla al layout le configuramos su *look & feel*: indicamos que no se vea la cabecera vertical (lo que sería el número de cada fila), y le indicamos que a la cabecera horizontal se le puede cambiar los tamaños (permitiendo ajustar el ancho de cada columna), y que la última columna se “estire” hasta ocupar el ancho de toda la tabla. Noten como en ese código no hay nada con respecto a los datos en sí, sólo se los pasamos al modelo al instanciar `TableModel`.

```

25 class TableModel(QAbstractTableModel):
26     """Modelo para la tabla de datos fuente."""
27
28     def __init__(self, data):
29         super(TableModel, self).__init__()
30         self._src_data = data
31
32     def headerData(self, section_index, orientation, role):
33         """Devuelve información para armar los headers."""
34         if orientation == Qt.Orientation.Horizontal and role == Qt.ItemDataRole.DisplayRole:
35             return ["x", "y"][section_index]
36
37     def data(self, index, role):
38         """Devuelve información necesaria para mostrar los datos.
39
40         - DisplayRole: el dato a mostrar
41         - TextAlignmentRole: la alineación
42         """
43         if role == Qt.ItemDataRole.DisplayRole:
44             # devolvemos el dato fuente según fila y columna, formateado lindo
45             datum = self._src_data[index.row()][index.column()]
46             return format(datum, ".5f")
47
48         if role == Qt.ItemDataRole.TextAlignmentRole:
49             # al centro en vertical, a la derecha en horizontal
50             return Qt.AlignmentFlag.AlignVCenter + Qt.AlignmentFlag.AlignRight
51
52     def rowCount(self, index):
53         """Devuelve el total de filas."""
54         return len(self._src_data)
55
56     def columnCount(self, index):
57         """Devuelve el total de columnas."""

```

```
58     return len(self._src_data[0])
```

Nuestro modelo es una clase que hereda de `QAbstractTableModel` y su funcionamiento se basa en proveer distintos métodos estándares que la vista va a consultar para obtener distintas informaciones. En nuestra clase definimos algunos métodos, los necesarios para proveer información sobre los datos de nuestra tabla simple y estática. Y el `__init__` para inicializar la clase, donde guardamos la tabla en la instancia luego de inicializar la clase ancestral.

Cada uno de estos métodos están bien definidos en la documentación: para qué sirve cada uno, qué información reciben al ser ejecutados y qué tienen que devolver. No vamos a explicar todo lo que podría proveer `QAbstractTableModel` porque sería demasiado largo, pero entremos en detalle al menos con los que usamos en nuestro modelo para ayudarles a entender el código de nuestro ejemplo y facilitarles la entrada a la documentación más compleja cuando la tengan que usar.

- `headerData`: este método devuelve información relacionada con las cabeceras; en nuestro caso sólo nos interesa proveer los “títulos” de cada columna, por eso sólo contestamos información para el caso de la cabecera horizontal (`orientation == Qt.Orientation.Horizontal`) y con respecto a cómo se muestran (`role == Qt.ItemDataRole.DisplayRole`), devolviendo `X` para la primer columna e `Y` para la segunda.
- `data`: este método es el que usa la vista para obtener los datos de cada celda; implementamos dos roles, `DisplayRole` donde devolvemos cada dato (formateado) según fila y columna, y `TextAlignmentRole` para la alineación, mucho más sencillo porque todas las celdas están alineadas igual (al centro en lo vertical y a la derecha en lo horizontal).
- `rowCount` y `columnCount`: devuelven la cantidad de filas y de columnas, dato que se obtiene fácilmente de la tabla fuente más allá de por qué índice hayan sido consultados (ya que la tabla es rectangular).

El próximo paso es usar una función para ajustar los datos a una función polinómica, generar el gráfico y mostrarlo. Para ello, ya en el código `07.fitgraph.py`, tenemos una función `fit` que recibe los datos y el grado de la función a utilizar.

```
37 # funciones para ajustar los datos; lineal, cuadrática y cúbica
38 functions_to_fit = {
39     1: lambda x, a, b: a * x + b,
40     2: lambda x, a, b, c: a * x**2 + b * x + c,
41     3: lambda x, a, b, c, d: a * x**3 + b * x**2 + c * x + d,
42 }
43
44
45 def fit(data, degree):
46     """Ajusta los datos con una función según el grado especificado."""
47     try:
48         fit_function = functions_to_fit[degree]
49     except KeyError:
50         raise ValueError(f"No hay una función para el grado requerido: {degree!r}")
51
```

```

52     # los datos fuentes es una tabla de dos columnas, x e y; necesitamos esas dos columnas
53     # por separado como entrada a la función de ajuste
54     x_data, y_data = (np.array(x) for x in zip(*data))
55
56     params, pcovs = curve_fit(fit_function, x_data, y_data)
57
58     param_texts = []
59     for idx, (pvalue, pname) in enumerate(zip(params, string.ascii_lowercase)):
60         pcov = pcovs[idx, idx]
61         param_texts.append(f"{pname} = {pvalue:.3f} ± {pcov:.3f}")
62
63     fig = plt.figure()
64     ax = fig.subplots()
65     ax.plot(x_data, y_data, '.')
66     ax.plot(x_data, fit_function(x_data, *params))
67     ax.set_xlabel("x")
68     ax.set_ylabel("y")
69
70     return param_texts, fig

```

A las funciones polinómicas las tenemos definidas en el diccionario `functions_to_fit` de acuerdo a lo que vamos a necesitar para `curve_fit` de la biblioteca SciPy. La función `fit` toma entonces la función polinómica según el grado recibido, separa los datos en dos columnas y obtiene (usando SciPy) los parámetros de la función polinómica y las covarianzas, y con esta información genera dos productos: los textos indicando el valor y error de cada parámetro, y una figura de matplotlib con todo dibujado (los puntos originales y la función polinómica calculada).

Para utilizar esta función conectamos a los radiobuttons que creamos anteriormente a un método nuestro.

```

168     for degree, name in enumerate(["Lineal", "Cuadrática", "Cúbica"], 1):
169         radio = QRadioButton(name, self)
170         radio.clicked.connect(partial(self.calculate, degree))
171         options.addWidget(radio)

```

Vemos en la línea 170 que conectamos la señal `clicked` de cada radiobutton a una función creada ahí mismo con `partial`, que al ejecutarse llamará a `self.calculate` pasando automáticamente `degree` como parámetro

```

176     def calculate(self, function_degree):
177         """Pone a realizar los cálculos."""
178         self.main_window.set_status("Calculando")
179
180         try:
181             result = fit(self.data, function_degree)
182         except Exception as exc:
183             self.on_error(exc)
184         else:
185             self.show_result(result)
186
187     def on_error(self, exc):
188         """Muestra el error que sucedió al hacer el cálculo."""

```

```

189     # un resumen del problema en la interfaz
190     self.main_window.show_error(f"Error al calcular! {exc!r}")
191     # y el traceback completo por la terminal
192     print("ERROR al calcular:")
193     traceback.print_exception(exc)
194
195     def show_result(self, result):
196         """Arma el panel con el resultado."""
197         self.main_window.set_status("Mostrando el resultado")
198         fit_parameters, matplotlib_figure = result
199         result_panel = ResultPanel(self, fit_parameters, matplotlib_figure)
200
201         if self.result_panel is None:
202             # primera vez, no hay resultados de antes
203             self.right_layout.addWidget(result_panel)
204         else:
205             # reemplazamos los resultados previos
206             self.right_layout.replaceWidget(self.result_panel, result_panel)
207             self.result_panel.close()
208         self.result_panel = result_panel

```

Este método `calculate` es el que ejecutará la función `fit`. En caso de error llamará a otro método que mostrará el error utilizando nuestro método de la ventana principal, más todo el *traceback* por la terminal.

En caso de que haya salido todo bien pasará ese resultado a otro método, `show_result`, cuya responsabilidad es crear un panel de resultados (que veremos a continuación) y agregarlo al layout de la derecha, o en caso de tener otro panel de antes, cerrar el viejo y reemplazarlo.

Veamos entonces `ResultPanel` que es el que se encarga de mostrar la info resultado. Recibe la instancia del panel principal, los parámetros de la función, y la figura de `matplotlib`:

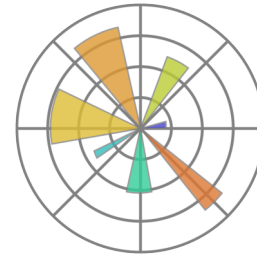
```

109 class ResultPanel(QWidget):
110     """Panel para mostrar el resultado: parámetros y el gráfico en sí."""
111     def __init__(self, parent, fit_parameters, matplotlib_figure):
112         super().__init__()
113         self.parent = parent
114         layout = QVBoxLayout()
115         self.setLayout(layout)
116
117         # los parámetros de la ecuación
118         params_textedit = QPlainTextEdit()
119         params_textedit.setPlainText("\n".join(fit_parameters))
120         layout.addWidget(params_textedit)
121
122         # el gráfico, y su toolbar relacionado que lo agregamos a la ventana principal
123         figure_canvas = FigureCanvasQTAgg(matplotlib_figure)
124         layout.addWidget(figure_canvas)
125         self.toolbar = NavigationToolbar2QT(figure_canvas, self)
126         self.parent.main_window.addToolBar(self.toolbar)
127
128     def close(self):
129         """Remueve la toolbar antes de cerrar el widget."""
130         self.parent.main_window.removeToolBar(self.toolbar)
131         super().close()

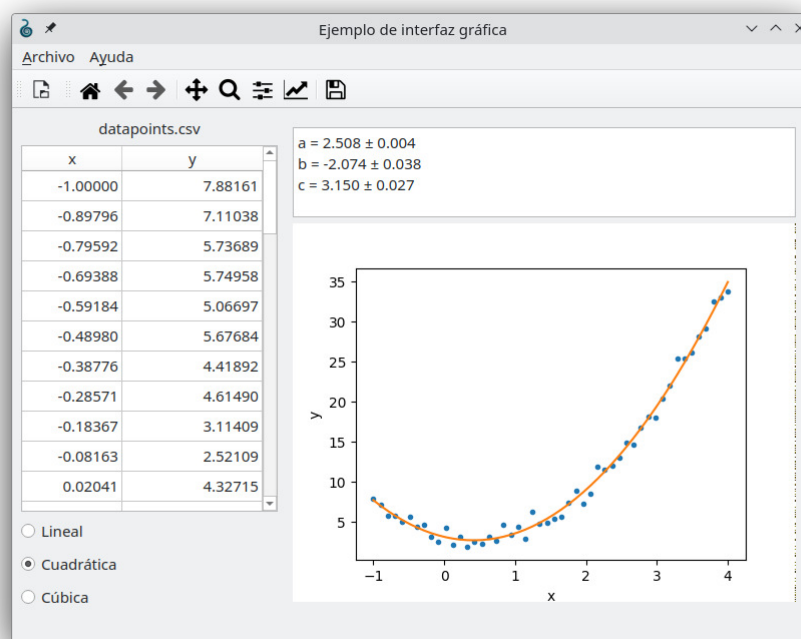
```

Como ya vimos en otros casos, la instancia primero inicializa el widget del que hereda, guarda algún parámetro recibido y crea un layout para agregar contenido. En este caso tenemos dos elementos: un *text edit* al que le ponemos varias líneas con los parámetros recibidos, y el gráfico.

Para el gráfico usaremos funcionalidad que *matplotlib* misma provee para integrarse con Qt: un *canvas* para agregar la figura y una barra de herramientas (conectada a esa *canvas*) para que se pueda interactuar con el gráfico. A la barra de tareas la podemos ubicar en cualquier lado, pero a nivel usabilidad lo mejor es que sea una barra más de la ventana principal, entonces en la línea 126 la agregamos allí. Pero al hacer esto tenemos que tener la precaución de que cuando se cambie el gráfico tenemos que sacar la barra del gráfico anterior, y eso lo hacemos en el método *close* de nuestra clase (línea 130).



Matplotlib - Biblioteca para la generación de gráficos estáticos, animados e interactivos en Python.



La próxima iteración de código, en `08.notblocking.py`, está pensada para ejemplificar cómo podemos soportar aquellas situaciones donde algo tarda demasiado. Como Qt es un sistema asíncrono en realidad no podemos bloquearlo con ninguna operación que tarde más que algunas decenas de milisegundos, porque mientras el sistema está bloqueado no va a poder procesar ningún evento, y esto a nivel de experiencia de usabilidad se traduce en que la ventana se “congela”, no responde al teclado o el mouse, e incluso puede aparecer a medio dibujar en la pantalla (recordemos que Qt no sólo procesa eventos nuestros sino también del sistema operativo, y si no puede procesar nada porque está bloqueado no se va a comportar correctamente).

En nuestro ejemplo el *cálculo científico* que hacemos es bastante rápido, ya que es un simple

ajuste de datos, pero simulemos un potencial cálculo más costoso agregando una demora artificial de cinco segundos en la función `fit`:

```
72 # simulamos que la tarea tarda mucho tiempo, y es bloqueante, así mostramos cómo
73 # lidiar con esta situación a nivel GUI
74 time.sleep(5)
```

Para soportar casos de funciones que tardan mucho en ejecutarse y no bloquear la interfaz gráfica debemos ejecutar esa función en otro hilo o *thread*, para lo cual creamos una clase auxiliar, `ThreadedTask`:

```
115 class TaskSignals(QObject):
116     """Señales para conectar de nuestra ThreadedTask."""
117     success = pyqtSignal(object)
118     error = pyqtSignal(Exception)
119
120
121 class ThreadedTask(threading.Thread):
122     """Ejecuta una función en un hilo.
123
124     Recibe la función y los argumentos. En 'signals' provee 'success' para cuando la función
125     se completa satisfactoriamente (envía el resultado) y 'error' para cuando no (envía
126     la excepción).
127     """
128     def __init__(self, func, *args, **kwargs):
129         self.func = func
130         self.args = args
131         self.kwargs = kwargs
132         self.signals = TaskSignals()
133         super().__init__()
134
135     def run(self):
136         try:
137             result = self.func(*self.args, **self.kwargs)
138         except Exception as exc:
139             self.signals.error.emit(exc)
140         else:
141             self.signals.success.emit(result)
```

Esta clase es bien simple, pero el truco es que hereda de `threading.Thread`, que es una de las maneras de armar hilos en Python (más sobre hilos en el capítulo de procesamiento paralelo 9.2). En el método de inicialización recibe la función a ejecutar y cualquier cantidad de argumentos posicionales y nombrados, y guarda todo en la instancia. Además inicializa un objeto con dos señales (definido en las líneas 115 a 118), para los casos de terminar satisfactoriamente o en error, y por supuesto llama al inicializador de la clase ancestral (requisito fundamental para que el hilo funcione).

Luego en el método `run` (que es el punto de entrada cuando se ejecuta el hilo) simplemente se ejecuta la función y guarda el resultado, con la precaución de soportar cualquier error. Si todo estuvo bien se emite la señal `success` pasando el resultado, y si algo salió mal se emite la señal `error` pasando la excepción ocurrida, y luego el hilo termina.

Volvamos al código de `MainPanel` para ver cómo usamos esa clase auxiliar. Aquí tenemos que reescribir el método `calculate` que es donde ejecutábamos la función:

```

211 def calculate(self, function_degree):
212     """Pone a realizar los cálculos en un hilo."""
213     self.main_window.set_status("Calculando")
214
215     # disparamos el nuevo cálculo y conectamos la función que muestra el resultado
216     tt = ThreadedTask(fit, self.data, function_degree)
217     tt.signals.success.connect(self.show_result)
218     tt.signals.error.connect(self.on_error)
219     tt.start()

```

Ahora, además de poner el estado en “Calculando”, inicializamos `ThreadedTask` pasando la función `fit`, los datos y el grado de la función de ajuste, y conectamos sus señales: la de éxito para mostrar el resultado, y la de error para mostrar la excepción ocurrida (¡notar que esas dos funciones no cambiaron! ya recibían el resultado o la excepción desde antes). Finalmente llamamos al método `start` de la clase auxiliar para que el hilo arranque.

Todo este trabajo que hicimos no tiene un impacto directo en la interfaz del usuario (no hay un botón nuevo, o se muestra distinta alguna información), pero es fundamental a la hora de tener una aplicación que no se congele y que no de la sensación al usuario de que “algo salió mal”.

Por otro lado, aunque ahora la aplicación se mantiene responsiva, en realidad no nos está dando ninguna indicación que nos ayude a darnos cuenta que está haciendo un cálculo que lleva tiempo y que tenemos que esperar. Implementamos esto en la próxima iteración del código, `09.complete.py`, que es la versión final de nuestra aplicación.

Históricamente la forma de las interfaces gráficas de mostrar que “se está realizando alguna tarea y hay que esperar” es mediante un *throbber* (un término sin traducción al castellano para esta acepción), que es una imagen animada, en general circular, que va “girando” mientras tenemos que esperar.

Agreguemos entonces un *throbber* a nuestro programa. El mismo tiene que verse cuando arranca el cálculo, y sacarse cuando tenemos el gráfico resultado. Si elegimos otro grado para la función de ajuste tenemos que sacar el gráfico y poner el *throbber* para esperar el nuevo cálculo. En la práctica, lo que sucede es que en el layout derecho, luego de iniciar el primer cálculo, vamos a ir alternando entre mostrar un *throbber* y el panel de resultados.

En el método `MainPanel.calculate` tenemos entonces que poner el *throbber* antes de crear e iniciar el hilo con el cálculo:

```

216 self.throbber = QLabel()
217 movie = QMovie("throbber.gif")
218 self.throbber.setMovie(movie)
219 movie.start()
220
221 if self.result_panel is None:
222     # primera vez, no hay resultados de antes, sólo agregamos el throbber
223     self.right_layout.addWidget(self.throbber, stretch=1)
224 else:
225     # reemplazamos los resultados previos con el throbber

```

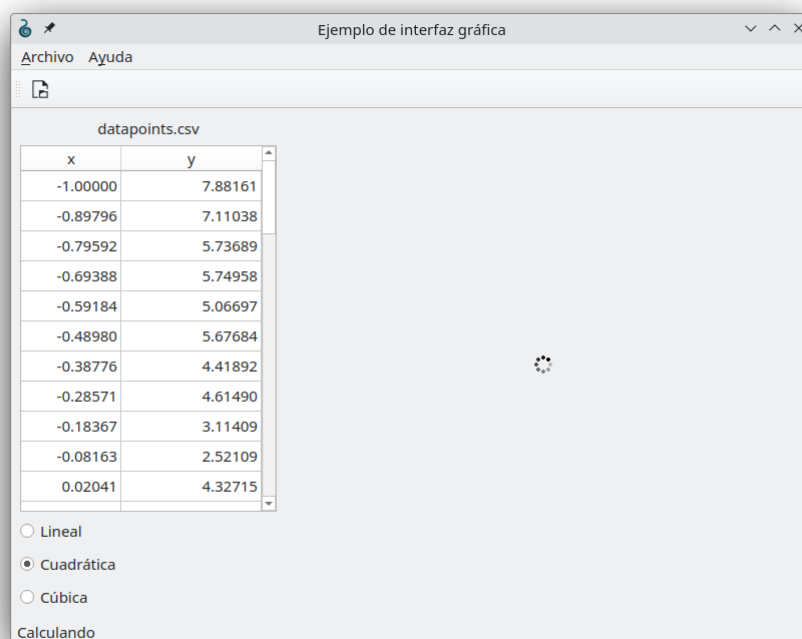
```

226         self.right_layout.replaceWidget(self.result_panel, self.throbber)
227         self.result_panel.close()

```

Crear el throbber es sencillo (líneas 216 a 219) porque nos basamos en tener un GIF animado en un archivo `throbber.gif` que cargamos con `QMovie` (un widget para mostrar animaciones simples sin sonido), que para insertarlo en la interfaz ponemos dentro de un `QLabel` (como haríamos con un texto o una imagen).

Inmediatamente después en las líneas 221 a 227 agregamos el throbber al layout derecho, directamente si no había un panel de antes, o reemplazando un panel anterior (y cerrándolo).



Luego de agregarlo, tenemos que considerar cuando sacarlo. Esto sucede en los métodos `on_error` y `show_result`, que se ejecutan cuando terminó el cálculo por error o satisfactoriamente, cerrando y eliminando el throbber en el primer caso...

```

243         self.right_layout.removeWidget(self.throbber)
244         self.throbber.close()

```

..., o cerrándolo y reemplazándolo por el panel de resultados en el segundo.

```

250         self.result_panel = ResultPanel(self, fit_parameters, matplotlib_figure)
251         self.right_layout.replaceWidget(self.throbber, self.result_panel)
252         self.throbber.close()

```

Llegamos entonces al final de nuestro recorrido, con una interfaz completa y funcional.

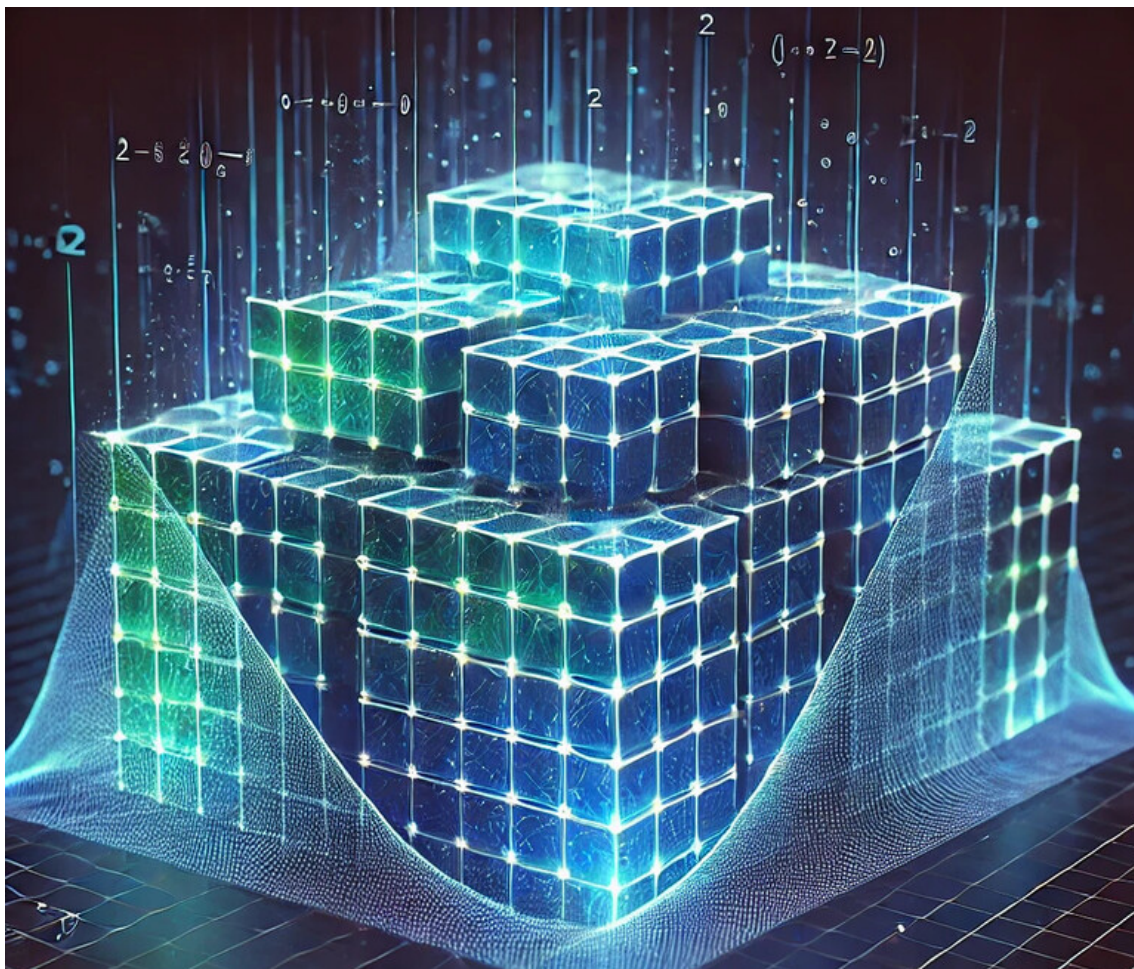
Vuestras necesidades de interfaz van a ser distintas, obviamente, pero siempre serán en mayor o menor medida alguna variación de lo que ya vimos en este capítulo, usando también otros widgets más o menos complejos.

Lo importante es que ya saben cómo arrancar.

Parte II

Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte [III](#) al abordar temas de aplicaciones específicas.



11 | Numpy

NumPy[2] es una biblioteca muy utilizada en Python para trabajar con vectores multidimensionales (arreglos, matrices, etc.) que ofrece también una gran colección de funciones matemáticas de alto nivel para operar con ellos.

En realidad es la base fundamental para el análisis numérico en Python y sustenta todo el ecosistema científico de este lenguaje, es un pilar esencial para otras bibliotecas científicas como SciPy[3] y Matplotlib[4], y habilita el desarrollo de proyectos en campos como la ciencia de datos, la inteligencia artificial y la simulación numérica.

El punto de entrada a NumPy es la estructura array, que nos permite manejar matrices de cualquier dimensión (incluso de una). Estos arreglos, a diferencia de las listas integradas en el lenguaje, son homogéneos: todos sus elementos deben ser del mismo tipo (lo cual es clave en la eficiencia y potencia de cálculo de esta biblioteca).

Podemos crear un arreglo utilizando un iterable como fuente:

Módulo	Versión
NumPy	1.26.4

[Código disponible](#)

CELDA 1 DE NUMPY.IPYNB

```
import numpy as np
```

CELDA 03

```
np.array([0, 1, 2, 3, 4, 5])
```

```
array([0, 1, 2, 3, 4, 5])
```



Es común importar el módulo principal de NumPy llamándolo `np`: es más corto y conciso, y no confunde porque todo el mundo está acostumbrado a eso.

NumPy nos ofrece formas rápidas de crear arreglos de distintos tamaños, por ejemplo con los números secuenciales como el conocido `range`, pero también inicializado con ceros y con unos (lo cual es muy normal, ya que son la identidad aditiva y multiplicativa respectivamente):



NumPy - Proporciona soporte para arreglos y matrices multidimensionales, junto con una amplia colección de funciones matemáticas para operar con ellos.

CELDA 04

```
np.arange(6)
```

```
array([0, 1, 2, 3, 4, 5])
```

CELDA 05

```
np.ones(4)
```

```
array([1., 1., 1., 1.])
```

CELDA 06

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

En este capítulo vamos a ver otras funciones que posee NumPy, como estas que acabamos de mostrar, pero siempre es interesante explorar y tener a mano la [Documentación de Referencia](#).

Vemos que por default NumPy crea todos los arreglos con `floats`. Como decíamos antes, todos los elementos tienen que ser del mismo tipo, pero tenemos control sobre cual tipo es ese; por ejemplo, si vamos a trabajar con enteros:

CELDA 07

```
np.ones(4, dtype=np.int32)
```

```
array([1, 1, 1, 1], dtype=int32)
```


11.1. Forma de trabajo

Hagamos una comparación entre las listas integradas y este nuevo tipo de dato, para empezar a notar las diferencias. Supongamos que queremos trabajar sobre un millón de números al mismo tiempo (pero por simplificación para el libro, usemos sólo 10... para el caso es lo mismo). Entonces tenemos nuestro millón de números:

CELDA 08
<pre>clasica = list(range(10)) clasica</pre>
<pre>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</pre>

Esto, en numpy sería:

CELDA 09
<pre>nueva = np.arange(10) nueva</pre>
<pre>array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])</pre>

Ahora, supongamos que queremos los cuadrados de esos números. Como vimos en la subsección 3.3, podemos hacer una list comprehension. Ingenuamente, intentamos lo mismo para ambas estructuras.

CELDA 10
<pre>[x ** 2 for x in clasica]</pre>
<pre>[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]</pre>

CELDA 11
<pre>np.array([x ** 2 for x in nueva])</pre>
<pre>array([0, 1, 4, 9, 16, 25, 36, 49, 64, 81])</pre>

Y acá es donde descubrimos el inmenso poder de NumPy: nos permite procesar los arreglos directamente, manejando simultaneamente todos sus elementos internos.

En el ejemplo que acabamos de mostrar, Python tiene que recorrer el iterable, ir recibiendo uno por uno los elementos, elevar ese elemento al cuadrado, e irlo agregando a una lista. Con diez elementos no pasa nada, pero recordemos, nuestro ejemplo imaginario tiene un millón de números.

Numpy, repetimos, nos permite procesar todos los elementos del arreglo al mismo tiempo:

CELDA 12
<pre>nueva</pre>
<pre>array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])</pre>

CELDA 13

```
nueva ** 2

array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Acá tenemos en Python solamente una llamada a una función que estará implementada en C, Fortran o algún lenguaje de bajo nivel, y fuertemente optimizada. Y se realiza todo el procesamiento necesario solamente con esa llamada a función, obteniendo efectivamente performances comparables con esos lenguajes de más bajo nivel. De cualquier manera, la idea de usar NumPy es justamente despreocuparnos sobre cómo está implementada y optimizada, y enfocarnos en usar la biblioteca correctamente, de la misma manera que al usar Python en sí por ejemplo nos despreocupamos sobre la administración dinámica que hace de la memoria.

Como ejercicio, comparemos las performances de los dos casos de nuestro ejemplo anterior.

CELDA 14

```
def f():
    clasica = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    cuads = [x ** 2 for x in clasica]

def g():
    nueva = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    cuads = nueva ** 2

print("Método clásico:")
%timeit f()
print("Usando numpy:")
%timeit g()
```

Método clásico:
1.73 µs ± 9.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
Usando numpy:
1.79 µs ± 17.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

¡Epa! Vemos que es más lento en NumPy, ¿cómo puede ser eso? Simple, porque no tiene sentido usar NumPy para conjuntos pequeños de datos, realmente estamos moviendo toda una infraestructura para procesar pocos casos. Tengamos en cuenta que `timeit.timeit` mide lo que pedimos un millón de veces por default, así que el número que estamos viendo ahí es en microsegundos.

Veamos un ejemplo más real, haciendo lo mismo pero con un millón de números:

CELDA 15

```
def f():
    clasica = range(1_000_000)
    cuads = [x ** 2 for x in clasica]

def g():
    nueva = np.arange(1_000_000)
    cuads = nueva ** 2

print("Método clásico:")
%timeit f()
```

```
print("Usando numpy:")
%timeit g()
```

Método clásico:

190 ms ± 2.34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Usando numpy:

1.97 ms ± 65.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Como trabajar sobre un millón de números obviamente tarda mucho más, le decimos a `timeit.timeit` que mida sólo una determinada cantidad de veces, y luego dividimos por esa cantidad, así que ahora estamos viendo segundos.

Y encontramos que para este caso más real, NumPy es más de cien veces más rápido.

Entonces, es responsabilidad nuestra saber cuando utilizar esta biblioteca. Por un lado, no vale la pena para algunos pocos datos, pero especialmente tenemos que tener el cuidado de NO caer en la tentación de procesar los arreglos “a mano en Python”, sino siempre utilizar las herramientas correctas de NumPy.

¿Tenemos que sumar los números usando el `sum` de Python? No. ¿Sacar el máximo usando `max`? Tampoco. ¿Y si queremos aplicar alguna función matemática? De nuevo, usar las herramientas de NumPy:

CELDA 16

```
arr = np.arange(10)
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

CELDA 17

```
arr.max()
```

```
9
```

CELDA 18

```
arr.sum()
```

```
45
```

CELDA 19

```
np.sin(arr)
```

```
array([ 0.          ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
        -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

11.2. Multidimensionalidad

Hasta ahora sólo armamos un arreglo de una sola dimensión, pero como decíamos arriba, NumPy permite que sean de varias dimensiones.

Incluso esas dimensiones pueden no ser iguales. Armemos por ejemplo una matriz de dos dimensiones que nos quede como una “tabla rectangular”:

	CELDA 20
<pre>np.ones((2, 5))</pre>	
<pre>array([[1., 1., 1., 1., 1.], [1., 1., 1., 1., 1.]])</pre>	

No estamos limitados en cantidad de dimensiones, armemos un arreglo de tres dimensiones: un paralelepípedo de 2x2x5:

	CELDA 21
<pre>np.ones((2, 2, 5))</pre>	
<pre>array([[[1., 1., 1., 1., 1.], [1., 1., 1., 1., 1.]], [[1., 1., 1., 1., 1.], [1., 1., 1., 1., 1.]])</pre>	

También, podemos cambiarle la forma. Armemos la misma matriz pero con números secuenciales:

	CELDA 22
<pre>lineal = np.arange(20) lineal</pre>	
<pre>array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])</pre>	

	CELDA 23
<pre>lineal.shape</pre>	
<pre>(20,)</pre>	

	CELDA 24
<pre>matriz = lineal.reshape((2, 2, 5)) matriz</pre>	
<pre>array([[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]], [[10, 11, 12, 13, 14], [15, 16, 17, 18, 19]])</pre>	

	CELDA 25
<pre>matriz.shape</pre>	
<pre>(2, 2, 5)</pre>	

En el ejemplo vemos como podemos preguntar la forma de un arreglo: en el primer caso es de largo 20 en la única dimensión que tiene, mientras que ya la matriz tiene largos 2, 2 y 5 en cada una de sus dimensiones.

También podemos preguntar la cantidad de dimensiones, y el tamaño del arreglo (que es sencillamente la cantidad de elementos que contiene, no confundir con los otros atributos recién mencionados):

	CELDA 26
<code>lineal.ndim</code>	
1	

	CELDA 27
<code>matriz.ndim</code>	
3	

	CELDA 28
<code>lineal.size</code>	
20	

	CELDA 29
<code>matriz.size</code>	
20	

Para acceder a los elementos adentro de los arreglos, NumPy nos permite explotar la natural multidimensionalidad de los mismos, y utilizaremos una sintaxis muy similar a la que estábamos acostumbrados con las secuencias de Python, pero con este gran detalle: podemos expresar al mismo tiempo el índice de cada dimensión, separándolos por coma.

En el caso de los arreglos de una sola dimensión nos queda exactamente como siempre:

	CELDA 30
<code>lineal[4]</code>	
4	

	CELDA 31
<code>lineal[:4]</code>	
<code>array([0, 1, 2, 3])</code>	

Es en el caso de los arreglos de más de una dimensión donde podemos explotar esto, si es necesario. En el siguiente ejemplo accedemos a la matriz con un sólo índice, que nos dará la “submatriz” correspondiente:

CELDA 32
<pre>matriz.shape</pre>
<pre>(2, 2, 5)</pre>

CELDA 33
<pre>subm = matriz[0] subm.shape</pre>
<pre>(2, 5)</pre>

CELDA 34
<pre>subm</pre>
<pre>array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])</pre>

Si queremos acceder a la primer submatriz, y luego a un subarreglo de la misma, y de ahí a un elemento, podríamos hacer lo que estamos acostumbrados en Python, o utilizar la forma de NumPy (obviamente, el segundo caso es más rápido, porque no vamos y venimos de NumPy, sino que se procesa todo en una misma llamada):

CELDA 35
<pre>matriz[1][0][3]</pre>
<pre>13</pre>

CELDA 36
<pre>matriz[1,0,3]</pre>
<pre>13</pre>

El verdadero potencial de esta forma se expresa cuando trabajamos con slices, ya que nos permite trabajar con varias dimensiones al mismo tiempo (como recién, separando cada dimensión con coma, ¡pero con slices!). En este caso lo complicado es ajustar el ojo para leer lo que está dentro de los corchetes, porque cuando en Python clásico tenemos algo como `[2:5]`, separamos el contenido con los dos puntos, leyendo el 2 primero, luego el 3, y entendiendo que es un "desde-hasta".

Pero con NumPy la separación realmente es "por dimensión". Entonces si vemos algo como `matriz[:,1:,2]` lo tenemos que separar por las comas, entender que para la primer dimensión tenemos `:` (todo el contenido en esa dimensión, que en el caso del ejemplo serían las dos matrices de 2 x 5 que se muestran), luego para la segunda dimensión tenemos `1:` (todo el contenido luego desde el primer item, que para el ejemplo sería exceptuar la primer línea de cada grupo), y finalmente para la tercer dimensión tenemos `2` (el elemento en la posición dos, en el caso del ejemplo nos quedamos con el 7 y con el 17).

Veamos progresivamente esto, y al final el detalle de cómo luego de recortar las distintas dimensiones de la matriz original, el resultado queda con su propia "forma":

CELDA 37
<pre>matriz array([[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]], [[10, 11, 12, 13, 14], [15, 16, 17, 18, 19]])</pre>
CELDA 38
<pre>matriz[:] array([[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]], [[10, 11, 12, 13, 14], [15, 16, 17, 18, 19]])</pre>
CELDA 39
<pre>matriz[:, 1:] array([[[5, 6, 7, 8, 9]], [[15, 16, 17, 18, 19]])</pre>
CELDA 40
<pre>matriz[:, 1:, 2] array([[7], [17]])</pre>
CELDA 41
<pre>matriz[:, 1:, 2].shape (2, 1)</pre>

Cuando hacemos *slicing* de arreglos de Numpy tenemos que tener un gran detalle en cuenta: a diferencia del mismo procedimiento en las listas integradas de Python, donde efectivamente tenemos una copia de la lista original, en el caso de NumPy lo que obtenemos es una “vista” del arreglo original.

Esta vista es afectada si modificamos el arreglo original, y viceversa. Esto es algo que hace NumPy en pos de la velocidad de procesamiento, ya que evita estar copiando objetos en memoria todo el tiempo.

Veamos este comportamiento. Primero un ejemplo usando las listas integradas en Python, para resaltar el efecto “copia” de las mismas:

CELDA 42
<pre>l1 = [1, 2, 3, 4] l2 = l1[:2] l2</pre>
<pre>[1, 2]</pre>

CELDA 43
<pre>l1[0] = 7 l2</pre>
<pre>[1, 2]</pre>

CELDA 44
<pre>l2[1] = 9 l1</pre>
<pre>[7, 2, 3, 4]</pre>

Y ahora el caso de los arreglos de NumPy, con el comportamiento de las vistas:

CELDA 45
<pre>a1 = np.array([1, 2, 3, 4]) a2 = a1[:2] a2</pre>
<pre>array([1, 2])</pre>

CELDA 46
<pre>a1[0] = 7 a2</pre>
<pre>array([7, 2])</pre>

CELDA 47
<pre>a2[1] = 9 a1</pre>
<pre>array([7, 9, 3, 4])</pre>

11.3. Indización avanzada

NumPy ofrece una funcionalidad aún más interesante que lo que recién vimos sobre indizar con varias dimensiones al mismo tiempo: ¡podemos también indizar usando otros arreglos! Esto explota en distintas funcionalidades que vemos en esta sección.

Si el arreglo-índice tiene números, esos números indicarán la posición de los elementos que queremos del arreglo que estamos indizando.

En el siguiente ejemplo usamos esta funcionalidad para sacar los cuadrados de las posiciones 2, 3, 15 y 7:

CELDA 48
<pre>cuadrados = np.arange(20) ** 2 cuadrados</pre>
<pre>array([0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361])</pre>

CELDA 49
<pre>índice = [2, 3, 15, 7] cuadrados[índice]</pre>
<pre>array([4, 9, 225, 49])</pre>

Vemos que no hace falta que el índice sea un arreglo de NumPy, pero obviamente eso también es soportado.

Tengamos en cuenta que los índices pueden ser repetidos (lo cual es útil en estadística para hacer muestreos eficientemente, por ejemplo):

CELDA 50
<pre>cuadrados[[0, 1, 2, 3, 2, 1, 0]]</pre>
<pre>array([0, 1, 4, 9, 4, 1, 0])</pre>

Por otro lado, si el arreglo-índice está compuesto por booleanos, los mismos indicarán qué elementos elegimos del arreglo que estamos indizando (en este caso el índice tiene que tener el mismo largo que el arreglo indizado).

CELDA 51
<pre>nros = np.array([0, 1, 2, 3, 4]) índice = [True, False, False, True, False] nros[índice]</pre>
<pre>array([0, 3])</pre>

Esto es muy útil para elegir elementos de un arreglo en función de si cumplen alguna condición (lo cual en muchos contextos se denomina *máscara*).

Veamos un ejemplo donde tenemos muchos números y queremos calcular el logaritmo sólo de los positivos (claro que lo podríamos hacer con una *list comprehension* eliminando a los negativos y luego calculando el logaritmo de cada uno, pero recordemos que la idea es siempre quedarnos adentro de NumPy, para aprovechar al máximo la potencia de esta biblioteca).

CELDA 52
<pre>nros = np.random.randint(-10, 10, 20) nros</pre>

```
array([ -2,   3,   9,  -4,  -1,  -1,   0,   3,  -6,   0,   6,   2,   2,
        3,   4,  -6,   2,   6,  -5, -10])
```

CELDA 53

```
cuales_positivos = nros > 0
cuales_positivos
```

```
array([False,  True,  True, False, False, False, False,  True, False,
        False,  True,  True,  True,  True,  True,  True, False,  True,  True,
        False, False])
```

CELDA 54

```
positivos = nros[cuales_positivos]
positivos
```

```
array([3, 9, 3, 6, 2, 2, 3, 4, 2, 6])
```

CELDA 55

```
np.log(positivos)
```

```
array([1.09861229, 2.19722458, 1.09861229, 1.79175947, 0.69314718,
        0.69314718, 1.09861229, 1.38629436, 0.69314718, 1.79175947])
```

NumPy también nos da mecanismos para trabajar con arreglos correlacionados (esto es, dos o más arreglos donde cada elemento de un arreglo tiene una relación con el elemento de la misma posición en los otros arreglos).

Por ejemplo, podemos tener unas mediciones para graficar, con los valores del eje x en un arreglo y los correspondientes del eje y en otro. Por algún motivo necesitamos ordenar los puntos del eje x, pero obviamente debemos mantener la relación con los valores correspondientes del eje y.

Para ello usaremos la función `np.argsort`, que nos devuelve un arreglo-índice que ordenaría el arreglo indicado, y luego usamos ese índice con ambos arreglos x e y:

CELDA 56

```
x = np.array([7, 3, 5, 9, 0, 6, 4, 1, 2, 8])
y = np.array([-8, 1, -3, -2, 3, 6, 5, -3, 2, 9])
índice = np.argsort(x)
índice # este nos dejaría a X ordenado

array([4, 7, 8, 1, 6, 2, 5, 0, 9, 3])
```

CELDA 57

```
x[índice]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

CELDA 58
<code>y[índice]</code>
<code>array([3, -3, 2, 1, 5, -3, 6, -8, 9, -2])</code>

Vemos que al indizar `x` obtenemos el arreglo ordenado, y al indizar `y` obtenemos los valores correspondientes según el nuevo orden de `x` (se sigue manteniendo la relación entre los elementos de `x` e `y` para cada posición).

En todos estos casos de indización avanzada tenemos que prestar atención al hecho de que aunque arrancamos con un arreglo y terminamos en un arreglo, no hay relación entre estos dos (a diferencia de cuando hacíamos slicing, que teníamos una vista). En definitiva no es más que una forma expresiva y eficiente de indizar el arreglo por posición varias veces. Es por esto que si modificamos el arreglo que tenemos como resultado no estaremos modificando el arreglo original.

11.4. Broadcasting

El término *broadcasting* (que podríamos traducir como “transmisión”) describe como NumPy trata durante las operaciones aritméticas a los arreglos que tienen distintas formas.

Porque cuando los arreglos tienen la misma forma es sencillo, los elementos mapean uno a uno y la operación se realiza entre los correspondientes:

CELDA 59
<pre>x = np.array([0, 1, 2, 3, 4]) y = np.array([7, 6, 5, 4, 3]) x + y</pre>
<code>array([7, 7, 7, 7, 7])</code>

CELDA 60
<code>x * y</code>
<code>array([0, 6, 10, 12, 12])</code>

Las reglas se complican (y el concepto de broadcasting aparece) cuando los arreglos **no** tienen el mismo tamaño.

Sin embargo, no cualquier combinación funciona. Cuando NumPy opera sobre dos arreglos compara la cantidad de elementos en cada dimensión (arrancando por las últimas), y en cada caso se tiene que cumplir una de dos condiciones:

- ambas dimensiones tienen la misma cantidad de elementos: en este caso se opera entre cada elemento de esa dimensión
- alguna de las dimensiones tiene sólo un elemento: en este caso se “estira” esa dimensión (se repite ese único elemento para igualar la otra dimensión)

El ejemplo más sencillo de esto es cuando operamos aritméticamente entre un arreglo y un escalar:

CELDA 61
<pre>x = np.array([0, 1, 2, 3, 4]) x * 2</pre>
<pre>array([0, 2, 4, 6, 8])</pre>

La operación que realizó allí es equivalente a haber repetido ese escalar la cantidad suficiente de veces, similar a lo que mostramos a continuación:

CELDA 62
<pre>x = np.array([0, 1, 2, 3, 4]) x * np.array([2, 2, 2, 2, 2])</pre>
<pre>array([0, 2, 4, 6, 8])</pre>

El *broadcasting*, entonces, provee una forma de vectorizar operaciones de arreglos, de manera que los loops ocurran en C en vez de Python. Y hace eso sin realizar copias innecesarias de los datos, lo que normalmente lleva a implementaciones muy eficientes de los algoritmos (hay casos, sin embargo, donde el broadcasting lleva a un uso ineficiente de la memoria y termina ralentizando el cálculo).

Veamos un ejemplo donde broadcasting funciona porque las dimensiones tienen la misma cantidad de elementos.

Supongamos un sensor óptico (pero simplificando, en vez de tener una matriz de 4096x4096, tenemos una matriz de 2x2, cada pixel con los tres valores correspondientes a RGB), Ahora, lo que queremos es aplicar un efecto donde reducimos el rojo muchísimo, dejando el verde y el azul en los valores originales (nuestro filtro tiene una sola dimensión y tres valores).

Prestemos atención al comentario luego de pedir cada forma, porque ahí estamos alineando las dimensiones arrancando por la última, que es como NumPy las compara. Entonces broadcasting funciona porque ambos arreglos tienen la dimensión con la misma cantidad de elementos.

CELDA 63
<pre>foto = np.arange(12).reshape((2, 2, 3)) # 2x2 pixeles, x 3 valores para RGB foto</pre>
<pre>array([[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]])</pre>

CELDA 64
<pre>filtro = np.array([.01, 1., 1.]) # solamente bajamos rojo muchísimo filtro</pre>
<pre>array([0.01, 1. , 1.])</pre>

CELDA 65

```
# mostramos las formas alineadas a la izquierda
print("foto : {:>6s}".format(' '.join(str(x) for x in foto.shape)))
print("filtro: {:>6s}".format(' '.join(str(x) for x in filtro.shape)))
```

```
foto : 2 2 3
filtro: 3
```

CELDA 66

```
foto * filtro

array([[[ 0. ,  1. ,  2. ],
        [ 0.03,  4. ,  5. ]],

       [[ 0.06,  7. ,  8. ],
        [ 0.09, 10. , 11. ]]])
```

Entonces, como la última dimensión tiene tres elementos en cada caso, multiplica los tres valores del filtro por los tres valores para cada uno de los cuatro píxeles de la foto (va haciendo lo mismo en las otras dimensiones, porque “filtro” sólo tenía una.

Como vimos al principio, cuando el escalar se repetía varias veces en un arreglo monodimensional para ser multiplicado por el otro arreglo con la misma forma, acá sucede lo mismo. No es tan fácil de ver, sin embargo, pero podemos pedirle a NumPy que nos muestre cómo quedaría filtro si lo broadcasteamos a la forma de la foto:

CELDA 67

```
np.broadcast_to(filtro, foto.shape)

array([[[0.01,  1. ,  1. ],
        [0.01,  1. ,  1. ]],

       [[0.01,  1. ,  1. ],
        [0.01,  1. ,  1. ]]])
```

Veamos ahora un caso donde ambos arreglos tienen más de una dimensión:

CELDA 68

```
nros = np.arange(8).reshape((2, 4))
nros

array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

CELDA 69

```
nros.shape

(2, 4)
```

CELDA 70
<pre>factor = np.array([[3], [5]]) factor</pre>
<pre>array([[3], [5]])</pre>

CELDA 71
<pre>factor.shape</pre>
<pre>(2, 1)</pre>

En este caso también aplica broadcasting, porque cada par de dimensiones compara bien: arrancando desde atrás (que es lo que hace NumPy) `nros` tiene 4 elementos y `factor` tiene 1 (válido, porque una de las dos tiene un elemento), y luego la anteúltima dimensión en cada caso tiene 2 elementos (válido, porque tienen la misma cantidad).

CELDA 72
<pre>nros * factor</pre>
<pre>array([[0, 3, 6, 9], [20, 25, 30, 35]])</pre>

Vemos que al operar, la dimensión de `factor` que tenía un elemento se “estira” hasta cubrir los cuatro elementos del `nros`, entonces nos queda que 0, 1, 2 y 3 multiplican cada uno por 3, mientras que 4, 5, 6 y 7 multiplican cada uno por 5.

Cerremos con un ejemplo más complicado:

CELDA 73
<pre>n1 = np.arange(12).reshape((2, 1, 2, 3)) n1</pre>
<pre>array([[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]])</pre>

CELDA 74
<pre>n2 = np.array([100, 200, 300, 400, 500, 600]).reshape((2, 1, 3)) n2</pre>
<pre>array([[[100, 200, 300]], [[400, 500, 600]])</pre>

En este caso también broadcasting va a funcionar: de atrás para adelante tenemos 3 y 3 (iguales), 2 y 1 (uno es 1), 1 y 2 (uno es 1), y dejamos de comparar porque para un arreglo se nos acabaron las dimensiones.

CELDA 75
<pre> n1 + n2 array([[[[100, 201, 302], [103, 204, 305]], [[400, 501, 602], [403, 504, 605]]], [[[106, 207, 308], [109, 210, 311]], [[406, 507, 608], [409, 510, 611]]]]) </pre>

11.5. Vectores y matrices

Venimos hablando de arreglos, tanto unidimensionales como multidimensionales, que son estructuras genéricas que NumPy nos ofrece para hacer un montón de cosas. Pero nosotros en álgebra lineal tenemos los conceptos de vectores y matrices, que sí, pueden ser considerados arreglos de determinadas dimensiones, pero tienen sus semánticas específicas.

Veamos entonces qué podemos hacer si miramos y usamos los arreglos desde este ángulo.

Arrancamos con la advertencia de que el operador `*` es para multiplicar los elementos en sí, cuyo uso ya vimos más arriba, y aplican las reglas de broadcasting que también comentamos. Algunos ejemplos:

CELDA 76
<pre> nros = np.array([0, 1, 2, 3, 4]) nros * 3 array([0, 3, 6, 9, 12]) </pre>

CELDA 77
<pre> nros1 = np.array([2, 3, 4]) nros2 = np.array([10, 0, 20]) nros1 * nros2 array([20, 0, 80]) </pre>

CELDA 78
<pre> mat = np.array([[1, 1, 1], [2, 2, 2], [3, 3, 3]]) nros = np.array([10, 20, 30]) mat * nros array([[10, 20, 30], [20, 40, 60], [30, 60, 90]]) </pre>

Por otro lado, si queremos "multiplicación de matrices", desde Python 3.5 tenemos el operador arroba (@) justamente pensado para esta operación (antes teníamos que usar la función `np.matmul`).

Veamos distintos ejemplos usando esta operación, escalando en complejidad. Si la operación es entre dos vectores, el resultado es la suma de la multiplicación entre sí de los escalares de cada posición (tengamos en cuenta que los dos vectores deben tener el mismo largo)

CELDA 79
<pre>nros1 = np.array([2, 3, 4]) nros2 = np.array([10, 0, 20]) nros1 @ nros2</pre>
100

Si multiplicamos una matriz por un vector, la regla es que el largo del vector tiene que coincidir con la cantidad de columnas de la matriz, y el resultado será un vector de largo igual a la cantidad de filas de la matriz, y en cada posición la suma del producto los escalares del vector y esa fila de la matriz:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + \cdots + a_{1n}b_n \\ a_{21}b_1 + a_{22}b_2 + \cdots + a_{2n}b_n \\ \vdots \\ a_{m1}b_1 + a_{m2}b_2 + \cdots + a_{mn}b_n \end{bmatrix}$$

Probemos eso en numpy:

CELDA 80
<pre>mat = np.array([[30, 10], [40, 40], [50, 50]]) # 2 columnas mat</pre>
<pre>array([[30, 10], [40, 40], [50, 50]])</pre>

CELDA 81
<pre>vec = np.array([1, 2]) # largo 2 vec</pre>
<pre>array([1, 2])</pre>

CELDA 82
<pre>mat @ vec # el resultado es largo 2 como el vector</pre>
<pre>array([50, 120, 150])</pre>

Para el caso bien genérico de multiplicar dos matrices, la regla a tener en cuenta es que el número de columnas de la primer matriz sea igual al número de filas de la segunda, y el resultado será otra matriz (con el número de filas de la primera y la cantidad de columnas de la segunda)

donde cada elemento será:

$$c_{ij} = \sum_{r=1}^n a_{ir} b_{rj}$$

Entonces en el caso genérico de $\mathbb{A}\mathbb{B} = \mathbb{C}$, si $\mathbb{A}$ tiene m filas y n columnas, $\mathbb{B}$ deberá tener n filas (y cualquier cantidad de columnas, digamos p); entonces el resultado $\mathbb{C}$ será una matriz de m filas y p columnas, donde:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{bmatrix}$$

En código:

CELDA 83
<pre>mat1 = np.array([[1, 1, 1], [2, 2, 2]]) mat1</pre>
<pre>array([[1, 1, 1], [2, 2, 2]])</pre>

CELDA 84
<pre>mat2 = np.array([[30, 10], [40, 40], [50, 50]]) mat2</pre>
<pre>array([[30, 10], [40, 40], [50, 50]])</pre>

CELDA 85
<pre>mat1 @ mat2</pre>
<pre>array([[120, 100], [240, 200]])</pre>

Más allá de todas estas operaciones entre matrices y/o vectores, también hay otros cálculos que generalmente se realizan en álgebra lineal para una matriz, y que por supuesto podemos hacer fácilmente en NumPy. Veamos algunos de ellos...

Podemos calcular la determinante:

CELDA 86
<pre>mat1 = np.array([[1, 2], [3, 4]]) np.linalg.det(mat1)</pre>
<pre>-2.0000000000000004</pre>

CELDA 87
<pre>mat2 = np.array([[6, 2, 3], [1, -2, 0], [12, 2, -1]]) np.linalg.det(mat2)</pre>
92.00000000000001

Las inversas de esas mismas matrices:

CELDA 88
<pre>np.linalg.inv(mat1)</pre>
<pre>array([[-2. , 1.], [1.5, -0.5]])</pre>

CELDA 89
<pre>np.linalg.inv(mat2)</pre>
<pre>array([[0.02173913, 0.08695652, 0.06521739], [0.01086957, -0.45652174, 0.0326087], [0.2826087 , 0.13043478, -0.15217391]])</pre>

O su normas, en toda su diversidad; no vamos a mostrar todas aquí, se puede explorar el item correspondiente en la [Referencia de NumPy](#), pero mostremos como ejemplo unas de las más usadas, la Frobenius 2:

CELDA 90
<pre>np.linalg.norm(mat1)</pre>
5.477225575051661

CELDA 91
<pre>np.linalg.norm(mat2)</pre>
14.247806848775006

E incluso podemos resolver un sistema de ecuaciones lineales con varias incógnitas.

Por ejemplo resolvamos el siguiente sistema con tres ecuaciones y tres incógnitas:

$$\begin{aligned} 3x + y &= -3 \\ 4y - z &= 5 \\ x + 3y - 2z &= -7 \end{aligned}$$

Lo primero es cargar una matriz con las ecuaciones y un vector con los resultados, de manera que toda la resolución sea:

$$\mathbb{A} \cdot \mathbf{x} = \mathbf{b}$$

En código:

CELDA 92

```
A = np.array([[3, 1, 0], [0, 4, -1], [1, 3, -2]])
A
```

```
array([[ 3,  1,  0],
       [ 0,  4, -1],
       [ 1,  3, -2]])
```

CELDA 93

```
b = np.array([-3, 5, -7])
b
```

```
array([-3,  5, -7])
```

Luego, NumPy nos calculará x :

CELDA 94

```
x = np.linalg.solve(A, b)
x
```

```
array([-2.,  3.,  7.])
```

Podemos validar fácilmente el resultado:

CELDA 95

```
A @ x
```

```
array([-3.,  5., -7.])
```

Tengamos en cuenta sin embargo que estamos lidiando con punto flotante, y como vimos en el capítulo correspondiente [12](#) podemos tener errores mínimos en las operaciones que hacen que “comparar por igual” sea engañoso:

CELDA 96

```
A @ x == b
```

```
array([False,  True, False])
```

CELDA 97

```
(A @ x)[0]
```

```
-3.0000000000000004
```

CELDA 98

```
(A @ x)[0] == .3
```

```
False
```

Para seguir con este tema y profundizarlo, les recomendamos el Capítulo de Ecuaciones Algebraicas [18](#).



Código disponible

Para nuestro ejemplo, supongamos que nuestra estructura de punto flotante tiene una precisión de 5 y la base es 10, por lo que los dos números indicados arriba se convierten en:

$$\begin{array}{l} 5.9720 \times 10^{24} \\ 9.1093 \times 10^{-28} \end{array}$$

Almacenar los números de esta manera nos da varias ventajas, no sólo la de ocupar poco espacio. Es evidente que con esta estructura podemos representar números de órdenes de magnitud muy diferentes (estamos sólo limitados por el exponente, como veremos abajo), y nos proporciona la misma precisión relativa para todos esos números (como mencionamos arriba, limitados por la longitud de la mantisa).

También nos permite algunos cálculos entre esos números muy grandes y muy pequeños (multiplicarlos, por ejemplo), pero tenemos que tener en cuenta que la suma y la resta posiblemente se vean afectadas por la precisión. Siguiendo el ejemplo de arriba, si sumamos la masa de la Tierra y la del electrón, realmente no veremos cambio en el resultado:

[illegible]

Esto es lo que se llama “error de representación” en los números de punto flotante, algo que no podemos evitar mientras manejemos una precisión finita. Tengamos en cuenta que este error no sólo se presenta en casos poco reales como el del ejemplo (nadie anda por la vida sumando la masa de la Tierra y la de un electrón), sino todo el tiempo, en operaciones mucho más triviales. Por ejemplo, en nuestra estructura ficticia “un tercio” sería 33333×10^{-5} , y claramente para que el número sea exacto necesitaríamos “infinitos tres” en esa mantisa. Hablaremos más sobre estos errores luego.

También tenemos errores de cálculo cuando llegamos al límite del exponente. Por ejemplo si en nuestra estructura ficticia determinamos que tenemos (además de los cinco para la mantisa) dos dígitos para el exponente, el número más grande que soportará la estructura es el 99999×10^{99} y el número más chico el 00001×10^{-99} . Si el número es mayor al máximo seguramente tengamos un error de *overflow*, y si el número es menor al mínimo lo más probable es que redondee a cero (aunque los sistemas de manejo de punto flotante permiten mostrar que sucedió un *underflow* e incluso se pueden configurar para que eso sea un error, no un redondeo).

Además, en punto flotante, tenemos los valores especiales “infinito” (positivo y negativo, resultado por ejemplo de dividir algún número por cero), y el valor especial NaN, que significa que no es un número (por el inglés *not a number*) sino el resultado de una operación indefinida

(como multiplicar cero por infinito). Tengamos en cuenta que como mencionaba arriba, muchos de estos casos por default generan un error, pero puede configurarse el sistema para manejarlos.

El tema da para mucho más, se nos escapa del alcance del libro. Para profundizar pueden explorar [la página de Wikipedia al respecto](#) o la [especificación de aritmética para punto flotante decimal](#), que detalla toda la implementación de un punto flotante de forma genérica sin entrar en la complicación de hacerlo súper optimizado en binario pensado para hardware, como lo hace [la norma IEEE 754](#), para punto flotante binario, que veremos a continuación.

12.1. Punto flotante binario

Hasta ahora hablamos genéricamente de estructuras de punto flotante, y en los ejemplos con los que jugamos usamos números decimales.

Aunque la base puede ser cualquiera, en la práctica se manejan dos grupos, los puntos flotantes decimales y los binarios. Obviamente los primeros son con dígitos en base 10 y los segundos son en base 2.

El punto flotante binario es el punto flotante nativo en todos los lenguajes modernos, debido a que su aritmética está incluida en los procesadores de las computadoras desde hace más de medio siglo, y desde la década de 1990 la mayoría de las implementaciones se basan en el estándar IEEE 754. El punto flotante decimal, aunque hubieron algunos procesadores que lo traían nativo, en la mayoría de los lenguajes está implementado en software, y normalmente siguiendo todos [el mismo estándar](#).

En Python al punto flotante binario lo tenemos como tipo de dato integrado en el lenguaje, es el `float`, mientras que al punto flotante decimal lo tenemos implementado en el módulo `decimal`, como ya mencionamos en la subsección de Números 2.1.

Ojo, que el punto flotante binario que usa Python, aunque se llame *float*, corresponde en precisión al que en otros lenguajes (como C) se denomina *double* (por “doble precisión”, que usa 8 bytes por número), mientras que allí el *float* usa precisión simple, ocupando 4 bytes por número. Mientras que el *float* integrado en Python está bien que sea de doble precisión (porque eso minimiza todo lo posible los errores), en NumPy podemos indicar explícitamente cual queremos utilizar:

CELDA 1 DE PUNTO_FLOTANTE.IPYNB
<pre>import numpy as np np.single</pre>
<pre>numpy.float32</pre>



[Leonardo Torres Quevedo](#) - Ingeniero de caminos, matemático e inventor español (1852–1936). Prolífico y versátil innovador, tuvo amplias contribuciones en el ámbito de la mecánica y la matemática aplicada así como en otras áreas de la ingeniería, incluidos los teleféricos, los dirigibles, o el radiocóntrol. Describió la idea de una máquina de funcionamiento secuencial para realizar cálculos y procesamiento y también la aritmética de punto flotante.

CELDA 02
<code>np.double</code>
<code>numpy.float64</code>

Más allá de esos detalles, la característica más importante que debemos mencionar aquí es que el punto flotante binario no puede representar fracciones decimales de forma exacta, por lo que si usamos punto flotante binario no podemos garantizar los mismos resultados que si usáramos aritmética decimal.

Muchas veces no vemos esto porque Python redondea el número usando los 17 dígitos más significativos, entonces si escribimos $1/10$ vemos 0.1 , aunque realmente ese número no puede representarse en binario, de la misma manera que no podemos representar $1/3$ en notación decimal.

CELDA 03
<code>1 / 10</code>
<code>0.1</code>

El tipo de datos Decimal, cuando lo inicializamos con un float, nos muestra el número exacto que ese float representa:

CELDA 04
<code>from decimal import Decimal</code>
<code>Decimal(0.1)</code>
<code>Decimal('0.1000000000000000055511151231257827021181583404541015625')</code>

Incluso nos puede sorprender que punto flotante binario represente de la misma manera dos números distintos en decimal:

CELDA 05
<code>Decimal(0.1)</code>
<code>Decimal('0.1000000000000000055511151231257827021181583404541015625')</code>

CELDA 06
<code>Decimal(0.100000000000000001)</code>
<code>Decimal('0.1000000000000000055511151231257827021181583404541015625')</code>

Veamos por ejemplo qué sucede al dividir varias veces un número por diez, primero directamente:

CELDA 07
<code>x = 9</code>
<code>for _ in range(10):</code>
<code> print(x)</code>


```

x /= 10
-----
9
0.9
0.09
0.009
0.0009
8.999999999999999e-05
8.999999999999999e-06
8.999999999999999e-07
8.999999999999999e-08
8.999999999999998e-09

```

Esto parece tener mucho sentido, pero si usamos Decimal para ver cual es el número exacto que está manejando el procesador, vemos que son sólo aproximaciones (a veces por arriba, a veces por abajo):

```
x = 9
for _ in range(10):
    print(Decimal(x))
    x /= 10
```

```
9
0.90000000000000000000002220446049250313080847263336181640625
0.089999999999999999999966693309261245303787291049957275390625
0.00899999999999999999931998839741709161899052560329437255859375
0.0008999999999999999997536692664112933925935067236423492431640625
0.000089999999999999999211568180168541175589780323207378387451171875
0.0000089999999999999999853394182236510090433512232266366481781005859375
8.99999999999999999853394182236510090433512232266366481781005859375E-7
8.99999999999999999853394182236510090433512232266366481781005859375E-8
8.999999999999999997872197332322678764437995369007694534957408905029296875E-9
```

Entonces, realizar más o menos operaciones vamos incorporando (de mayor o menor manera) un error en el resultado final. Podemos acotar este error, sin embargo, ya que según la implementación del punto flotante binario tendremos un límite superior del error relativo debido al redondeo; este valor se llama “épsilon” y en Python lo podemos encontrar (junto a otros valores que dependen de la implementación) en la estructura `sys.float_info`:

```
import sys
sys.float_info.epsilon
-----
2.220446049250313e-16
```

En definitiva esto hace que reglas que aprendimos en la escuela como “sumar un número N veces es lo mismo que multiplicar ese número por N” no se cumplen en punto flotante binario

CELDA 10
$0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3$
2.9999999999999996

CELDA 11
$0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 == 0.3 * 10$
False

Esto es aún más sorprendente en operaciones más cortas, donde a veces “parece” estar todo bien pero en otros casos, para la misma operación, encontramos que no:

CELDA 12
$5.4 + 0.3 == 5.7$
True

CELDA 13
$5.4 + 0.4 == 5.8$
False

Por esto es que siempre recomendamos no comparar por igualdad números de punto flotante, sino que lo ideal es evaluar si la diferencia entre los dos números que queremos comparar está acotada a un error máximo que definamos (técnica llamada “comparación con épsilon”):

CELDA 14
<pre> epsilon = 0.000000000001 a = 5.4 + 0.3 b = 5.7 abs(a - b) < epsilon </pre>
True

CELDA 15
<pre> a = 5.4 + 0.4 b = 5.8 abs(a - b) < epsilon </pre>
True

En estos ejemplos el épsilon fue elegido en función de los números con los que operábamos, pero en realidad no podemos usar siempre un valor absoluto, ya que si los números que estamos comparando son muy pequeños, ese épsilon nos quedó demasiado grande. Por eso siempre se usa un épsilon relativo, en función de los números a comparar. Ojo que este épsilon no es el mismo número que mencionábamos arriba, con el mismo nombre, pero que hacía referencia al máximo error relativo en función de la implementación.

Tenemos que tener en cuenta que todo esto está en la naturaleza básica del punto flotante binario: no es un bug en Python, ni en el procesador, ni en nuestro código, y por eso encontramos

el mismo comportamiento en todos los lenguajes que soportan la aritmética de punto flotante de los procesadores (aunque los distintos lenguajes pueden tener distintas reglas sobre cuando y cómo representar estas diferencias).

Estos detalles (y otros) hacen extremadamente difícil desarrollar y probar valores comerciales y financieros, y es por eso que se usa punto flotante decimal en esos casos.

¿Pero qué pasa con aplicaciones científicas?

Cuando tenemos que realizar miles o millones de operaciones, no tenemos alternativa a utilizar punto flotante binario, porque es el que está implementado en hardware: cualquier otra opción sería demasiado lenta.

Entonces, tenemos que lidiar con los errores inherentes a estos números. No los podemos evitar, pero sí podemos controlar su magnitud, no sólo para obtener un resultado final lo más preciso posible, sino también para poder conocer y acotar el error que tenemos al final de una serie de cálculos.

El análisis que debemos hacer se suma, obviamente, al que ya realizábamos sobre los errores de los números originales de esos cálculos (que si son valores teóricos son exactos, pero cuando manejamos mediciones reales, esos mismos números ya vienen con un determinado error que necesitamos conocer).

Por lo general los errores de redondeo son muy pequeños cuando se trabaja con doble precisión y operaciones aisladas. Sin embargo hay situaciones donde pueden aparecer problemas con operaciones simples o al acumularse en cálculos repetidos. La multiplicación y la división son operaciones seguras, pero la suma y la resta son peligrosas (porque para realizarlas hay que operar sobre la mantisa luego de igualar los exponentes, entonces volvemos a tener el problema mostrado al inicio del capítulo cuando mezclamos números de magnitudes muy diferentes).

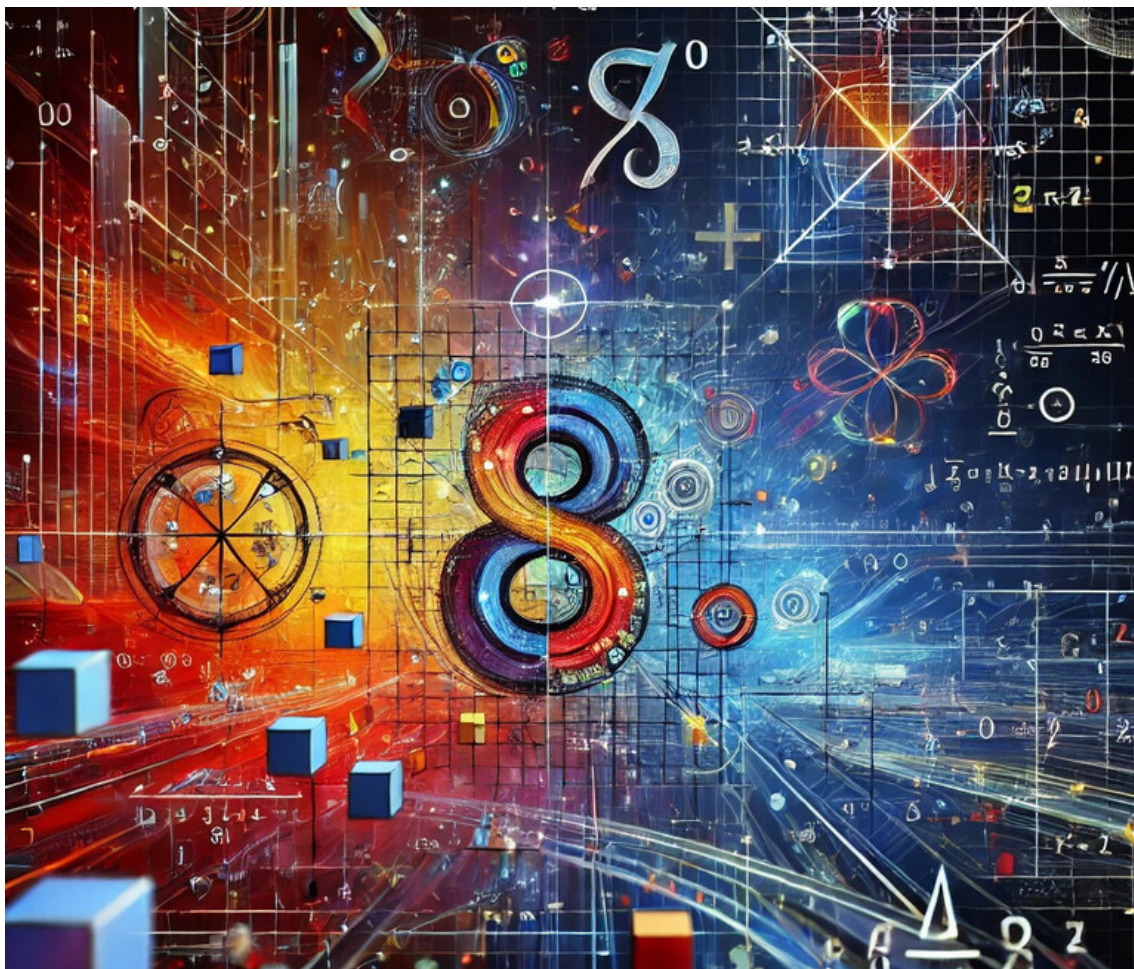
Veamos un ejemplo de esto, mostrando como no es lo mismo el orden cuando sumamos o restamos:

CELDA 16	
<pre>a = 168163.005 b = 168163.004 c = .0000000000123 a - b</pre>	
0.0010000000183936208	
CELDA 17	
<pre>a - c - b # restar "c" se perdió totalmente!!</pre>	
0.0010000000183936208	
CELDA 18	
<pre>a - b - c # acá sí influyó "c"</pre>	
0.0010000000060936209	

Cuantas más operaciones se realice en un cálculo más atención hay que darle a este problema (especialmente con los métodos iterativos de resolución de algoritmos), por eso es que siempre es

recomendable utilizar algoritmos diseñados específicamente para minimizar estos errores (particularmente, en lo posible, siempre que se puede utilicemos las bibliotecas de NumPy y SciPy en vez de escribir los algoritmos nosotros).

Para profundizar en el control de error en operaciones aritméticas usando punto flotante les recomendamos el documento [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#), de David Goldberg.



13 | Matemática simbólica

Es común utilizar las computadoras para realizar cálculo numérico, que implica realizar veloces operaciones aritméticas utilizando muchas veces aproximaciones numéricas y siempre números con precisión determinada. Sin embargo, es posible también en muchos casos realizar operaciones exactas en forma simbólica (como resolver ecuaciones, derivar, integrar, simplificar expresiones) utilizando para ello símbolos y variables. En Python, esto es posible gracias al módulo (o biblioteca?) SymPy [5], que provee lo que se denomina un «sistema de álgebra computacional» (o CAS, del inglés *computer algebra system*).

Una característica de SymPy que lo diferencia de otros CAS es que está escrito en Python y puede ejecutarse completamente en Python como cualquier otro módulo (NumPy, SciPy, etc.), tanto en forma interactiva como dentro de un programa. En particular, al usarlo en un *notebook* de Jupyter, podemos mostrar las salidas interpretando $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ lo cual facilita enormemente la lectura de expresiones complejas.

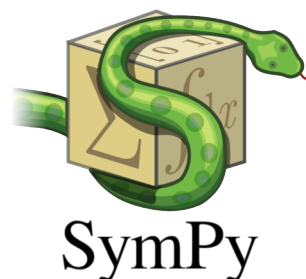
Módulo	Versión
SymPy	1.13.1
Código disponible	

13.1. Conceptos y operaciones básicas

Un concepto central en SymPy es el de símbolo. Éste representa, por medio de una variable simbólica, un valor abstracto o desconocido que puede ser utilizado en expresiones matemáticas. De este modo, es posible realizar cálculos simbólicos con dichas variables que son tratadas como entidades abstractas en vez de números específicos.

Los objetos de tipo símbolo se pueden instanciar a partir de `sympy.Symbol()` o `sympy.symbols()`. En el primer caso solo se instancia un único símbolo, mientras que el segundo caso permite instanciar varios símbolos en una sola instrucción. En ambos casos, además del nombre del símbolo creado, dichos objetos contienen atributos y métodos que describen sus propiedades y permiten operar sobre ellos. Del mismo modo que los objetos en Python, es posible asignarles nombres a los símbolos de modo de poder usarlos posteriormente.

Veamos cómo instanciar un símbolo x al que le asignaremos el nombre x , luego de importar el módulo `sympy` en la primera línea:



SymPy - Biblioteca para realizar cálculos matemáticos simbólicos, como álgebra, cálculo y teoría de números, directamente en Python.

CELDA 1 DE CONCEPTOS_BASICOS.IPYNB
<pre>import sympy as sym x = sym.Symbol('x') (x + 1)**2</pre>
$(x + 1)^2$

El método `Symbol()` de SymPy recibe como argumento una cadena e instancia un objeto `Symbol` al que le asignamos el nombre x . Este nombre puede ser utilizado en expresiones, y al no tener un valor numérico asignado estas expresiones quedan en forma simbólica (y en el *notebook* se muestra «renderizado» a partir de la salida en $\text{\LaTeX}$). En la celda siguiente instanciamos varios objetos utilizando `sympy.symbols()` mediante una cadena que tiene palabras (separadas por espacios, pero también pueden separarse mediante comas) compuestas por una o varias letras, incluyendo algunas que denotan letras del alfabeto griego. A dichos objetos les asignamos nombres que son utilizados para construir una expresión:

CELDA 02
<pre>y, z, a, b = sym.symbols('y z alpha beta') z + a + a + b * b</pre>
$2\alpha + \beta^2 + z$

Es posible, naturalmente, asignar un símbolo a un nombre arbitrario (por ejemplo el símbolo z a la variable h), sin embargo esto no constituye una buena práctica. Es importante señalar la diferencia entre un símbolo y el nombre que le asignamos. Veamos el siguiente ejemplo:

CELDA 03
<pre>expresión = x + 2 x = 2 expresión</pre>
$x + 2$

Este ejemplo muestra el procedimiento habitual de Python para asociar nombres a objetos tal como describimos en la Sección 2.5. Primero asignamos al nombre *expresión* una operación suma en la que participa la variable *x*, que a su vez «apunta» hacia el símbolo *x*. En la línea siguiente «reorientamos» el nombre *x* hacia el objeto de tipo entero 2. Cuando mostramos *expresión*, tal vez podríamos esperar obtener el valor 4, pero no es lo que ocurre, sino que recuperamos la definición inicial de *expresión* con el símbolo *x*, sin que el nuevo valor asignado al nombre *x* sea tenido en cuenta en la evaluación de *expresión*. Si esto último es lo que estábamos buscando, lo que tenemos que hacer es sustituir el símbolo *x* por el valor 2 en *expresión*:

CELDA 04
<pre>x = sym.symbols('x') expresión = x + 2 expresión.subs(x, 2)</pre>
4

En esta última celda, el nombre *x* representa un símbolo abstracto *x*, que «podría» ser un número entero, real, complejo, una función o cualquier otro ente matemático que pueda ser usado en expresiones y que permita su tratamiento simbólico. Es posible suministrar más información sobre el objeto de modo que SymPy pueda manipularlo apropiadamente. Podemos ver un ejemplo de esto en las celdas siguientes:

CELDA 05
<pre>x = sym.Symbol('x') y = sym.Symbol('y', positive=True) sym.sqrt(x ** 2)</pre>
$\sqrt{x^2}$

CELDA 06
<pre>sym.sqrt(y ** 2)</pre>
y

Aquí hemos definido dos variables simbólicas, estableciendo además que *y* es positiva. De este modo, al aplicar la función `sqrt()` sobre ambas, para el caso de *x* no se produce la simplificación automática de la expresión como si sucede con *y*. Del mismo modo, podemos definir variables que representen números enteros, pares e impares. Con esta información, SymPy puede simplificar o evaluar analíticamente las expresiones:

CELDA 07

```
m = sym.Symbol('m')
n = sym.Symbol('n', integer=True)
l = sym.Symbol('l', odd=True)
sym.cos(m * sym.pi)
```

$$\cos(\pi m)$$

CELDA 08

```
sym.cos(n * sym.pi)
```

$$(-1)^n$$

CELDA 09

```
sym.cos(l * sym.pi)
```

$$-1$$

Es posible preguntar si un símbolo posee determinados atributos. Por ejemplo, en el último caso, podemos interrogar al símbolo l si es un número par o impar:

CELDA 10

```
print(f' l es impar: {l.is_odd}, es par: {l.is_even}')
```

```
l es impar: True, es par: False
```

Como hemos visto en la celda 2, es posible definir múltiples símbolos en una sola instrucción utilizando `symbols()`. En este caso, al establecer condiciones como las de las celdas precedentes, éstas aplican a todo el conjunto de símbolos creados. Es posible obtener el conjunto de atributos asignados a un determinado símbolo mediante `assumptions0`, que contiene un diccionario cuyas claves son las propiedades y los valores son **True**, **False** o **None**, dependiendo si esta propiedad ha sido definida o no:

CELDA 11

```
import pprint

pprint.pprint(l.assumptions0)
```

```
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'even': False,
 'extended_nonzero': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'integer': True,
```



```
'irrational': False,
'noninteger': False,
'nonzero': True,
'odd': True,
'rational': True,
'real': True,
'transcendental': False,
'zero': False}
```

Una característica que puede originar cierta confusión es que, dado que SymPy no extiende la sintaxis de Python, el símbolo `=` no representa una igualdad en SymPy, sino que sigue siendo el operador de asignación usual de Python. Tampoco `==` sirve para evaluar la igualdad matemática en SymPy, tal como se ve a continuación:

CELDA 12
<code>(x + 2) ** 2 == x ** 2 + 4 * x + 4</code>
False

En este caso, podemos identificar claramente que estamos desarrollando el cuadrado de un binomio, por lo que matemáticamente las expresiones a izquierda y derecha del operador de comparación son iguales. Sin embargo, el operador `==` compara estrictamente expresiones, que en el ejemplo de la celda 12 son estructuralmente diferentes en ambos lados del operador: a la izquierda tenemos un binomio elevado al cuadrado, mientras que a la derecha vemos la suma de tres términos. Para realizar la comparación en sentido matemático, podemos ver que $(x + 2)^2 - x^2 - 4x - 4 = 0$, entonces podemos intentar simplificar la diferencia entre ambos miembros y verificar la igualdad cuando esa simplificación dé como resultado un cero. Para realizar la simplificación algebraica de expresiones utilizamos el método `simplify()`:

CELDA 13
<pre>izquierda = (x + 2) ** 2 derecha = x ** 2 + 4 * x + 4 sym.simplify(izquierda - derecha)</pre>
0

CELDA 14
<pre>derecha_2 = x ** 2 + 2 ** 2 sym.simplify(izquierda - derecha_2)</pre>
4x

En la celda 14 podemos comprobar que las expresiones no son matemáticamente iguales excepto para $x = 0$. Este método para verificar igualdades de expresiones no es infalible, dado que se puede probar teóricamente que es imposible determinar si dos expresiones simbólicas son idénticamente iguales en general¹. No obstante funciona aceptablemente para expresiones habituales no muy complejas. SymPy también dispone del método `equals()`, que comprueba la igualdad de expresiones por medio de evaluaciones numéricas en puntos aleatorios:

¹ Ver la [entrada](#) del teorema de Richardson en Wikipedia.

CELDA 15
<pre>print(izquierda.equals(derecha)) print(izquierda.equals(derecha_2))</pre>
<pre>True False</pre>

En la celda 4 vimos que podemos reemplazar un símbolo por un valor numérico utilizando el método `subs()`. En general, podemos usar este método para sustituir todas las instancias de algo en una expresión con alguna otra cosa. Por ejemplo:

CELDA 16
<pre>expresión = 1 + sym.cos(x) expresión.subs(x, 0)</pre>
2

CELDA 17
<pre>expresión.subs(sym.cos(x), sym.sqrt(1 - sym.sin(x) ** 2))</pre>
$\sqrt{1 - \sin^2(x)} + 1$

CELDA 18
<pre>expresión</pre>
$\cos(x) + 1$

En el caso de la celda 16, simplemente reemplazamos el símbolo x por el valor numérico 0, y como resultado se muestra la evaluación de la expresión. En la celda 17, la sustitución consiste en reemplazar $\cos(x)$ por $\sqrt{1 - \sin^2(x)}$, sin realizar evaluaciones numéricas. Finalmente, vemos que `subs()` devuelve una nueva expresión, dado que los objetos de SymPy son inmutables, tal como se ve al mostrar el valor que contiene la variable `expresión`.

Es posible también construir expresiones simbólicas de SymPy a partir de cadenas, utilizando para ello el método `sympify()` (no confundir con `simplify()`):

CELDA 19
<pre>cadena = 'x ** 2 - 5 * x + 1' expr = sym.sympify(cadena) expr</pre>
$x^2 - 5x + 1$

Como dijimos al inicio de esta sección, los símbolos en SymPy permiten representar diversos objetos matemáticos, entre los que se encuentran particularmente los números. Dado que éstos comparten atributos y métodos de `Symbol`, deben ser objetos distintos a los números propios de Python (como `int` y `float`). La conversión de números de Python a objetos de SymPy se realiza automáticamente por SymPy, o podemos hacerlo en forma explícita, y luego podemos acceder a

diversos atributos, por ejemplo:

CELDA 20
<pre>i = sym.Integer(13) type(i)</pre>
<pre>sympy.core.numbers.Integer</pre>

CELDA 21
<pre>i.is_Integer, i.is_real, i.is_odd</pre>
<pre>(True, True, True)</pre>

CELDA 22
<pre>f = sym.Float(3.14) type(f)</pre>
<pre>sympy.core.numbers.Float</pre>

CELDA 23
<pre>f.is_Integer, f.is_real, f.is_odd</pre>
<pre>(False, True, False)</pre>

Podemos volver a convertir instancias de números de SymPy en los correspondientes tipos intrínsecos de Python con las funciones usuales `int()` y `float()`:

CELDA 24
<pre>i_py = int(i) f_py = float(f) type(i_py), type(f_py)</pre>
<pre>(int, float)</pre>

En la celda 7 vimos que podemos instanciar un objeto de `Symbol` pasando el parámetro `integer=True`. De esta forma, este objeto representa «algún» entero, mientras que una instancia de `Integer` representa un entero específico. En ambos casos, el atributo `is_integer` es `True`, pero solo en el segundo es `True` el atributo `is_Integer` (notar la «I» mayúscula). Por ejemplo:

CELDA 25
<pre>i = sym.Symbol('i', integer=True) j = sym.Integer(4) print(f"i: {i.is_integer}, {i.is_Integer}, {i.is_positive}, {i.is_Symbol}") print(f"j: {j.is_integer}, {j.is_Integer}, {j.is_positive}, {j.is_Symbol}")</pre>
<pre>i: True, False, None, True j: True, True, True, False</pre>

Los números enteros en SymPy son de precisión arbitraria, de modo que no tienen límites inferior y superior fijos como ocurre cuando se representan enteros con un número fijo de bits

CELDA 30
$(a * b).p, (a * b).q$
$(2, 5)$

Finalmente, SymPy contiene también constantes numéricas que se pueden utilizar en expresiones, como π , e , ∞ (que se representa con dos letras «o» minúsculas seguidas), entre otras, así como constantes físicas y unidades:

CELDA 31
<code>print(sym.pi.evalf(), sym.E.evalf(), sym.oo.evalf())</code>
3.14159265358979 2.71828182845905 oo

13.2. Simplificación de expresiones

Una de las características más utilizadas en un CAS es la manipulación algebraica de expresiones complejas para obtener expresiones más simples. SymPy contiene muchas funciones que realizan diferentes tareas de simplificación. Una de ellas, que mostramos en un ejemplo anterior, es `simplify()`. Esta función intenta aplicar, en forma heurística, todas las funciones específicas que contiene SymPy con el propósito de obtener la forma más simple de una expresión. Veamos algunos casos:

CELDA 1 DE ALGEBRA.IPYNB
<code>import sympy as sym</code> <code>x, y, z = sym.symbols('x y z')</code>

CELDA 02
<code>sym.simplify((x ** 3 - 2 * x ** 2 - 13 * x - 10) / (x ** 2 - 4 * x - 5))</code>
$x + 2$

CELDA 03
<code>sym.simplify(2 * sym.cos(x)**2 + 2 * sym.sin(x)**2)</code>
2

CELDA 04
<code>sym.simplify(sym.gamma(x)/sym.gamma(x - 1))</code>
$x - 1$

En la salida de la celda 2 obtenemos una expresión simple para un cociente de polinomios. En la celda siguiente el resultado se obtiene a partir de la simplificación de una identidad trigonométrica. En el último ejemplo, `sp.gamma(x)` representa la función gamma $\Gamma(x)$. Si bien `simplify`

intenta encontrar la expresión más simple, a veces no estamos de acuerdo con el resultado que nos ofrece. Por ejemplo:

CELDA 05
<code>sym.simplify(x ** 2 + 4 * x + 4)</code>
$x^2 + 4x + 4$

Como vemos, la salida de la función es simplemente su argumento, cuando tal vez esperábamos obtener $(x + 2)^2$. Veremos a continuación que podemos intentar realizar la simplificación utilizando un método más específico.

13.2.1. Simplificación de funciones polinómicas y racionales

Una de las funciones de simplificación más utilizadas en SymPy es `expand()`, que dado un polinomio, lo expresa en la forma canónica como suma de monomios:

CELDA 06
<code>sym.expand((x - 2) * (x + 3) * (x - 1))</code>
$x^3 - 7x + 6$

CELDA 07
<code>sym.expand((x + 1) ** 4)</code>
$x^4 + 4x^3 + 6x^2 + 4x + 1$

Tal como su nombre lo indica, `expand()` no parece realizar tareas de simplificación dado que devuelve expresiones más grandes. Generalmente este es el caso, pero en ocasiones se obtienen expresiones más simples debido a cancelaciones de términos:

CELDA 08
<code>sym.expand((x + 1)*(x - 2) - (x - 1)*x)</code>
-2

Otra función de simplificación muy útil es `factor()`, que recibe polinomios como argumentos y los expresa como productos de factores irreducibles sobre los números racionales. Por ejemplo:

CELDA 09
<code>sym.factor(x ** 3 + x ** 2 - x - 1)</code>
$(x - 1)(x + 1)^2$

CELDA 10
<code>sym.factor(x ** 2 * z + 4 * x * y * z + 4 * y ** 2 * z)</code>
$z(x + 2y)^2$

Es posible también acceder a la lista de los factores de una expresión, junto con sus correspondientes exponentes, por medio de `factor_list()`:

CELDA 11
<code>sym.factor_list(x ** 2 * z + 4 * x * y * z + 4 * y ** 2 * z)</code>
<code>(1, [(z, 1), (x + 2*y, 2)])</code>

Para polinomios, `factor()` y `expand()` son funciones opuestas. Sin embargo, los argumentos de estas funciones no necesariamente deben ser polinomios, ya que ambas intentarán factorizar o expandir cualquier tipo de expresiones (aunque los factores pueden no ser irreducibles si los argumentos no son polinomios sobre los racionales):

CELDA 12
<code>sym.expand((sym.cos(x) + sym.sin(x)) ** 2)</code>
$\sin^2(x) + 2 \sin(x) \cos(x) + \cos^2(x)$

CELDA 13
<code>sym.factor(sym.cos(x) ** 2 + sym.sin(x) ** 2 - 2 * sym.sin(x) * sym.cos(x))</code>
$(-\sin(x) + \cos(x))^2$

La función `collect()` permite agrupar las potencias de una variable determinada en una expresión. Por ejemplo:

CELDA 14
<code>expresión = x ** 4 + 3 * (x * z) ** 3 + x ** 2 + x * y - z * x</code> <code>sym.collect(expresión, x)</code>
$x^4 + 3x^3z^3 + x^2 + x(y - z)$

Si queremos aislar el coeficiente de una potencia específica de una expresión, podemos usar el método `coeff()` tal como hacemos en la celda siguiente, para obtener el coeficiente de x^3 de expresión:

CELDA 15
<code>expresión.coeff(x, 3)</code>
$3z^3$

La función `cancel()` expresa una función racional en su argumento en la forma canónica p/q ,

donde numerador y denominador son polinomios sin factores comunes y los coeficientes de las máximas potencias de p y q son enteros (no fracciones). Por ejemplo:

CELDA 16
<code>sym.cancel((x ** 2 + 4 * x + 4) / (x ** 2 + 2 * x))</code>
$\frac{x+2}{x}$

CELDA 17
<code>expresión = 1 / x + (5 * x / 4 - 4) / (x - 2)</code> <code>expresión</code>
$\frac{\frac{5x}{4} - 4}{x - 2} + \frac{1}{x}$

CELDA 18
<code>sym.cancel(expresión)</code>
$\frac{5x^2 - 12x - 8}{4x^2 - 8x}$

Por último, mencionamos la función `apart()`, que realiza una descomposición en fracciones simples de una expresión racional:

CELDA 19
<code>e_racional = (x ** 2 + 3 * x + 1) / (x + 1) ** 3</code> <code>e_racional</code>
$\frac{x^2 + 3x + 1}{(x + 1)^3}$

CELDA 20
<code>sym.apart(e_racional)</code>
$\frac{1}{x+1} + \frac{1}{(x+1)^2} - \frac{1}{(x+1)^3}$

13.2.2. Simplificaciones trigonométricas

Para el caso de las funciones trigonométricas, podemos utilizar `trigsimp()`, que intenta simplificar expresiones utilizando identidades trigonométricas, y `expand_trig()` que aplica identidades para la suma o productos de ángulos:

CELDA 21
<code>sym.trigsimp(sym.sin(x) ** 2 + sym.cos(x) ** 2)</code>
1

CELDA 22
<code>sym.trigsimp(sym.sin(x) / sym.tan(x))</code>
$\cos(x)$

CELDA 23
<code>sym.trigsimp(sym.sinh(x) ** 2 + sym.cosh(x) ** 2)</code>
$\cosh(2x)$

CELDA 24
<code>sym.expand_trig(sym.sin(x) + sym.sin(3 * x))</code>
$-4 \sin^3(x) + 4 \sin(x)$

13.2.3. Simplificación de potencias

SymPy es capaz de simplificar expresiones utilizando las reglas usuales de potenciación:

1. $x^a x^b = x^{a+b}$
2. $x^a y^a = (xy)^a$
3. $(x^a)^b = x^{ab}$

No obstante, no realiza simplificaciones cuando dichas expresiones no son válidas en general. Por ejemplo: $x^a x^b = x^{a+b}$ es una igualdad válida, pero $x^a y^a = (xy)^a$ solo lo es cuando x y y no son negativos y a es real. Por defecto, SymPy asume que los símbolos sobre los que opera son complejos, pero como vimos previamente, es posible indicar atributos durante la creación de estos símbolos. Por ejemplo:

CELDA 25
<code>x, y = sym.symbols('x y', positive=True)</code> <code>a, b = sym.symbols('a b', real=True)</code> <code>m, n, l = sym.symbols('m n l')</code>

CELDA 26
<code>sym.powsimp(x ** a * x ** b)</code>
x^{a+b}

CELDA 27
<code>sym.powsimp(x ** a * y ** a)</code>
$(xy)^a$

CELDA 28
<code>sym.powsimp(m ** l * n ** l)</code>
$m^l n^l$

En el último caso, no hemos indicado atributos en la creación de m , n , y l que permitan a SymPy simplificar la expresión, sin embargo, podemos forzar a que se realice la simplificación independientemente de los atributos:

CELDA 29
<code>sym.powsimp(m ** l * n ** l, force=True)</code>
$(mn)^l$

Las operaciones de expansión de expresiones con potencias aplican las identidades 1 y 2 de derecha a izquierda, respectivamente, mediante las funciones `expand_power_exp()` y `expand_power_base()`:

CELDA 30
<code>sym.expand_power_exp(x ** (a + b))</code>
$x^a x^b$

CELDA 31
<code>sym.expand_power_base((x * y) ** a)</code>
$x^a y^a$

Al igual que en la simplificación, la identidad 2 no se aplica si no es válida, aunque podemos forzar a que se produzca:

CELDA 32
<code>sym.expand_power_base((m * n) ** l)</code>
$(mn)^l$

CELDA 33
<code>sym.expand_power_base((m * n) ** l, force=True)</code>
$m^l n^l$

Para finalizar, podemos aplicar la identidad 3 usando `powdenest()`, siempre que los atributos de los símbolos garanticen su validez:

CELDA 34	
<code>sym.powdenest((x ** a) ** b)</code>	
x^{ab}	

CELDA 35	
<code>sym.powdenest((m ** a) ** b)</code>	
$(m^a)^b$	

CELDA 36	
<code>sym.powdenest((m ** a) ** b, force=True)</code>	
m^{ab}	

13.2.4. Simplificación de exponenciales y logaritmos

Las principales identidades que intenta aplicar SymPy para simplificar expresiones con logaritmos son:

1. $\log(xy) = \log(x) + \log(y)$
2. $\log(x^n) = n \log(x)$

Estas identidades no son válidas para valores complejos arbitrarios de x y y , por lo que para que las expresiones puedan simplificarse es necesario establecer los atributos correspondientes:

CELDA 37	
<pre>x, y = sym.symbols('x y', positive=True) n = sym.symbols('n', real=True) p, q = sym.symbols('p q')</pre>	

CELDA 38	
<code>sym.expand_log(sym.log(x * y))</code>	
$\log(x) + \log(y)$	

CELDA 39	
<code>sym.expand_log(sym.log(x / y))</code>	
$\log(x) - \log(y)$	

CELDA 40

`sym.expand_log(sym.log(x ** 3))` $3 \log(x)$

CELDA 41

`sym.expand_log(sym.log(x ** n))` $n \log(x)$

CELDA 42

`sym.expand_log(sym.log(p * q))` $\log(pq)$

CELDA 43

`sym.expand_log(sym.log(p * q), force=True)` $\log(p) + \log(q)$

La operación inversa a la expansión se realiza con `logcombine()`:

CELDA 44

`sym.logcombine(sym.log(x) + sym.log(y))` $\log(xy)$

CELDA 45

`sym.logcombine(n * sym.log(x))` $\log(x^n)$

CELDA 46

`sym.logcombine(n * sym.log(p))` $n \log(p)$

CELDA 47

`sym.logcombine(n * sym.log(p), force=True)` $\log(p^n)$

13.2.5. Funciones especiales

Para finalizar esta sección mencionaremos algunas funciones más que permiten manipular expresiones para expresarlas de un modo más simple (o conveniente). Una de ellas es `rewrite()`, que sirve para escribir una expresión en términos de una función, tal como vemos en los ejemplos siguientes:

CELDA 48
<code>sym.tan(x).rewrite(sym.sin)</code>
$\frac{2 \sin^2(x)}{\sin(2x)}$

CELDA 49
<code>sym.factorial(x).rewrite(sym.gamma)</code>
$\Gamma(x + 1)$

Es posible también simplificar expresiones combinatorias por medio de `combsimp()`:

CELDA 50
<code>n, k = sym.symbols('n k', integer=True)</code> <code>sym.combsimp(sym.factorial(n + 1) / sym.factorial(n - 1))</code>
$n(n + 1)$

CELDA 51
<code>sym.combsimp(sym.binomial(n + 2, k + 2) / sym.binomial(n, k))</code>
$\frac{(n + 1)(n + 2)}{(k + 1)(k + 2)}$

Del mismo modo podemos intentar transformar expresiones que contienen funciones gamma o combinatorias con argumentos no enteros, usando la función `gammasimp()`:

CELDA 52
<code>sym.gammasimp(sym.gamma(2 + x) * sym.gamma(1 - x))</code>
$\frac{\pi x (x + 1)}{\sin(\pi x)}$

Por último, mostramos un ejemplo en el que expandimos una función especial (gamma) utilizando alguna identidad conocida, utilizando la función `expand_func()`:

CELDA 53
<code>sym.expand_func(sym.gamma(x + 4))</code>
$x(x+1)(x+2)(x+3)\Gamma(x)$

13.3. Solución de ecuaciones

SymPy es capaz de resolver, en forma simbólica, ecuaciones lineales y no lineales, sistemas de ecuaciones lineales, desigualdades, ecuaciones diofánticas y ecuaciones diferenciales². En caso que la solución analítica de una ecuación (o sistema de ecuaciones) no exista, SymPy provee la posibilidad de resolverlas en forma numérica.

Antes de iniciar un recorrido sobre algunas de estas posibilidades, es necesario recordar que una ecuación en SymPy no se representa con un símbolo $=$ o $==$, sino a través de un objeto Eq. Por ejemplo:

CELDA 1 DE ECUACIONES.IPYNB
<code>import sympy as sym</code>
<code>x, y = sym.symbols('x y')</code>
<code>sym.Eq(2 * x + 8, 16 * y)</code>
$2x + 8 = 16y$

Por otra parte, cualquier expresión que no sea una instancia de Eq se asume igual a cero en el contexto de funciones que resuelven ecuaciones (usualmente denominadas «*solvers*»). Dado que en el ejemplo anterior $2x + 8 = 16y$ es equivalente a $2x + 8 - 16y = 0$, podemos pasar esta última expresión como argumento a un *solver* para intentar obtener una solución a dicha ecuación:

CELDA 02
<code>sym.solve(sym.Eq(2 * x + 8, 16 * y), x)</code>
$\{8y - 4\}$

CELDA 03
<code>sym.solve(sym.Eq(2 * x + 8 - 16 * y, 0), x)</code>
$\{8y - 4\}$

CELDA 04
<code>sym.solve(2 * x + 8 - 16 * y, x)</code>

²Para ver cómo resolver ecuaciones diferenciales, ver el Capítulo 20.

$$\{8y - 4\}$$

Existen dos funciones para resolver ecuaciones algebraicas: `solve()` y `solveset()`. La primera opción es útil para obtener representaciones simbólicas de las posibles soluciones de una ecuación, y para sustituir dichas representaciones en otras ecuaciones o expresiones. `solveset()` es una alternativa que representa las soluciones utilizando conjuntos matemáticos, y permite obtener todas las soluciones incluso si el conjunto es infinito (las recomendaciones de uso de una función u otra se pueden ver [aquí](#)). Veamos algunos ejemplos.

CELDA 05

```
sym.solve(x ** 2 + 2 * x - 3)
```

```
[-3, 1]
```

CELDA 06

```
sym.solveset(x ** 2 + 2 * x - 3, x)
```

```
{-3, 1}
```

En las dos celdas precedentes resolvemos la ecuación $x^2 + 2x - 3 = 0$ (o lo que es lo mismo, determinamos las raíces del polinomio). En el caso de `solve()`, obtenemos una lista enumerando las soluciones para la variable x , mientras que con `solveset()` lo que resulta es un conjunto finito. En la expresión que representa al polinomio, el único símbolo «indeterminado» es la variable independiente x , mientras que los coeficientes y potencias son números. Por este motivo es opcional indicar sobre qué símbolo debe resolverse la ecuación (no especificado en la celda 5). Naturalmente, podemos utilizar expresiones que contengan otros símbolos además de la variable que queremos determinar como solución, y en este caso debemos indicar cuál es esa variable:

CELDA 07

```
a, b, c = sym.symbols('a b c')
sym.solveset(a * x ** 2 + b * x + c, x)
```

$$\left\{ -\frac{b}{2a} - \frac{\sqrt{-4ac + b^2}}{2a}, -\frac{b}{2a} + \frac{\sqrt{-4ac + b^2}}{2a} \right\}$$

SymPy permite, en algunos casos, hallar también la solución de ecuaciones trigonométricas. En el ejemplo siguiente podemos apreciar la diferencia en los resultados de `solve()` y `solveset()`:

CELDA 08

```
sym.solve(sym.cos(x) - sym.sin(x), x)
```

```
[pi/4]
```

CELDA 09

```
sym.solve(sym.cos(x) - sym.sin(x), x)
```

$$\left\{2n\pi + \frac{5\pi}{4} \mid n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{\pi}{4} \mid n \in \mathbb{Z}\right\}$$

En los casos en que no exista una solución analítica, o que SymPy sea incapaz de hallarla, devolverá una solución formal que eventualmente puede ser resuelta en forma numérica:

CELDA 10

```
sym.solve(x ** 5 - x ** 2 + 1, x)
```

```
[CRootOf(x**5 - x**2 + 1, 0),  
CRootOf(x**5 - x**2 + 1, 1),  
CRootOf(x**5 - x**2 + 1, 2),  
CRootOf(x**5 - x**2 + 1, 3),  
CRootOf(x**5 - x**2 + 1, 4)]
```

CELDA 11

```
sym.solve(sym.tan(x) + x, x)
```

$$\{x \mid x \in \mathbb{C} \wedge x + \tan(x) = 0\}$$

mientras que si no existe la solución de una ecuación, estas funciones devolverán una lista vacía o un conjunto vacío:

CELDA 12

```
sym.solve(sym.exp(x), x)
```

```
[]
```

CELDA 13

```
sym.solve(sym.exp(x), x)
```

```
∅
```

La solución de un sistema de ecuaciones consiste en extender el procedimiento que utilizamos para ecuaciones de una sola incógnita. Simplemente es necesario pasar como argumento una lista de las ecuaciones a resolver, y otra con la lista de símbolos para los que se busca la solución (o las soluciones). Por ejemplo:

CELDA 14

```
ecuación_1 = 2 * x - 5 * y + 4  
ecuación_2 = -3 * x + 4 * y + 1  
sym.solve([ecuación_1, ecuación_2], [x, y], dict=True)
```

```
[{x: 3, y: 2}]
```

CELDA 15

```

ecuación_1 = x ** 2 - 2 * y
ecuación_2 = y ** 2 - 2 * x
soluciones = sym.solve([ecuación_1, ecuación_2], [x, y], dict=True)
soluciones

[{x: 0, y: 0},
 {x: 2, y: 2},
 {x: (-1 - sqrt(3)*I)**2/2, y: -1 - sqrt(3)*I},
 {x: (-1 + sqrt(3)*I)**2/2, y: -1 + sqrt(3)*I}]

```

En estas últimas dos celdas, `solve()` devuelve una lista en la que cada elemento representa una solución del sistema. Además, hemos pasado como argumento opcional `dict=True` de forma que la respuesta de SymPy sea en forma de diccionario, lo que resulta útil para reutilizar los valores obtenidos de las soluciones. Por ejemplo, podemos verificar que se satisfacen las ecuaciones:

CELDA 16

```

[ecuación_1.subs(sol).simplify() == 0 and ecuación_2.subs(sol).simplify() == 0
 for sol in soluciones]

[True, True, True, True]

```

Por defecto, SymPy ofrecerá soluciones en el campo complejo para las incógnitas:

CELDA 17

```

sym.solve(x ** 4 - 2 ** 4, x)

[-2, 2, -2*I, 2*I]

```

CELDA 18

```

sym.solveset(x ** 4 - 2 ** 4, x)

{-2, 2, -2i, 2i}

```

Es posible restringir el dominio de búsqueda de soluciones, por ejemplo limitando la incógnita al campo real o a un intervalo dado. Las funciones `solve()` y `solveset()` tienen abordajes diferentes para ello.

En el caso de `solve()`, podemos limitar el dominio al campo real incorporando una condición a la definición de la incógnita:

CELDA 19

```

x = sym.Symbol('x', real=True)
sym.solve(x ** 4 - 2 ** 4, x)

[-2, 2]

```

o utilizando un método de Python para restringir las soluciones a un determinado intervalo:

CELDA 20

```
polinomio = x ** 4 + 2 * x ** 3 - 25 * x ** 2 - 26 * x + 120
solución = sym.solve(polinomio, x)
solución
```

```
[-5, -3, 2, 4]
```

CELDA 21

```
sol_intervalo = [v for v in solución if (v.is_real and sym.Or(v < 0 , v > 3))]
sol_intervalo
```

```
[-5, -3, 4]
```

donde hemos hecho uso de la disyunción lógica `Or`. Para restringir el dominio al utilizar `solveset()`, dicho dominio se debe pasar como argumento:

CELDA 22

```
sym.solveset(x ** 4 - 2 ** 4, x, domain=sym.S.Reals)
```

```
{-2, 2}
```

CELDA 23

```
sym.solveset(polinomio, x, sym.Interval(0, 3))
```

```
{2}
```

En la celda 21, `sym.S.Reals` es un «singleton» que representa los números reales desde $-\infty$ a $+\infty$.

Como vimos en un ejemplo precedente, cuando SymPy no es capaz de resolver una ecuación en forma analítica devuelve una expresión sin evaluar o una excepción:

CELDA 24

```
sym.solve(sym.cos(x) - x, x)
```

```
NotImplementedError                                Traceback (most recent call last)
Cell In[24], line 1
----> 1 sym.solve(sym.cos(x) - x, x)

File ~/config/jupyterlab-desktop/jlab_server/lib/python3.12/site-
  packages/sympy/solvers/solvers.py:1170, in solve(f, *symbols, **flags)
    1168     solution = _solve_undetermined(f[0], symbols, flags)
    1169     if not solution:
-> 1170         solution = _solve(f[0], *symbols, **flags)
    1171 else:
    1172     linear, solution = _solve_system(f, symbols, **flags)

File ~/config/jupyterlab-desktop/jlab_server/lib/python3.12/site-
  packages/sympy/solvers/solvers.py:1729, in _solve(f, *symbols, **flags)
    1726 # ----- end of fallback -----
    1728 if result is False:
```

```
-> 1729     raise NotImplementedError('\n'.join([msg, not_impl_msg % f]))
      1731 result = _remove_duplicate_solutions(result)
      1733 if flags.get('simplify', True):
```

```
NotImplementedError: multiple generators [x, cos(x)]
No algorithms are implemented to solve equation -x + cos(x)
```

En estos casos, podemos resolver la ecuación en forma numérica con la función `nsolve()`, a la que es necesario pasar como argumento la expresión a resolver, el símbolo que representa la incógnita, y un valor próximo a la solución que inicia la secuencia de aproximaciones numéricas:

CELDA 25

```
sym.nsolve(sym.cos(x) - x, x, 1)
```

```
0.739085133215161
```

Es habitual representar un sistema de ecuaciones lineales en forma matricial. SymPy permite resolver ecuaciones de esta forma, ya sea con un *solver* como `solve()` y también con factorizaciones como la descomposición LU o QR. Mostramos algunos ejemplos de esta forma de resolver ecuaciones matriciales en el capítulo 18.

13.4. Funciones y cálculo diferencial e integral

En esta sección exploraremos superficialmente las capacidades de SymPy para el cálculo diferencial e integral, herramientas fundamentales en el análisis matemático. Comenzaremos con el concepto de funciones en el contexto de SymPy y a continuación abordaremos los cálculos de límites, derivadas e integrales. Finalmente mostraremos como evaluar expresiones que contienen sumas, productos y la posibilidad de expandir funciones en series de potencia o de Fourier.

13.4.1. Funciones

Así como podemos definir símbolos en SymPy, podemos también definir funciones. SymPy permite definir funciones abstractas (o indefinidas), y funciones definidas. Las funciones indefinidas son instancias de `sympy.Function` y, al igual que con los símbolos, el primer argumento del constructor de `Function` es el nombre que le asignamos.

Una función indefinida no se puede evaluar, ya que no tiene definido el cuerpo de la función. Al igual que con los símbolos, se pueden establecer atributos sobre funciones que permitan su manipulación aprovechando propiedades derivadas de estos atributos. Veamos algunos ejemplos.

CELDA 1 DE CALCULO.IPYNB

```
import sympy as sym

x, y, z = sym.symbols('x y z')
f = sym.Function('f')
type(f)
```

```
sympy.core.function.UndefinedFunction
```

En la celda precedente, luego de importar el módulo `sympy`, definimos los símbolos x , y y z como hicimos anteriormente, y luego asignamos a la variable `f` una función indefinida. Esta función, además de carecer de cuerpo, no ha sido aplicada a ningún conjunto de símbolos que representen sus argumentos, pero podemos hacerlo sobre los símbolos ya definidos:

CELDA 02
<code>f(x)</code>
$f(x)$

CELDA 03
<code>f(x, y, z)</code>
$f(x, y, z)$

También podemos definir funciones indefinidas que dependan explícitamente de argumentos:

CELDA 04
<code>g = sym.Function('g')(x, y, z)</code> <code>g</code>
$g(x, y, z)$

Podemos acceder al conjunto de argumentos establecidos para f y g con el método `free_symbols`:

CELDA 05
<code>f.free_symbols, g.free_symbols</code>
<code>(set(), {x, y, z})</code>

resultando el conjunto vacío para el caso de f . Se pueden establecer propiedades sobre funciones del mismo modo que como hicimos para símbolos:

CELDA 06
<code>f_real = sym.Function('f', real=True)</code> <code>f_real(x).is_real</code>
<code>True</code>

o también heredando propiedades de símbolos:

CELDA 07
<code>f_simbolo = sym.Symbol('f', real=True)</code> <code>f_real_herencia = sym.Function(f_simbolo)</code> <code>f_real_herencia.is_real</code>
<code>True</code>

Las funciones indefinidas son útiles en diversos contextos, por ejemplo, para escribir ecuaciones diferenciales en las que no conocemos la forma de la función, y justamente resolver la ecuación es lo que permite establecer cuál es su cuerpo. Por otro lado, es posible también definir funciones definidas cuya dependencia operacional con sus argumentos se encuentra implementada, ya sea como subclase de `sympy.Function` o como combinación de las múltiples funciones definidas en SymPy. Por ejemplo, la función coseno se encuentra entre las disponibles en SymPy, y representa una función no evaluada que se puede aplicar a un símbolo, a un número o a una expresión:

CELDA 08	
<code>type(sym.cos)</code>	
<code>sympy.core.function.FunctionClass</code>	

CELDA 09	
<code>sym.cos(sym.pi * 2)</code>	
1	

Cuando el argumento es un símbolo, la función se mantiene sin evaluar excepto que sea posible su evaluación hacia un valor numérico o una expresión a partir de las propiedades de su argumento, tal como se muestra en la celda siguiente:

CELDA 10	
<code>n = sym.Symbol('n', integer=True, odd=True)</code> <code>sym.cos(n * sym.pi)</code>	
-1	

Podemos definir el cuerpo de nuestra propia función al estilo usual de funciones de Python. En este caso, si tenemos previamente definidos como símbolos los argumentos de la función, lo que devuelve `return` en la función `f_py(x, y)` siguiente es una expresión válida de dichos símbolos:

CELDA 11	
<code>def f_py(x, y):</code> <code>return (x + y) ** 3</code>	
<code>print(type(f_py))</code> <code>f_py(x, y)</code>	
<class 'function'>	
$(x + y)^3$	

En este ejemplo, `f_py` devuelve una expresión simbólica dado que los argumentos de esta función ya fueron definidos como símbolos x y y . No obstante, podemos también evaluar numéricamente a `f_py` si le pasamos como argumentos dos números de un tipo de datos intrínseco del lenguaje, que en el siguiente ejemplo son del tipo `int`:

	CELDA 12
<code>f_py(1, 2)</code>	
27	

mientras que si los argumentos son símbolos, podemos realizar algún tipo de manipulación simbólica sobre `f_py`:

	CELDA 13
<code>sym.expand(f_py(x, 3))</code>	
$x^3 + 9x^2 + 27x + 27$	

Por último, SymPy permite definir funciones anónimas, o funciones lambda, que no tienen nombre pero implementan el cuerpo de la función de forma que permitan su evaluación. Estas funciones son instancias de `sympy.Lambda` y reciben como primer argumento una tupla con los símbolos que representan las variables independientes, y como segundo argumento la expresión que representa el cuerpo de la función:

	CELDA 14
<code>h = sym.Lambda((x, y), (x + y) ** 2)</code> <code>h</code>	
$((x, y) \mapsto (x + y)^2)$	

	CELDA 15
<code>h(3, y)</code>	
$(y + 3)^2$	

13.4.2. Límites

SymPy es capaz de calcular simbólicamente el límite de una expresión. Para obtener

$$\lim_{x \rightarrow x_0} f(x)$$

se utiliza la función `sympy.limit()`, pasando como primer argumento la expresión a la que se le calcula el límite, como segundo argumento la variable, y por último el valor al que se aproxima la variable, tal como muestra la celda siguiente:

	CELDA 16
<code>sym.limit(sym.sin(x) / x, x, 0)</code>	
1	

Naturalmente, en situaciones en que el límite dé origen a una expresión indefinida, no es válido evaluar este límite reemplazando el valor de la variable independiente por el valor al que

tiende. Por ejemplo, en la celda siguiente queremos calcular el límite al que tiende $x^3 - 3x$ cuando $x \rightarrow \infty$ (que se representa en SymPy como `sym.oo`):

CELDA 17
<pre>f = x ** 3 - 3 * x f.subs(x, sym.oo)</pre>
NaN

Dado que esta expresión resulta indeterminada, obtenemos un NaN (*Not-a-Number*) como respuesta. La forma correcta de realizar el cálculo es:

CELDA 18
<pre>sym.limit(f, x, sym.oo)</pre>
∞

Es posible también calcular límites simbólicos definidos sobre funciones abstractas, por ejemplo para el cálculo de derivadas por definición, que luego al utilizar funciones específicas obtenemos como resultado la derivada en cuestión:

CELDA 19
<pre>f = sym.Function('f') x, h = sym.symbols('x h') cociente_incremental = (f(x + h) - f(x)) / h sym.limit(cociente_incremental.subs(f, sym.sin), h, 0)</pre>
$\cos(x)$

CELDA 20
<pre>sym.limit(cociente_incremental.subs(f, sym.exp), h, 0)</pre>
e^x

Es posible generar una expresión que contenga el límite sin evaluar, que permite una evaluación posterior de la expresión obtenida. Estos límites sin evaluar también son utilizados por SymPy cuando no puede obtener un resultado, por ejemplo, cuando la expresión contiene una función abstracta. Para dejar sin evaluar un límite se utiliza la función `sympy.Limit` (con «L» mayúscula) con idéntica sintaxis que `sympy.limit()`:

CELDA 21
<pre>derivada = sym.Limit(cociente_incremental, h, 0) derivada</pre>
$\lim_{h \rightarrow 0^+} \left(\frac{-f(x) + f(h+x)}{h} \right)$

y para realizar la evaluación sobre una función definida, podemos utilizar primero el método `subs` que reemplaza la función abstracta f por una específica (en la celda siguiente, `sym.cos`), y

realizamos la evaluación con `doit()`:

CELDA 22
<code>derivada.subs(f, sym.cos).doit()</code>
$-\sin(x)$

Para finalizar esta sección, mostraremos que también es posible calcular límites laterales, estableciendo como cuarto parámetro el sentido desde el que nos aproximamos al valor límite del argumento: '+' para límite por la derecha, y '-' para el límite por izquierda:

CELDA 23
<code>l_sup = sym.limit(1 / (x - 2), x, 2, '+')</code> <code>l_inf = sym.limit(1 / (x - 2), x, 2, '-')</code> <code>l_sup, l_inf</code>
$(\infty, -\infty)$

13.4.3. Derivadas

Vimos en la sección anterior que podemos calcular la derivada de una función mediante su definición utilizando límites. SymPy permite calcular la derivada usando directamente la función `diff()`:

CELDA 24
<code>sym.diff(sym.cos(x) ** 2, x)</code>
$-2 \sin(x) \cos(x)$

o utilizando el método `.diff()` de una expresión:

CELDA 25
<code>expr = sym.cos(x) ** 2</code> <code>expr.diff()</code>
$-2 \sin(x) \cos(x)$

Para realizar múltiples derivadas se puede repetir la variable las veces que sean necesarias:

CELDA 26
<code>sym.diff(sym.exp(x ** 2), x, x, x)</code>
$4x(2x^2 + 3)e^{x^2}$

o simplemente indicar el orden de la derivación con un número entero:

CELDA 27

```
sym.diff(sym.exp(x ** 2), x, 3)
```

$$4x(2x^2 + 3)e^{x^2}$$

También es posible realizar el cálculo de derivadas parciales de múltiples variables. Por ejemplo, para calcular

$$\frac{\partial^8}{\partial x^2 \partial y^2 \partial z^4} e^{xyz}$$

podemos escribir:

CELDA 28

```
sym.diff(sym.exp(x * y * z), x, 2, y, y, z, 4)
```

$$x^2 y^2 (x^4 y^4 z^4 + 20x^3 y^3 z^3 + 122x^2 y^2 z^2 + 256xyz + 144) e^{xyz}$$

Dado que el cálculo de derivadas es una tarea relativamente simple, es posible evaluar las derivadas de la mayoría de las funciones matemáticas definidas en SymPy, incluso las que involucran expresiones complicadas:

CELDA 29

```
expresion = sym.hermite(x, 0)
expresion.diff(x)
```

$$\frac{2^x \sqrt{\pi} \text{polygamma}\left(0, \frac{1}{2} - \frac{x}{2}\right)}{2\Gamma\left(\frac{1}{2} - \frac{x}{2}\right)} + \frac{2^x \sqrt{\pi} \log(2)}{\Gamma\left(\frac{1}{2} - \frac{x}{2}\right)}$$

Del mismo modo que para el caso de los límites, podemos crear una expresión que contenga una derivada sin evaluar:

CELDA 30

```
derivada = sym.Derivative(sym.exp(x * y * z), x, 2, y, y, z, 4)
derivada
```

$$\frac{\partial^8}{\partial z^4 \partial y^2 \partial x^2} e^{xyz}$$

Cuando queramos efectivamente evaluar la derivada, invocamos el método `doit()`

CELDA 31

```
derivada.doit()
```

$$x^2 y^2 (x^4 y^4 z^4 + 20x^3 y^3 z^3 + 122x^2 y^2 z^2 + 256xyz + 144) e^{xyz}$$

Desde luego, podemos utilizar la función o el método `diff()` sobre funciones abstractas:

CELDA 32

```
f, g = sym.symbols('f g', cls=sym.Function)
f(x).diff()
```

$$\frac{d}{dx}f(x)$$

CELDA 33

```
g(x, y, z).diff(x, x, y, 3)
```

$$\frac{\partial^5}{\partial y^3 \partial x^2} g(x, y, z)$$

CELDA 34

```
f(x).diff(y)
```

$$0$$

Las derivadas de expresiones abstractas son útiles porque permiten utilizarlas para representar, por ejemplo, ecuaciones diferenciales. Para ilustrar un caso de uso, podemos resolver la ecuación diferencial

$$f''(x) - f(x) = \sin(x)$$

Definimos entonces una expresión que representa esta ecuación:

CELDA 35

```
ecdif = sym.Eq(f(x).diff(x, 2) - f(x), sym.sin(x))
ecdif
```

$$-f(x) + \frac{d^2}{dx^2}f(x) = \sin(x)$$

y pasamos esta expresión como argumento de la función `dsolve()`:

CELDA 36

```
sym.dsolve(ecdif, f(x))
```

$$f(x) = C_1 e^{-x} + C_2 e^x - \frac{\sin(x)}{2}$$

Observamos que `dsolve()` devuelve una instancia de `Eq` debido a que, por lo general, las soluciones de una ecuación diferencial no se pueden resolver explícitamente por medio de una función. Como es usual, las constantes C_1 y C_2 deben determinarse a partir de las condiciones de contorno de la ecuación.

13.4.4. Integrales

El cálculo de integrales se realiza mediante la función `sympy.integrate()`, mientras que la representación formal de integrales se realiza mediante `sympy.Integral()` (que al igual que para los límites y las derivadas, pueden evaluarse explícitamente mediante el método `doit()`). De esta forma es posible calcular integrales definidas o indefinidas de una o más variables utilizando para ello algoritmos heurísticos para el reconocimiento de patrones, el algoritmo de [Risch-Norman](#) y consultas en tablas. De esta manera se pueden obtener las integrales de funciones algebraicas elementales y trascendentales, y una variedad de funciones especiales.

Para el caso de las integrales indefinidas, es necesario pasar como argumento de `integrate()` la función integrando y un símbolo que representa la variable de integración:

CELDA 37
<code>sym.integrate(sym.cos(x), x)</code>
$\sin(x)$

Debe observarse que el resultado no contiene la constante de integración. Para calcular integrales definidas, como segundo argumento hay que pasar una tupla con el símbolo correspondiente a la variable de integración, y los límites inferior y superior de la integral definida:

CELDA 38
<code>a, b = sym.symbols('a b')</code> <code>sym.integrate(sym.cos(x), (x, a, b))</code>
$-\sin(a) + \sin(b)$

Si `integrate()` es incapaz de determinar la primitiva del integrando, devolverá un objeto `Integral` no evaluado:

CELDA 39
<code>sym.integrate(x ** x, x)</code>
$\int x^x dx$

Tanto para el caso de integrales definidas o indefinidas, es posible pasar como argumento de `integrate()` múltiples tuplas para evaluar una integral múltiple. En el ejemplo siguiente definimos primero una integral múltiple que dejamos sin evaluar, y a continuación invocamos el método `mipdoit()` para obtener el resultado:

CELDA 40
<code>integral = sym.Integral(sym.exp(-x ** 2 - y ** 2), (x, 0, sym.oo), (y, 0, sym.oo))</code> <code>integral</code>

$$\int_0^{\infty} \int_0^{\infty} e^{-x^2-y^2} dx dy$$

CELDA 41

```
integral.doit()
```

$$\frac{\pi}{4}$$

SymPy provee rutinas para la evaluación numérica de integrales, que son especialmente útiles cuando no es posible determinar el resultado en forma analítica. Por ejemplo:

CELDA 42

```
integral_numérica = sym.integrate(sym.exp(-x ** 2 * abs(x-1)), (x, 0, sym.oo))
integral_numérica
```

$$\int_0^{\infty} e^{-x^2|x-1|} dx$$

CELDA 43

```
integral_numérica.evalf()
```

1,322

Por último, mencionamos también que SymPy es capaz de resolver integrales de línea mediante la función `line_integrate()`. En el ejemplo siguiente, definimos una curva paramétrica:

$$x(t) = e^t + 1, y(t) = e^t - 1$$

para t en el intervalo $[0, \ln 2]$, y posteriormente integramos la función $f(x, y) = x^2 + y^2$ a lo largo de esa curva:

CELDA 44

```
x, y, t = sym.symbols('x y t', real=True)
C = sym.Curve([sym.E ** t + 1, sym.E ** t - 1], (t, 0, sym.ln(2)))
sym.line_integrate(x ** 2 + y ** 2, C, [x, y])
```

$$\frac{20\sqrt{2}}{3}$$

13.4.5. Sumas, productos y series

Es posible representar simbólicamente sumas y productos con SymPy a través de instancias de las clases `Sum` y `Product`. Los constructores de ambas clases reciben como primer argumento una expresión, y una tupla $(n, n1, n2)$ como segundo argumento, siendo n un símbolo y $n1$ y $n2$ los límites inferior y superior para la suma o producto, respectivamente. Después de instanciar estos objetos, la evaluación de las sumas o productos se realiza con el método `doit()`. Por ejemplo, para el caso de sumas:

CELDA 45

```
n = sym.symbols('n', integer=True)
S1 = sym.Sum((-1) ** n / (n + 1), (n, 0, sym.oo))
S1
```

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{n+1}$$

CELDA 46

```
S1.doit()
```

$$\log(2)$$

CELDA 47

```
S2 = sym.Sum(1 / n ** 2, (n, 1, sym.oo))
S2
```

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

CELDA 48

```
S2.doit()
```

$$\frac{\pi^2}{6}$$

CELDA 49

```
S3 = sym.Sum(1 / n ** 4, (n, 1, 10))
S3
```

$$\sum_{n=1}^{10} \frac{1}{n^4}$$

CELDA 50
<pre>S3.doit()</pre>
$\frac{43635917056897}{40327580160000}$

Podemos notar que en las celdas 45 y 46 utilizamos ∞ (sym.oo) como límite superior, y claramente la operación no puede evaluarse sumando explícitamente. SymPy realiza el cálculo en forma analítica, incluso en casos en los que el sumando contiene variables simbólicas además del índice de suma, como en el ejemplo a continuación:

CELDA 51
<pre>S4 = sym.Sum(x ** n / sym.factorial(n), (n, 1, sym.oo)) S4.doit().simplify()</pre>
$e^x - 1$

Para el caso de los productos, es posible evaluar expresiones con valores acotados del índice:

CELDA 52
<pre>P = sym.Product(1 / n, (n, 1, 10)) P</pre>
$\prod_{n=1}^{10} \frac{1}{n}$

CELDA 53
<pre>P.doit()</pre>
$\frac{1}{3628800}$

Sin embargo, para productos de infinitos factores resulta muy difícil obtener en forma directa una expresión cerrada. Si consideramos el [producto de Wallis](#):

CELDA 54
<pre>W = sym.Product(2 * n / (2 * n - 1) * 2 * n / (2 * n + 1), (n, 1, sym.oo)) W.doit()</pre>
$\prod_{n=1}^{\infty} \frac{4n^2}{(2n-1)(2n+1)}$

vemos que el cálculo directo falla y devuelve la expresión sin evaluar. Podemos recorrer un camino indirecto si definimos el producto de Wallis con un número finito de factores

CELDA 55

```
m = sym.Symbol('m', integer=True, positive=True)
wallis = sym.Product(2 * n / (2 * n - 1) * 2 * n / (2 * n + 1), (n, 1, m))
WE = wallis.doit()
WE
```

$$\frac{2^{-2m} 4^m m!^2}{\left(\frac{1}{2}\right)^{(m)} \left(\frac{3}{2}\right)^{(m)}}$$

y luego evaluamos el límite cuando $m \rightarrow \infty$:

CELDA 56

```
sym.limit(WE, m, sym.oo)
```

$$\frac{\pi}{2}$$

lo que nos produce el resultado correcto.

Dada una función, se puede obtener su expansión en serie de potencias mediante el método `series()` o invocando a la función `series()`, indicando en ambos casos el valor alrededor del cual se expande la serie, y el orden hasta el que se obtienen los términos:

CELDA 57

```
f = sym.cos(x)
f.series(n=10)
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320} + O(x^{10})$$

CELDA 58

```
sym.series(f, x0=sym.pi)
```

$$-1 + \frac{(x - \pi)^2}{2} - \frac{(x - \pi)^4}{24} + O\left((x - \pi)^6; x \rightarrow \pi\right)$$

Al omitir el valor de `x0` se asume que $x_0 = 0$, mientras que si omitimos el valor para `n`, toma por defecto el valor 6. El término $O(x^{10})$ representa todos los términos de la suma con potencia mayor o igual a x^{10} . Esta notación también se utiliza para otros puntos límite arbitrarios (por ejemplo $x \rightarrow \pi$ en la celda 58). Para evitar incorporar ese término a la expresión se utiliza el método `remove0()`:

CELDA 59

```
f.series(n=10).remove0()
```

$$\frac{x^8}{40320} - \frac{x^6}{720} + \frac{x^4}{24} - \frac{x^2}{2} + 1$$

El método (o la función) `series()` devuelve una expresión. Pero si especificamos `n=None`, lo que obtenemos como resultado es un generador, que nos permite coleccionar, por ejemplo, un número determinado de términos de la serie:

CELDA 60
<pre>serie = f.series(n=None) [next(serie) for i in range(7)]</pre>
<pre>[1, -x**2/2, x**4/24, -x**6/720, x**8/40320, -x**10/3628800, x**12/479001600]</pre>

Para finalizar esta sección, mostramos que SymPy es capaz de generar expansiones en series de Fourier. Si $f(x)$ está definida sobre el intervalo (a, b) , la serie de Fourier correspondiente es:

$$\frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2n\pi x}{L}\right) + b_n \sin\left(\frac{2n\pi x}{L}\right) \right]$$

donde:

$$\begin{aligned} L &= b - a \\ a_0 &= \frac{2}{L} \int_a^b f(x) dx \\ a_n &= \frac{2}{L} \int_a^b f(x) \cos\left(\frac{2n\pi x}{L}\right) dx \\ b_n &= \frac{2}{L} \int_a^b f(x) \sin\left(\frac{2n\pi x}{L}\right) dx \end{aligned}$$

Dada entonces una función, el método `fourier_series()` realiza la expansión. Podemos visualizar los términos hasta un cierto orden por medio del método `truncate()`:

CELDA 61
<pre>f = x ** 2 g = x Ff = sym.fourier_series(f, (x, -sym.pi, sym.pi)) Fg = sym.fourier_series(g, (x, -sym.pi, sym.pi)) Ff.truncate(n=5)</pre>
$-4 \cos(x) + \cos(2x) - \frac{4 \cos(3x)}{9} + \frac{\cos(4x)}{4} + \frac{\pi^2}{3}$

CELDA 62
<pre>Fg.truncate(n=4)</pre>
$2 \sin(x) - \sin(2x) + \frac{2 \sin(3x)}{3} - \frac{\sin(4x)}{2}$

13.5. Evaluaciones numéricas y gráficos

Hemos visto en las secciones precedentes una introducción muy superficial de las capacidades de SymPy para realizar cálculo simbólico. Las posibilidades que ofrece este módulo son mucho más extensas y profundas de lo que podemos abarcar en este capítulo, por lo que invitamos a quienes estén interesados en conocer más a leer la [documentación](#).

Sin embargo, es muy común que en alguna instancia del cálculo necesitemos ver números, o gráficos, generados por las expresiones simbólicas. Para realizar evaluaciones numéricas de expresiones podemos usar el método `evalf()` (que mencionamos al final de la primera sección para evaluar constantes) o la función `N()`.

CELDA 63
<pre>f = sym.sin(x) + sym.pi f_numérico = f.evalf(subs={x:1}) print(f"f(1) = {f_numérico}")</pre>
<pre>f(1) = 3.98306363839769</pre>

Se puede especificar la precisión de las evaluaciones numéricas (por defecto establecido en 15 dígitos decimales) pasando como primer argumento un entero positivo:

CELDA 64
<pre>f_preciso = f.evalf(50, subs={x:1}) print(f"f(1) = {f_preciso}")</pre>
<pre>f(1) = 3.9830636383976897451151457049098018838197324601735</pre>

CELDA 65
<pre>sym.N(f.subs({x:1}), 50)</pre>
<pre>3,9830636383976897451151457049098018838197324601735</pre>

De esta forma se pueden evaluar expresiones en determinados valores de algún símbolo que contenga la expresión que resulta reemplazado mediante `subs()`. Esto no resulta práctico cuando necesitamos realizar evaluaciones numéricas masivas. Para ello es conveniente utilizar la función `lambdify()`, que permite realizar cálculos vectorizados veloces por medio del módulo NumPy. Entre otras aplicaciones, estas evaluaciones masivas permiten convertir expresiones simbólicas en arrays numéricos para su visualización con Matplotlib (aunque SymPy tiene su propia función `plot()`, no resulta tan versátil como el popular módulo de visualización). Aprovecharemos las expansiones en series de Fourier de la sección anterior para realizar evaluaciones numéricas vectorizadas y visualizar estos resultados.

En la celda 66 desarrollamos un pequeño script para ver cómo se aproximan a la función $f(x) = x$ las sucesivas sumas parciales de la expansión en serie de Fourier con n términos. Primero importamos los módulos Matplotlib y NumPy (para realizar el gráfico y tener soporte para el uso de arrays, respectivamente).

A continuación asignamos a Fx la transformada de Fourier, entre $-\pi$ y π , de la función f , y construimos un array de 500 puntos equiespaciados entre estos límites, asignándolo a `x_array`.

Inmediatamente después graficamos este array en función de si mismo (lo que representa $f(x) = x$).

Para generar las sumas parciales incorporando un número creciente de términos, recorremos un bucle asignando a la variable n los valores de la lista `[1, 3, 5, 7]`. Dentro de este bucle, la primera instrucción consiste en truncar la expansión de Fourier en el número de términos indicados por n . Este paso convierte un objeto del tipo `FourierSeries` (devuelto por `fourier_series()`) en una instancia de `Expr`, ya que `lambdify()` acepta expresiones que representan funciones como argumento, pero no un objeto del tipo `FourierSeries`.

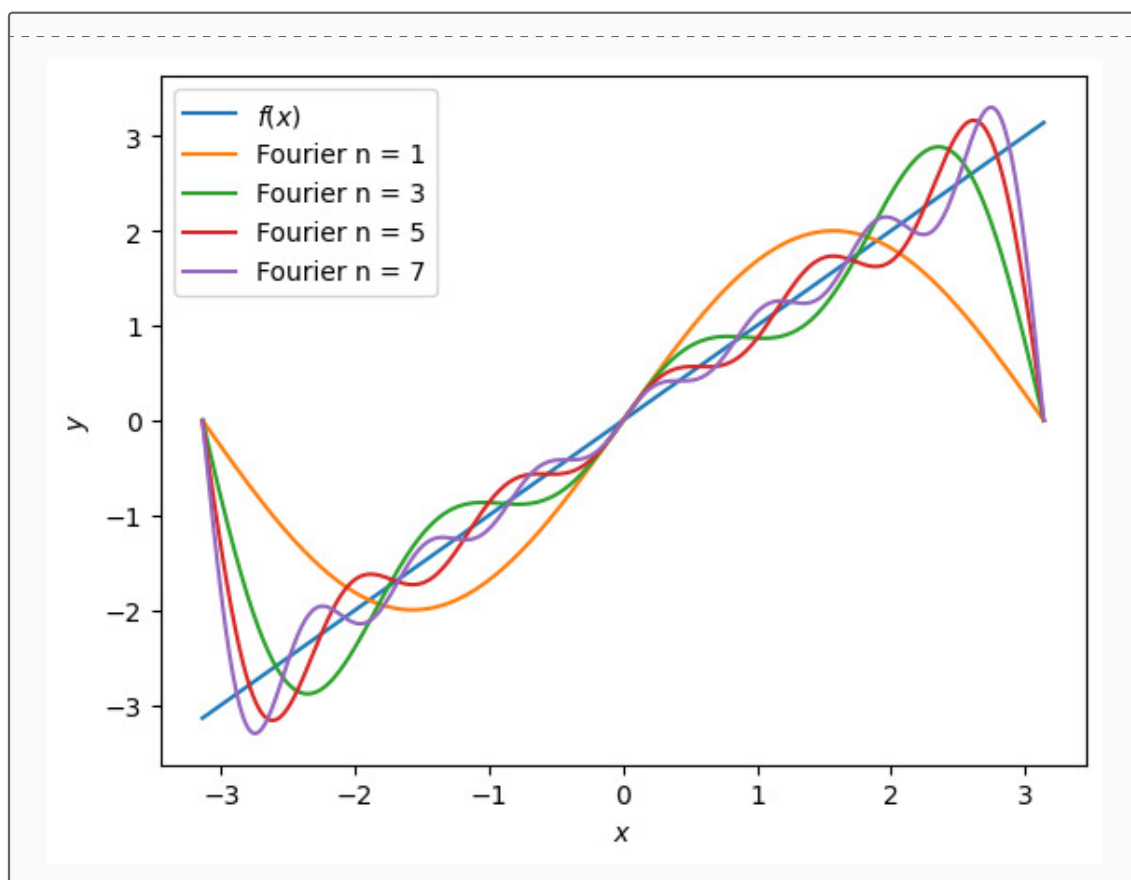
Una vez truncada la serie, le pasamos a `lambdify()` el símbolo que representa la variable que evaluaremos numéricamente (x), la expresión que representa la función a evaluar (`F_trunc`) y el módulo que se utilizará para realizar la evaluación numérica de la expresión simbólica. En este caso usaremos el módulo `'numpy'`, pero son posibles otras opciones como `'math'`, que es el módulo por defecto de Python, o `'mpmath'` para cálculos de precisión arbitraria. Asignamos a la variable `F_numerica` la función que devuelve `lambdify`.

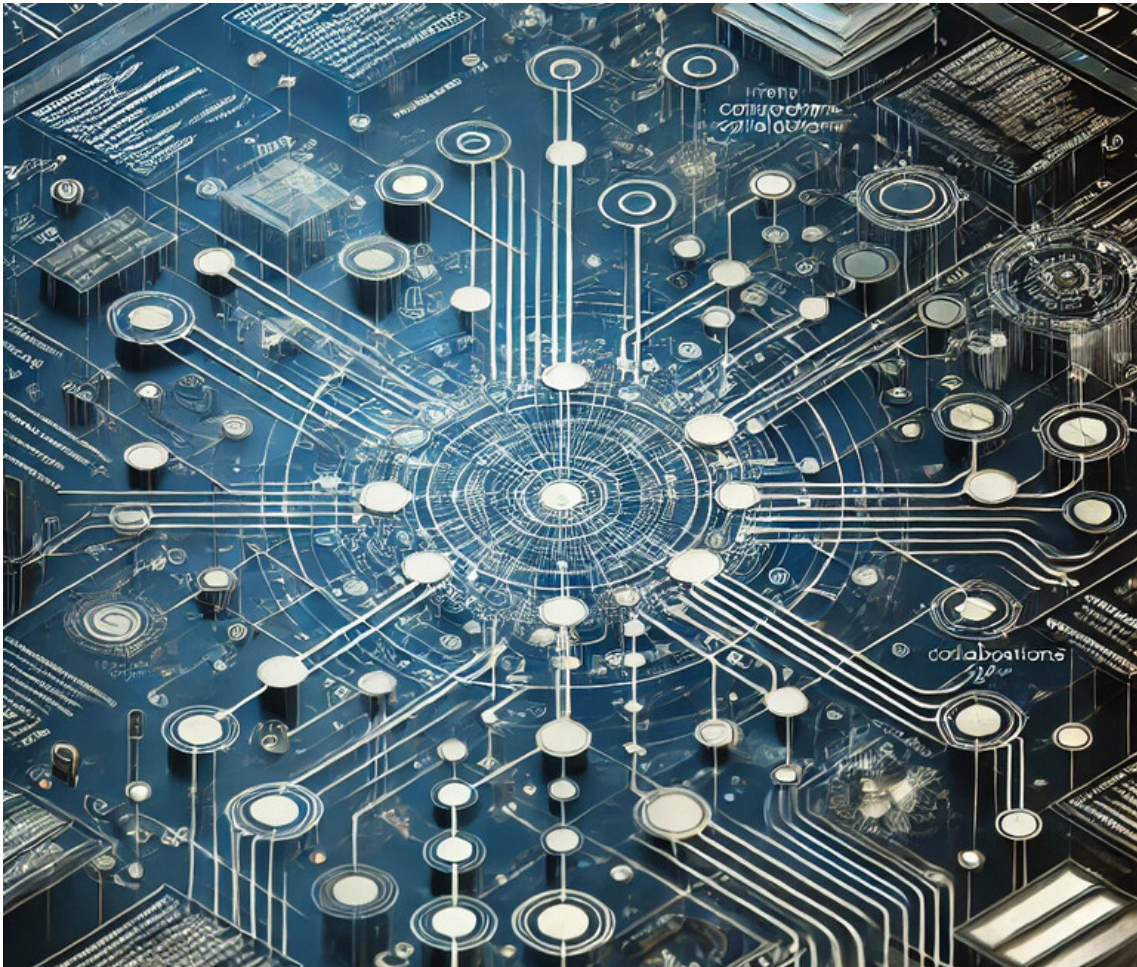
La última instrucción dentro del bucle construye la representación gráfica, evaluando en forma vectorial `F_numérica` para cada valor del array `x_array`. Luego completamos el gráfico agregando las etiquetas de los ejes y la leyenda que informa los colores asignados a cada n .

CELDA 66

```
import matplotlib.pyplot as plt
import numpy as np
Fx = sym.fourier_series(x, (x, -sym.pi, sym.pi))
x_array = np.linspace(-np.pi, np.pi, 500)
plt.plot(x_array, x_array, label=r'$f(x)$')
for n in [1, 3, 5, 7]:
    F_trunc = Fx.truncate(n=n)
    F_numérica = sym.lambdify(x, F_trunc, modules='numpy')
    plt.plot(x_array, F_numérica(x_array), label=f"Fourier n = {n}")

plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.legend()
plt.show()
```





14 | Versionado de código

En un universo ideal, cuando queremos construir un proyecto de software, escribimos los sistemas de punta a punta, sin errores, los terminamos, funcionan perfecto, y no hay nada más que hacer.

Los universos ideales no existen.

En realidad, escribimos y reescribimos muchas veces cada programa, cada documento, cada archivo. Los creamos, los renombramos, quedan obsoletos y los borramos. Tenemos ideas que apuntan para un lado y las exploramos, luego nos damos cuenta que la decisión no era la más acertada, a veces corregimos un poco, a veces cambiamos totalmente la dirección.

Muchas veces, nos damos cuenta luego de modificar fuertemente un archivo, o incluso luego de borrarlo, que el estado anterior del mismo no estaba tan mal, y queremos volver a eso. Las primeras veces que nos pasa no tenemos alternativa, ya perdimos "lo viejo", pero luego empezamos a tratar de mantener algunas versiones de los archivos, antes de modificarlos, y empezamos generar copias con numeros o fechas en los nombres.

Y si a la situación le agregamos que podemos tener colaboradores en el proyecto, modificando

Aplicación	Versión
git	2.27.0

algunos archivos sin que nos enteremos, o incluso modificando las mismas líneas de los mismos archivos que nosotros, todo se vuelve inmanejable.

Entran los Sistemas de Control de Versiones.

Un sistema de control de versiones es una herramienta que nos permite administrar los cambios que realizamos a los archivos de nuestros proyectos (no importa qué tipo de archivos sean), registrando qué cambios hizo qué persona en qué archivo, contribuyendo incluso a minimizar los conflictos por cambios simultáneos a las mismas líneas de un archivo.

En detalle, nos permite:

- grabar el estado del proyecto en un determinado momento ("sacar una foto" del estado de todos los archivos), pudiendo indicar anotaciones con respecto a ese estado
- revisar y comparar esos estados a través del tiempo, e incluso obtener el contenido de los archivos en esos distintos momentos
- trabajar en distintas ramas" de nuestro proyecto (manteniendo distintos cambios en paralelo, sin cruzarse); explicaremos este concepto más adelante [14.1](#)
- colaborar múltiples personas en el proyecto de forma distribuida

Notemos que algunas de estas características son provistas sólo en los sistemas modernos de versionado de código, como Git, Bazaar, o Mercurial. Sistemas más viejos, ya en desuso (Subversion, SourceSafe, CVS), no proveen todas estas características.

Debemos hacer énfasis en que es muy importante tener nuestros proyectos bajo algún sistema de versionado de código, incluso si trabajamos sólo nosotros en el proyecto. ¿Qué sistema elegir? Depende muchas veces del contexto y del grupo de trabajo, hoy en día el más difundido es Git, del cual hablaremos en detalle luego [14.2](#).

14.1. Ramas

¿Qué es una "rama" en este contexto? Vamos a enfocarnos un poco en este concepto, que es central para entender los sistemas de control de versiones modernos (pero al mismo tiempo tratando de no entrar en las especificidades de ninguna herramienta en particular).

Es una técnica de desarrollo que surgió a partir de usar los sistemas de control de versiones, ya que sin ellos es imposible administrar los cambios. Nos permite explorar distintos caminos evolutivos de nuestro sistema de forma simple e intuitiva.

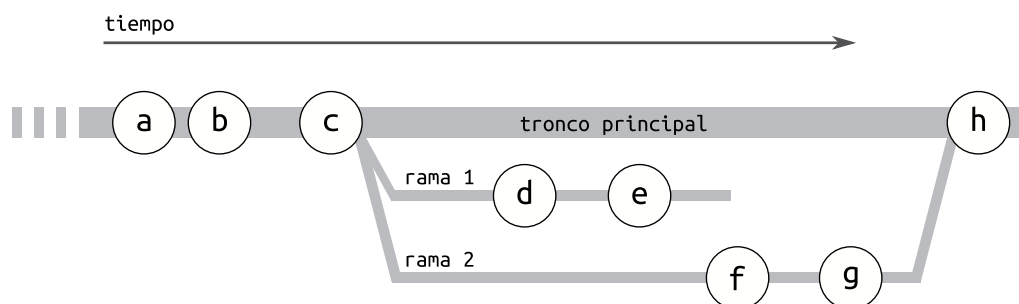
Supongamos que tenemos un proyecto con una veintena de archivos, estamos contentos con el estado actual, pero tenemos que implementar un cambio para el cual tenemos dos ideas distintas de cómo hacerlo. Decidimos ir por la primera idea y empezamos a implementarla, pero luego de unos días de trabajo nos damos cuenta que no es el mejor camino y decidimos probar la segunda idea.

El concepto de "rama" nos permite gestionar eso de forma muy limpia.

Tenemos que pensar en la evolución principal de nuestro proyecto como un "tronco" del cual salen estas ramas. En el momento en que queremos probar la primer idea, creamos una rama que bifurca el estado del proyecto. Trabajamos en esa rama, grabando los estados en distintos

momentos, hasta que como dijimos antes, nos damos cuenta que no es lo que queremos. Ahí volvemos al estado del proyecto como estaba en el tronco principal, efectivamente "deshaciendo" todos los cambios que hicimos en la rama (pero no los perdemos, ¡quedan allí guardados!), y abrimos una segunda rama. Trabajamos en esta segunda rama por varios días, guardando los distintos estados en distintos momentos, hasta que terminamos el cambio. En ese momento integramos estos cambios en el tronco principal, y allí ya podemos pensar en el próximo cambio (el cual implementaremos en otra rama, y así...).

Veamos este proceso representado en el siguiente gráfico:



Allí vemos que en el tronco principal de desarrollo habíamos guardado los estados a, b y c. Luego, decidimos abrir la primera rama, desarrollamos, guardamos los estados d y e, pero decidimos abandonar esa línea de desarrollo. Entonces volvemos a abrir una rama desde el mismo punto anterior, guardamos los estados f y g, y como nos parece que está todo terminado correctamente, juntamos esa rama con el tronco principal, y grabamos el estado h antes de ponernos a trabajar con el siguiente cambio.

14.2. Git

Git es el sistema de versionado de código más difundido en la actualidad.

Pensado originalmente para manejar proyectos distribuidos muy grandes, tuvo una gran difusión gracias al sitio Github que permite almacenar y gestionar los distintos proyectos, aunque hoy en día la mayoría de los sitios de manejo de proyectos (Launchpad, Gitlab, SourceForge, etc.) soportan git como mecanismo.

Aunque git es un sistema puramente distribuido, en la práctica el desarrollo se centraliza en una o más ramas principales en algunos de esos sitios, definidas en función de cada proyecto.

Las bibliografías oficiales de Git detallan un mecanismo de trabajo llamado "gitflow", donde recomiendan tener muchas ramas con funciones específicas, como se muestra en el siguiente gráfico:

Sin embargo en nuestra opinión y experiencia ese flujo de trabajo es demasiado complejo y se justifica solamente en proyectos y equipos muy grandes. En la gran mayoría de los casos es suficiente con un modelo mucho más simple de entender y ejecutar en el día a día, basado en las siguientes premisas:

- tenemos una rama principal, que siempre es "correcta"; es buena idea que no todos los colaboradores puedan escribir en esa rama principal, y la mayoría de los sitios de adminis-

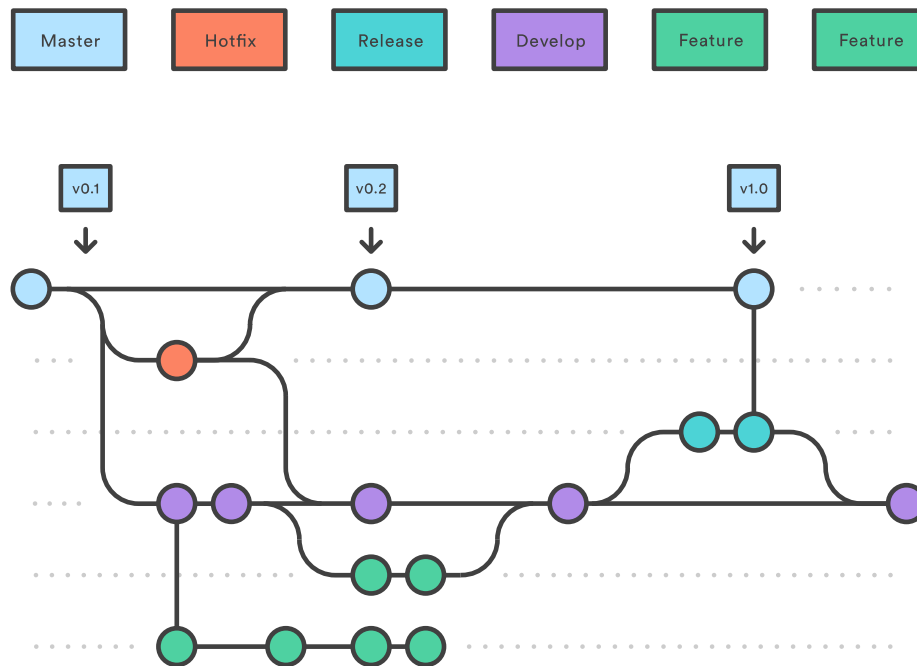


FIGURA 14.1: Flujo [recomendado por Atlassian](#) para trabajar con git

tracción de proyectos proveen herramientas para que sólo se escriba allí luego de ejecutar pruebas y verificaciones que podemos definir por proyecto.

- de esa rama principal sacamos ramas de trabajo para cada cambio que necesitemos hacer en el proyecto, y a esa rama principal volvemos con código terminado, revisado y correcto.
- marcamos puntos temporales en esa rama principal usando etiquetas (por ejemplo, cuando liberamos o ponemos en servicio el código)

Hablamos sobre la dinámica de trabajo con ramas arriba, cuando las presentamos, pero entre-mos más en detalle sobre cómo utilizarlas en función del modelo que estamos explicando ahora.

Entones, el “ciclo de vida” de una rama es:

- la creamos desde la rama principal para comenzar a trabajar en algún cambio (una nueva característica, solucionar algún problema, etc.)
- trabajamos en el código durante un tiempo (un rato, un día, una semana), muy posiblemente grabando el estado del código en varios momentos durante ese trabajo
- eventualmente, consideramos que el trabajo está hecho, entonces proponemos esa rama para que otros desarrolladores la evalúen y nos den feedback; mientras esto sucede podemos sacar una nueva rama desde la principal y comenzar otra tarea
- si el feedback que nos dan sobre nuestro trabajo implica que tenemos que realizar cambios en el mismo, volvemos a esa rama y hacemos las modificaciones necesarias y volvemos a proponer nuestro trabajo para que sea revisado

- en algún momento el equipo decidirá que el trabajo está completo de forma satisfactoria, entonces llevaremos todas las modificaciones de esa rama a la principal

En verdad, todo este flujo de trabajo es independiente de la herramienta. Se puede lograr con `git`, como explicaremos abajo, pero también con `bzr` (Bazaar), `hg` (Mercurial), o cualquier sistema de control de versiones más o menos moderno que permita trabajar con ramas de forma decente.

14.3. Comenzando con Git

En esta sección mostraremos lo básico de la utilización de `git` para lograr el flujo descripto arriba (y más, porque son operaciones genéricas que podemos combinar como querramos).

No es un tutorial completo sobre la herramienta, ni tampoco entra en detalles específicos sobre las estructuras internas de `git`. Para una utilización más avanzada de este sistema de control de versiones se deberán aprender otras opciones de la misma, y entender cómo funciona internamente en más detalle, pero creemos que dejando de lado muchas de esas especificidades permitimos un acercamiento más pedagógico a la utilización por primera vez de un control de versiones, sea `git` o el elegido en nuestro grupo de trabajo.

Entonces, el objetivo es bajar la barrera de entrada lo más posible para que todos podamos aprovechar esta herramienta, ya habrá tiempo para profundizar.

El primer paso es arrancar con un proyecto. Si el mismo ya está creado en otro lado, podemos clonarlo directamente:

```
$ git clone <la url que corresponda del proyecto>
```

Por ejemplo, para clonar el proyecto de este mismo libro, se puede hacer:

```
$ git clone https://github.com/facundobatista/libro-pyciencia.git
```

Si estamos arrancando un proyecto desde cero, a veces es mejor ir al sitio de administración que elijamos (Github, Launchpad, etc.), crear un repositorio vacío allí, y clonarlo, pero también podemos arrancar uno nuevito en nuestra máquina local:

```
$ mkdir miproyecto
$ cd miproyecto
$ git init -b trunk
```

En este procedimiento apareció una nueva palabra: “repositorio”. Un repositorio es el lugar donde el sistema de control de versiones almacena todos nuestros archivos más un montón de información extra para poder darnos las funcionalidades intrínsecas al control de versiones.

También cuando creamos el repositorio en nuestra máquina le dijimos a `git` que la rama principal se va a llamar `trunk`. Históricamente las ramas principales de Git se llamaron siempre `master`, pero en los últimos años surgieron y se afianzaron una serie de movimientos en el mundo que apuntan a evitar referenciar desigualdades históricas, principalmente raciales, reflejadas en muchos términos que se usan en la informática en el día a día, y en función de esto Github por

ejemplo empezó a llamar `main` a la rama principal de un nuevo proyecto. Tanto *main* (“principal”) como *trunk* (“tronco”) son alternativas interesantes, porque en un caso hace referencia a que es la rama principal, y en el otro es de donde salen las otras ramas.

Sólo para arrancar, creemos un archivo en nuestro nuevo proyecto. Con algún editor de texto grabemos un archivo `saludo.txt` con algún contenido. Luego de eso, si hacemos `git status` veremos que Git nos reporta que ese archivo está “sin seguimiento”, o sea que no está siendo administrado/controlado por el sistema de versionado.

Le indicamos entonces a Git que queremos este archivo en nuestro proyecto (lo cual sólo tenemos que hacer la primera vez que creamos o traemos un archivo al proyecto):

```
$ git add saludo.txt
```

Ahora `git status` nos mostrará que tenemos un nuevo archivo, pero con cambios a ser confirmados. Es hora de usar uno de las órdenes de Git más utilizadas, y que es la responsable de “grabar el estado” de nuestro proyecto, cómo explicábamos arriba:

```
$ git commit -m "Archivo de prueba."
```

El argumento `-m` nos permite adjuntar un mensaje al *commit* que acabamos de realizar. Si omitimos esa opción, Git tratará de abrir un editor de texto para que escribamos dicho mensaje, lo cual funcionará o no ya dependiendo de cómo tenemos configurado nuestro sistema (si tenemos un editor por default, por ejemplo, y si el mismo se puede utilizar de la manera que Git lo está intentando usar).

Un nuevo `git status` nos mostrará que “el árbol de trabajo está limpio”. Estamos listos para agregar nuevos cambios a nuestro proyecto.

Un detalle importante que omitimos mencionar hasta ahora es que no estamos trabajando realmente con ramas como explicamos arriba, sino que hicimos el `commit` directamente a la rama principal. No está del todo mal, pero cuando el proyecto empieza a crecer en complejidad (y queremos explorar distintos cambios al mismo tiempo) o tenemos un equipo de trabajo con varios colaboradores, si se trabaja de esta manera realmente se pierden muchas de las funcionalidades que el sistema de control de versiones tiene para ofrecernos.

Entonces, creemos una nueva rama, y posicionémosnos para trabajar en ella:

```
$ git branch cambio-1  
$ git checkout cambio-1
```

Podemos validar que estamos en dicha rama con `git status`. Ahora hagamos un cambio en nuestro único archivo (cambiémosle el contenido). Para grabar el nuevo estado del proyecto, primero deberíamos especificarle a git cuales archivos cambiados queremos registrar (con `add`) y luego efectivamente registrarlos (con `commit`), pero podemos hacer ambas acciones directamente:


```
$ git commit -am "Nuevo contenido de saludo."
```

Notemos que ahora al `commit` le pasamos dos opciones: `a` para agregar los cambios (por default, todos ellos), y `m` para el mensaje, que está escrito a continuación.

Realicemos el proceso de cambiar el archivo y volver a `commit`ear los cambios. Luego, para ver todo el historial de trabajo, usamos `git log`:

```
$ git log
commit 5a5f0369b89a6e5b0689f13d55671669c35ce9e4 (HEAD -> cambio-1)
Author: Facundo Batista <facundo@supermail.org>
Date: Mon Feb 8 17:05:01 2021 -0300

    Más cambios en el saludo.

commit 4804a799ddaa714e1f9e4f301f2a668364e4e908
Author: Facundo Batista <facundo@supermail.org>
Date: Mon Feb 8 16:56:34 2021 -0300

    Nuevo contenido de saludo.

commit a543a7406bca6ac08cbfda2bb291a7f25ec7cbc4 (trunk)
Author: Facundo Batista <facundo@supermail.org>
Date: Thu Feb 4 20:24:21 2021 -0300

    Archivo de prueba.
```

Como vemos, tenemos un bloque de información por cada *commit* que hicimos. Ese bloque de información tiene dos partes, un encabezado con información del *commit* y el mensaje que nosotros incluimos (que en estos casos es corto, pero podrían ser varias líneas o párrafos).

Ese encabezado también tiene varias partes. El autor y la fecha son obvios, enfoquémonos en la primer línea: vemos que arranca con `commit` y luego hay una serie de letras y números: ese es el *hash* del *commit*, un identificador único que nos permite trabajar sobre ese *commit* o hacer referencia al mismo.

Finalmente, tenemos entre paréntesis el dato sobre las ramas y donde estamos parados. Vemos que el primer *commit* que hicimos originalmente en *trunk* menciona esa rama entre paréntesis: esto es porque la rama en la que estamos parados ahora “salió desde ese punto” de la rama de donde nació (en este caso, justamente, *trunk*, pero piensen que se puede sacar una rama desde cualquier otra rama). En el último *commit*, además de la indicación de que es el último *commit* de la rama *cambio-1*, vemos que dice `HEAD`: es la indicación de que estamos viendo el estado del proyecto en ese *commit*.

Hagamos la prueba de movernos de *commit* para explorar el pasado:

```
$ git checkout 4804a799ddaa714e1f9e4f301f2a668364e4e908
```

Al ejecutar eso recibimos un mensaje de Git diciéndonos que estamos en modo *detached HEAD* (*HEAD* desacoplada), que significa que no estamos actualmente dentro de ninguna rama,

sino en ese commit específico. Podemos ver eso mismo haciendo `git log`, e incluso revisar el archivo de nuestro proyecto y ver que su contenido es el que “pertenece al pasado”.

Podemos volver a la rama en que estábamos haciendo `git checkout cambio-1`.

Y en este punto es donde dejamos de repetir que se puede utilizar tanto `git log` como `git status` para ver y entender cómo los distintos comandos afectan nuestro trabajo en el proyecto.

Ahora, siguiendo la idea del ejemplo original del capítulo, supongamos que nos arrepentimos de explorar este cambio en nuestro proyecto (digamos que entendimos que estuvo mal cambiar el contenido de `saludo.txt`, debíamos agregar un segundo archivo). Para arrancar limpios (y dejar intacto todo lo trabajado hasta ahora, que nos puede servir en el futuro), volvemos a `trunk` y arrancamos una nueva rama desde allí.

```
$ git checkout trunk
$ git checkout -b cambio-2
```

Atención al detalle de que, en lugar de crear la rama y después cambiar a ella, hacemos las dos operaciones en un sólo paso utilizando la opción `-b` del comando `checkout`.

Podemos ver las ramas que tenemos creadas en nuestro proyecto (Git nos indicará con un asterisco en cual estamos parados en este momento):

```
$ git branch
cambio-1
* cambio-2
trunk
```

Ahora podemos agregar otro archivo, incorporarlo al proyecto, hacer todos nuestros cambios, commitear varias veces, revisar cómo nos queda, hasta estar contentos con nuestro trabajo. Una mímica de todo este proceso sería lo siguiente:

```
$ echo "hola hola" > saludo-extra.txt
$ git add saludo-extra.txt
$ git commit -am "Segundo archivo"
[cambio-2 1f779d7] Segundo archivo
1 file changed, 1 insertion(+)
create mode 100644 saludo-extra.txt
$ echo "hola hola chau" > saludo-extra.txt
$ git commit -am "Con despedida"
[cambio-2 df39361] Con despedida
1 file changed, 1 insertion(+), 1 deletion(-)
$ echo "¡hola!" > saludo.txt
$ git commit -am "Énfasis en el saludo original"
[cambio-2 524dd0a] Énfasis en el saludo original
1 file changed, 1 insertion(+), 1 deletion(-)
$ git log
commit 524dd0afca1525f57658f81a0304e64112bbe977 (HEAD -> cambio-2)
Author: Facundo Batista <facundo@supermail.org>
Date: Tue Feb 9 11:30:12 2021 -0300

    Énfasis en el saludo original

commit df3936143e018d4e95d8149eddab9e29593765a7
```

```
Author: Facundo Batista <facundo@supermail.org>
Date:   Tue Feb 9 11:29:25 2021 -0300

Con despedida

commit 1f779d77cb748e7b3f95091f60ad9b50996233ac
Author: Facundo Batista <facundo@supermail.org>
Date:   Tue Feb 9 11:28:59 2021 -0300

Segundo archivo

commit a543a7406bca6ac08cbfda2bb291a7f25ec7cbc4 (trunk)
Author: Facundo Batista <facundo@supermail.org>
Date:   Thu Feb 4 20:24:21 2021 -0300

Archivo de prueba.
$
```

Vemos que esta segunda rama existe “en paralelo” con la otra, sin mezclarse la serie de cambios en cada caso.

¿Y ahora? Una vez que terminamos con nuestro cambio, es hora de llevarlo a la rama principal. Siempre es buena idea revisar cuál es la diferencia entre lo que tenemos y la rama a la que llevaremos los cambios:

```
$ git diff trunk
diff --git a/saludo-extra.txt b/saludo-extra.txt
new file mode 100644
index 00000000..e12be5d
--- /dev/null
+++ b/saludo-extra.txt
@@ -0,0 +1 @@
+hola hola chau
diff --git a/saludo.txt b/saludo.txt
index 5c1b149..8d1966f 100644
--- a/saludo.txt
+++ b/saludo.txt
@@ -1 +1 @@
-hola
+¡hola!
```

Es hora de llevar todo esto a la rama principal. En verdad no “llevamos” los cambios, sino que los “traemos”, por lo que primero tenemos que pararnos donde queremos los cambios, y luego indicamos que queremos “combinar” a la otra rama.

```
$ git checkout trunk
$ git merge cambio-2
```

Listo, tenemos todos los commits de cambio-2 también en trunk. Debemos mencionar que no siempre el *merge* funciona tan limpio, porque podemos tener conflictos, que aparecen cuando en distintas ramas se modificaron zonas similares de algún archivo.

Si en una rama tocamos un archivo y en la otra rama tocamos otro archivo, no hay problema. Incluso si tocamos el mismo archivo en las dos ramas, pero en zonas lo suficientemente separadas,

tampoco es un inconveniente. Pero si en las distintas ramas modificamos las mismas líneas del mismo archivo, Git no podrá combinar ese archivo, y tendremos un conflicto que habrá que resolver.

Esta y muchas otras situaciones se pueden presentar en el día a día, ya que el manejo central de un sistema de control de versiones es bastante directo y entendible, pero tiene muchas puntas afiladas y zonas oscuras que implicará un aprendizaje continuo al usar la herramienta, al menos al principio. Lo importante es que arranquemos a utilizarla con los comandos aprendidos en este capítulo, el resto irá viniendo solo.

Para profundizar les dejamos dos guías en sistemas de control de versiones en general y git en particular: el [Git Handbook](#) (parte de las Github Guides) es interesante y cortito, pero en inglés, y el [tutorial de Atlassian](#) es muy completo, entra en muchos detalles y esta en castellano, pero no trabaja con Github sino con Bitbucket, lo cual puede ser confuso en algún punto, pero interesante para aprender variado.



15 | Manipulación de datos

En este capítulo hablaremos en cómo manipular datos, independientemente del procesamiento de los mismos. Nuestro foco estará en cómo cargar o leer datos y herramientas para procesarlos, no en las operaciones que se realizarán sobre esos datos. En otras palabras y a modo de ejemplo, veremos cómo cargar una lista de datos en memoria como para calcular su promedio, pero no hablaremos del cálculo que se necesita para obtener el promedio.

La fuente natural de datos son archivos si los datos ya están generados en una etapa previa, o algún dispositivo de entrada si los datos se están generando en el momento de forma continua. Notablemente, cuando queremos descargar datos de internet nos podemos encontrar con la misma situación: si los datos son pocos los podemos leer y procesar todos en la misma acción, o también podemos irlos bajando y procesando de a poco.

En cualquier caso los datos pueden estar en forma de “texto” o ser “binarios”. En general

Módulo	Versión
h5py	3.11.0
netCDF4	1.7.1
numpy	1.26.4
pandas[all]	2.2.2
requests	2.32.3

[Código disponible](#)

(aunque no siempre) si los datos están en formato de texto estarán pensados para ser consumidos línea a línea, mientras que los datos binarios se procesarán byte a byte. Por razones pedagógicas nos enfocaremos en ambos casos por separado.

15.1. Leyendo archivos

Usamos el concepto “archivo” de forma muy genérica y al tiempo de forma muy poderosa, porque los sistemas informáticos utilizan ese concepto para proveer información desde varias fuentes. Entonces, trabajaremos como si fuesen archivos no sólo a archivos reales que tenemos en el dispositivo de almacenamiento de la computadora sino también a otras representaciones como dispositivos de entrada/salida de la computadora o respuestas de pedidos por la red.

En particular está el concepto de *file-like object*, “objetos que se parecen a archivos”, que son objetos que podemos tener como resultados de operaciones en nuestro programa y que por simplicidad pueden tratarse como archivos aunque realmente no lo sean.

La forma más simple de abrir un archivo es utilizando la función integrada `open`. Sus parámetros más comunes son: el nombre del archivo, el modo, y la codificación.

El nombre del archivo (o el *path* completo) apunta al archivo en disco, que puede existir o no según cómo vayamos a abrirlo.

El modo es justamente la forma en que vamos a abrir el archivo; es una cadena de varias letras cada una con un significado específico que se pueden combinar de algunas maneras puntuales.

- `'r'`: abre para solamente leer
- `'w'`: abre para solamente escribir, truncando el archivo a cero si es que ya existía
- `'x'`: abre para solamente escribir, pero exclusivamente creando el archivo (falla si el archivo ya existía)
- `'a'`: abre para solamente escribir, agregando el nuevo contenido al final
- `'b'`: modo binario
- `'t'`: modo texto
- `'+'`: abre para actualizar (leer y escribir)

En general construimos el modo indicando lectura/escritura y binario/texto, por ejemplo `'wb'` para escribir un archivo en binario, `'rt'` para leer un archivo en texto, etc. El `'+'` se usa relativamente poco, es para aquellos casos en que queremos escribir y leer al mismo tiempo, y se combina con las otras letras de lectura/escritura más para indicar qué sucedería con un archivo que ya existe o no.

CELDA 1 DE ARCHIVOS.IPYNB

```
# abrimos para escritura, escribimos, cerramos
fh = open("/tmp/archivoprueba", "wb")
fh.write(b"holá")
fh.close()
```



```
# abrimos para lectura, leemos, cerramos
fh = open("/tmp/archivoprueba", "rb")
print(fh.read())
fh.close()
```

```
b'holá'
```

CELDA 02

```
# abrimos para *agregar* al archivo anterior, escribimos, cerramos
fh = open("/tmp/archivoprueba", "ab")
fh.write(b" mundo")
fh.close()

# vemos que en el archivo tenemos lo anterior y lo nuevo
fh = open("/tmp/archivoprueba", "rb")
print(repr(fh.read()))
fh.close()
```

```
b'holá mundo'
```

En realidad los archivos siempre son binarios; sólo pueden tener bytes dentro porque los medios de almacenamiento guardan bytes. Abrir el archivo en modo “texto” es solamente una forma cómoda de trabajar con texto Unicode en nuestros programas sin pasar por una etapa donde el contenido está en binario: entonces en el programa siempre manejaremos texto y la conversión a o desde bytes la hará Python automáticamente, y es por eso que en estos casos hay que indicarle la codificación que usará para codificar o decodificar (en realidad es opcional, pero el default depende del sistema, entonces siempre es recomendable que esté indicado de forma explícita).

CELDA 03

```
# notar como para texto especificamos la codificación
fh = open("/tmp/archivoprueba", "wt", encoding="utf8")
fh.write("moño")
fh.close()

# y aunque el archivo realmente tiene bytes, accedemos como texto
fh = open("/tmp/archivoprueba", "rt", encoding="utf8")
print(repr(fh.read()))
fh.close()
```

```
'moño'
```

CELDA 04

```
# si accedemos al archivo en modo binario vemos lo que realmente tiene
fh = open("/tmp/archivoprueba", "rb")
data = fh.read()
fh.close()
print(repr(data))
```

```
b'mo\xc3\xb1o'
```

CELDA 05

```
# obviamente podemos decodificarlo a mano, pero la idea es
# que si trabajamos con texto nunca manejarlo como bytes
print(repr(data.decode("utf8")))

-----

'moño'
```

Hay algunas opciones más para lidiar con distintas especificidades (que surgen especialmente en distintas plataformas) pero ya con estas podemos comenzar a trabajar. Para profundizar lo mejor es [su documentación](#).

La metodología de abrir y cerrar el archivo como mostramos antes es muy útil cuando el archivo se utiliza por tiempo indefinido o estructuramos algún objeto alrededor de su uso y los dos ciclos de vida están atados. Pero en general se usa otra técnica, ya que es muy común cerrar el archivo lo antes posible en el mismo bloque de código que se abrió; recordemos que mientras el archivo está abierto se está utilizando un recurso del sistema operativo, que hay límites con respecto a la cantidad de archivos abiertos, etc., por lo que siempre conviene prestar atención en cerrar el archivo lo antes posible cuando ya no lo utilizamos más.

Para simplificar eso Python nos permite usar un administrador de contexto [5.2](#) para interactuar con archivos:

CELDA 06

```
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:
    print(f"Estado dentro del contexto: closed={fh.closed}")
    data = fh.read()
print(f"Estado luego del contexto: closed={fh.closed}")
print("Leído:", data)

-----

Estado dentro del contexto: closed=False
Estado luego del contexto: closed=True
Leído: moño
```

Vemos como usando el **with** tenemos al *file handler* abierto dentro del contexto y automáticamente cerrado luego de salir del mismo: de esta manera es clara la zona donde trabajamos con el archivo, con la seguridad que siempre se cerrará al archivo al salir de esa zona (incluso en el caso de una excepción, lo cual no sucedía en los ejemplos anteriores).

15.2. Trabajando con datos en formato textual

Cuando mencionamos archivos de forma genérica en [15.1](#) vimos la operatoria básica: abrir, cerrar, escribir, leer; entraremos en esta sección en detalles particulares al trabajar con texto.

Habiendo dicho eso, volveremos a mostrar aquí cómo escribir texto en un archivo aunque sea similar a lo que ya vimos. En el siguiente ejemplo bajamos un texto que tenemos como lista de cadenas a un archivo (respetando las líneas, para lo cual tenemos que agregar manualmente los saltos de línea).

CELDA 1 DE TEXTOS.IPYNB

```
lines = [  
    "En este film velado en blanca noche",  
    "El hijo tenaz de tu enemigo",  
    "El muy verdugo cena distinguido",  
    "Una noche de cristal que se hace añicos",  
]  
  
with open("/tmp/archivoprueba", "wt", encoding="utf8") as fh:  
    for line in lines:  
        fh.write(line + "\n") # notar el agregado del salto de linea  
  
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:  
    print(fh.read())
```

```
En este film velado en blanca noche  
El hijo tenaz de tu enemigo  
El muy verdugo cena distinguido  
Una noche de cristal que se hace añicos
```

Para leer, por otro lado, aprovechamos la característica de que el *file handler* que devuelve `open` es iterable y nos devolverá línea por línea del archivo, lo cual es al mismo tiempo la forma más práctica y más eficiente de recorrer las líneas de un archivo:

CELDA 02

```
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:  
    for line in fh:  
        print(len(line), repr(line))
```

```
36 'En este film velado en blanca noche\n'  
28 'El hijo tenaz de tu enemigo\n'  
32 'El muy verdugo cena distinguido\n'  
40 'Una noche de cristal que se hace añicos\n'
```

Hay otros métodos para trabajar con el *file handler*, por ejemplo `.readlines` que leerá todas las líneas del archivo almacenándolas en una lista, pero en general siempre será mejor leer línea por línea y en todo caso armar una lista con el resultado de un primer procesamiento de eso archivo.

Por supuesto, cuando trabajamos con texto también podemos leer carácter a carácter (similar a como vamos a ver cuando trabajemos con binarios, más adelante), buscando el salto de línea y armando las líneas a mano, pero en la casi absoluta cantidad de veces es suficiente la manera ya vista.

La iteración de cada línea ni siquiera tiene que ser explícita, ya que muchas funciones de Python trabajan iterando naturalmente. En el siguiente ejemplo vemos una simple forma de obtener la línea más larga del archivo que recién creamos, usando `max` y aprovechando que se le puede indicar cual es indicador de orden (en nuestro caso el largo de cada línea, que es lo que devolverá la función `len` en cada caso):

CELDA 03

```
with open("/tmp/archivoprueba", "rt", encoding="utf8") as fh:
    longest = max(fh, key=len)
    print("Más larga:", repr(longest))
```

```
Más larga: 'Una noche de cristal que se hace añicos\n'
```

Veamos un ejemplo más complejo: sacar el top 5 de mantenedores de paquetes de Debian. La fuente de datos será un archivo público, [la lista de maintainers](#), que en cada línea indica el paquete, uno o más espacios, y el nombre y mail de su mantenedor (para más detalles de la estructura que procesa el código ver la descripción en el Jupyter Notebook `textos.ipynb` de este capítulo).

CELDA 04

```
from collections import Counter

names_counter = Counter()

with open("Maintainers", "rt", encoding="utf8") as fh:
    for line in fh:
        # separamos por blancos, pero como mucho una vez para tener intacto
        # el texto de todos los mantenedores
        package, maintainers = line.strip().split(maxsplit=1)

        # separamos por mayor-que y coma, para no cortar por la coma entre los nombres
        for maintainer in maintainers.split(">,"):
            if "<" in maintainer:
                # sacamos el resto del mail
                name, _ = maintainer.split("<")
            else:
                # el nombre *es* el mail
                name = maintainer.replace("<", "").replace(">", "")

            # contamos
            name = name.strip()
            names_counter[name] += 1

for name, quantity in names_counter.most_common(5):
    print(f"{quantity:3d} {name}")
```

```
9002 Debian Kernel Team
7392 Debian GCC Maintainers
6544 Debian Rust Maintainers
5614 Debian Python Team
5010 Debian Haskell Group
```

Por supuesto, la estructura que acabamos de procesar es muy específica. Hay muchos formatos textuales que son más estándar, veamos algunos casos.

15.2.1. Tres formatos textuales muy comunes

Mencionaremos particularmente tres formatos que se utilizan mucho en los sistemas informáticos, CSV, JSON y XML.

15.2.1.1. CSV

Los archivos CSV, por *Comma-Separated Values*, o “valores separados por coma” es un tipo de archivo muy utilizado para guardar datos en forma de tabla.

Aunque no tiene prácticamente soporte para manejar distintos tipos de datos (la conversión a Unicode de los textos dependerá de cómo abramos el archivo, y sólo convertirá algunos campos numéricos en algún caso, como veremos luego), pero su simplicidad y facilidad de uso hizo que sea uno de los formatos más utilizados desde siempre.

Su estructura, a priori, es muy sencilla: cada línea es una fila, y cada columna se separa de la otra por comas. Pero, en la práctica, es todo mucho más complicado, porque no hay un estándar fuerte sobre distintos aspectos como qué codificación se usa para guardar textos Unicode, cómo separar las columnas cuando el punto decimal en los números es la coma, o cuando los textos usan comillas o incluso si tienen saltos de línea dentro.

Entonces siempre recomendamos evitar caer en la tentación y procesar un archivo CSV “a mano” (leyendo línea por línea asumiendo que sólo hay saltos de línea al final de cada fila y directamente separar por comas para obtener las columnas). El procedimiento correcto es utilizar el módulo `csv` provisto por la Biblioteca Estándar de Python.

Veamos esto con un ejemplo. Construyamos un caso particularmente complicado donde tenemos cadenas Unicode, un texto vacío, y otros que tienen caracteres que a priori serían “de control” en el formato: comillas dobles y coma (y tilde y punto y coma, por las dudas), y un salto de línea.

CELDA 05

```
import csv

original = [
    ["algún texto", 234, "", 'con comilla ("')'],
    ["con tilde (')", 14.3, "coma y punto y coma (,;)", "con newline (\n)"],
]

with open("/tmp/prueba.csv", "wt") as fh:
    writer = csv.writer(fh)
    for row in original:
        writer.writerow(row)

with open("/tmp/prueba.csv", "rt") as fh:
    print(fh.read())
```

```
algún texto,234,,,"con comilla ("")"
con tilde ('),14.3,"coma y punto y coma (,;)",,"con newline (
)"
```

Grabamos el archivo en formato CSV usando el módulo ya mencionado para asegurarnos que el formato final sea el correcto, y luego lo abrimos y mostramos sin procesar. Allí vemos que aunque grabamos dos filas, tenemos efectivamente tres líneas y otros detalles que nos complicarían el procesamiento “a mano”. Veamos entonces qué sucede si lo procesamos “mal”:

CELDA 06

```
with open("/tmp/prueba.csv", "rt") as fh:
    for line in fh:
        print(line.split(","))
```

```
['algún texto', '234', '', '"con comilla (")"\n']
["con tilde (')", '14.3', '"coma y punto y coma (, ;)"', '"con newline (\n)'\n']
['')\n']
```

Por supuesto, si utilizamos el módulo `csv` para procesar el archivo volvemos a tener cada uno de los textos de forma correcta:

CELDA 07

```
with open("/tmp/prueba.csv", "rt") as fh:
    reader = csv.reader(fh)
    for row in reader:
        print(row)
```

```
['algún texto', '234', '', 'con comilla (")']
["con tilde (')", '14.3', 'coma y punto y coma (, ;)', 'con newline (\n)']
```

Decíamos arriba que los archivos CSV pueden tener distintos formatos. El módulo `csv` trae soporte para interpretar los “dialectos” más comunes (`excel`, `unix`, etc.), y de esta manera podemos tener control sobre qué se usa para separar columnas, con qué se envuelven los campos que tienen caracteres especiales, cómo se escapan a su vez esos caracteres, el fin de línea, etc.

Podemos elegir cuál dialecto preferimos, armar uno nosotros, o incluso pasar modificadores al escribir o leer el CSV para afectar puntualmente algún comportamiento. En el siguiente ejemplo indicamos que al escribir se agreguen comillas alrededor de todos los campos no numéricos, entonces luego al leer el resto de los campos serán convertidos a `float` directamente:

CELDA 08

```
with open("/tmp/prueba.csv", "wt") as fh:
    writer = csv.writer(fh, quoting=csv.QUOTE_NONNUMERIC)
    for row in original:
        writer.writerow(row)
```

```
with open("/tmp/prueba.csv", "rt") as fh:
    reader = csv.reader(fh, quoting=csv.QUOTE_NONNUMERIC)
    for row in reader:
        print(row)
```

```
['algún texto', 234.0, '', 'con comilla (")']
["con tilde (')", 14.3, 'coma y punto y coma (, ;)', 'con newline (\n)']
```

Hasta este momento estamos mostrando fila por fila de datos sin acceder a una columna en particular. Habrán notado que cada fila termina siendo una lista, por lo que para acceder a cada dato debemos hacerlo posicionalmente:

CELDA 09
<pre>with open("/tmp/prueba.csv", "rt") as fh: reader = csv.reader(fh, quoting=csv.QUOTE_NONNUMERIC) for row in reader: print(f"Cadena {row[3]!r} con valor {row[1]:.2f}")</pre>
Cadena 'con comilla (")' con valor 234.00 Cadena 'con newline (\n)' con valor 14.30

Podemos mejorar la legibilidad de nuestro código si en vez de usar el *reader* común usamos *DictReader*, al cual le pasamos el nombre de los campos y en vez de darnos una tupla nos dará un diccionario:

CELDA 10
<pre>fieldnames = ["column1", "value", "column3", "text"] with open("/tmp/prueba.csv", "rt") as fh: reader = csv.DictReader(fh, fieldnames=fieldnames, quoting=csv.QUOTE_NONNUMERIC) for row in reader: print("Cadena {text!r} con valor {value:.2f}".format_map(row))</pre>
Cadena 'con comilla (")' con valor 234.00 Cadena 'con newline (\n)' con valor 14.30

Incluso *DictReader* soporta tomar el nombre de las columnas de la primer fila del archivo; lo hace automáticamente si evitamos pasarle el parámetro *fieldnames*.

15.2.1.2. JSON

La ubicuidad del lenguaje JavaScript en la gran mayoría de navegadores web hizo que un formato pensado para ser evaluado de forma sencilla por ese lenguaje se hiciera moneda común a la hora de intercambiar información entre distintas partes.

Este formato es un subconjunto de la notación de JavaScript, llamado JSON por *JavaScript Object Notation*, “notación de objeto de JavaScript”, y es a la vez flexible (porque soporta los tipos de datos más utilizados por los distintos lenguajes) y simple (porque es una cadena de caracteres mayormente legible y también soporta solamente unos pocos tipos de datos), lo cual impactó fuertemente en su interoperabilidad y por lo tanto en su adopción en el tiempo.

Los siguientes son los tipos de datos soportados por JSON, con la aclaración de a qué tipo corresponden en Python:

- Números enteros (*int*) y con parte fraccional separada por punto (*float*), en ambos casos positivos y negativos
- Cadenas de texto Unicode (*str*)
- Booleanos (*True* y *False*)
- Null (*None*)
- Arreglos (o *arrays*) con cualquier otro tipo de dato soportado dentro (*list*)

- Objetos o colecciones no ordenadas de pares clave-valor, donde la clave tiene que ser una cadena sí o sí y el valor puede ser cualquier tipo (`dict`, con el detalle de restricción del tipo de la clave)

Veamos algunos ejemplos codificando estructuras de Python al formato JSON:

CELDA 11
<pre>import json structures = ["una cadena unicode áÑ", # eso 35, # un entero 2.3, # punto flotante [1, "hola", [3, 2], 'comilla "'], # lista con cosas (inclusive otra lista) (1, 2, 3, True, False, None), # tupla {'a': 3, 'b': [], 'c': ''} # diccionario, ojo a las claves siempre string] for structure in structures: encoded = json.dumps(structure) print(f"Python: {structure!r} - JSON: {encoded!r}")</pre> <pre>Python: 'una cadena unicode áÑ' - JSON: '"una cadena unicode \\u00e1\\u00d1"' Python: 35 - JSON: '35' Python: 2.3 - JSON: '2.3' Python: [1, 'hola', [3, 2], 'comilla "'] - JSON: '[1, "hola", [3, 2], "comilla \\\"]' Python: (1, 2, 3, True, False, None) - JSON: '[1, 2, 3, true, false, null]' Python: {'a': 3, 'b': [], 'c': ''} - JSON: '{"a": 3, "b": [], "c": ""}'</pre>

Podemos ver en el ejemplo algunos detalles que vale la pena mencionar. Las cadenas siempre se rodean con comilla doble (cuando la cadena en sí tiene comilla doble se escapa con barra invertida) y cuando tienen caracteres no-ASCII los mismos se codifican a UTF-16 y se incluyen los valores escapados de los bytes. `True`, `False` y `None` se escriben casi igual en JSON, lo que sumado a muchas otras similitudes hacen que sea un formato muy legible si estamos acostumbrados a Python.

Por último, tanto la tupla como la lista se transforman en el mismo tipo de *array* (que usa como delimitador los mismos corchetes que Python para las listas). Un efecto secundario de esto es que podemos inadvertidamente “perder” las tuplas al pasar por JSON:

CELDA 12
<pre>the_tuple = (1, 2, 3, True, False, None) print("Original:", repr(the_tuple)) encoded = json.dumps(the_tuple) print("El JSON:", repr(encoded)) decoded = json.loads(encoded) print("Volvimos:", repr(decoded))</pre> <pre>Original: (1, 2, 3, True, False, None) El JSON: '[1, 2, 3, true, false, null]' Volvimos: [1, 2, 3, True, False, None]</pre>

Hasta ahora convertimos de Python a JSON como cadena de texto (y volvimos de esa cadena

de texto a Python) usando las funciones `dumps` y `loads` del módulo `json`. La mayoría de las veces no nos interesa obtener esa cadena *per se* sino que su destino final es un archivo o enviarla a través de la red; en esos casos podemos grabar o leer directamente de un archivo (o de un objeto *file like*) con las funciones `dump` y `load`.

CELDA 13

```
somestuff = dict(answer=42, message="hola campeón", sequence=[1, 2, 3.14])
with open("/tmp/prueba.json", "wt", encoding="utf8") as fh:
    json.dump(somestuff, fh)

with open("/tmp/prueba.json", "rt", encoding="utf8") as fh:
    loaded = json.load(fh)
print(loaded)

{'answer': 42, 'message': 'hola campeón', 'sequence': [1, 2, 3.14]}
```

También podemos trabajar con otros objetos que no sean realmente archivos pero que expongan la misma interfaz (los llamamos *file-like objects*). Un caso típico es la respuesta de un request HTTP, que expone el método `read` sobre el cual podemos operar.

En el siguiente ejemplo usamos la biblioteca `requests` para obtener la temperatura actual del Museo de Ciencias Naturales de La Plata, Buenos Aires, Argentina:

CELDA 14

```
import requests

forecast_url = "https://api.open-meteo.com/v1/forecast"

# pedimos el clima actual en las coordenadas del museo
lat, long = -34.9153435, -57.9331994
params = dict(latitude=lat, longitude=long, current_weather=True)

# para que no venga comprimido
headers = {"Accept-Encoding": "identity"}

resp = requests.get(forecast_url, stream=True, headers=headers, params=params)
print(resp.headers["Content-Type"])
print(resp.raw)

application/json; charset=utf-8
<urllib3.response.HTTPResponse object at 0x7352b41ff580>
```

Cuando hacemos el `get` con `stream=True` la biblioteca no va a leer toda la respuesta inmediatamente, solamente leerá las cabeceras, y vemos también que tenemos el atributo `raw` que es el objeto-respuesta a más bajo nivel, la cual usamos luego para leer directamente decodificando el JSON de la respuesta:

CELDA 15

```
data = json.load(resp.raw)
weather = data["current_weather"]
weather
```

```
{'time': '2024-09-24T12:00',
 'interval': 900,
 'temperature': 16.0,
 'windspeed': 23.3,
 'winddirection': 107,
 'is_day': 1,
 'weathercode': 0}
```

JSON soporta todos los tipos de datos que mencionamos antes pero si lo usamos con continuidad notaremos que siempre en algún momento necesitamos enviar en este formato algún tipo de dato no soportado. Incluso en el ejemplo que acabamos de ver tenemos un caso de esto: la “fecha hora” es una cadena y no un objeto `datetime`.

Para este ejemplo alcanza con convertir esa cadena al objeto correspondiente, pero si va a ser recurrente podemos armar un *decoder* específico para nuestras necesidades:

CELDA 16

```
import datetime

def decoder(json_dict):
    for key, value in json_dict.items():
        try:
            json_dict[key] = datetime.datetime.fromisoformat(value)
        except (ValueError, TypeError):
            pass
    return json_dict

resp = requests.get(forecast_url, stream=True, headers=headers, params=params)
data = json.load(resp.raw, object_hook=decoder)
weather = data["current_weather"]
weather

{'time': datetime.datetime(2024, 9, 24, 12, 0),
 'interval': 900,
 'temperature': 16.0,
 'windspeed': 23.3,
 'winddirection': 107,
 'is_day': 1,
 'weathercode': 0}
```

Vemos como podemos indicar una función al momento de decodificar que recibirá todos los “objetos” a nivel JSON (que en Python conceptualizamos como diccionarios). No hay manera de recibir las cadenas directamente, pero con recibir todo el objeto nos alcanza, podemos iterar las claves y valores del mismo y convertir a fecha-hora lo que se pueda.

Para intercambios de datos más complejos y estructurados es normal utilizar claves con nombres particulares que indican el tipo de objeto que está representado en ese JSON e instanciar un objeto Python de ese tipo con esos atributos.

15.2.1.3. XML

El XML, por sus siglas en inglés *eXtensible Markup Language* (“lenguaje de marcado extensible”) nos permite codificar datos de una manera extremadamente flexible y compatible, ya que

se pueden tener reglas para cualquier tipo de dato y al mismo tiempo no está atado a ningún lenguaje de programación o plataforma.

Veremos que su estructura es parecida a la del mucho más conocido HTML. Ambos provienen de un lenguaje adoptado por la ISO llamado SGML (*Standard Generalized Markup Language*, una evolución del GML creado por IBM en la década de 1970).

En comparación con los formatos que vimos anteriormente, XML es por lejos el más flexible, ya que permite codificar no sólo la información que necesitamos almacenar o transmitir sino también todo tipo de metadata necesaria para interpretar esa información. Esta flexibilidad tiene el costo asociado de la complejidad del formato y su muy baja legibilidad.

Su estructura se basa en una estructura de árbol donde cada nodo puede tener un dato u otros nodos, además de “marcas” que proveen información extra sobre esos datos o nodos.

Para visualizar más fácil esto armemos en formato XML lo que podría ser la respuesta del servicio que usamos antes para saber el clima de una ciudad (escribiendo sólo la parte que usamos de clima actual, por brevedad):

```
<?xml version="1.0"?>
<data>
  <current_weather>
    <temperature unit="celsius">20.3</temperature>
    <windspeed unit="knots">10.7</windspeed>
    <winddirection>132.0</winddirection>
    <weathercode>3</weathercode>
    <is_day>1</is_day>
    <time format="ISO">2023-04-22T15:00</time>
  </current_weather>
  <etc>
    ...
  </etc>
</data>
```

Allí vemos la estructura de árbol que mencionamos, así como también información extra asociada a cada dato, útil tanto para parsear el texto del dato (como en el caso de la fecha-hora) como para también potencialmente mostrarle esa información al usuario (como la unidad de temperatura).

Veamos un caso más real: el XML resultante al exportar la página de Argentina de Wikipedia en castellano:



[requests](#) - Simplifica las solicitudes HTTP en Python, permitiendo enviar solicitudes con métodos como GET y POST de manera sencilla.

CELDA 17

```
resp = requests.get("https://es.wikipedia.org/wiki/Especial:Exportar/Argentina")
fullxml = resp.text
len(fullxml)

118595
```

No la mostramos toda (¡117 mil bytes!) pero usemos el módulo `textwrap` para ver de forma sencilla y prolija la primera parte del contenido:

CELDA 18

```
import textwrap

for chunk in textwrap.wrap(fullxml, width=80, max_lines=10):
    print(chunk)

<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.11/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.mediawiki.org/xml/export-0.11/
http://www.mediawiki.org/xml/export-0.11.xsd" version="0.11" xml:lang="es">
<siteinfo>      <sitename>Wikipedia</sitename>      <dbname>eswiki</dbname>
<base>https://es.wikipedia.org/wiki/Wikipedia:Portada</base>
<generator>MediaWiki 1.43.0-wmf.23</generator>      <case>first-letter</case>
<namespaces>      <namespace key="-2" case="first-letter">Medio</namespace>
<namespace key="-1" case="first-letter">Especial</namespace>      <namespace
key="0" case="first-letter" />      <namespace key="1" case="first- [...]
```



Una buena manera de ver un XML completo bien estructurado y con la facilidad extra de ocultar/mostrar distintas partes del árbol es abriendo ese XML en un navegador web; prueben esto abriendo la URL del XML que acabamos de descargar.

Pasemos a procesar este XML.

En Python tenemos distintas herramientas para trabajar con este formato. Veamos los distintos casos todos sobre el mismo contenido que acabamos de descargar, siempre con el objetivo de encontrar el título de la página, el identificador y timestamp de esta revisión en particular, y el largo del texto en sí de la página.

La primer biblioteca que usaremos es `ElementTree`, que nos da una forma simple y eficiente para parsear XMLs, ya que mapea directamente el concepto de “árbol de nodos” de este formato con dos tipos de objetos: el `ElementTree` que representa el documento XML completo, y `Element` que nos permite trabajar con cada nodo.

CELDA 19

```
from xml.etree import ElementTree

node = ElementTree.fromstring(fullxml)
print("El primer nodo:", node)
```

```

print("Su tag:", node.tag)
print("Atributos:")
for key, value in node.attrib.items():
    print(f"    {key!r}: {value!r}")
print("Texto útil no tiene:", repr(node.text))
print("Pero sí nodos hijos:")
for child in node:
    print("    ", child)

```

```

El primer nodo: <Element '{http://www.mediawiki.org/xml/export-0.11/}mediawiki' at 0x7352af5dd210>
Su tag: {http://www.mediawiki.org/xml/export-0.11/}mediawiki
Atributos:
  '{http://www.w3.org/2001/XMLSchema-instance}schemaLocation':
    4 '{http://www.mediawiki.org/xml/export-0.11/ http://www.mediawiki.org/xml/export-0.11.xsd'
  'version': '0.11'
  '{http://www.w3.org/XML/1998/namespace}lang': 'es'
Texto útil no tiene: '\n '
Pero sí nodos hijos:
  <Element '{http://www.mediawiki.org/xml/export-0.11/}siteinfo' at 0x7352af5df380>
  <Element '{http://www.mediawiki.org/xml/export-0.11/}page' at 0x7352af5b9850>

```

Como ya tenemos el XML completo en una cadena lo parseamos de allí (ElementTree, como el resto de las bibliotecas que veremos aquí nos permite tanto leer el contenido desde una cadena como de un archivo). Nos devuelve directamente el primer nodo del documento, que cómo es una estructura de árbol tendemos a llamar “raíz”, pero atención que en este caso tenemos más de un nodo en el primer nivel.

Vemos que su tag es {http://www.mediawiki.org/xml/export-0.10/}mediawiki, no simplemente mediawiki. Esto es porque ElementTree nos muestra siempre (y obliga a trabajar) cada nombre en su espacio de nombre. Los espacios de nombre son para identificar unívocamente a qué documento pertenece un nombre; sería muy útil si mezclamos este XML con otro documento que también tiene los tags mediawiki o page, por ejemplo, pero para manejar un sólo documento puede ser un poco engorroso.

Más allá de eso, ElementTree nos deja acceder de forma bastante directa a toda la información del nodo: tenemos sus atributos en .attrib, el texto en .text, y lo iteramos directamente para ir a por los hijos. Y también tenemos el método .find para buscar un nodo hijo en particular, usémoslo para recorrer el árbol y obtener la información que queríamos:

```

page = node.find("{http://www.mediawiki.org/xml/export-0.11/}page")
title = page.find("{http://www.mediawiki.org/xml/export-0.11/}title")
revision = page.find("{http://www.mediawiki.org/xml/export-0.11/}revision")
rev_id = revision.find("{http://www.mediawiki.org/xml/export-0.11/}id")
rev_tstamp = revision.find("{http://www.mediawiki.org/xml/export-0.11/}timestamp")
rev_text = revision.find("{http://www.mediawiki.org/xml/export-0.11/}text")
textlen = rev_text.attrib["bytes"]
print(f"Página {title.text!r} r.{rev_id.text} ({rev_tstamp.text}) largo={textlen}")

```

```

Página 'Argentina' r.162365205 (2024-09-09T23:59:56Z) largo=115597

```

La segunda herramienta es muy parecida a la anterior pero su API es estándar, definida por la *World Wide Web Consortium (W3C)*. De allí viene el nombre del módulo, DOM, por *Document*

Object Model.

Cuando parseamos el XML nos devuelve un “documento”, al cual le pedimos el primer nodo hijo:

CELDA 21

```

from xml.dom.minidom import parseString

dom = parseString(fullxml)
node = dom.firstChild

print("El primer nodo:", node)
print("Su tag:", node.nodeName)
print("Atributos:")
for key, value in node.attributes.items():
    print(f"    {key!r}: {value!r}")
print('Los nodos "hijos":')
for child in node.childNodes:
    print("    ", child)

```

El primer nodo: <DOM Element: mediawiki at 0x7352b425fad0>
 Su tag: mediawiki
 Atributos:
 'xmlns': 'http://www.mediawiki.org/xml/export-0.11/'
 'xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance'
 'xsi:schemaLocation': 'http://www.mediawiki.org/xml/export-0.11/
 L http://www.mediawiki.org/xml/export-0.11.xsd'
 'version': '0.11'
 'xml:lang': 'es'
 Los nodos "hijos":
 <DOM Text node "'\n '">
 <DOM Element: siteinfo at 0x7352b425f770>
 <DOM Text node "'\n '">
 <DOM Element: page at 0x7352af5d7020>
 <DOM Text node "'\n'">

Si comparamos los atributos de este nodo con lo que teníamos en `ElementTree` vemos que esta otra biblioteca no nos está manejando los espacios de nombres: los tenemos sin expandir en los atributos del primer nodo y no está incluido en el tag.

Al iterar un nodo del DOM obtenemos tanto los nodos hijos como otros “pseudo nodos” de tipo `Text`, lo cual es un poco confuso de trabajar. Este detalle es aún más notorio cuando queremos obtener la información necesaria del XML, donde aparece incluso cuando queremos el largo del texto: en el atributo `bytes` del nodo `text` para sacar el valor debemos obtener una especie de nodo hijo y acceder a sus datos:

CELDA 22

```

page = dom.getElementsByTagName("page")[0]
title = page.getElementsByTagName("title")[0]
revision = page.getElementsByTagName("revision")[0]
rev_id = revision.getElementsByTagName("id")[0]
rev_tstamp = revision.getElementsByTagName("timestamp")[0]
rev_text = revision.getElementsByTagName("text")[0]
textlen = rev_text.attributes["bytes"].firstChild.data

```

```

title_text = title.firstChild.data
rev_id_text = rev_id.firstChild.data
rev_tstamp_text = rev_tstamp.firstChild.data

print(f"Página {title_text!r} r.{rev_id_text} ({rev_tstamp_text}) largo={textlen}")

```

```

Página 'Argentina' r.162365205 (2024-09-09T23:59:56Z) largo=115597

```

En este caso de uso es explícito que cuando buscamos cada nodo hijo (por ejemplo `page`) realmente nos estamos quedando con el primero, potencialmente podría haber más. Es lo mismo que pasaba con `ElementTree`, donde usábamos el método `.find` que también nos trae el primero que encuentre.

Finalmente, la tercer biblioteca nos provee un conjunto de herramientas que implementan una forma de acceder a la información de cada nodo al parsear el XML: la *Simple API for XML* (SAX). En este caso no tendremos toda la estructura del XML como un árbol sino que tendremos que implementar nuestros propios mecanismos para acceder a la información correspondiente.

La forma más fácil de lograr esto es heredar del `ContentHandler` que trae el módulo por default e implementar tres métodos que se irán llamando automáticamente al parsear el XML: `startElement` para cuando arranca un elemento, `endElement` para cuando termina, y `characters` para el texto de cada nodo.

En la siguiente celda vemos como armamos nuestra clase siguiendo ese esquema, y luego usamos esa clase para parsear el XML, mostrando directamente la información necesaria al final:

CELDA 23

```

import xml.sax

class Handler(xml.sax.handler.ContentHandler):

    def __init__(self):
        self.for_text = {
            ("mediawiki", "page", "title"): "title",
            ("mediawiki", "page", "revision", "id"): "rev_id",
            ("mediawiki", "page", "revision", "timestamp"): "rev_tstamp",
        }
        self.for_attribs = {
            ("mediawiki", "page", "revision", "text"): "text_attr",
        }
        self.data = {}
        self.branch = []

    def startElement(self, name, attrs):
        self.branch.append(name)
        branch = tuple(self.branch)
        if branch in self.for_attribs:
            key = self.for_attribs[branch]
            self.data[key] = dict(attrs)

    def endElement(self, name):
        assert self.branch[-1] == name
        self.branch.pop()

```

```

def characters(self, content):
    branch = tuple(self.branch)
    if branch in self.for_text:
        key = self.for_text[branch]
        self.data[key] = self.data.get(key, "") + content

handler = Handler()
xml.sax.parseString(fullxml, handler)
print("Página {title!r} r.{rev_id} ({rev_tstamp}) largo={text_attr[bytes]}".format_map(handler.data))
-----
Página 'Argentina' r.162365205 (2024-09-09T23:59:56Z) largo=115597

```

En el método de inicialización tenemos algunas estructuras necesarias para el procesamiento, basadas en el concepto de “rama” del árbol del XML: una secuencia de nodos desde una “raíz” hasta la “hoja” que necesitamos:

- `data`: un diccionario donde guardaremos los resultados del procesamiento
- `for_text` y `for_attribs`: dos diccionarios donde la clave es la rama para indicar la hoja que nos interesa (en el primer caso guardar su texto y en el segundo sus atributos) y el valor es a su vez la clave con que guardaremos esa información en el atributo `data`.
- `branch`: una lista que nos representa “la rama actual”

En `startElement` y `endElement` agregamos y sacamos el nodo en cuestión de `self.branch` para construir/mantener en qué rama estamos; además, en el primero de los dos (donde entramos al nodo y recibimos sus atributos) verificamos si la rama actual es interesante justamente para guardar esos atributos.

En `characters` recibimos el texto de los distintos nodos, el cual guardaremos si la rama está indicada para ello. En este método hay que tener en cuenta que podemos recibir el contenido de un nodo en varias llamadas, así que no alcanza con guardarlo directamente en el diccionario de resultados: tenemos que concatenar las posibles partes.

Vemos que para casos simples es mucho más fácil utilizar alguna de las dos bibliotecas anteriores. Pero el fuerte de SAX es que no tiene que cargar y procesar el XML completo de una sola vez, ya que irá llamando a nuestros métodos y será responsabilidad de nuestro código de extraer sólo la información relevante. De esta manera podemos procesar eficientemente XMLs muy grandes, lo cual sería muy costoso si hay muchos y grandes, o directamente imposible si alguno es más grande que la memoria de nuestra computadora.

15.3. Trabajando con datos binarios

Cuando mencionamos archivos de forma genérica en [15.1](#) vimos la operatoria básica: abrir, cerrar, escribir, leer; entraremos en esta sección en detalles particulares al trabajar con formatos binarios.

Habiendo dicho eso, volveremos a mostrar aquí cómo escribir bytes en un archivo aunque sea similar a lo que ya vimos.

CELDA 1 DE BINARIOS.IPYNB

```
some_bytes = b"abcd\x01\x02\x03\x04"

with open("/tmp/archivoprueba", "wb") as fh:
    fh.write(some_bytes)

with open("/tmp/archivoprueba", "rb") as fh:
    print(fh.read())
```

```
b'abcd\x01\x02\x03\x04'
```

Para leer, a diferencia de cuando trabajamos con texto, en general no iteraremos el *file handler*; el mismo sí es iterable, y nos devolverá el valor de cada byte uno a uno, pero es raro necesitar los bytes individualmente.

Además de leer *todo* el archivo (como recién mostramos para visualizar lo que acabamos de escribir) también podemos leer por partes. Si al `read` le pedimos la cantidad de bytes devolverá solamente lo indicado (a menos que se le termine un archivo, en cual caso devolverá todo lo que pueda), dejando un puntero en donde dejó de leer. Si volvemos a leer, leerá desde allí (y si no indicamos la cantidad, leerá desde esa posición hasta el final).

CELDA 02

```
fh = open("/tmp/archivoprueba", "rb")
# leemos dos bytes
fh.read(2)
```

```
b'ab'
```

CELDA 03

```
# vemos donde está el puntero
fh.tell()
```

```
2
```

CELDA 04

```
# leemos tres bytes más
fh.read(3)
```

```
b'cd\x01'
```

CELDA 05

```
# leemos hasta el final
fh.read()
```

```
b'\x02\x03\x04'
```

A menos que el archivo sea trivial siempre estaremos leyendo sólo lo que necesitamos y no todo el binario en una sola acción. En función de la estructura de los datos a leer podemos incluso evitar leer secuencialmente el archivo para llegar a la parte que necesitamos: si sabemos en qué posición del archivo está la información que necesitamos podemos mover el puntero hasta allí y leer.

CELDA 06
<pre>with open("/tmp/archivoprueba", "rb") as fh: # vamos hasta la cuarta posición fh.seek(4) # leemos print(fh.read(2))</pre> <pre>b'\x01\x02'</pre>

En el ejemplo mostrado el seek posiciona al puntero en la cuarta posición desde el *principio* del archivo. Pero también podemos movernos relativamente a la posición actual o al final del archivo:

CELDA 07
<pre>import os with open("/tmp/archivoprueba", "rb") as fh: # leemos los últimos dos bytes fh.seek(-2, os.SEEK_END) print(fh.read())</pre> <pre>b'\x03\x04'</pre>

No es raro al trabajar con archivos binarios ir “saltando” por distintas partes del archivo para leer sólo lo necesario (en contraposición a una lectura puramente secuencial, como es normal en los archivos de texto).

Más allá de la forma de lectura, el siguiente paso es “interpretar” la información que estamos leyendo. Normalmente no nos interesan los bytes “crudos” al leer un archivo binario, sino convertirlos a números, texto, o las estructuras pertinentes para los tipos de datos con los que estamos trabajando.

Para ilustrar esto tomemos una secuencia de bytes simple (que grabamos primero en el archivo para simular el ejemplo) y luego leemos en función de la siguiente estructura predefinida:

- el primer byte nos indica la longitud de un texto guardado (codificado en UTF-8)
- el segundo byte nos indica la cantidad de números de 32 bits a encontrar
- luego viene el texto (longitud variable, indicada al principio)
- y por último los números que mencionamos

CELDA 08
<pre># escribimos la secuencia que vamos a leer sequence = (b"\x05\x07A\xc3\xb1os" b"\x00\x00\x07\xe3\x00\x00\x07\xe7" b"\x00\x00\x07\xe2\x00\x00\x07\xd5" b"\x00\x00\x07\xc9\x00\x00\x07\xcb" b"\x00\x00\x07\xd6") with open("/tmp/archivoprueba", "wb") as fh:</pre>


```

fh.write(sequence)

# leemos e interpretamos los bytes del archivo
with open("/tmp/archivoprueba", "rb") as fh:
    len_text = int.from_bytes(fh.read(1), byteorder="big")
    quant_numbers = int.from_bytes(fh.read(1), byteorder="big")
    text_bytes = fh.read(len_text)
    print(f"(debug: {text_bytes=})")
    numbers = []
    for _ in range(quant_numbers):
        number = int.from_bytes(fh.read(4), byteorder="big")
        numbers.append(number)
    print(f"(debug: {numbers=})")

text = text_bytes.decode("utf8")
num_sequence = ', '.join(map(str, numbers))
print(f"{text}: {num_sequence}")

```

```

(debug: text_bytes=b'A\xc3\xbls')
(debug: numbers=[2019, 2023, 2018, 2005, 1993, 1995, 2006])
Años: 2019, 2023, 2018, 2005, 1993, 1995, 2006

```

Vemos que luego de abrir el archivo leemos un sólo byte y lo interpretamos como número para el largo del texto, y luego hacemos lo mismo para la cantidad de números. Para obtener el texto leemos la cantidad necesaria de bytes y decodificamos. Para los números vamos leyendo de a 4 bytes y convirtiendo a entero (indicando que están ordenados según *Big-endian*). Más allá de dos líneas de *debug* que dejamos para que se entienda mejor el proceso, finalmente mostramos los datos de forma “elegante”.



Al interpretar una secuencia de bytes como números tenemos que saber si estamos leyendo primero los bytes más o menos significativos. Por ejemplo, el número 23875 se descompone en dos bytes con valores 93 y 67 (porque $93 \times 256 + 67 = 23875$), en hexadecimal 0x5d y 0x43. Estos dos bytes los podemos escribir como b'\x5d\x43' (el más significativo primero, lo que se denomina *Big-endian*) o b'\x43\x5d' (el menos significativo primero, llamado *Little-endian*). Dato curioso: estos nombres provienen de la novela “Los viajes de Gulliver”, del siglo XVIII.

Esta técnica es sencilla y nos puede resultar efectiva para estructuras pequeñas. Pero para casos más complejos tiende a ser muy propensa a errores, porque las distintas lecturas e interpretaciones están distribuidas a lo largo de varias líneas de código.

Hay una forma más simple de realizar esto, usando la función `unpack` del módulo `struct`, que nos permite especificar el formato de la estructura que queremos interpretar desde la cadena de bytes y luego hace todo el trabajo:

CELDA 09

```
import struct

with open("/tmp/archivoprueba", "rb") as fh:
    len_text, quant_numbers = struct.unpack("BB", fh.read(2))
    complex_format = f">{len_text}s{quant_numbers}i"
    text_bytes, *numbers = struct.unpack(complex_format, fh.read())
    print(f"(debug: {text_bytes=})")
    print(f"(debug: {numbers=})")

text = text_bytes.decode("utf8")
num_sequence = ', '.join(map(str, numbers))
print(f"{text}: {num_sequence}")

(debug: text_bytes=b'A\xc3\xbls')
(debug: numbers=[2019, 2023, 2018, 2005, 1993, 1995, 2006])
Años: 2019, 2023, 2018, 2005, 1993, 1995, 2006
```

La primera vez que lo usamos es para sacar las dos cantidades: el largo del texto y cuantos números hay. El formato usado es 'BB', que indica que hay dos enteros sin signo de 1 byte cada uno.

Luego armamos un formato más complejo, donde con '>' indicamos que es *Big-endian* (tiene que estar al principio del formato), luego el largo del texto y 's' para indicar una cadena de cuantos bytes, y finalmente la cantidad de números y 'i' para la cantidad de enteros con signo de 4 bytes de largo cada uno. Termina armando '>5s7i', que nos devolverá una cadena de cinco bytes y los siete números. Cabe aclarar que la repetición del formato 's' funciona distinto que la de la mayoría de los otros formatos (como el 'i'); en el primer caso devuelve *una* cadena de N de largo, mientras que en el caso más común la repetición devuelve *N* elementos.

Entonces cuando desempacamos usando ese formato complejo obtendremos una cadena de bytes y algunos números, los cuales asignamos respectivamente a text y numbers (que como tiene un * al principio del nombre consume todos los elementos remanentes, armando efectivamente una lista). Podemos validar en las líneas de *debug* que obtenemos la misma información que en el ejemplo anterior, y obviamente la línea final mostrada será también igual.

Un efecto colateral fantástico de usar struct para interpretar estructuras de bytes (o para armarlas) es que una sola llamada a nivel Python alcanza para todo el procesamiento, que se realiza a nivel de C compilado, con lo cual es extremadamente eficiente y permite procesar una gran cantidad de información en poco tiempo.

Veamos otro ejemplo apenas más complejo pero más real: interpretar el contenido de una imagen con formato PNG.

CELDA 10

```
from collections import namedtuple

IHDR_struct = '>IbBBBB'
IHDR = namedtuple('IHDR', 'width height bit_depth color_type compression filter interlace')

with open("logo.png", "rb") as fh:
    header = fh.read(8)
    assert header == b"\x89PNG\r\n\x1A\n"
```

```

while True:
    octet = fh.read(8)
    if not octet:
        break
    length, chunk_type = struct.unpack(">I4s", octet)
    print(f"Chunk {chunk_type!r} len={length}")
    chunk_data = fh.read(length)
    if chunk_type == b"IHDR":
        ihdr = IHDR._make(struct.unpack(IHDR_struct, chunk_data))
        print(f"    width={ihdr.width} height={ihdr.height}")

    fh.read(4) # CRC, not for this example

```

```

Chunk b'IHDR' len=13
    width=1280 height=640
Chunk b'pHYs' len=9
Chunk b'tEXt' len=25
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=8192
Chunk b'IDAT' len=1393
Chunk b'IEND' len=0

```

Al principio del código definimos dos estructuras que nos servirán para parsear el campo IHDR de un PNG (que nos da información sobre la imagen como su ancho y alto, que es lo que mostraremos); primero el formato para struct y luego una tupla con nombres para guardar los distintos items resultantes al parsear ese header con el formato indicado. El PNG [tiene otros campos](#) pero en este ejemplo sólo parsearemos el ya nombrado.

Luego abrimos el archivo y antes que nada validamos que los primeros bytes del mismo correspondan a la firma que todos los PNG tienen, para confirmar que el formato es ese. Luego entramos en un bucle “infinito” realizando los siguientes pasos:

- tratamos de leer ocho bytes; si no obtuvimos nada es que se terminó el archivo y salimos
- interpretamos esos ocho bytes como un entero de 32 bits (el largo de la cantidad de bytes a continuación) y una cadena de cuatro bytes-caracteres (su tipo); mostramos ambos datos
- leemos la porción de bytes que viene a continuación, usando el largo recién obtenido
- si el tipo es IHDR, lo interpretamos como dos enteros de 32 bits y cuatro enteros de 8 bits, según la estructura definida arriba, metiendo todos esos datos directamente en una namedtuple para poder acceder a la info por nombre
- de la estructura IHDR mostramos el ancho y alto del PNG
- consumimos los 4 bytes de la verificación del bloque (que no usamos en este simple ejemplo)

Veamos otro caso donde leemos directamente desde un dispositivo de hardware: el mouse. Al mismo podemos acceder, en Linux, abriendo un *path* especial (con permisos de *root*) como si fuese un archivo:

```

1 import struct
2
3 with open("/dev/input/mice", "rb") as fh: # needs root permissions
4     while True:
5         try:
6             data = fh.read(3)
7             button, x, y = struct.unpack(">bbb", data)
8
9             left = button & 0x1
10            right = button & 0x2
11            middle = button & 0x4
12
13            print(f"x={x}, y={y}, {left=}, {middle=}, {right=}")
14        except KeyboardInterrupt:
15            break

```

Para leer del dispositivo tenemos una estructura de “bucle infinito hasta que se interrumpe con CTRL-C”, donde en cada ciclo leemos 3 bytes; cada uno es un número entero: el primero tiene información de los botones, y el segundo y tercero el delta X e Y de la posición. Interpretamos y mostramos.

Cabe la pena mencionar que aunque `struct` nos facilita enormemente el interpretar distintas secuencias de bytes no nos ayuda para interpretar distintos bits dentro de un mismo byte. Es por eso que en las líneas 9 a 11 tenemos que aplicar distintas máscaras para detectar si algunos de los tres bits menos significativos (que corresponden a los tres botones más normales de un mouse) están activados.

Una salida posible de este programa, al mover un poco el mouse y luego apretar el botón derecho, sería algo como lo siguiente:

```

\ $ sudo python3 raton.py
x=0, y=-1, left=0, middle=0, right=0
x=-1, y=-1, left=0, middle=0, right=0
x=-1, y=0, left=0, middle=0, right=0
x=-3, y=-2, left=0, middle=0, right=0
x=-1, y=0, left=0, middle=0, right=0
x=-2, y=0, left=0, middle=0, right=0
x=-1, y=-1, left=0, middle=0, right=0
x=-2, y=0, left=0, middle=0, right=0
x=-1, y=0, left=0, middle=0, right=0
x=-2, y=-1, left=0, middle=0, right=0
x=-2, y=0, left=0, middle=0, right=0
x=-2, y=-1, left=0, middle=0, right=0
x=0, y=0, left=0, middle=0, right=1

```

15.3.1. Un formato binario muy usado: HDF5

El formato HDF5 es la versión 5 del HDF (*Hierarchical Data Format*, “formato jerárquico de datos”), un formato de archivo diseñado para almacenar y organizar grandes cantidades de datos.

En esta versión la estructura del archivo está simplificada para incluir dos tipos de objetos: los conjuntos de datos (*datasets*) dispuestos como arreglos multidimensionales con el tipo defi-

nido, y grupos, que son básicamente contenedores con un nombre que dentro pueden tener a los conjuntos de datos u otros grupos.

La biblioteca `h5py` representa toda esa estructura como un diccionario, donde cada clave será el nombre de un grupo y el valor su contenido, que será otro diccionario si es un grupo o directamente el conjunto de datos.

Para ejemplificar su uso abriremos y usaremos un archivo HDF5 con [información atmosférica de Estados Unidos](#) (descargamos el `all.h5` de allí). Abramos entonces el archivo y exploremos cuales son sus grupos en el primer nivel:

CELDA 11
<pre>import h5py fh = h5py.File("all.h5", "r") fh.keys()</pre>
<pre><KeysViewHDF5 ['datetime', 'latitude', 'longitude', 'mean', 'n', 'std', 'wk']></pre>

Cabe aclarar que al archivo lo podemos abrir usando la sintaxis de administrador de contexto (como ya vimos 5.2 para otros archivos), pero en este caso lo abrimos de forma clásica para poder mostrarlo mejor en estos ejemplos.

Este archivo es bastante simple, tiene directamente los conjuntos de datos sin ningún sub-grupo:

CELDA 12
<pre>for dataset in fh.values(): print(dataset)</pre>
<pre><HDF5 dataset "datetime": shape (1, 61368), type " S14"> <HDF5 dataset "latitude": shape (581777,), type "<f4"> <HDF5 dataset "longitude": shape (581777,), type "<f4"> <HDF5 dataset "mean": shape (581777, 1), type "<f4"> <HDF5 dataset "n": shape (581777, 1), type "<i4"> <HDF5 dataset "std": shape (581777, 1), type "<f4"> <HDF5 dataset "wk": shape (581777, 1), type "<f4"></pre>

Podría ser incluso más simple, ya que casi todos los arreglos tienen una dimensión extra (de valor uno) que no tiene sentido. Veamos cómo acceder en los distintos casos y exploremos apenas los valores del grupo `latitude`:

CELDA 13
<pre>print("datetime:", fh["datetime"][0]) # el primer elemento tiene todo el array print("latitude:", fh["latitude"][42]) # aquí lo tenemos directamente print("mean:", fh["mean"][123]) # y acá cada valor está dentro de una lista fh["latitude"][:20]</pre>
<pre>datetime: [b'20110801230000' b'20110801220000' b'20110801210000' ... b'20110402020000' b'20110402010000' b'20110402000000'] latitude: 24.026184 mean: [7.5135026]</pre>

```
array([24.01368 , 24.014069, 24.01445 , 24.014832, 24.01519 , 24.015564,
       24.015923, 24.016289, 24.016632, 24.01699 , 24.017326, 24.017662,
       24.018005, 24.018349, 24.01867 , 24.018997, 24.019302, 24.019623,
       24.019936, 24.020256], dtype=float32)
```

HDF5 permite agregarle atributos a cada conjunto de datos (para “anotarlos” con cualquier información relevante), así como también nombrar sus distintas dimensiones, pero en este archivo esas facilidades no fueron utilizadas:

CELDA 14

```
print("Dimensiones sin nombre:")
for dim in fh["mean"].dims:
    print("    ", dim)
print("Tampoco hay atributos:", len(fh["latitude"].attrs))
```

```
Dimensiones sin nombre:
<" dimension 0 of HDF5 dataset at 129675587088320>
<" dimension 1 of HDF5 dataset at 129675587088320>
Tampoco hay atributos: 0
```

Los *datasets* de HDF5 son conceptualmente similares a los arreglos de NumPy 11 y hp5y nos da un objeto que se les parece mucho, pero no lo son exactamente. De cualquier manera es trivial crear un arreglo de NumPy a partir del *dataset* para hacer análisis más complejos, como el siguiente ejemplo donde calculamos la media de temperatura para los puntos con latitud entre 27 y 28 y en cualquier longitud:

CELDA 15

```
import numpy as np

latitudes = np.array(fh["latitude"])
latitudes

array([24.01368 , 24.014069, 24.01445 , ..., 49.291092, 49.283897,
       49.27669 ], dtype=float32)
```

CELDA 16

```
lat_27 = (latitudes >= 27) & (latitudes < 28)
np.count_nonzero(lat_27)
```

```
52402
```

CELDA 17

```
means = np.squeeze(fh["mean"])
means

array([7.4238734, 7.4596395, 7.5191693, ..., 9.926428 , 9.930207 ,
       9.933213 ], dtype=float32)
```

	CELDA 18
<code>means[lat_27].mean()</code>	
6.9832783	

Tomamos las latitudes como arreglo de NumPy para poder crear una máscara con **True** cuando el valor está entre 27 y 28, mostrando que hay más de cincuenta mil puntos con esa característica. Luego tomamos los valores medios de cada punto, pero como ese *dataset* tiene una dimensión extra que no queremos tenemos que usar `squeeze` de NumPy, viendo cómo nos queda un arreglo listo para usar. Luego aplicamos la máscara quedándonos con los valores que corresponden a la latitud que nos interesa, y calculamos el valor medio del total de esos datos.



pandas - Ofrece estructuras de datos y herramientas de análisis de datos de alto rendimiento y fáciles de usar, similares a las tablas de bases de datos o hojas de cálculo.

15.4. Pandas

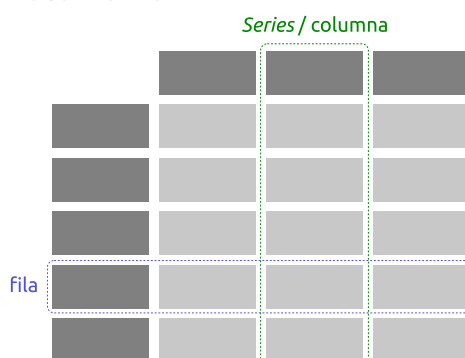
La biblioteca Pandas se ha establecido como la gran herramienta para trabajar con tablas de datos, ofreciendo utilidades que permite realizar diversos procesamiento de forma simple pero poderosa.

Se enfoca en permitir manejar tablas de dos dimensiones. A priori parece una restricción pero tengamos en cuenta que esto modela exactamente la mayoría de las fuentes de datos como pueden ser bases de datos, planillas de cálculo, archivos separados por coma, etc. Pandas denomina *DataFrame* a esta tabla de datos.

Cada *dataframe* puede ser considerado entonces como un conjunto de columnas donde cada una de estas últimas será una serie de escalares de algún tipo de datos específico. A estas se las llama *Series*.

Más allá de esas dos denominaciones puntuales y en función de la naturaleza tabular de la estructura de datos, al trabajar con Pandas también es costumbre hablar de “columnas” (una serie de datos puntual) y “filas” (los datos para la misma posición de las distintas columnas).

DataFrame



Veamos estos conceptos en un ejemplo. Descargamos un CSV de [la tasa de Natalidad por 1000 habitantes registrada en la República Argentina](#) y lo abrimos con Pandas:

CELDA 1 DE PANDAS.IPYNB

```
import pandas as pd

nats = pd.read_csv("tasa-natalidad-deis-2000-2022.csv")
type(nats)

-----

pandas.core.frame.DataFrame
```

Uno de los primeros pasos al trabajar con estructuras de datos es entender exactamente esas estructuras: qué columnas hay y con qué tipo de dato cada una, si contienen valores nulos, cuantas filas hay en total, etc.

Podemos pedirle a Pandas que nos de un resumen:

CELDA 02

```
nats.info()

-----

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
Data columns (total 26 columns):
 #   Column                                Non-Null Count  Dtype
---  -
 0   indice_tiempo                        23 non-null    object
 1   natalidad_argentina                  23 non-null    float64
 2   natalidad_ciudad_autonoma_de_buenos_aires 23 non-null    float64
 3   natalidad_buenos_aires               23 non-null    float64
 4   natalidad_catamarca                 23 non-null    float64
 5   natalidad_cordoba                   23 non-null    float64
 6   natalidad_corrientes                 23 non-null    float64
 7   natalidad_chaco                      23 non-null    float64
 8   natalidad_chubut                    23 non-null    float64
 9   natalidad_entre_rios                 23 non-null    float64
10  natalidad_formosa                    23 non-null    float64
11  natalidad_jujuy                      23 non-null    float64
12  natalidad_la_pampa                   23 non-null    float64
13  natalidad_la_rioja                   23 non-null    float64
14  natalidad_mendoza                    23 non-null    float64
15  natalidad_misiones                   23 non-null    float64
16  natalidad_neuquen                    23 non-null    float64
17  natalidad_rio_negro                  23 non-null    float64
18  natalidad_salta                      23 non-null    float64
19  natalidad_san_juan                    23 non-null    float64
20  natalidad_san_luis                   23 non-null    float64
21  natalidad_santa_cruz                 23 non-null    float64
22  natalidad_santa_fe                   23 non-null    float64
23  natalidad_santiago_del_estero         23 non-null    float64
24  natalidad_tucuman                    23 non-null    float64
25  natalidad_tierra_del_fuego           23 non-null    float64
dtypes: float64(25), object(1)
memory usage: 4.8+ KB
```

Vemos que tenemos un dataframe con 23 filas (numeradas del 0 al 22, también podemos

verlas con `df.index`) y 26 columnas (también podemos ver sus títulos con `df.columns`) donde todas menos la primera (las natalidades) son punto flotante y esa primera (el índice de tiempo) es un objeto genérico; el conteo de “no nulas” es igual al total de filas, o sea que tenemos una tabla completa.

A cada una de esas columnas (*series*) podemos referenciarlas por su nombre o título; aquí vemos los valores de la natalidad de toda Argentina (los primeros cinco, por brevedad):

CELDA 03	
<pre>natalidad_argentina = nats["natalidad_argentina"] natalidad_argentina.head()</pre>	
0	19.0
1	18.2
2	18.3
3	18.4
4	19.3
Name: natalidad_argentina, dtype: float64	

También podemos mencionar varias columnas, lo que efectivamente nos da otro dataframe (¿copiando los valores o es una vista que sólo los referencia? No lo podemos saber de antemano, Pandas hace lo más eficiente según el momento en cada caso):

CELDA 04	
<pre>natarg_con_tiempos = nats[["indice_tiempo", "natalidad_argentina"]] natarg_con_tiempos.head()</pre>	
indice_tiempo	natalidad_argentina
0	2000-01-01 19.0
1	2001-01-01 18.2
2	2002-01-01 18.3
3	2003-01-01 18.4
4	2004-01-01 19.3

Para entender las columnas de punto flotante (que son los valores medidos) podemos calcular un análisis de cada serie de manera muy sencilla:

CELDA 05	
<pre>natarg_con_tiempos.describe()</pre>	
natalidad_argentina	
count	23.000000
mean	16.956522
std	2.540549
min	10.700000
25%	16.350000
50%	17.900000
75%	18.500000
max	19.300000

Para filtrar parte de la tabla podemos trabajar con máscaras como ya vimos con NumPy:

CELDA 06		
<pre>mask = natarg_con_tiempos["indice_tiempo"] >= "2015-01-01" natarg_con_tiempos[mask]</pre>		
indice_tiempo	natalidad_argentina	
15	2015-01-01	17.9
16	2016-01-01	16.7
17	2017-01-01	16.0
18	2018-01-01	15.4
19	2019-01-01	13.9
20	2020-01-01	11.8
21	2021-01-01	11.6
22	2022-01-01	10.7

Volviendo al *dataframe* original, podemos hacer referencia a secciones de la tabla también por índice a través del método `iloc` (por *integer-location*), donde usamos la sintaxis de *slicing* también de NumPy (que es la normal de Python llevada a varias dimensiones) que para dos dimensiones podemos señalar como `rows_slice, cols_slice`.

En el siguiente paso entonces hacemos referencia a todas las filas (nada antes de la coma) y desde la segunda hasta la séptima columna, efectivamente obteniendo toda la info de las primeras cinco provincias (mostramos las primeras tres filas por brevedad, como en muchos de los próximos ejemplos):

CELDA 07

```
provinces = nats.iloc[:, 2:7]
provinces[:3]
```

	natalidad_ciudad_autonoma_de_buenos_aires	natalidad_buenos_aires \
0	14.3	17.5
1	13.9	16.9
2	13.6	17.0

	natalidad_catamarca	natalidad_cordoba	natalidad_corrientes
0	25.8	17.2	22.7
1	24.9	15.9	21.9
2	24.4	16.6	23.3

Los *dataframes* tienen muchos métodos para realizar operaciones en ellos, y en general nos permiten indicar sobre qué eje se realiza la operación. Por ejemplo para sacar el mínimo y máximo de cada fila le indicamos que es en el eje 1, y obtenemos una nueva columna que incorporamos directamente a nuestra tabla general de natalidades:

CELDA 08

```
nats["prov_min"] = provinces.min(axis=1)
nats["prov_max"] = provinces.max(axis=1)
reduced = nats[["indice_tiempo", "natalidad_argentina", "prov_min", "prov_max"]]
reduced[:3]
```

	indice_tiempo	natalidad_argentina	prov_min	prov_max
0	2000-01-01	19.0	14.3	25.8
1	2001-01-01	18.2	13.9	24.9
2	2002-01-01	18.3	13.6	24.4

Allí también vemos como sacamos una nueva tabla de la general (reduced) (porque ya no nos interesa el dato de cada provincia). Esta nueva tabla “reducida” de la general tiene solamente las cuatro columnas que indicamos, y podemos seguir operando sobre ella.

Pandas nos permite realizar operaciones entre las columnas (elemento por elemento a través de las filas), por ejemplo para agregar una nueva columna `range` que es la diferencia entre el máximo y el mínimo de cada fila:

CELDA 09						
<pre>reduced.loc[:, "range"] = reduced["prov_max"] - reduced["prov_min"] reduced.loc[:, "año"] = reduced["indice_tiempo"].str[:4] reduced[:3]</pre>						
	indice_tiempo	natalidad_argentina	prov_min	prov_max	range	año
0	2000-01-01	19.0	14.3	25.8	11.5	2000
1	2001-01-01	18.2	13.9	24.9	11.0	2001
2	2002-01-01	18.3	13.6	24.4	10.8	2002

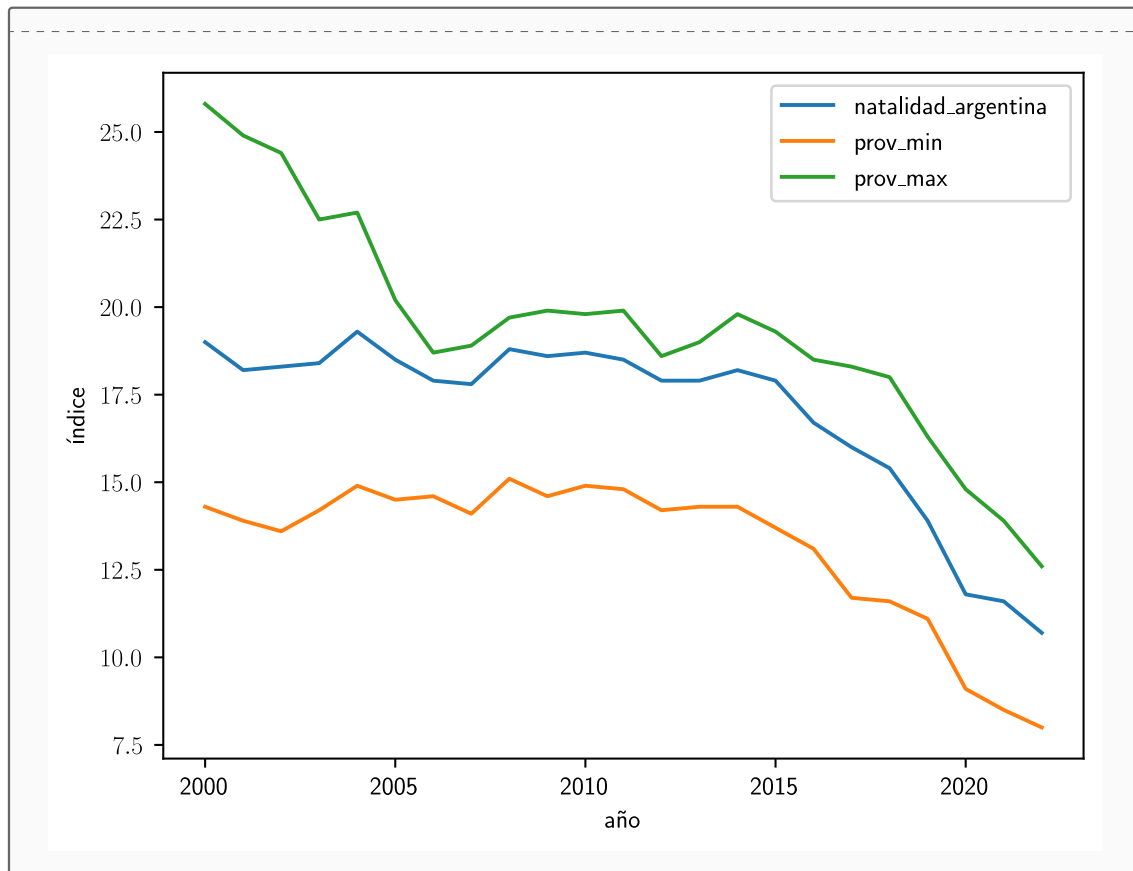
También tenemos acceso al atributo `str` para manejar cada valor como si fuese una cadena (con los métodos de las cadenas que estamos acostumbrados o directamente obteniendo parte de la misma como estamos haciendo en el ejemplo) e incluso un `apply` al que le podríamos pasar una función de Python creada por nosotros.

Noten como usamos otra sintaxis para crear una nueva columna en la tabla. Esto es un requerimiento de Pandas porque a priori no se sabe si `reduced` es efectivamente una vista de `nats` o una copia de una sección de la misma (Pandas resuelve qué hacer en cada caso en tiempo de ejecución).

Una vez que están los datos procesados en general vamos a querer sacar algún gráfico o guardarlos para un procesamiento posterior.

Pandas tiene una muy buena integración con Matplotlib, la biblioteca de gráficos que vimos en otras secciones del libro. Vemos como podemos plotear directamente desde un *dataframe*, incluso pasando opciones útiles para generar el gráfico según queramos.

CELDA 10	
<pre>cols = reduced[["año", "natalidad_argentina", "prov_min", "prov_max"]] subplot = cols.plot(x="año", ylabel="índice")</pre>	



Incluso obtenemos el *plot* de Matplotlib, así que en realidad tenemos todo su potencial al alcance de la mano.

	CELDA 11
<code>type(subplot)</code>	
<code>matplotlib.axes._axes.Axes</code>	

Si quisiéramos trabajar los datos a más bajo nivel podemos obtener un arreglo multidimensional de NumPy sin esfuerzo:

	CELDA 12
<code>reduced[:3].to_numpy()</code>	
<code>array([['2000-01-01', 19.0, 14.3, 25.8, 11.5, '2000'],</code>	
<code> ['2001-01-01', 18.2, 13.9, 24.9, 10.999999999999998, '2001'],</code>	
<code> ['2002-01-01', 18.3, 13.6, 24.4, 10.799999999999999, '2002']],</code>	
<code> dtype=object)</code>	

También podemos exportar los datos a diversos formatos, como por ejemplo los que ya vimos en este capítulo CSV y HDF5, pero hay muchas más opciones.

CELDA 13

```
csv_dumped = reduced[:3].to_csv()
for linea in csv_dumped.split("\n"):
    print(linea) # para mostrarlo más sencillamente aqui
```

```
,indice_tiempo,natalidad_argentina,prov_min,prov_max,range,año
0,2000-01-01,19.0,14.3,25.8,11.5,2000
1,2001-01-01,18.2,13.9,24.9,10.999999999999999,2001
2,2002-01-01,18.3,13.6,24.4,10.799999999999999,2002
```

CELDA 14

```
key = "/data" # el grupo base en la estructura de HDF5
reduced.to_hdf(path_or_buf="/tmp/testpandas.h5", key=key)
```

CELDA 15

```
import h5py

fh = h5py.File("/tmp/testpandas.h5", "r")
[title.decode("utf8") for title in fh["data"]["axis0"]]

['indice_tiempo',
 'natalidad_argentina',
 'prov_min',
 'prov_max',
 'range',
 'año']
```

Para terminar, unas palabras sobre el proceso de instalación de Pandas. Aunque es trivial instalarlo (usando `pip install` en un entorno virtual como ya vimos en 7.2) tenemos que ser conscientes que hay distintas “opciones”, donde cada opción instalará dependencias en función de la utilización que vamos a hacer de la biblioteca. Por ejemplo, si vamos a usar Pandas para realizar gráficos a partir de los datos necesitamos hacer `pip install pandas[plot]`, si vamos a exportar o importar datos de planillas de cálculo haremos `pip install pandas[excel]`, etc.

Pueden revisar las distintas [dependencias opcionales](#), aunque siempre es más sencillo (pero menos eficiente) indicarle que traiga directamente todo con `pip install pandas[all]`.

Parte III

Métodos numéricos y técnicas computacionales

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte [II](#).



16 | Elementos de estadística

16.1. Introducción

La estadística comprende la colección, organización y análisis de datos para comprender fenómenos y tomar decisiones. Los métodos estadísticos son necesarios cuando tenemos información incompleta sobre un fenómeno, debido a que es imposible obtener datos de todos los miembros de una “población”, o porque las mediciones que realizamos poseen intrínsecamente una incerteza. Cuando no podemos determinar completamente los valores de una población, podemos obtener una “muestra” elegida aleatoriamente (muestreo sistemático, estratificado, de conglomerados, etc.) y por medio de métodos estadísticos obtener parámetros que permitan hacer inferencias sobre las propiedades de la población, en forma sistemática y con estimaciones.

Los métodos estadísticos se fundamentan en la teoría de la probabilidad, con la cual podemos modelar incertezas e información incompleta usando variables aleatorias. Como un ejemplo muy

Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1

[Código disponible](#)

común, a partir de la selección aleatoria de una muestra de una población podemos inferir las propiedades de esta última. En la teoría de la probabilidad, cada salida posible de una observación tiene una probabilidad dada, y la probabilidad de todas las posibles salidas constituye la distribución de probabilidad. Dada una función de distribución de probabilidad, podemos calcular las propiedades de la población tales como la media y la varianza, pero si solo conocemos los valores para una muestra obtenida aleatoriamente podemos estimar los valores esperados, o medios, de la población.

16.2. Números aleatorios

La biblioteca estándar de Python provee el módulo `random` que implementa generadores de números pseudo aleatorios para diversas distribuciones, utilizando el algoritmo Mersenne-Twister como generador [6]. Por otra parte, el módulo `random` de NumPy brinda una funcionalidad similar con métodos que generan arrays de NumPy, además de ofrecer un número mayor de distribuciones de probabilidad.

Las rutinas para generar números pseudo aleatorios de NumPy utilizan una combinación de un `BitGenerator`, que son objetos que proveen secuencias de bits aleatorios¹ compuestos de “palabras” de enteros sin signo con secuencias de 32 o 64 bits, y `Generators`, que transforman las secuencias de bits aleatorios de un `BitGenerator` en secuencias de números con una distribución de probabilidad específica (uniforme, normal o binomial, por ejemplo), en un intervalo dado.

Un detalle importante es la semilla utilizada para iniciar la secuencia de números aleatorios. Esta semilla se pasa como argumento al objeto `BitGenerator` en forma de entero o array de enteros. En caso de no pasar ninguna semilla al inicializar el `BitGenerator`, se tomará un valor del generador de entropía del sistema operativo². En las aplicaciones científicas o tecnológicas es importante emular un proceso aleatorio, pero también lo es que estas secuencias sean reproducibles, por lo que el control y registro de las semillas conforman una buena práctica.

Veamos entonces cómo se utilizan estas herramientas. Como muestra el *jupyter-notebook* a continuación, en la celda 1 importamos `Generator` y `PCG64` desde `numpy.random`, además de NumPy para la utilización de arrays. `PCG64` es el objeto `BitGenerator` que inicializa el generador pseudo aleatorio PCG-64, que a su vez implementa el generador congruencial de permutación de O’Neill [7]. Luego establecemos primero la semilla del generador utilizando el número de Hardy-Ramanujan³, de modo de hacer reproducible los resultados, e instanciamos el objeto `rng` de `Generator`. Luego obtenemos el primer número de la secuencia con el método `random` de `rng`, que devuelve un real con distribución uniforme en $[0, 1)$.

¹ Sabiendo que los algoritmos solo son capaces de generar números “pseudo aleatorios”, utilizaremos indistintamente el término “aleatorio” para referirnos a los que estrictamente son números “pseudo aleatorios”.

² En los sistemas operativos del tipo Unix, estas semillas se originan de los archivos especiales `/dev/random`, `/dev/urandom` y `/dev/arandom`, que colectan ruido ambiental de los dispositivos de la computadora. Se puede ver la implementación en Linux [aquí](#).

³ 1729 es el número natural más pequeño que puede expresarse como suma de dos cubos positivos, de dos formas diferentes: $1729 = 1^3 + 12^3 = 9^3 + 10^3$. También es conocido como el “número Taxi”, por la anécdota que le da nombre.

CELDA 1 DE ESTADISTICA.IPYNB

```
import numpy as np
from numpy.random import Generator, PCG64

np.random.seed(1729)
rng = Generator(PCG64())
rng.random()

-----

0.9000273730257627
```

Si queremos obtener un número aleatorio con distribución uniforme en $[a, b)$ es necesario reescalar la salida de `random` multiplicándola por $(b-a)$ y sumándole a . Por ejemplo, si el intervalo es $[-3, 0)$:

CELDA 02

```
(0 + 3) * rng.random() - 3

-----

-0.8286282006850128
```

Tal como anticipamos, podemos obtener no solo números aleatorios aislados sino arrays con las dimensiones deseadas:

CELDA 03

```
rng.random((2,3))

-----

array([[0.65059468, 0.96076472, 0.5634431 ],
       [0.75887512, 0.7184905 , 0.8458265 ]])
```

El objeto `Generator` también ofrece la posibilidad de generar números enteros en un intervalo dado $[a, b)$ (por defecto semiabierto) mediante el método `integers()`. En el ejemplo siguiente generamos un array de cinco elementos en el intervalo cerrado $[10, 20]$, estableciendo que el extremo superior del intervalo puede ser generado con la opción `endpoint=True`.

CELDA 04

```
rng.integers(low=10, high=20, size=5, endpoint=True)

-----

array([16, 12, 14, 16, 18])
```

Otra funcionalidad muy útil que brinda `Generator` es la generación de una muestra aleatoria de un determinado array, usando para esto el método `choice()`. En el ejemplo a continuación simulamos una serie de 5 lanzamientos de tres dados. Para ello, en la celda 5 asignamos en la variable `dado` los posibles resultados del lanzamiento de un dado. En la celda 7 invocamos el método `choice` sobre el objeto `rng`, generando los 5 lanzamientos con tres dados, asumiendo que el dado es “honrado” y que cada cara tiene probabilidad $1/6$:

CELDA 05

```
dado = np.arange(6) + 1
dato

-----

array([1, 2, 3, 4, 5, 6])
```

CELDA 06

```
rng.choice(dado, (5, 3), replace=True)

array([[1, 2, 4],
       [3, 3, 5],
       [4, 6, 4],
       [3, 5, 2],
       [5, 6, 1]])
```

El argumento `replace` permite establecer si el muestreo es con o sin reemplazo. La opción por defecto es `True`, lo que significa que los elementos del array `dado` pueden ser elegidos en múltiples oportunidades.

Por supuesto, podemos asignar probabilidades diferentes para cada elemento del array del cual queremos tomar la muestra. Para ello es necesario especificar mediante otro array la distribución deseada, cumpliendo la condición obvia que la suma debe ser 1. Este array de probabilidades se pasa como argumento `p` del método `choice`. En las celdas 7–8 siguientes “cargamos” el dado en la cara 4 asignándole mayor probabilidad que al resto de las caras, y repetimos el experimento de los cinco lanzamientos de los tres dados:

CELDA 07

```
pesos = np.array([3/20, 3/20, 3/20, 1/4, 3/20, 3/20])
pesos.sum()

1.0
```

CELDA 08

```
rng.choice(dado, (5,3), p=pesos, replace=True)

array([[5, 3, 2],
       [3, 4, 4],
       [2, 4, 5],
       [3, 3, 2],
       [1, 4, 5]])
```

Podemos comparar la frecuencia con la que son seleccionados los diferentes elementos del array `dado` para ambos casos, generando un número relativamente grande de selecciones y graficando el histograma correspondiente, tal como vemos en la celda siguiente:

CELDA 09

```
import matplotlib.pyplot as plt

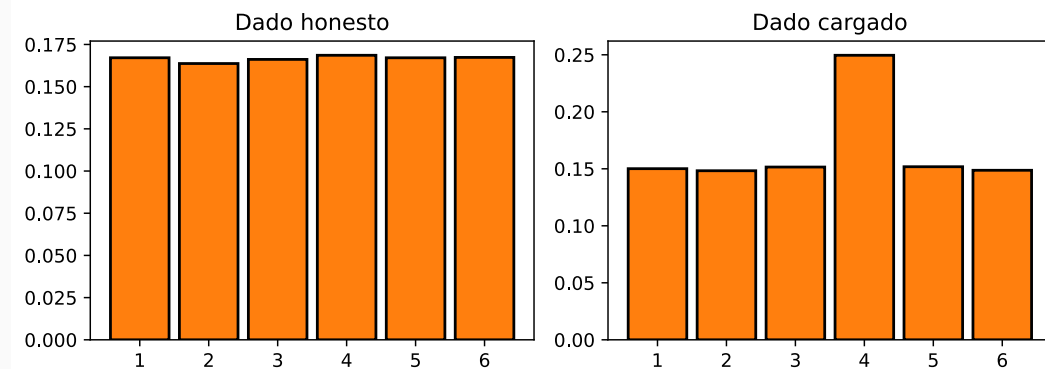
num_lanzamientos = 100000 # Cantidad de veces que seleccionamos una cara del dado
dado_honesto = rng.choice(dado, num_lanzamientos, replace=True)
dado_cargado = rng.choice(dado, num_lanzamientos, p=pesos, replace=True)
bins = [1, 2, 3, 4, 5, 6, 7] # Bins para construir el histograma

fig, ax = plt.subplots(1,2, figsize=(8,3))
ax[0].hist(dado_honesto, bins=bins, density=True, align='left', rwidth=0.85, ec='black',
           fc='tab:orange', linewidth=1.5)
ax[0].set_title("Dado honesto")
```

```

ax[0].set_xticks(bins[:-1])
ax[1].hist(dado_cargado, bins=bins, density=True, align='left', rwidth=0.85, ec='black',
          fc='tab:orange', linewidth=1.5)
ax[1].set_title("Dado cargado")
ax[1].set_xticks(bins[:-1])
plt.tight_layout()
plt.show()

```



Por último, Generator provee métodos para reordenar o permutar elementos de una secuencia: `shuffle` y `permutation`. La principal diferencia entre estos métodos consiste en que `shuffle` reordena los elementos de la secuencia modificando ésta (*in place*), por lo que la secuencia debe ser un objeto “mutable”, mientras que `permutation` devuelve una copia si como argumento se le pasa un array, o un rango reordenado si como argumento se le pasa un entero. Podemos ver cómo se reordenan arrays, listas o rangos en las siguientes celdas:

CELDA 10

```

array = np.arange(10)
rng.shuffle(array)
array

```

```
array([2, 8, 1, 5, 7, 4, 0, 6, 9, 3])
```

CELDA 11

```
rng.permutation([1, 2, 3, 4, 5])
```

```
array([3, 2, 1, 5, 4])
```

CELDA 12

```
rng.permutation(10)
```

```
array([1, 5, 2, 3, 4, 8, 9, 7, 0, 6])
```

En el caso que necesitemos reordenar arrays multidimensionales, podemos establecer sobre qué dimensión hacer el reordenamiento indicándolo en el argumento `axis`. Por ejemplo, si queremos ordenar “por columna”:

CELDA 13

```
x = np.arange(0, 12).reshape(3,4)
x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

CELDA 14

```
y = rng.permutation(x, axis=1)
y
```

```
array([[ 1,  0,  2,  3],
       [ 5,  4,  6,  7],
       [ 9,  8, 10, 11]])
```

mientras que si queremos hacerlo “por fila”:

CELDA 15

```
z = rng.permutation(x, axis=0)
z
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3]])
```

16.3. Variables aleatorias y distribuciones

En la teoría de la probabilidad, el conjunto de todos los resultados de un experimento aleatorio se denomina “espacio muestral”, y cada uno de los resultados se llama “punto muestral”. Por lo general habrá más de un espacio muestral para describir los resultados de un experimento, pero usualmente hay solo uno que suministra la mayoría de la información (por ejemplo, para el experimento de lanzar un dado podríamos tener como espacio muestral el número sobre la cara superior, o si esa cara representa un número par o impar).

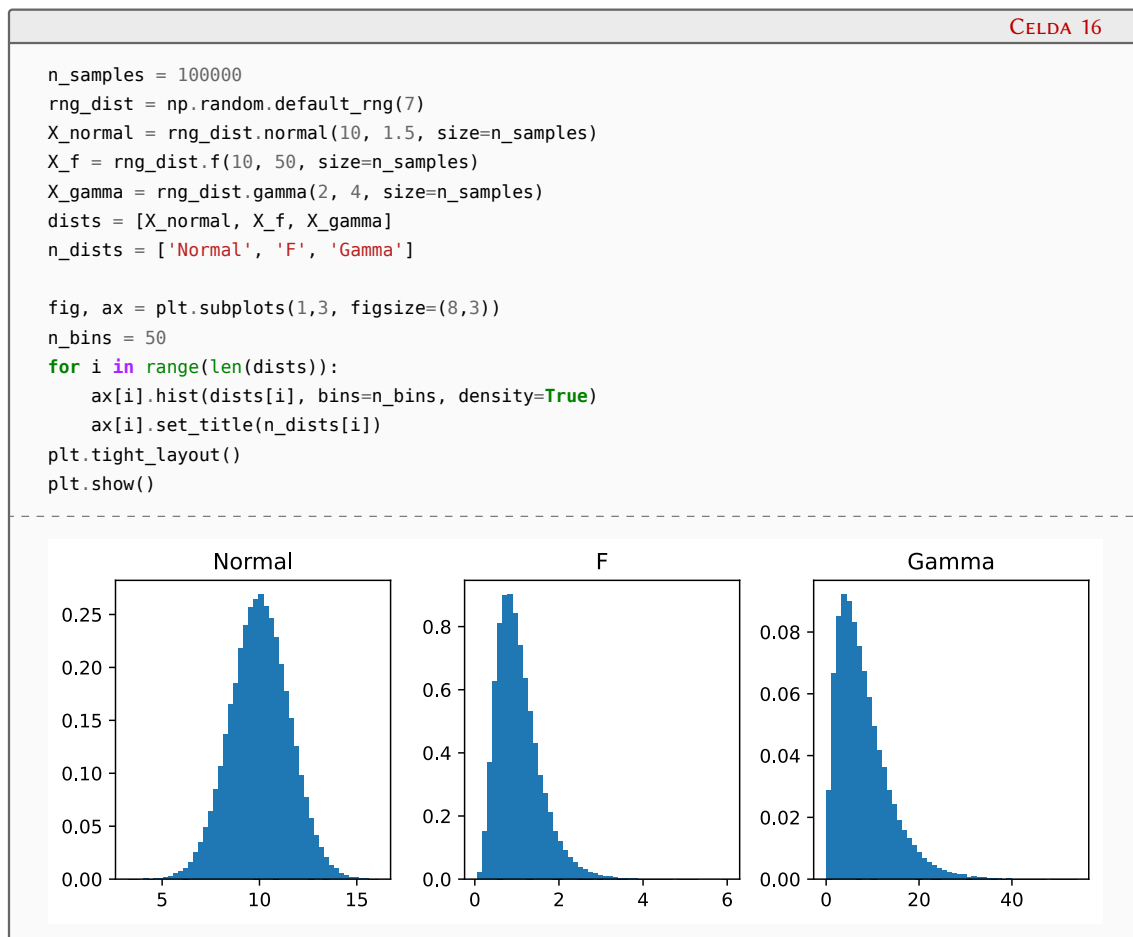
Si a cada punto muestral le asignamos un número, podemos definir una “función aleatoria” (o “estocástica”) que mapea desde el espacio muestral al conjunto de números reales o enteros, que de este modo constituye una “variable aleatoria” o “estocástica”. Por ejemplo, nuestro experimento podría ser lanzar una moneda dos veces de tal forma que nuestro espacio muestral es $E = \{CC, CS, SC, SS\}$ donde C representa cara y S , sello. Con cada punto muestral podemos asociar un número X que sea el número de caras. Nuestra variable aleatoria queda definida entonces por la siguiente tabla:

Punto muestral	CC	CS	SC	SS
X	2	1	1	0

Una variable aleatoria que toma un número finito o infinito contable de valores se denomina “variable aleatoria discreta”, mientras que una que toma un número infinito no contable de

valores se llama “variable aleatoria continua”. Un problema estadístico usual consiste en mapear el resultado de experimentos en valores numéricos y determinar la distribución de probabilidad de esos valores. En consecuencia, una variable aleatoria se caracteriza por los posibles valores que puede tomar (el “recorrido” de la variable aleatoria) y por su función de distribución de probabilidad, y en la práctica esto significa trabajar con estas distribuciones. Según la notación tradicional, en el caso discreto se denomina “función de distribución de probabilidades” mientras que para el caso continuo la denominación es “función de densidad de probabilidad”. En lo que sigue usaremos indistintamente la notación FDP según el caso sea discreto o continuo.

Los objetos Generator de NumPy permiten generar secuencias de números aleatorios a partir de numerosas distribuciones⁴. Como ejemplo, en la celda siguiente construimos un objeto Generator invocando (a diferencia del ejemplo previo) el constructor por defecto `default_rng()` que lo inicializa con el BitGenerator PCG64, y al que le pasamos la semilla del generador (lo cual es opcional⁵). Luego utilizamos ese objeto para generar arrays con un número `n_samples` de elementos con distintas distribuciones, que se muestran como histogramas en la figura.



Como vimos, un objeto Generator de NumPy es capaz de producir una secuencia de números aleatorios con una distribución dada. Esto es, de algún modo, equivalente a realizar experimentos

⁴ Se puede ver la lista completa de distribuciones disponibles en la [documentación](#) de NumPy.

⁵ En este ejemplo utilizamos el [primer primo feliz](#).

estocásticos con esa distribución y obtener salidas de las variables aleatorias que resultan de ese experimento. Una descripción más completa a nivel estadístico se obtiene a partir del submódulo `stats` de SciPy.

El módulo `scipy.stats` provee clases para representar variables aleatorias con una extensa lista de distribuciones de probabilidad. Tiene dos clases base para variables aleatorias continuas y discretas: `rv.continuous` y `rv.discrete`. Estas clases no son utilizadas directamente sino que se usan como clases base para variables estocásticas con distribuciones específicas, y definen una interfaz común para todas las variables aleatorias en este módulo. A continuación haremos una revisión breve de los principales conceptos que se utilizan para caracterizar una distribución, y la forma de obtener descriptores utilizando convenientemente métodos de NumPy o SciPy.



SciPy - Ofrece una colección de algoritmos y herramientas matemáticas avanzadas para tareas como optimización, álgebra lineal y procesamiento de señales.

16.3.1. Medidas de centralidad

Cuando tenemos un conjunto de datos como muestra de una distribución, podemos caracterizar el centro de la distribución con diferentes parámetros: según su valor o por su rango en función del orden en una lista de acuerdo a una magnitud.

16.3.1.1. Media

Por la media de una distribución nos estamos refiriendo a la media aritmética $\bar{x}$:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

Tomemos como ejemplos para analizar las distribuciones obtenidas en la celda 16. Con el método `mean()` de NumPy podemos obtener las medias:

CELDA 17	
<code>X_normal.mean(), X_f.mean(), X_gamma.mean()</code>	

<code>(9.998010521368922, 1.039862209298918, 8.001001879189166)</code>	

Un problema común que acontece es que el conjunto de datos con el que trabajamos tiene valores perdidos (por errores de medición, registro, etc.) que se representan por NaN (que significa “not a number”, o un no-número). En estos casos, NumPy posee métodos que pueden obtener resultados ignorando estos NaN. Por ejemplo, en la celda 18 agregamos un NaN (que numpy representa con `numpy.nan`) a nuestro array con distribución normal. Si calculamos la media de esta nueva distribución, obtenemos un NaN como resultado, pero con el método `nanmean()` recuperamos el resultado correcto:

CELDA 18	
<code>X_normal_Nan = np.hstack((X_normal, np.nan))</code>	
<code>X_normal_Nan.mean()</code>	

nan
CELDA 19
<code>np.nanmean(X_normal_Nan)</code>
9.998010521368922

16.3.1.2. Mediana

La mediana es el valor que separa la mitad superior de la mitad inferior de una lista ordenada de valores. En caso en que dicha lista tenga un número par de valores, la mediana es la media aritmética de los dos valores centrales.

En la celda siguiente podemos ver el resultado de obtener la mediana para dos arrays con números par e impar de elementos:

CELDA 20
<pre>valores_par = np.arange(10) valores_impar = np.arange(11) print(f'Mediana par: {np.median(valores_par)} Mediana impar: {np.median(valores_impar)}') print(f' Media par: {valores_par.mean()} Media impar: {valores_impar.mean()}')</pre>
<pre>Mediana par: 4.5 Mediana impar: 5.0 Media par: 4.5 Media impar: 5.0</pre>

Se puede observar que cuando las distribuciones de valores son simétricas, la media y la mediana coinciden, pero no así cuando las distribuciones no lo son:

CELDA 21
<pre>print(f' Media Normal: {X_normal.mean()}') print(f'Mediana Normal: {np.median(X_normal)}') print(f' Media F: {X_f.mean()}') print(f'Mediana F: {np.median(X_f)}')</pre>
<pre>Media Normal: 9.998010521368922 Mediana Normal: 9.996573186158033 Media F: 1.039862209298918 Mediana F: 0.9448978239181103</pre>

En el caso en que tengamos una función de densidad de probabilidad (variable continua), la mediana es el valor tal que si integramos dicha función desde $-\infty$ hasta la mediana, la integral da 0,5.

16.3.1.3. Moda

La moda de un conjunto de datos es el valor que aparece con mayor frecuencia (si la distribución es continua, es el valor máximo). Por ejemplo, si hacemos cinco experimentos lanzando cuatro dados cargados según la FDP que mostramos en la celda 7, podemos obtener la moda en

cada eje del array resultante, o de todas las salidas, utilizando el método `mode` de `scipy.stats` como se muestra en la celda 23:

CELDA 22
<pre> from scipy import stats samples = rng.choice(dado, (5, 4), p=pesos, replace=True) print(samples) # Salida del experimento print(stats.mode(samples)) # Por defecto axis=0 (conteo por columnas) print(stats.mode(samples, axis=None)) # Todos los datos </pre> <pre> [[4 1 4 5] [5 6 2 4] [3 6 4 1] [4 6 1 2] [3 5 3 6]] ModeResult(mode=array([3, 6, 4, 1]), count=array([2, 3, 2, 1])) ModeResult(mode=4, count=5) </pre>

En la primera salida del cálculo de la moda obtenemos un array que muestra cuál es el valor más frecuente en cada columna (`axis=0` por defecto) y un array con el número de veces que aparece el valor correspondiente. Si queremos obtener la moda de todos los valores que contiene el array `samples`, pasamos la opción `axis=None` como argumento.

En caso que el conjunto de datos a analizar incluya uno o varios NaN, es posible establecer la respuesta de `mode` mediante el parámetro opcional `nan_policy`. Si este parámetro es `'propagate'` el método `mode` devuelve NaN (y ésta es la opción por defecto). Si le pasamos `'omit'` obtenemos la moda de la distribución omitiendo los NaN y si el valor de este parámetro es `'raise'` se lanza una excepción. Cuando existen varias modas en el conjunto, `mode()` devuelve la menor.

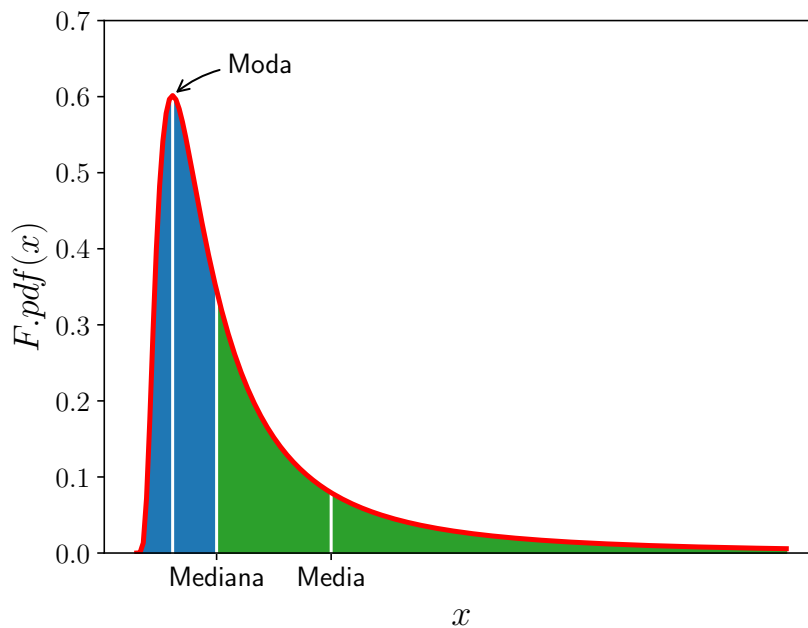
16.3.1.4. Media geométrica

En algunos casos es útil la media geométrica como medida de centralidad de una distribución. Esta magnitud se puede calcular mediante la media aritmética del logaritmo de los valores:

$$\bar{x}_g = \left(\prod_{i=1}^n x_i \right)^{1/n} = \exp \left(\frac{\sum_i \ln(x_i)}{n} \right)$$

Esta función se puede evaluar con el método `gmean` de `scipy.stats`. Obviamente, los valores deben ser positivos para que se pueda calcular la media geométrica. En la celda 23 podemos ver las distintas medidas de centralidad sobre la distribución F.

CELDA 23
<pre> print(f' Media F: {X_f.mean()}') print(f' Mediana F: {np.median(X_f)}') print(f'Media geom. F: {stats.gmean(X_f)}') print(f' Moda F: {stats.mode(X_f)}') </pre> <pre> Media F: 1.039862209298918 Mediana F: 0.9448978239181103 Media geom. F: 0.9189045833992728 </pre>

FIGURA 16.1: Moda, mediana y media para la distribución F .

```
Moda F: ModeResult(mode=0.05411090116941826, count=1)
```

En la figura 16.1 vemos una comparación gráfica de la moda, la mediana y la media para la distribución F cuya asimetría permite distinguir las tres magnitudes. Cabe enfatizar aquí que estas medidas de centralidad corresponden a muestras específicas, si cambiamos la semilla que usamos para el generador, obtendremos otros valores dado que estaríamos analizando una salida de esas mismas variables aleatorias.

16.3.2. Medidas de dispersión

Además de las medidas de centralidad que nos informan acerca de los valores centrales de una distribución de datos, es necesario también conocer alguna cantidad que represente la dispersión de los valores alrededor de las medidas centrales. Veremos a continuación algunas medidas frecuentes.

16.3.2.1. Rango

El rango de una distribución de datos es la diferencia entre los valores máximo y mínimo de dicha distribución, y se obtiene con el método `ptp` de NumPy (que proviene del inglés *peak-to-peak*):

CELDA 24

```
for name, distribucion in zip(n_dists, dists):
    print(f'Rango de {name}: {np.ptp(distribucion)}')
```

Rango de Normal: 12.845742827626554

Rango de F: 5.944760304157928

Rango de Gamma: 53.84036147711152

Esta medida de dispersión incluye los valores atípicos, u *outliers*, que difieren notoriamente del resto de los valores de nuestro conjunto de datos, y pueden ser originados por errores en la medición o registro. Existen criterios para detectar estos *outliers*, tal como el considerar que un valor es atípico cuando se encuentra a 1,5 *IQR* de distancia de los cuartiles Q_1 y Q_3 , siendo $IQR = Q_3 - Q_1$ el rango intercuartílico (que veremos a continuación).

16.3.2.2. Cuantiles

Los cuantiles son valores específicos de la variable aleatoria que dividen el rango de la distribución de probabilidad en intervalos continuos de igual probabilidad. Los cuantiles más usuales tienen nombre propio: cuartiles (cuatro grupos), deciles (diez grupos) y percentiles (100 grupos). Un caso particular es la mediana, que divide la distribución en dos regiones de igual probabilidad.

Para analizar cuantitativamente los cuantiles es útil definir la “función de distribución acumulada” CDF (por su sigla en inglés) como función de la integral de la FDP:

$$CDF(x) = \int_{-\infty}^x FDP(x') dx' \quad (16.1)$$

Conocer la CDF facilita el cálculo de la probabilidad de encontrar un valor de x en el intervalo (a, b) , dado que

$$P(a \leq X \leq b) = \int_a^b FDP(x) dx = CDF(b) - CDF(a)$$

Naturalmente, para distribuciones discretas, las integrales deben reemplazarse por sumas. La figura 16.2 muestra la representación gráfica de la $PDF(x)$ y $CDF(x)$ para una distribución normal, destacando en ambos paneles el valor de $CDF(x = 1)$.

Dado que la CDF es una función monótona creciente, su inversa CDF^{-1} existe, y a partir de ésta se determinan los cuantiles. Por ejemplo, la ya mencionada mediana es simplemente $CDF^{-1}(0,5)$. Del mismo modo se pueden encontrar rangos que incluyan un porcentaje dado de los datos. Si queremos obtener el rango $(x_{\inf}, x_{\sup})$ que incluye el 95 % de la distribución, tenemos:

$$x_{\inf} = CDF^{-1}(0,025), \quad x_{\sup} = CDF^{-1}(0,975)$$

Como mencionamos, otros cuantiles muy usados son los cuartiles Q_1 , Q_2 y Q_3 , que corresponden a los percentiles 25 ($CDF^{-1}(0,25)$), 50 ($CDF^{-1}(0,5)$) y 75 ($CDF^{-1}(0,75)$), respectivamente. Estos cuartiles se utilizan para generar los gráficos de caja (*boxplots*), y la diferencia $IQR = Q_3 - Q_1$ se denomina “rango intercuartílico”, que se utiliza en uno de los métodos para identificar *outliers*.

El submódulo stats de SciPy tiene la función `iqr()` que calcula, por defecto, el rango intercuartílico de una distribución. Sin embargo, podemos modificar este comportamiento estableciendo el rango correspondiente a otros percentiles. Por ejemplo, para calcular el rango de una

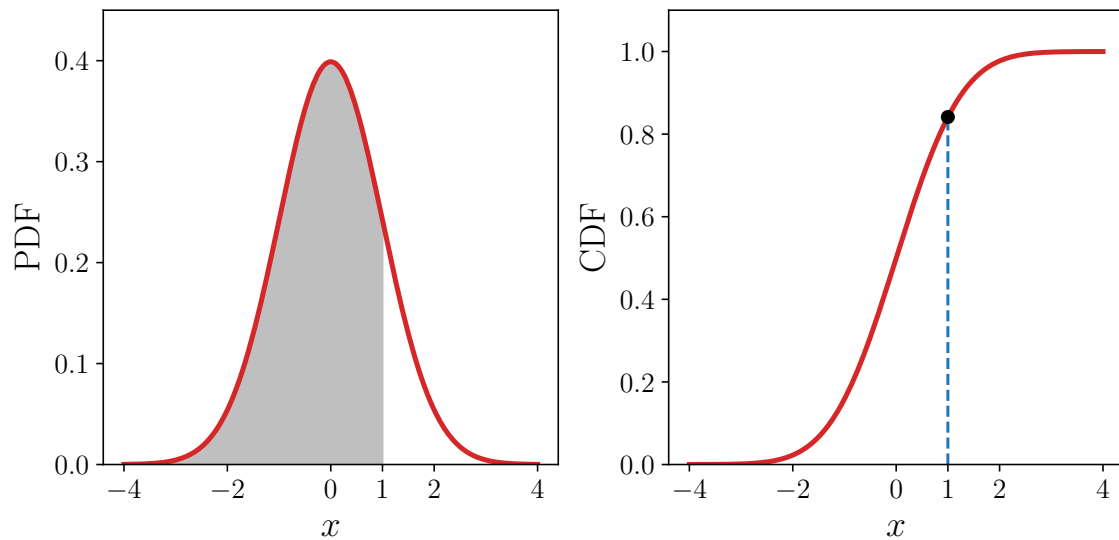


FIGURA 16.2: Función de densidad de probabilidad (FDP, panel izquierdo) y función de distribución acumulada (CDF, panel derecho), para una distribución normal con media $\mu = 0$ y desviación estándar $\sigma = 1$.

distribución podemos pasar como argumento `rng=(0, 100)` que equivale a utilizar la función `ptp`. En el ejemplo mostrado en la celda 25 vemos cómo podemos calcular la mediana como el percentil 50, y el rango intercuartílico como diferencia de los percentiles 75 y 25.

CELDA 25

```

IQR = stats.iqr(X_gamma)
p25 = stats.iqr(X_gamma, rng=(0, 25)) # Percentil 25
p50 = stats.iqr(X_gamma, rng=(0, 50)) # Percentil 50
p75 = stats.iqr(X_gamma, rng=(0, 75)) # Percentil 75
ptp = stats.iqr(X_gamma, rng=(0, 100)) # rango
print(f'{IQR = } \np75 - p25 = {p75 - p25} \nmediana = {p50} \n { ptp = }')

```

```

IQR = 6.916758838426253
p75 - p25 = 6.916758838426253
mediana = 6.721041139175036
ptp = 53.84036147711152

```

El uso de cuartiles permite representar distribuciones mediante los populares gráficos de caja o *boxplots*. En la celda 26 generamos nuevas distribuciones (normal, lognormal y Gumbel) y representamos los valores obtenidos mediante cajas que abarcan el 50 % de los datos, desde el primer cuartil Q_1 hasta el tercer Q_3 (es decir, el rango intercuartílico *IQR*). La línea horizontal dentro de la caja representa la mediana o segundo cuartil Q_2 . Los “bigotes” (líneas negras en ambos lados la caja) se extienden hasta $Q_1 - 1,5 IQR$ y $Q_3 + 1,5 IQR$, y establecen el rango fuera del cual los valores de la distribución se consideran *outliers*, que en la figura se muestran con cruces. Estos bigotes pueden representar valores alternativos como: los valores mínimo y máximo, una desviación estándar alrededor de la media de los datos, los percentiles 9 y 91 o los percentiles 2 y 98⁶.

⁶Para más información sobre cómo construir estos gráficos, invitamos a leer la [documentación de Matplotlib](#).

CELDA 26

```

n2_samples = 500 # Número de muestras

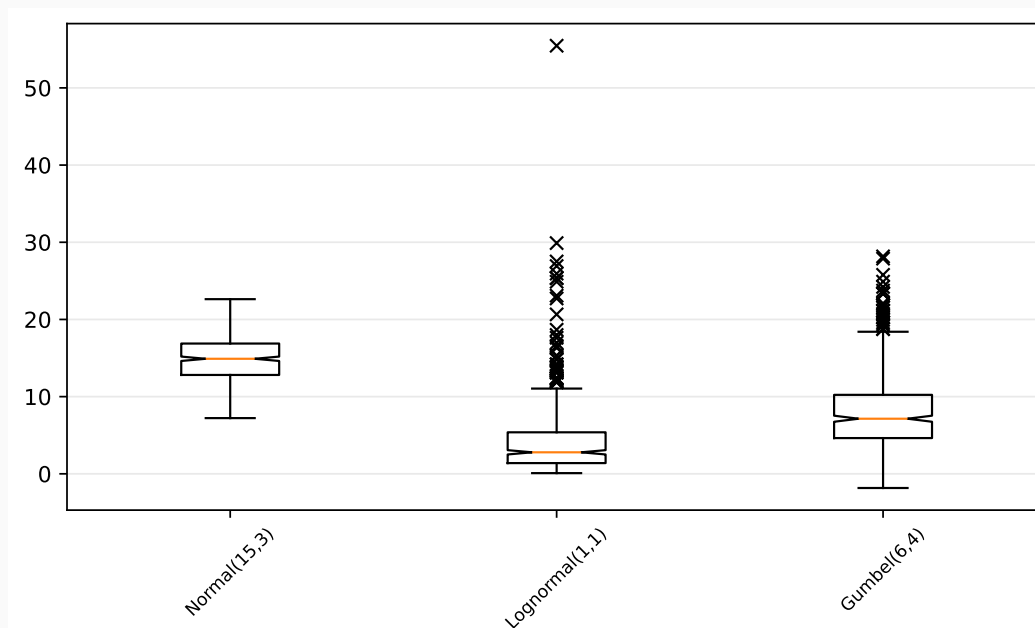
# Tomamos muestras de tres distribuciones
X2_normal = rng_dist.normal(15, 3, size=n2_samples)
X2_logn = rng_dist.lognormal(1, 1, size=n2_samples)
X2_gumb = rng_dist.gumbel(6, 4, size=n2_samples)
dists_2 = [X2_normal, X2_logn, X2_gumb]
nombres_dist = ['Normal(15,3)', 'Lognormal(1,1)', 'Gumbel(6,4)']

fig2, ax2 = plt.subplots(figsize=(8,4))
bp = plt.boxplot(dists_2, notch=True, sym='+', vert=True, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], marker='x')

# Trazamos una grilla horizontal de color tenue para no distraer
# la lectura de los datos.
ax2.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)

# Etiquetamos los boxplots
xtickNames = plt.setp(ax2, xticklabels=nombres_dist)
plt.setp(xtickNames, rotation=45, fontsize=8)
plt.show()

```



16.3.2.3. Varianza y desviación estándar

El cálculo de la varianza de una población con n valores discretos se define como

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (16.2)$$

y la correspondiente desviación estándar es simplemente σ . En caso en que los valores disponibles no correspondan a toda la población, sino solamente a una *muestra* de la misma, se sabe que la ecuación (16.2) es un estimador sesgado⁷ que subestima la varianza de la población. Se puede demostrar que un estimador no sesgado está dado por

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (16.3)$$

que se conoce como la varianza muestral, y la correspondiente raíz cuadrada como la desviación estándar muestral. La notación usual consiste en representar con letras del alfabeto griego los parámetros poblacionales y con letras latinas los correspondientes muestrales. La diferencia con la varianza poblacional es que dividimos por $n-1$ en vez de n . La razón de esto es que se ha eliminado un grado de libertad de la muestra al calcular el valor medio $\bar{x}$, por lo que quedan disponibles $n-1$ grados de libertad. Esta diferencia en la forma de calcular varianzas y desviaciones estándar se refleja en la forma en que invocamos funciones en NumPy. En la celdas 27 y 28 vemos cómo utilizar las funciones `var` y `std` de Numpy para calcular estas magnitudes en ambos casos.

CELDA 27

```
# Varianza y desviación estándar poblacional
var = np.var(X2_normal)
std = np.std(X2_normal)
print(f'σ² = {var}, σ = {std}')
```

σ² = 7.6544508825934985, σ = 2.7666678301873353

CELDA 28

```
# Varianza y desviación estándar muestral
var_m = np.var(X2_normal, ddof=1)
std_m = np.std(X2_normal, ddof=1)
print(f's²(m) = {var_m}, s(m) = {std_m}')
```

s²(m) = 7.669790463520539, s(m) = 2.7694386549480634

Al igual que en otras funciones, NumPy ofrece también versiones de estas funciones que ignoran la presencia de NaN en los datos (`nanvar()` y `nanstd()`).

16.3.2.4. Error estándar

El error estándar (EE) es la estimación de la desviación estándar de un parámetro que se calcula a partir de una dada muestra. Por ejemplo, si estimamos la media de una distribución a partir de las medias de diversas muestras obtenidas de esa distribución, la desviación estándar de esas medias muestrales es el error estándar de la muestra (EEM).

En las celda 29 construimos una distribución normal con media 5 y desviación estándar 1, y a continuación tomamos `n_muestras = 100` muestras de esa distribución, con `n_valores = 60` valores cada una, y guardamos en el array `medias` los valores medios de cada muestra. El valor

⁷ Un estimador es una función que se calcula a partir de los valores muestrales, utilizado para estimar un parámetro desconocido de la población. Un estimador es sesgado cuando la diferencia entre su esperanza matemática y el valor numérico es diferente de cero.

medio de este array es una buena estimación de la media de la distribución. La dispersión de estos valores se representa en estas figuras como la línea anaranjada que están a una desviación estándar de la media. Para valores normalmente distribuidos, el error estándar de la media (EEM) se define como

$$\text{EEM} = \frac{s}{\sqrt{n}} = \frac{1}{\sqrt{n}} \sqrt{\frac{\sum_{i=1}^n (x - \bar{x})^2}{n - 1}} \quad (16.4)$$

que se representa con las líneas de trazos verdes en la figura.

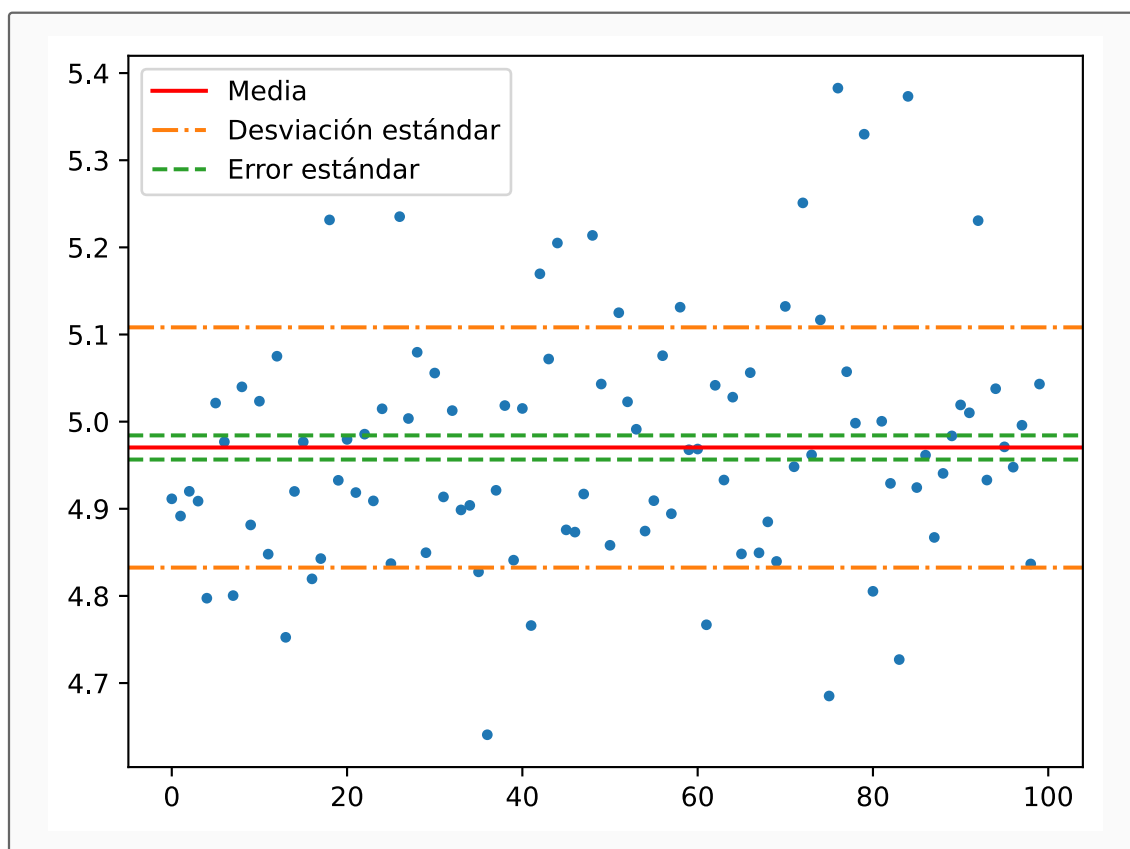
CELDA 29

```
from math import sqrt

rv = stats.norm(5, 1)

n_muestras = 100
n_valores = 60
medias = np.zeros(n_muestras)
for i in range(n_muestras):
    sample = rv.rvs(n_valores)
    medias[i] = sample.mean()
media = medias.mean()
std = medias.std()
EEM = np.std(medias, ddof=1) / sqrt(n_muestras)
print(f'Media de medias: {media}')
print(f'Error estándar de la media: {EEM}')
plt.plot(medias, '.')
plt.axhline(y = media, color='red', label='Media')
plt.axhline(y = media - std, color='tab:orange', linestyle='dashdot', label='Desviación estándar')
plt.axhline(y = media + std, color='tab:orange', linestyle='dashdot')
plt.axhline(y = media - EEM, color='tab:green', linestyle='--', label='Error estándar')
plt.axhline(y = media + EEM, color='tab:green', linestyle='--')
plt.legend()
plt.show()
```

```
Media de medias: 4.970356897826361
Error estándar de la media: 0.013853918499575835
```



El submódulo `stats` de SciPy tiene una función especializada para el cálculo del error estándar de la media (`sem`), que permite operar sobre los diferentes ejes del array de entrada y configurar la manipulación de NaN:

CELDA 30	
<code>stats.sem(medias)</code>	
0.013853918499575835	

16.3.2.5. Intervalos de confianza

Cuando se realiza un análisis estadístico de datos es usual informar el intervalo de confianza de la estimación de un parámetro. El intervalo de confianza (IC) del α % representa el rango que contiene el valor verdadero de ese parámetro con una probabilidad del α %.

Si la distribución muestral es simétrica y unimodal, es posible aproximar el intervalo de confianza de la media por:

$$\text{IC} = \bar{x} \pm Z_{\alpha} \frac{s}{\sqrt{n}}$$

Para determinar el valor de Z_{α} es necesario conocer la distribución teórica del parámetro que se intenta determinar. Sin embargo, para tamaños de muestras n grandes la media muestral está distribuida normalmente, por lo cual la determinación de Z_{α} consiste en encontrar el cuantil

$(1 - \alpha)/2$ de la distribución normal. Por ejemplo, para el intervalo de confianza del 95 %:

CELDA 31
<pre>print(stats.norm.interval(.95, np.mean(medias), scale=stats.sem(medias)))</pre>
<pre>(4.943203716522439, 4.997510079130282)</pre>

En caso que la muestra sea pequeña ($n < 30$) suele usarse como distribución teórica la distribución t de Student⁸.

CELDA 32
<pre>print(stats.t.interval(.95, np.mean(medias), scale=stats.sem(medias)))</pre>
<pre>(-0.03567660913926219, 0.03567660913926219)</pre>

16.4. Test de hipótesis

Es frecuente la necesidad de tomar decisiones sobre poblaciones a partir de la información muestral disponible de las mismas. En estos casos conviene hacer determinados supuestos o hipótesis acerca de las poblaciones que se estudian. Incluso en algunas ocasiones se formulan hipótesis con el solo propósito de rechazarlas. Por ejemplo, si se quiere demostrar que una moneda está “cargada”, se formula la hipótesis de que es “honesta”, lo que significa que la probabilidad de que un lanzamiento resulte cara o sello es 0,5, y se hace un ensayo para validar (o rechazar) estadísticamente esta hipótesis.

El proceso de ensayo o *test* de hipótesis involucra entonces la evaluación de dos afirmaciones mutuamente excluyentes utilizando para ello los datos de una muestra. La idea principal es que siempre existe una situación que puede ser considerada por defecto, de no cambio (respecto de la opinión actual), de no diferencia, no mejora, *statu quo*, etc., que se denomina “hipótesis nula” o H_0 . Por otro lado, la hipótesis alternativa H_1 asume que hay algún cambio respecto de la situación actual. El propósito del test de hipótesis es determinar si los datos muestrales respaldan la hipótesis nula o la alternativa.

En virtud de la naturaleza estadística del método de ensayo, las decisiones que se toman con un número limitado de datos (la muestra) pueden ser erróneas. Si se rechaza la hipótesis nula cuando debería ser aceptada se produce un “error tipo I”, mientras que si se no se rechaza H_0 siendo esta falsa en la población, se está cometiendo un “error tipo II”. Los ensayos de hipótesis deben diseñarse entonces de forma que minimicen los errores de decisión, pero esto no es tan simple ya que en general, un intento de disminuir un tipo de error produce un incremento en el otro. En la práctica un tipo de error suele tener más importancia que el otro (generalmente el error tipo I), por lo que se suele poner una limitación al error de mayor importancia. La única forma de reducir efectivamente ambos errores es aumentar el tamaño de la muestra, pero esto no siempre es posible.

La máxima probabilidad con la que estaríamos dispuestos a arriesgarnos a un error tipo I se denomina “nivel de significancia del test”. Esta probabilidad se suele establecer antes de generar la muestra, de modo que los resultados obtenidos no tengan influencia en esta decisión. Por lo

⁸ Se pueden ver más detalles de la distribución t de Student en la [entrada](#) de Wikipedia.

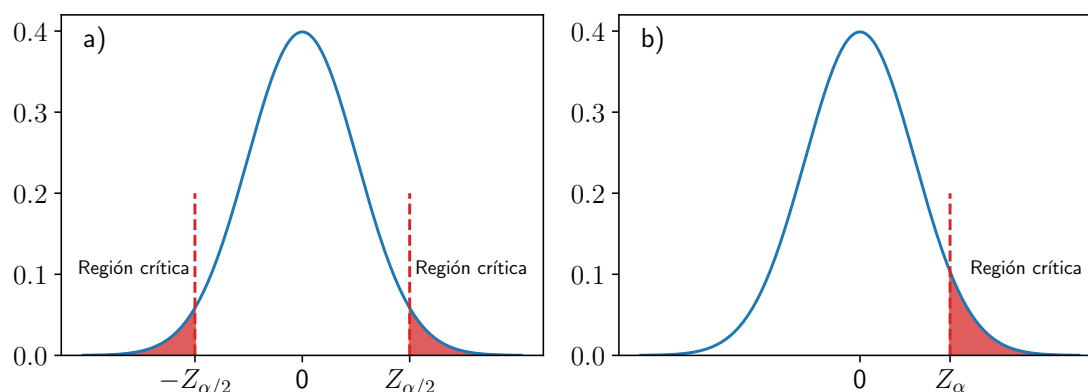


FIGURA 16.3: Test de hipótesis para una distribución normal ($\mu = 0$, $\sigma = 1$). a) Ensayo de dos colas. b) Ensayo de una cola. Las sumas de las áreas en rojo correspondientes a valores en las regiones críticas representan el nivel de significancia del test.

general se utiliza un nivel de significancia de 0,05 o 0,01. Si, por ejemplo, decidimos usar un nivel de 0,05 o 5 %, habrá 5 oportunidades en 100 de que rechacemos la hipótesis cuando debería ser aceptada. Es decir, siempre que la hipótesis nula sea verdadera, tenemos un 95 % de confianza en que tomaremos la decisión correcta.

Otro aspecto a tener en cuenta se refiere al tipo de comparación a realizar. Por ejemplo, podríamos establecer como hipótesis nula que el tiempo promedio que tarda una persona en llegar a su trabajo es de 30 minutos, o que este tiempo es superior a 30 minutos. En estos casos, los valores críticos que definen el rechazo de H_0 se toman como los valores extremos de una distribución de dos colas (con el correspondiente nivel de significancia), o como el valor extremo de una sola cola (que en este caso es la superior). En el primer caso, si el nivel de significancia es del 5 %, los valores críticos son los que limitan la distribución en 2,5 % de su área a cada lado, mientras que en el segundo el 5 % corresponde solo al límite superior de la distribución, tal como se ejemplifica en la figura 16.3.

Una vez que se definieron H_0 y H_1 , se deben coleccionar los datos de la muestra (por ejemplo a través de mediciones, encuestas, etc.). El paso siguiente consiste en encontrar un estadístico que se pueda calcular a partir de la muestra y cuya distribución de probabilidad sea conocida asumiendo la hipótesis nula. Luego calculamos, a partir de los datos, la probabilidad de obtener el valor observado del estadístico utilizando la función de distribución que implica asumir la hipótesis nula. Esta probabilidad se denomina p -valor. Si el p -valor es menor que el umbral predeterminado establecido por el nivel de significancia, podemos concluir que los datos observados probablemente no son descritos por la distribución correspondiente a H_0 , y en este caso podemos rechazar la hipótesis nula en favor de H_1 . El procedimiento para realizar el test de hipótesis consiste entonces en los siguientes pasos:



William Sealy Gosset - Estadístico inglés (1876–1937), conocido por su sobrenombre literario Student. Estudió química y matemática en Oxford.

1. Formular la hipótesis nula y la alternativa.
2. Seleccionar un estadístico tal que la distribución de la muestra bajo la hipótesis nula sea conocida (en forma exacta o aproximada).
3. Obtener los datos de la muestra.
4. Calcular el estadístico a partir de los datos y obtener su p -valor bajo la hipótesis nula.
5. Si el p -valor es menor que el nivel de significancia α predeterminado, rechazamos la hipótesis nula. Si es mayor, H_0 no puede ser rechazada.

En el procedimiento anterior, la mayor dificultad la plantea la determinación de la distribución de probabilidad del estadístico. Sin embargo, muchos ensayos de hipótesis pertenecen a unas pocas categorías usuales para las cuales estas distribuciones son conocidas. En la Tabla 16.1 se muestran algunos casos comunes para H_0 , las distribuciones asociadas y las funciones de SciPy que calculan el estadístico y el p -valor.

TABLA 16.1: Diversos ensayos de hipótesis y sus distribuciones asociadas.

Hipótesis nula	Distribuciones	Funciones de SciPy
Evaluar si la media de una población tiene un valor dado	Normal o t de Student	<code>stats.ttest_1samp</code>
Evaluar si la media de dos variables aleatorias son iguales	t de Student	<code>stats.ttest_ind</code> , <code>stats.ttest_rel</code>
Evaluar la calidad de ajuste de una distribución continua a datos	Kolmogorov-Smirnov	<code>stats.kstest</code>
Evaluar la frecuencia de ocurrencia de datos categóricos	χ^2	<code>stats.chisquare</code>
Evaluar la igualdad de varianza en muestras de dos o mas variables	F	<code>stats.barlett</code> , <code>stats.levene</code>
Evaluar la no correlación entre dos variables	Beta	<code>stats.pearsonr</code> , <code>stats.spearmanr</code>
Evaluar si dos o mas variables tienen la misma media poblacional (análisis de varianza - ANOVA)	F	<code>stats.f_oneway</code> , <code>stats.kuskal</code>

Veamos un ejemplo. Uno de los casos más frecuentes es el de afirmar que la media μ de una población tiene el valor μ_0 . Obtenemos entonces una muestra de esa población, y utilizamos la media muestral $\bar{x}$ para construir el estadístico

$$Z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}$$

donde como es usual, n es el tamaño de la muestra. Si la población es grande y se conoce su varianza σ , es razonable asumir que el estadístico Z está distribuido normalmente. Si la varianza es desconocida podemos reemplazarla por la varianza muestral s . Entonces el estadístico sigue una distribución t de Student, que en el límite de grandes números se aproxima a una distribución normal.

En la celda 33 generamos una distribución normal X con media μ y desviación estándar σ , y luego tomamos una muestra de n valores de esta distribución que almacenamos en $X\_muestra$. Entonces, nuestras hipótesis son:

- $H_0: \mu = \mu_0$
- $H_1: \mu \neq \mu_0$

CELDA 33
<pre>mu_0, mu, sigma = 2.0, 1.82, 0.5 X = stats.norm(mu, sigma) n = 100 X_muestra = X.rvs(n)</pre>

A continuación calculamos los estadísticos Z (asumiendo una distribución normal con σ conocida) y t (con desviación estándar muestral s). Si definimos que nuestro nivel de significancia es del 5 % ($\alpha = 0,05$), calculamos el valor crítico con una distribución normal de dos colas, utilizando para ello la función `norm.ppf()` que devuelve el valor de $Z_{\alpha/2}$, como se muestra en la figura 16.3 (panel a).

CELDA 34
<pre>Z = (X_muestra.mean() - mu_0) / (sigma / np.sqrt(n)) Z</pre>
-2.720519876366154

CELDA 35
<pre>t = (X_muestra.mean() - mu_0) / (X_muestra.std(ddof=1) / np.sqrt(n)) t</pre>
-2.7205218379193887

CELDA 36
<pre>valor_critico = stats.norm().ppf(0.025) valor_critico</pre>
-1.9599639845400545

Podemos ver que los valores de Z y t caen fuera del intervalo $(-1,96, 1,96)$ que corresponde a la zona de aceptación de H_0 , lo cual nos permite rechazar la hipótesis nula con nivel de significancia del 5 %. Podemos calcular también los p -valores utilizando para ello tanto la distribución normal como la t de Student, para calcular el área de las dos colas (por eso el factor 2) limitadas por los estadísticos correspondientes. Vemos que ambos valores están por debajo del nivel de significancia preestablecido, lo que confirma la decisión anterior de rechazar H_0 .

CELDA 37
<pre>p_valor_n = 2 * stats.norm().cdf(-abs(Z)) p_valor_t = 2 * stats.t(df=(n-1)).cdf(-abs(t)) p_valor_n, p_valor_t</pre>

```
(0.0065179353263593535, 0.007700231797543923)
```

Hemos hecho estos cálculos “manualmente” para comprender los detalles de cómo tomar la decisión de aceptar o rechazar la hipótesis nula, pero podemos ahorrarnos este trabajo utilizando directamente las funciones provistas por SciPy para ello. En la celda 38 utilizamos la función `ttest_1samp` de `stats` para comparar la media de la muestra con μ_0 tal como afirma H_0 . Esta función nos devuelve el valor del estadístico y el p -valor:

CELDA 38

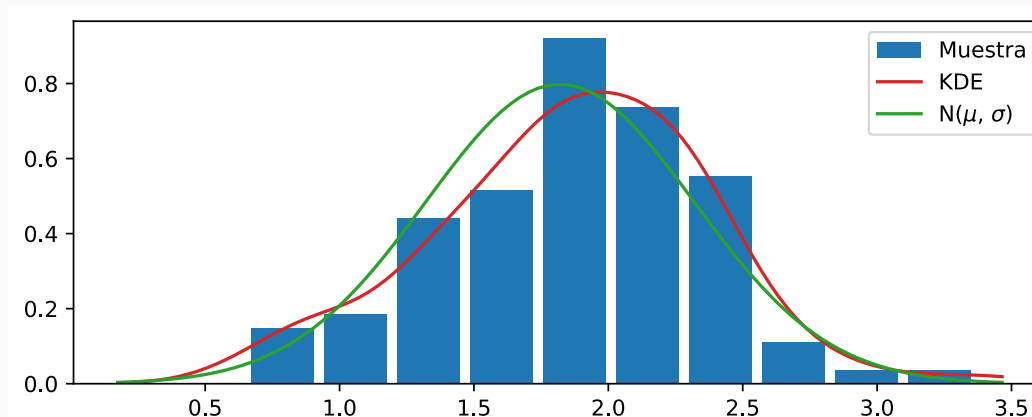
```
e, p = stats.ttest_1samp(X_muestra, mu_0)
e, p
```

```
(-2.7205218379193883, 0.00770023179754393)
```

Tal como vimos previamente, el p -valor devuelto nos permite rechazar la hipótesis nula. En la celda siguiente comparamos la distribución de datos de la muestra (histograma), con la distribución real (línea verde) y una distribución aproximada a partir de los datos de la muestra (línea roja, ver sección siguiente 16.5). Pese a que las distribuciones son muy similares, el *test* de hipótesis con solo 100 valores nos permite afirmar (con una probabilidad menor al 5 % de cometer un error tipo I) que la media muestral no es μ_0 .

CELDA 39

```
from scipy.stats import gaussian_kde
fig, ax = plt.subplots(figsize=(8,3))
ax.hist(X_muestra, density=True, rwidth=0.85, label='Muestra')
kde = gaussian_kde(X_muestra)
x = np.linspace(*X.interval(0.999), num=100)
ax.plot(x, kde(x), color='tab:red', label='KDE')
ax.plot(x, stats.norm(loc=mu, scale=sigma).pdf(x), color='tab:green', label=r'N($\mu$, $\sigma$)')
ax.legend()
plt.show()
```



Otro caso muy frecuente consiste en la comparación de las medias de dos muestras diferentes en la cual la hipótesis nula consiste en asumir que las medias poblacionales de las dos variables aleatorias representadas por las muestras son iguales. Para realizar el *test* de hipótesis, generare-

mos dos variables aleatorias con distribución normal y medias elegidas aleatoriamente. De cada una de estas distribuciones seleccionaremos muestras de 30 valores:

CELDA 40

```
n, sigma = 30, 1.0
mu_1, mu_2 = np.random.random(2)
X_1 = stats.norm(mu_1, sigma)
X_2 = stats.norm(mu_2, sigma)
X_1_muestra = X_1.rvs(n)
X_2_muestra = X_2.rvs(n)
```

La función `ttest_ind()` realiza un ensayo para la hipótesis nula de que dos muestras independientes tienen la misma media asumiendo por defecto que tienen idéntica varianza. La distribución utilizada en este caso es la *t* de Student con un *test* de dos colas, y obtenemos el valor del estadístico y el *p*-valor:

CELDA 41

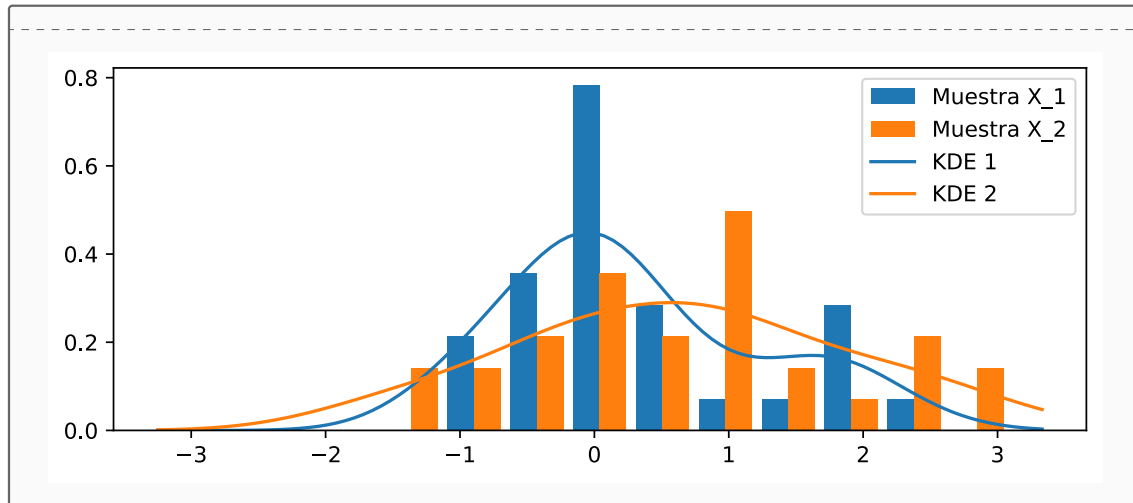
```
t, p = stats.ttest_ind(X_1_muestra, X_2_muestra)
t, p

(-1.2353053231429025, 0.2216974276096801)
```

Vemos que el *p*-valor está muy por encima de 0,05 (considerando un nivel de significancia del 5 %), lo cual no nos permite rechazar la hipótesis nula. En la celda siguiente graficamos los histogramas correspondientes a cada muestra, así como sus distribuciones aproximadas. La escasa cantidad de valores no permite afirmar estadísticamente que las diferencias sean significativas.

CELDA 42

```
fig, ax = plt.subplots(figsize=(8,3))
ax.hist([X_1_muestra, X_2_muestra], density=True, rwidth=0.85, label=['Muestra X_1', 'Muestra X_2'])
kde_1 = gaussian_kde(X_1_muestra)
kde_2 = gaussian_kde(X_2_muestra)
x = np.linspace(*X_1.interval(0.999), num=100)
ax.plot(x, kde_1(x), color='tab:blue', label='KDE 1')
ax.plot(x, kde_2(x), color='tab:orange', label='KDE 2')
ax.legend()
plt.show()
```



Sin embargo, si mostramos los valores elegidos estocásticamente para μ_1 y μ_2 vemos que son bien diferentes:

CELDA 43	
mu_1, mu_2	
(0.04154576848611169, 0.4911986914774764)	

Incrementando la muestra 50 valores, el p -valor obtenido es menor a 0,05, permitiendo en ese caso rechazar H_0 . Esto muestra la importancia del muestreo en la toma de decisiones basada en el *test* de hipótesis.

Por último mostraremos un ejemplo de *test* de hipótesis que generaliza el test anterior a dos o más poblaciones, siendo uno de los casos de análisis de varianza (ANOVA: “ANalysis Of VAriance”), y que se basa en la idea de dividir la varianza total en la varianza entre diferentes grupos y la varianza dentro de cada grupo, y ver si estas distribuciones satisfacen la hipótesis nula de que todos los grupos pertenecen a la misma distribución. Las variables que distinguen los diferentes grupos se denominan habitualmente “factores” o “tratamientos”. En el caso en que cada grupo corresponda a un valor diferente de una sola variable categórica, se utiliza un ANOVA “de una vía” o unidireccional (*one-way ANOVA*), mientras que si se intenta determinar el efecto de dos variables categóricas, se utiliza un ANOVA bidireccional (o *two-way ANOVA*). Es posible realizar también estudios de tres o más variables categóricas pero son poco comunes y puede ser difícil interpretar los resultados si se utilizan demasiados factores.

Un estudio ANOVA básico consiste en realizar los siguientes pasos. Primeramente se calcula la media de cada uno de los grupos, y a continuación se compara la varianza de estas medias (“intervarianza”) con la varianza promedio dentro de cada grupo (“intravarianza”, no explicada por la variable grupo, intervianza). Según la hipótesis nula, todas las observaciones proceden de la misma población con lo que la intervianza será igual a la varianza promedio dentro de los grupos. Se asumen las siguientes suposiciones:

- Las distribuciones están normalmente distribuidas.
- Homogeneidad de varianza, las variaciones dentro de cada grupo son similares en todos los grupos.

- Las observaciones de cada grupo son independientes entre sí, y las observaciones dentro de cada grupo se obtuvieron en forma aleatoria.

En estas condiciones, para un ANOVA unidireccional las hipótesis son:

- H_0 (hipótesis nula): $\mu_1 = \mu_2 = \dots = \mu_k$ (las medias de los k grupos son iguales).
- H_1 (hipótesis alternativa): al menos una media es diferente del resto.

Se construye entonces el estadístico F , que es el cociente entre la varianza de las medias de los grupos y el promedio de la varianza dentro de los grupos, y que tiene una distribución “F de Fisher-Snedecor”. Si H_0 se satisface, F adquiere un valor cercano a 1 debido a que la intervarianza será igual a la intravarianza. Por otro lado, si las medias de los grupos difieren, mayor será la varianza entre medias en comparación con el promedio de las varianzas dentro de los grupos, generando valores de F mayores a 1 y en consecuencia, menor es la probabilidad de que la distribución adquiera valores tan extremos (obteniéndose de este modo un p -valor menor).

Veamos cómo realizar un ANOVA unidireccional mediante la función `f_oneway` del módulo `scipy.stats`. Para ello queremos comparar el rendimiento en un examen de un grupo de 30 alumnos, en función de dónde se sientan cuando asisten a clase: adelante, al medio o atrás. La hipótesis nula consiste en afirmar que la ubicación en el aula no afecta al rendimiento. La hipótesis alternativa indica entonces que la ubicación afecta al rendimiento.

En la celda 44 agrupamos el resultado de una evaluación en tres grupos según la ubicación habitual en el aula durante las clases de 30 alumnos, con 10 alumnos por grupo. Para tener una idea gráfica de estos rendimientos, generamos un gráfico con los *boxplots* de cada grupo a los que hemos superpuesto los resultados individuales, lo que nos da una idea de la dispersión de los resultados.

A continuación, en la celda 45 importamos e invocamos la función `f_oneway` (pasando como argumentos los grupos de datos) que realiza todos los cálculos necesarios para obtener el estadístico F y calcular el p -valor a partir de los resultados muestrales. Vemos que el valor de F es cercano a 1, y el p -valor es suficientemente grande como para evitar el rechazo de H_0 . Podemos interpretar esto afirmando que para los datos disponibles, no hay evidencia estadística que permita distinguir el efecto de la ubicación en las clases con el rendimiento de los alumnos⁹.

CELDA 44

```
frente = [92, 79, 70, 76, 91, 80, 93, 82, 81, 84]
medio = [80, 84, 76, 72, 85, 79, 81, 85, 82, 89]
atras = [74, 67, 81, 78, 69, 80, 89, 84, 90, 67]
grupos = [frente, medio, atras]
nombres_grupos = ['Frente', 'Medio', 'Atrás']

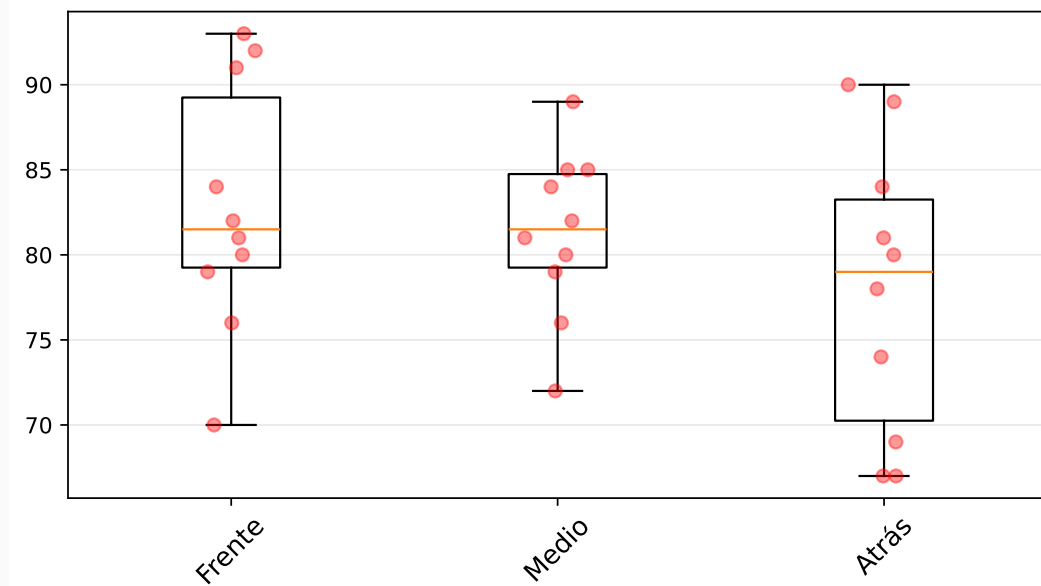
fig, ax = plt.subplots(figsize=(8,4))
bp = plt.boxplot(grupos, sym='+', vert=True, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')

for i, g in enumerate(grupos):
    x = np.random.normal(i + 1, 0.05, size=len(g))
```

⁹Por supuesto, esto es solo un ejemplo de utilización de la función `f_oneway()` y no representa un estudio real (y serio) del rendimiento en los exámenes según la ubicación de los alumnos en clase.

```
plt.plot(x, g, 'ro', alpha=0.4)
ax.yaxis.grid(True, linestyle='-', which='major', color='lightgrey', alpha=0.5)

# Etiquetamos los boxplots
xtickNames = plt.setp(ax, xticklabels=nombres_grupos)
plt.setp(xtickNames, rotation=45, fontsize=12)
plt.show()
```



CELDA 45

```
from scipy.stats import f_oneway
f_oneway(frente, medio, atras)
```

```
F_onewayResult(statistic=1.2490092470277412, pvalue=0.30283594892105703)
```

16.5. Estimación no paramétrica de la FDP

Dada una muestra, podemos asumir que representan valores aleatorios que siguen una determinada distribución de probabilidad caracterizada por ciertos parámetros (por ejemplo, media y varianza de una distribución normal). Estos parámetros pueden ser ajustados a partir de los valores muestrales usando, por ejemplo, una optimización de máxima verosimilitud. De este modo, los métodos estadísticos basados en estas funciones de distribución (tales como los *tests* de hipótesis) representan “métodos paramétricos”. En estos casos estamos realizando una suposición fuerte al determinar la función de distribución que sigue nuestro proceso aleatorio.

Una forma alternativa consiste en intentar describir la distribución a partir de los datos sin hacer ninguna suposición sobre su distribución. Tal vez la manera más frecuente es la de realizar un histograma que divide el rango de valores que contiene la muestra en intervalos iguales (o *bins*) y calcula la cantidad de datos en cada intervalo, o la frecuencia si normalizamos el histograma de modo que el área total de todos los *bins* sume 1. Como resultado obtenemos una

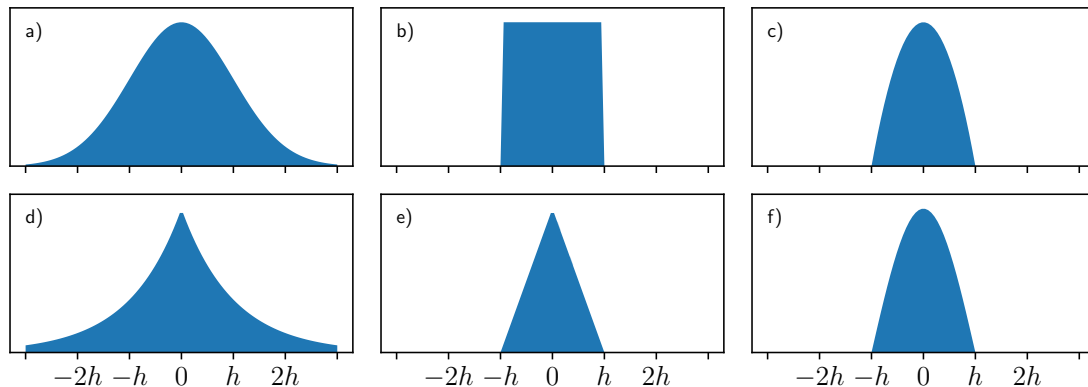


FIGURA 16.4: Algunos kernels más utilizados. a) Gaussiano; b) uniforme; c) Epanechnikov; d) exponencial; e) lineal y f) coseno.

descripción empírica discontinua de la distribución que depende del número de intervalos y de sus localizaciones, por lo que la elección de estos parámetros pueden conducir a representaciones cuantitativamente diferentes de la distribución.

Una *kernel density estimation* (KDE) es una función cuyo propósito es el de estimar la función de distribución de probabilidad de una manera no paramétrica, que puede expresarse como:

$$f(x) = \frac{1}{nh} \sum_{i=0}^n K\left(\frac{x - x_i}{h}\right) \quad (16.5)$$

donde como es usual, n es el tamaño de la muestra (el número de valores), x_i son los valores que componen la muestra, K es una función *kernel* y h es el ancho de banda o *bandwidth*. Un kernel es una función de distribución de probabilidad que debe satisfacer tres condiciones:

1. Simetría: $K(x) = K(-x)$.
2. No negativa: $\forall x, K(x) \geq 0$.
3. Área igual a 1: $\int_{-\infty}^{\infty} K(x) dx = 1$.

Las funciones kernel determinan la forma en que se distribuye la influencia de cada observación. En la gran mayoría de los casos se usa un kernel gaussiano, pero existen otras posibilidades como las que se enumeran en la Tabla 16.2 y sus correspondientes representaciones gráficas en la figura 16.4. En SciPy está disponible el kernel gaussiano, mientras que todas las que aparecen en la Tabla 16.2 pueden utilizarse con el módulo scikit-learn¹⁰

El ancho de banda h controla el ancho de las distribuciones que se ubican alrededor de cada x_i , lo que afecta en forma importante el suavizado del KDE. Para muestras pequeñas, es recomendable usar un h grande de modo que la inclusión de más puntos provea un contexto adecuado. Para muestras grandes, es mejor usar un h pequeño ya que cada punto está rodeado de muchos



scikit-learn - Ofrece herramientas simples y eficientes para análisis de datos y minería de datos, incluyendo algoritmos de clasificación, regresión y clustering.

¹⁰ Se puede ver la documentación [aquí](#).

TABLA 16.2: Kernels usuales.

Kernel	Ecuación
Gaussiano	$\frac{1}{h\sqrt{2\pi}} \exp(-x^2/(2h^2))$
Uniforme	$\frac{1}{2h}, \quad x \leq h$
Epanechnikov	$\frac{3}{4h}(1 - x^2/h^2), \quad x \leq h$
Exponencial	$\frac{1}{2h} \exp(- x /h)$
Lineal	$\frac{1}{2h}(1 - x /h), \quad x \leq h$
Coseno	$\frac{\pi}{4h} \cos(\pi x/2h), \quad x \leq h$

otros, lo cual brinda un buen contexto. Existen diversos criterios para obtener el h óptimo, pero para KDE gaussianos existen algunas “reglas del pulgar” como la regla de Silverman:

$$h = \frac{0,9}{\sqrt[5]{n}} \min\left(s, \frac{IQR}{1,349}\right)$$

(recordemos que IQR es el rango intercuartílico, ver subsección 16.3.2.2). Veamos cómo generamos un KDE gaussiano a partir de una muestra y el efecto que tiene la elección de h .

En la celda 46 definimos la función `k_gauss()` que utilizaremos para graficar las funciones gaussianas que componen la KDE (notar que no están normalizadas, ya que las usaremos solo con fines de visualización) y una función auxiliar, `hacer_figura()`, para construir los gráficos de las celdas siguientes. Esta última función grafica un histograma que agrupa los datos en `n_bins`, traza la KDE y por defecto traza también los kernels sobre los valores individuales (que podemos desactivar con `plt_kernels=False`). Adicionalmente, esta función añade una pequeña marca en cada valor de la muestra sobre el eje horizontal (lo que se denomina un *rug plot*). En la celda 49 generamos una muestra pequeña (`x_chica`), establecemos el valor de $h = 0,05$ y calculamos la KDE sobre la muestra).

CELDA 46

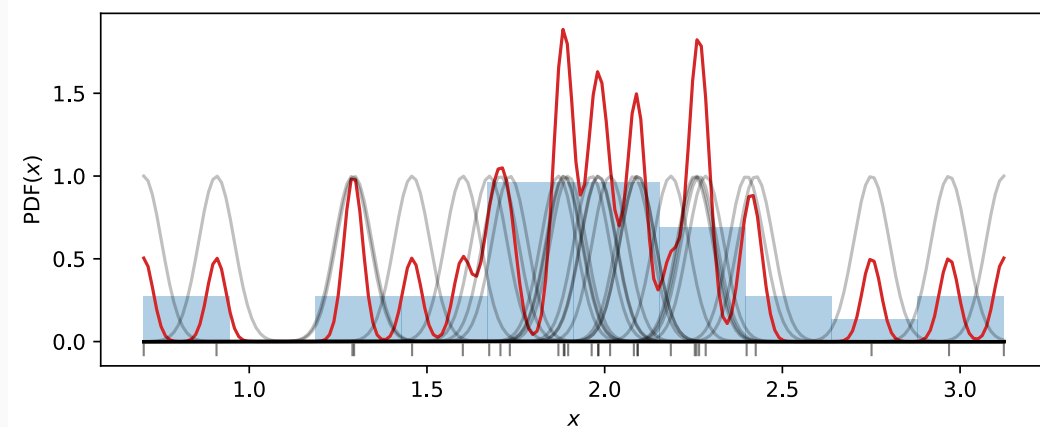
```
def k_gauss(x, xi, h):
    return np.exp(-(x-xi)**2/(2*h**2))

def hacer_figura(datos, kde, n_bins, plot_kernels=True):
    """Función que grafica un histograma de los datos y el KDE asociado,
    junto con las funciones gaussianas de cada valor y un rug plot."""
    fig, ax = plt.subplots(figsize=(8,3))
    ax.hist(datos, bins=n_bins, density=True, alpha=0.35)
    x_c = np.linspace(datos.min(), datos.max(), num=200)
    ax.plot(datos, np.full_like(datos, -0.05), 'k|', markeredgewidth=1.0, alpha=0.5)
    ax.plot(x_c, kde(x_c), color='tab:red', label='KDE')
    if plot_kernels:
        for x_i in datos:
            ax.plot(x_c, k_gauss(x_c, x_i, h), color='k', alpha=0.25)
```

```
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$PDF(x)$')
plt.show()
```

CELDA 47

```
n_chico = 30
X_chica = X.rvs(n_chico)
h = 0.05
kde_c = gaussian_kde(X_chica, bw_method=h)
hacer_figura(X_chica, kde_c, 10)
```

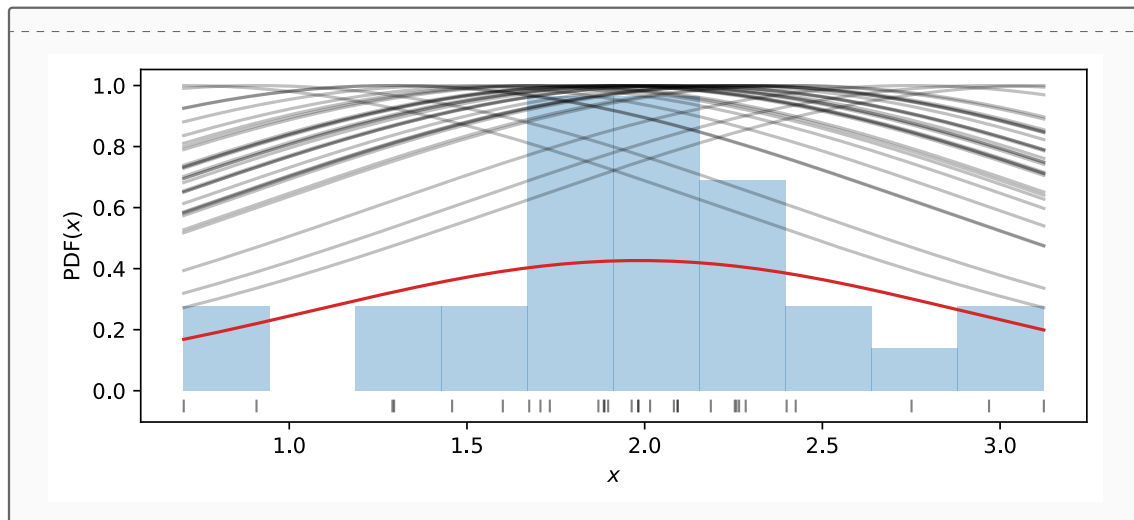


En la figura generada, las curvas grises tenues representan cada gaussiana centrada en los valores de la muestra, que componen (cuando están correctamente normalizadas, no como en la figura) la función KDE que se muestra en línea roja. Dado que usamos un valor de h muy chico, cada función gaussiana abarca una zona angosta alrededor de los valores x_i , dando como resultado una KDE excesivamente estructurada.

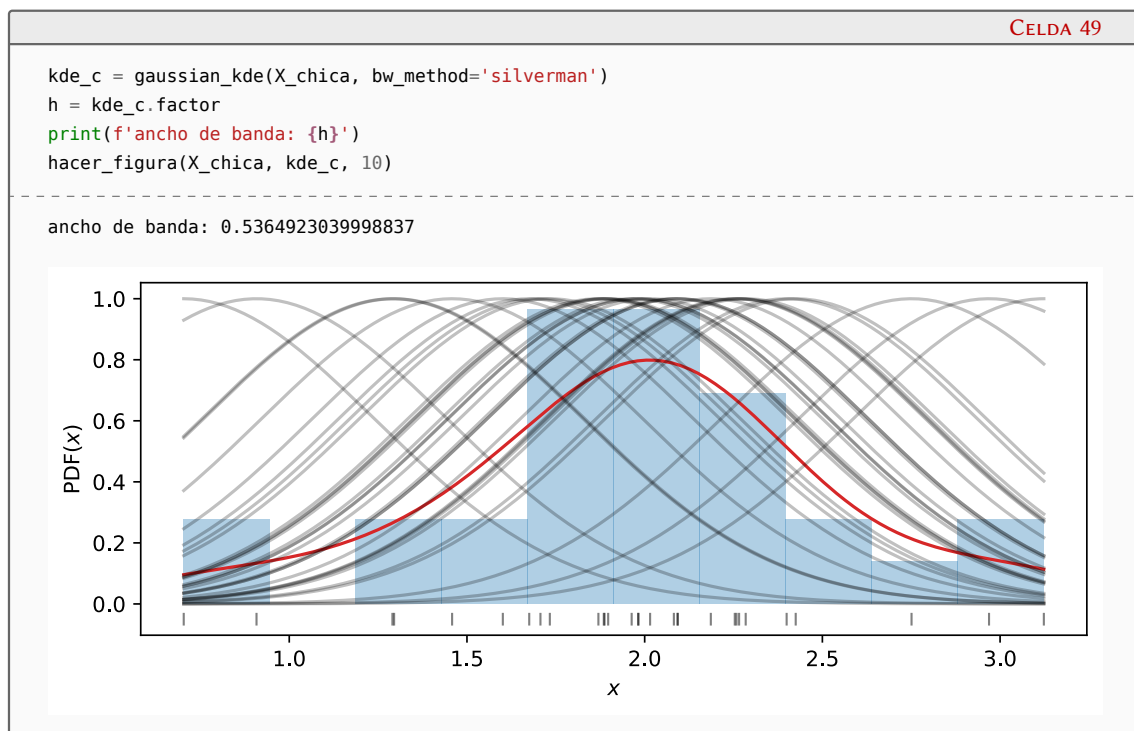
Si ahora elegimos un valor de h exageradamente grande, como en la celda 48, vemos que las gaussianas individuales son tan anchas que todas abarcan al conjunto entero que compone la muestra, y en consecuencia la KDE resulta muy “plana”.

CELDA 48

```
h = 1.5
kde_c = gaussian_kde(X_chica, bw_method=h)
hacer_figura(X_chica, kde_c, 10)
```



Finalmente, en la celda 49 seleccionamos como argumento de `bw_method` la cadena '`silverman`', que aplicará esa regla para determinar el h óptimo. Podemos mostrar el valor de h obtenido mediante el atributo `factor` de la KDE, que ahora resulta una distribución que representa mejor la distribución de los datos. Tal como dijimos previamente, dado que la muestra es chica resulta un ancho de banda relativamente grande de modo que las gaussianas individuales son relativamente anchas.



Para finalizar, veamos un ejemplo en el que tenemos una muestra bimodal de $n = 1000$ valores que construimos en la celda 50. En este caso, el método por defecto para el cálculo del ancho de banda óptimo es el de Scott, utilizado en el cálculo de la KDE de la celda 51. Se puede ver que la KDE obtenida representa de una manera suave el histograma de los datos.

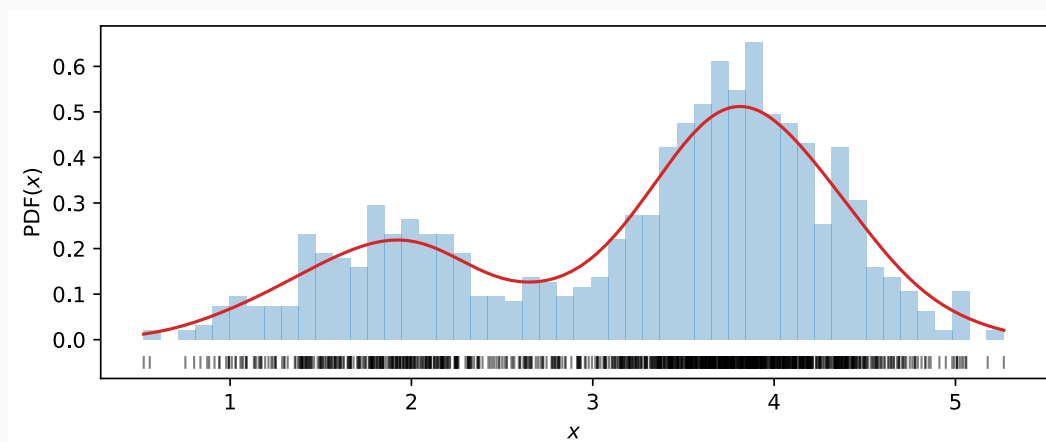
CELDA 50

```
n = 1000
X_1 = stats.norm(2.5, 0.5)
X_muestra = X.rvs(n)
X_muestra[int(0.3 * n):] += 2
```

CELDA 51

```
kde = gaussian_kde(X_muestra)
print(f'Ancho de banda: {kde.factor}')
hacer_figura(X_muestra, kde, 50, plot_kernels=False)
```

Ancho de banda: 0.251188643150958



16.6. Lecturas recomendadas

- Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.^a ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: [10.1007/978-3-319-28316-6](https://doi.org/10.1007/978-3-319-28316-6).
- John J. Schiller, R. Alu Srinivasan y Murray R. Spiegel. *Schaum's outline of probability and statistics*. 4.^a ed. Schaum's outline series. New York, NY: McGraw-Hill Professional, dic. de 2012.



17 | Modelado estadístico

17.1. Introducción

Los modelos estadísticos, construidos a partir de un conjunto de datos observados, son utilizados en las diferentes disciplinas científicas con dos propósitos: explicación causal y predicción empírica. Ambos enfoques no son, naturalmente, mutuamente excluyentes.

Estos modelos generalmente contienen uno o varios parámetros, que pueden determinarse a partir de los datos a través de un procedimiento de ajuste para obtener los valores que mejor expliquen los datos observados. Posteriormente se pueden utilizar para predecir los valores de nuevas observaciones dados los valores de las variables independientes del modelo.

Una vez establecido el modelo, se pueden realizar comparaciones estadísticas entre los valores observados y los generados por el modelo para determinar cuán bien el modelo explica los datos, y qué parámetros contribuyen más al poder predictivo.

Módulo	Versión
Matplotlib	3.9.2
pandas	2.2.2
SciPy	1.14.1
statsmodels	0.14.3

[Código disponible](#)

Abordaremos el siguiente problema: dado un conjunto de variables dependientes Y (o respuesta), y variables independientes X , queremos determinar una relación matemática (o “modelo”) como función $Y = f(X)$.

Si no conocemos la función f pero tenemos acceso a datos provenientes de observaciones como pares $\{x_i, y_i\}$, podemos parametrizar la función y ajustar los valores de dichos parámetros utilizando los datos. Por ejemplo, podríamos tener el modelo lineal $f(X) = \beta_0 + \beta_1 X$ cuyos parámetros son β_0 y β_1 . Por lo general tenemos más datos que parámetros en el modelo, y en tales casos podemos realizar un ajuste de mínimos cuadrados que minimiza la norma del residuo $r = Y - f(X)$.

Hemos presentado el modelo matemático representado por la función f , pero el ingrediente que hace al modelo *estadístico* lo constituye cierta incertidumbre contenida en los datos $\{x_i, y_i\}$ en virtud de, por ejemplo, errores en las mediciones u otros factores fuera de control. Esta incerteza en los datos puede ser descrita en términos de variables aleatorias, por ejemplo, $Y = f(X) + \varepsilon$, donde ε es una variable estocástica. Dependiendo de cómo aparece la variable aleatoria en el modelo, y de la distribución que la describe, podemos obtener distintos tipos de modelos estadísticos, los cuales requieren diferentes abordajes para sus análisis.

Una situación de uso habitual de los modelos estadísticos consiste en describir un conjunto y_i de valores experimentales que se obtienen registrando en cada observación x_i variables de control. Un elemento x_i puede o no ser relevante para predecir (o explicar) la respuesta y_i , por lo que un aspecto importante del modelado estadístico consiste en determinar cuáles son las variables independientes relevantes. Por supuesto, también puede suceder que haya factores relevantes que no hayan sido incluidos en el conjunto de variables independientes x_i , pero que aporten a la determinación de y_i . En este caso puede no ser posible explicar con precisión los datos obtenidos con el modelo. La determinación de cuán precisa es la explicación de los datos con el modelo es otro aspecto importante del modelado estadístico.

17.2. Coeficiente de correlación

Al explorar conjuntos de datos de dos variables es natural preguntarse si existe alguna relación entre ellas. Por ejemplo, querríamos saber si al cambiar una de las variables, la otra también lo hace. Cuando detectamos que existe correlación entre las variables, es posible obtener una relación predictiva que puede resultar útil. Por ejemplo, la correlación existente entre la temperatura ambiente y el consumo de energía eléctrica permite anticipar que habrá un pico de consumo en días de calor extremo.

Si ambas variables están distribuidas normalmente, la medida mas común de dependencia entre estas cantidades es el coeficiente de correlación producto-momento de Pearson, o simplemente “coeficiente de correlación de Pearson”. Para el caso de muestras representadas por pares $\{x_i, y_i\}$, este coeficiente se define como:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

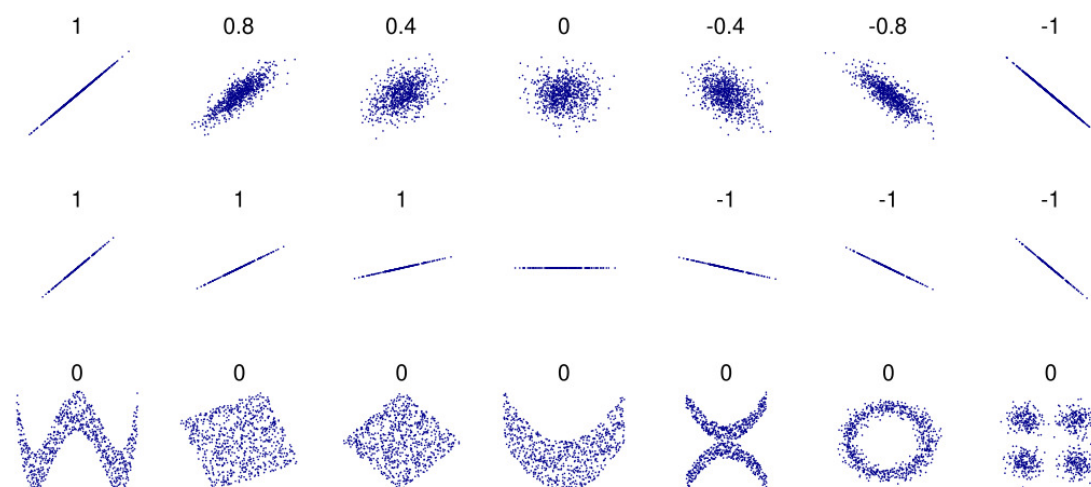


FIGURA 17.1: Distintos conjuntos de valores $\{x, y\}$ con el coeficiente de Pearson para cada caso. En la primera file puede notarse que el coeficiente refleja tanto la dirección de una relación lineal como el apartamiento de la misma, pero no la pendiente de la relación lineal (como se aprecia en la línea media, no tampoco relaciones no lineales (línea inferior). Notar que la figura en el centro tiene pendiente nula, pero el coeficiente de correlación es indefinido. Fuente: [Wikipedia](#) (accedida el 20.06.2022).

Si definimos la covariancia de la muestra como

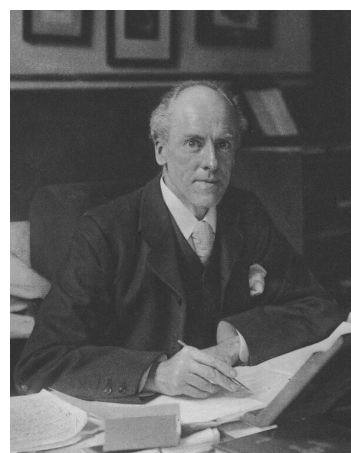
$$s_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n - 1}$$

y s_x y s_y como las desviaciones estándar muestrales de x y y , respectivamente, el coeficiente de correlación r_{xy} se puede escribir como

$$r_{xy} = \frac{s_{xy}}{s_x \cdot s_y}$$

El coeficiente de correlación de Pearson resulta entonces una cantidad adimensional que puede tomar cualquier valor entre -1 y $+1$. Un coeficiente de correlación nulo indica que no existe una relación lineal entre las variables, mientras que los valores extremos indican correlación lineal perfecta. Si la correlación es un número positivo, las variables están directamente relacionadas (esto es, el aumento de una variable implica el aumento de la otra), mientras que si el número es negativo, las variables están inversamente relacionadas (cuando una aumenta la otra disminuye su valor). La figura 17.1 muestra varios conjuntos de puntos y el valor del coeficiente de correlación correspondiente.

Veremos un ejemplo de cálculo de coeficiente de correlación de Pearson analizando un conjunto de datos de atletas olímpicos recolectados desde los juegos de Atenas de 1896 hasta Río de Janeiro en 2016, con el propósito de obtener una estimación cuantitativa de las correlaciones



Karl Pearson - Prominente científico inglés (1857-1936), matemático y pensador socialista que estableció la disciplina de la estadística matemática. Desarrolló una intensa investigación sobre la aplicación de los métodos estadísticos en la biología y fue el fundador de la bioestadística.

entre el peso de los atletas y sus alturas y edades. Estos datos pueden descargarse libremente de [kaggle](#), y analizaremos esta información utilizando las funcionalidades de pandas.

En la celda 1 importamos los módulos requeridos para este ejemplo y generamos el *dataset* a partir del archivo de datos descargados.

CELDA 1 DE CORRELATION.IPYNB

```
import pandas as pd

columns = ['Sex', 'Age', 'Height', 'Weight', 'Year', 'Sport', 'NOC']
olymp = pd.read_csv('athlete_events.csv', usecols=columns).dropna()
olymp.head()
```

	Sex	Age	Height	Weight	NOC	Year	Sport
0	M	24.0	180.0	80.0	CHN	1992	Basketball
1	M	23.0	170.0	60.0	CHN	2012	Judo
4	F	21.0	185.0	82.0	NED	1988	Speed Skating
5	F	21.0	185.0	82.0	NED	1988	Speed Skating
6	F	25.0	185.0	82.0	NED	1992	Speed Skating

En la celda 2 exploramos la estructura del *dataset*, y vemos que disponemos de 206.165 registros para analizar.

CELDA 02

```
olymp.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 206165 entries, 0 to 271115
Data columns (total 7 columns):
#   Column  Non-Null Count  Dtype
---  -
0   Sex      206165 non-null    object
1   Age      206165 non-null    float64
2   Height   206165 non-null    float64
3   Weight   206165 non-null    float64
4   NOC      206165 non-null    object
5   Year     206165 non-null    int64
6   Sport    206165 non-null    object
dtypes: float64(3), int64(1), object(3)
memory usage: 12.6+ MB
```

Para inspeccionar la forma en que se agrupan los puntos tomando los pares altura-peso y edad-peso, los graficamos en forma de puntos:

CELDA 03

```
import matplotlib.pyplot as plt

# Creamos los objetos figura y ejes
fig, ax = plt.subplots(1, 2, figsize=(10,5))

# Creamos los scatterplots
ax[0].plot(
    olymp['Height'], olymp['Weight'], marker='o', linestyle='',
    alpha=0.1, markersize=2)
```

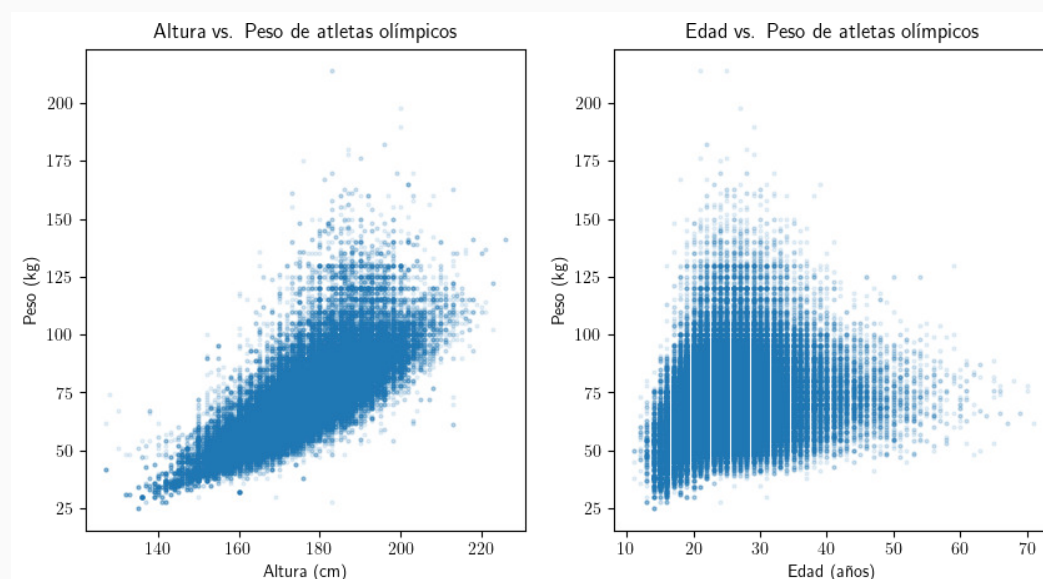
```

ax[1].plot(
    olymp['Age'], olymp['Weight'], marker='o', linestyle='',
    alpha=0.1, markersize=2)

# Títulos y ejes
ax[0].set(
    title='Altura vs. Peso de atletas olímpicos',
    xlabel='Altura (cm)', ylabel='Peso (kg)', rasterized=True)
ax[1].set(
    title='Edad vs. Peso de atletas olímpicos',
    xlabel='Edad (años)', ylabel='Peso (kg)', rasterized=True)

plt.show()

```



Obtenemos el coeficiente de correlación de Pearson entre las variables peso, altura y edad simplemente utilizando el método `corr()` de los `dataset` de pandas:

```
olymp[['Weight', 'Height', 'Age']].corr()
```

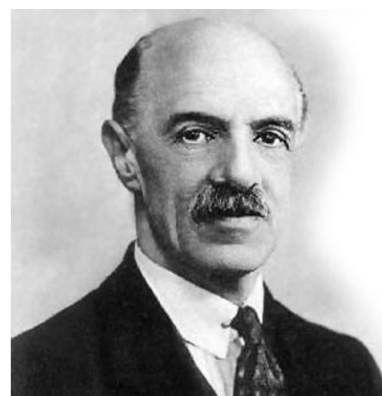
	Weight	Height	Age
Weight	1.000000	0.796573	0.212041
Height	0.796573	1.000000	0.141684
Age	0.212041	0.141684	1.000000

Naturalmente, la correlación de una variable con si misma es perfecta, por lo que los elementos diagonales toman el valor uno. Tal como anticipa la figura de la celda 3, existe una mayor correlación entre la altura y peso de los atletas que entre la edad y el peso. Esta tabla muestra también una menor correlación aún entre la edad y la altura de los atletas.

El coeficiente de correlación de Pearson puede verse afectado por la existencia de valores extremos, que pueden exagerar o atenuar la intensidad de la relación lineal, y por lo tanto es inapropiado su uso cuando una o ambas variables no están normalmente distribuidas. Una de

las formas más usadas para evaluar la correlación entre dos variables, tanto continuas como discretas, que obedecen a distribuciones sesgadas es la ρ de Spearman. Esta métrica utiliza el “orden” (o *ranking*) en que se presentan los datos, en vez de sus valores en sí.

Por ejemplo, podemos crear un *dataframe* reducido del original conservando solo los primeros diez registros de las columnas peso y altura, y luego agregar tres columnas más conteniendo el orden de los valores peso y altura (generados con el método `rankdata()` de `scipy.stats`, y la diferencia d entre ambos valores para el mismo registro:



Charles Edward Spearman - Psicólogo inglés (1863-1945). Estudió en las universidades de Leipzig, Wurzburg y Göttingen y enseñó e investigó en la Universidad de Londres. Formuló la teoría de que la inteligencia se compone de un factor general y otros específicos.

CELDA 05					
<pre>from scipy.stats import rankdata oly_10 = olymp.loc[:10, ['Weight', 'Height']] oly_10['r_Weight'] = rankdata(oly_10['Weight']) oly_10['r_Height'] = rankdata(oly_10['Height']) oly_10['d'] = oly_10['r_Weight'] - oly_10['r_Height'] oly_10</pre>					
	Weight	Height	r_Weight	r_Height	d
0	80.0	180.0	3.0	2.0	1.0
1	60.0	170.0	1.0	1.0	0.0
4	82.0	185.0	6.5	5.5	1.0
5	82.0	185.0	6.5	5.5	1.0
6	82.0	185.0	6.5	5.5	1.0
7	82.0	185.0	6.5	5.5	1.0
8	82.0	185.0	6.5	5.5	1.0
9	82.0	185.0	6.5	5.5	1.0
10	75.0	188.0	2.0	9.0	-7.0

Lo que muestra la celda 5 es que, para el registro 1, tanto el peso como la altura tienen el orden más bajo (1). Cuando los valores se repiten se toma el valor promedio del orden (en este caso, el valor de altura 185 ocupa los órdenes desde el 3 hasta el 8: $(3+4+5+6+7+8)/6 = 5,5$). Con esta diferencia de orden d se calcula el coeficiente de correlación de Spearman como:

$$\rho = - \frac{6 \sum_{i=1}^n d_i^2}{n(n^2 - 1)}$$

donde d_i es la diferencia de orden entre los valores de x y y . Podemos obtener el coeficiente de correlación de Spearman para el *dataset* completo de los atletas olímpicos invocando nuevamente el método `corr()` de pandas, pero esta vez indicando que el método que queremos utilizar para el cálculo es 'spearman':

CELDA 06			
<pre>olymp[['Weight', 'Height', 'Age']].corr(method='spearman')</pre>			
	Weight	Height	Age
Weight	1.000000	0.827500	0.216952
Height	0.827500	1.000000	0.148001
Age	0.216952	0.148001	1.000000

El coeficiente de correlación de Spearman es más robusto ante la presencia de valores extremos, y es una medida del grado de correlación entre funciones monótonas (más que de la correlación lineal como el coeficiente de Pearson). También toma los valores extremos $+1$ y -1 cuando la correlación es perfecta (directa o inversa, respectivamente).

17.3. Definición de modelos estadísticos con *patsy*

Al realizar un modelado estadístico es necesario especificar la relación matemática entre las variables dependientes Y y las variables independientes X . Por ejemplo, en el caso de los modelos lineales, la variable Y puede expresarse como una combinación lineal de las variables independientes X , o de funciones de estas variables. Por ejemplo, podríamos proponer que esta relación es:

$$Y = \beta_1 X + \beta_2 X^3 + \cos(X)$$

Es importante notar que no necesariamente Y debe ser función lineal de X en los modelos lineales, sino que debe ser lineal respecto de los coeficientes desconocidos β_i . Por otro lado, el modelo $Y = \exp(\beta_0 + \beta_1 X)$ es un ejemplo de modelo no lineal, ya que en este caso Y no es una función lineal de β_0 y β_1 . Sin embargo, podemos transformar este modelo en uno lineal tomando logaritmos en ambos miembros lo que conduce a $\tilde{Y} = \beta_0 + \beta_1 X$ con $\tilde{Y} = \log Y$. La clase de problemas que pueden transformarse en modelos lineales pueden ser tratados con el modelo lineal generalizado.

El paquete *patsy* de Python provee el acceso a un mini lenguaje que permite describir fórmulas para representar relaciones entre variables dependientes e independientes, y está basado en la notación introducida por Wilkinson y Rogers en 1973[10]. Con este paquete, podemos escribir

$$Y \sim X$$

para representar la relación directa entre la variable Y y X . Podemos representar una relación más compleja en la que Y depende de las variables X , a y b de la siguiente manera:

$$Y \sim X + a + b + a : b$$

Los símbolos que se pueden utilizar para construir estas expresiones pueden verse en la Tabla 17.1. La descripción detallada y completa del lenguaje se puede ver en la documentación del paquete.

Una vez definida la relación funcional entre las variables que constituyen el modelo, el paso siguiente es construir las llamadas matrices de diseño. Para un modelo lineal simple $Y = X\beta + \epsilon$ son necesarias dos matrices: una con los valores de la variable dependiente Y y otra con las variables dependientes del segundo miembro (más una columna opcional de unos que representan la ordenada al origen)¹. Los elementos X_{ij} de la matriz de diseño X son los valores de funciones de las variables independientes correspondientes a cada coeficiente β_i y observación y_i . Por ejemplo, si los valores observados son $y = [1, 2, 3, 4, 5]$ para dos variables independientes con los valores correspondientes $x_1 = [6, 7, 8, 9, 10]$ y $x_2 = [13, 14, 15, 16, 17]$, y si el modelo

¹ Se suelen nombrar también “vector observación” y y dejar específicamente “matriz de diseño” para X . Usaremos estas notaciones indistintamente.

TABLA 17.1: Elementos principales de la sintaxis de fórmulas con *patsy*.

Operador	Significado
$\sim$	Separa el lado derecho del izquierdo en la expresión. Si se omite se asume que solo existe el lado derecho.
$+$	Combina términos en ambos lados de la expresión (unión de conjuntos).
$-$	Elimina términos en el lado derecho del conjunto de términos del lado izquierdo (diferencia de conjuntos).
$:$	Término de interacción (por ejemplo $a : b$ denota $a \cdot b$)
$*$	$a * b$ es una forma abreviada de $a + b + a : b$
$/$	a/b es una forma abreviada de $a + a : b$
$**$	Toma un conjunto de términos en el lado izquierdo y un entero n en el derecho, y calcula el resultado de $*$ sobre el conjunto de términos con sí mismo n veces.
$f(x)$	Función arbitraria de Python (o NumPy) utilizada para transformar términos en la expresión (por ejemplo: <code>np.log(x)</code>)
$C(x)$	Trata a la variable x como categórica

en consideración es $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$, la matriz de diseño para el lado derecho es $X = [\mathbf{1}, x_1, x_2, x_1 x_2]$. En las celdas del siguiente *notebook* podemos ver de qué forma construimos el modelo con *patsy* y obtenemos las matrices de diseño.

CELDA 1 DE PATSY_MATRICES.IPYNB
<pre>import numpy as np y = [1, 2, 3, 4, 5] x1 = [6, 7, 8, 9, 10] x2 = [13, 14, 15, 16, 17] data = {'y': np.array(y), 'x1': np.array(x1), 'x2': np.array(x2)}</pre>
CELDA 02
<pre>from patsy import dmatrices dmatrices("y ~ x1 + x2 + x1:x2", data) (DesignMatrix with shape (5, 1)) y 1 2 3 4 5 Terms: 'y' (column 0), DesignMatrix with shape (5, 4) Intercept x1 x2 x1:x2 1 6 13 78 1 7 14 98 1 8 15 120 1 9 16 144</pre>

```

      1  10  17   170
Terms:
  'Intercept' (column 0)
  'x1' (column 1)
  'x2' (column 2)
  'x1:x2' (column 3)

```

En las celdas 1 definimos los valores de nuestras variables dependiente e independientes, respectivamente. En la celda 2 utilizamos la función `dmatrixes()` para obtener las matrices de diseño correspondientes al modelo definido como argumento de dicha función, que se construyen utilizando data ingresado como segundo argumento. Si solo queremos la matriz de diseño sin los valores de y , podemos usar `dmatrix()` como se muestra en la celda siguiente:

CELDA 03

```

from patsy import dmatrix

X = dmatrix("x1 + x2 + x1:x2")
X

DesignMatrix with shape (5, 4)
Intercept  x1  x2  x1:x2
      1   6  13    78
      1   7  14    98
      1   8  15   120
      1   9  16   144
      1  10  17   170

Terms:
  'Intercept' (column 0)
  'x1' (column 1)
  'x2' (column 2)
  'x1:x2' (column 3)

```

La definición del modelo con *patsy* consiste en definir una cadena de caracteres que representa la relación matemática entre las variables (nótese que no figuran los coeficientes β_i en la fórmula ya que se asume implícitamente que cada término de la fórmula tiene un parámetro del modelo como coeficiente), mientras que los datos utilizan una representación con diccionarios.

La matriz de diseño obtenida con `dmatrix` es un objeto del tipo `DesignMatrix` de *patsy*. Este objeto es completamente compatible con los arrays de NumPy, e incluso se muestra como tal como podemos ver a continuación:

CELDA 04

```

print(type(X))
print(X)

<class 'patsy.design_info.DesignMatrix'>
[[ 1.   6.  13.  78.]
 [ 1.   7.  14.  98.]
 [ 1.   8.  15. 120.]
 [ 1.   9.  16. 144.]
 [ 1.  10.  17. 170.]]

```

En forma alternativa, y dada la popularidad de *pandas* para la representación de conjuntos de

datos, podemos establecer el argumento `return_type` de `dmatrixes` como `"dataframe"`. Además, dado que los objetos `DataFrame` se comportan como diccionarios, los podemos utilizar para definir los datos como segundo argumento de la función `dmatrixes`:

CELDA 05				
<pre>import pandas as pd df_data = pd.DataFrame(data) y, X = dmatrixes("y ~x1 + x2 + x1:x2", df_data, return_type="dataframe") X</pre>				
	Intercept	x1	x2	x1:x2
0	1.0	6.0	13.0	78.0
1	1.0	7.0	14.0	98.0
2	1.0	8.0	15.0	120.0
3	1.0	9.0	16.0	144.0
4	1.0	10.0	17.0	170.0

Una vez obtenidas las matrices de diseño con *patsy*, requeridas para resolver un modelo estadístico, podemos utilizar un método de mínimos cuadrados como el de NumPy `np.linalg.lstsq`:

CELDA 06				
<pre>beta, res, rank, sval = np.linalg.lstsq(X, y, rcond=None) beta</pre>				
<pre>array([[-3.33333333e-01], [1.66666667e+00], [-6.66666667e-01], [-4.44089210e-16]])</pre>				

O también usar cualquiera de los muchos modelos estadísticos provistos por la biblioteca `statsmodels`.

17.4. Regresión lineal

El modelado estadístico suele involucrar un análisis interactivo de los datos, comenzando con una inspección visual, buscando correlaciones y relaciones funcionales. A partir de esta primera inspección, se propone un modelo estadístico que podría describir los datos. En los casos más simples, las relaciones entre los datos se puede describir con un modelo lineal:

$$Y = kX + d$$

A continuación se procede a determinar los parámetros del modelo (en este caso, k y d). Luego es necesario determinar la calidad del modelo, y finalmente se analizan los residuos (diferencias entre los valores observados y los predichos) para verificar si el modelo propuesto falla en describir algunas características de los datos.

Si los residuos presentan valores altos u *outliers*, o si sugieren una relación funcional diferente a la propuesta, es necesario modificar el modelo. Este procedimiento se repite hasta que los resultados sean satisfactorios.

La biblioteca statsmodels [11] provee diversos modelos estadísticos que comparten el mismo modo de uso, lo que facilita cambiar entre diferentes modelos. Estos modelos estadísticos están representados por clases, que pueden ser inicializadas usando tanto las matrices de diseño como expresiones en patsy, y un *dataframe* u otro objeto de tipo diccionario conteniendo los datos. Los pasos para establecer y analizar un modelo son los siguientes (asumiendo que importamos statsmodels.api como sm y statsmodels.formula.api como smf):



statsmodels - Permite la estimación y evaluación de modelos estadísticos en Python, ofreciendo herramientas para análisis econométricos y estadísticos.

1. Crear una instancia de la clase modelo, por ejemplo, `modelo = sm.MODEL(Y, X)` o `modelo = smf.model(formula, data)`, utilizando las mayúsculas cuando usamos las matrices de diseño y las minúsculas para expresiones de patsy. MODEL o model denotan el nombre de un modelo particular como OLS (mínimos cuadrados ordinarios), GLS (mínimos cuadrados generalizados), etc.
2. Instanciar un modelo no realiza ningún cálculo. Para ajustar el modelo a los datos se utiliza el método `fit`, por ejemplo, `resultados = modelo.fit()`, que realiza el ajuste y devuelve un objeto que contiene atributos y métodos para análisis posteriores.
3. Inspeccionar el resumen de las estadísticas contenidas en el objeto resultado. Este objeto varía levemente entre diferentes modelos, pero la mayoría implementa el método `summary` que produce una salida que describe el resultado del ajuste.
4. Postprocesamiento de los resultados. Además del método `summary`, el objeto resultado tiene métodos y atributos para obtener los parámetros ajustados (`params`), los residuos (`resid`), los valores ajustados (`fittedvalues`) y un método para predecir los valores de las variables dependientes para nuevos valores de las variables independientes (`predict`).
5. Opcionalmente, suele ser útil visualizar el resultado del ajuste. Para ello se puede utilizar Matplotlib o algunas de las rutinas gráficas provistas por la biblioteca statsmodels.

Recorreremos estos pasos utilizando un ejemplo en el cual conocemos la relación entre la variable dependiente y las independientes, y ajustaremos un modelo que genere una predicción cercana a estos datos a los que agregaremos ruido distribuido normalmente.

Vamos a considerar que la relación matemática que vincula las variables independientes x_1 y x_2 con la variable dependiente y está dada por

$$y = 4 + x_1 - 2x_2 + 3x_1x_2 \quad (17.1)$$

Para ello generaremos un conjunto de números aleatorios (estableciendo previamente un valor de la semilla del generador para poder reproducir los valores muestreados) y construiremos a partir de estos valores los correspondientes a la variable dependiente según la ecuación (17.1). En la siguiente celda del *notebook* importamos los módulos necesarios y construimos un *dataframe* de pandas conteniendo los valores de las variables dependientes x_1 y x_2 y la variable dependiente y :

CELDA 1 DE MODELO_LINEAL.IPYNB

```
import numpy as np
import pandas as pd

N = 100 # Cantidad de datos
np.random.seed(13) # Semilla del generador de números aleatorios
x1 = np.random.randn(N)
x2 = np.random.randn(N)
data = pd.DataFrame({"x1": x1, "x2": x2})
def y(x1, x2):
    return 4 + x1 - 2 * x2 + 3 * x1 * x2
data["y_exacto"] = y(x1, x2)
```

En la celda siguiente generamos un array de números aleatorios (epsilon) que sumaremos a la variable dependiente, de modo de simular la presencia de “ruido” en los datos que queremos ajustar:

CELDA 02

```
epsilon = 0.5 * np.random.randn(N)
data["y"] = data["y_exacto"] + epsilon
```

Construimos ahora un primer modelo en el que suponemos que la relación entre las variables independientes y la dependiente es solo la suma de las primeras, omitiendo el término que contiene el producto x_1x_2 de la ecuación (17.1), es decir, $Y = \beta_0 + \beta_1x_1 + \beta_2x_2$ que representamos con la fórmula de patsy “ $y \sim x1 + x2$ ” en la que hacemos uso de los datos “con ruido” de la variable dependiente. Utilizaremos el método de mínimos cuadrados ordinarios (u OLS por su sigla en inglés: *Ordinary Least Squares*) para ajustar el modelo, para lo cual instanciaremos un objeto modelo_1 de la clase `smf.ols`, e invocaremos al método `fit()` de ese objeto. Para ver el resultado del ajuste, invocaremos el método `summary()`:

CELDA 03

```
import statsmodels.formula.api as smf

modelo_1 = smf.ols("y ~ x1 + x2", data)
resultado_1 = modelo_1.fit()
print(resultado_1.summary())
```

```

OLS Regression Results
=====
Dep. Variable:          y      R-squared:          0.094
Model:                OLS      Adj. R-squared:      0.075
Method:             Least Squares      F-statistic:      5.041
Date:               jue, 17 oct 2024      Prob (F-statistic): 0.00827
Time:                18:56:51      Log-Likelihood:    -252.20
No. Observations:      100      AIC:              510.4
Df Residuals:          97      BIC:              518.2
Df Model:               2
Covariance Type:      nonrobust
=====

```

	coef	std err	t	P> t	[0.025	0.975]
Intercept	4.1240	0.311	13.240	0.000	3.506	4.742

x1	0.5549	0.328	1.690	0.094	-0.097	1.207
x2	-1.0186	0.378	-2.693	0.008	-1.769	-0.268
=====						
Omnibus:		26.557	Durbin-Watson:			2.067
Prob(Omnibus):		0.000	Jarque-Bera (JB):			86.445
Skew:		0.811	Prob(JB):			1.69e-19
Kurtosis:		7.256	Cond. No.			1.32
=====						
Notes:						
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.						

La descripción detallada de la salida de `summary()` excede el alcance de este libro, pero señalaremos algunos indicadores de la calidad del ajuste en la primera tabla. Entre ellos, `R-squared` es un estadístico que indica la proporción de variación en la variable dependiente debida a la variación en las variables independientes, y comprende el rango entre 0 y 1, tomando el valor 1 cuando el ajuste es perfecto. Además de estar informado en la salida de `summary()`, está contenido en el atributo `rsquared` de `resultado_1`, tal como se ve en la celda siguiente:

CELDA 04	
<code>resultado_1.rsquared</code>	

0.09414676944969613	

En este caso vemos que el ajuste es muy pobre, ya que apenas llega a 0,094. El valor ajustado de R^2 (`Adj. R-squared`) es una corrección de R^2 que penaliza el número de parámetros en el modelo. Otros indicadores útiles para comparar la calidad de ajustes entre modelos son AIC (*Akaike Information Criterion*) y BIC (*Bayesian Information Criterion*). Estos dos indicadores no nos dicen qué tan bueno es un ajuste individual, pero son útiles para decidir entre diferentes modelos ya que resulta mejor el que tiene valores menores de estos indicadores.

La segunda tabla de `summary()` contiene los coeficientes β_i obtenidos por el ajuste del modelo. Estos coeficientes (o pesos) de la regresión lineal están contenidos en el atributo `params` de `resultado_1` y se devuelve como un objeto `Series` de `pandas`:

CELDA 05	
<code>resultado_1.params</code>	

Intercept	4.124015
x1	0.554879
x2	-1.018580
dtype:	float64

Asumiendo que los residuos están normalmente distribuidos, la columna `std err` ofrece una estimación de los errores estándar de los coeficientes del modelo, así como el estadístico t y su correspondiente p -valor para el test estadístico según el cual la hipótesis nula es que el correspondiente coeficiente es cero. Esta columna entonces ($P > |t|$) nos permite juzgar cuáles variables independientes tienen coeficientes con probabilidad de ser diferente de cero, lo que significa que tienen un poder predictivo significativo. La hipótesis alternativa es que el predictor contribuye a la respuesta. Como es usual, establecemos un umbral $\alpha = 0,05$ o $0,001$, y si el p -valor es menor que el umbral ($P(T \geq |t|) < \alpha$), podemos rechazar la hipótesis nula. El t -test nos permite evaluar

la importancia de los diferentes predictores siempre que los residuos del modelo tengan una distribución normal alrededor de cero. Si esto no es el caso, el *t-test* sugiere la existencia de alguna no linealidad entre las variables y por lo tanto no se puede utilizar para estimar la importancia de los predictores individuales.

Para tener una idea más precisa sobre el supuesto de que los errores están normalmente distribuidos, es necesario analizar los residuos del modelo que ajustamos a los datos. Los errores estándar de los coeficientes son las raíces cuadradas de los elementos diagonales de la matriz de covarianza que resulta de

$$C = \text{cov}(\beta) = \sigma^2(\mathbf{X}\mathbf{X}^T)^{-1}$$

donde $\mathbf{X}$ es la matriz de diseño y σ^2 es la varianza, o error cuadrático medio, de los residuos.

Podemos analizar los residuos para justificar la suposición de distribución normal. Éstos son accesibles mediante el atributo `resid` del objeto `resultado_1`:

CELDA 06

```
resultado_1.resid.head()

0    2.772770
1   -1.074751
2    0.563161
3   -0.057005
4   -0.949921
dtype: float64
```

Podemos invocar la función `normaltest()` de `scipy.stats` para verificar la hipótesis que los residuos constituyen una muestra de una variable aleatoria con distribución normal (hipótesis nula). Si establecemos un $\alpha = 0,001$, y calculamos el *p*-valor:

CELDA 07

```
from scipy import stats

z, p = stats.normaltest(resultado_1.resid)
p

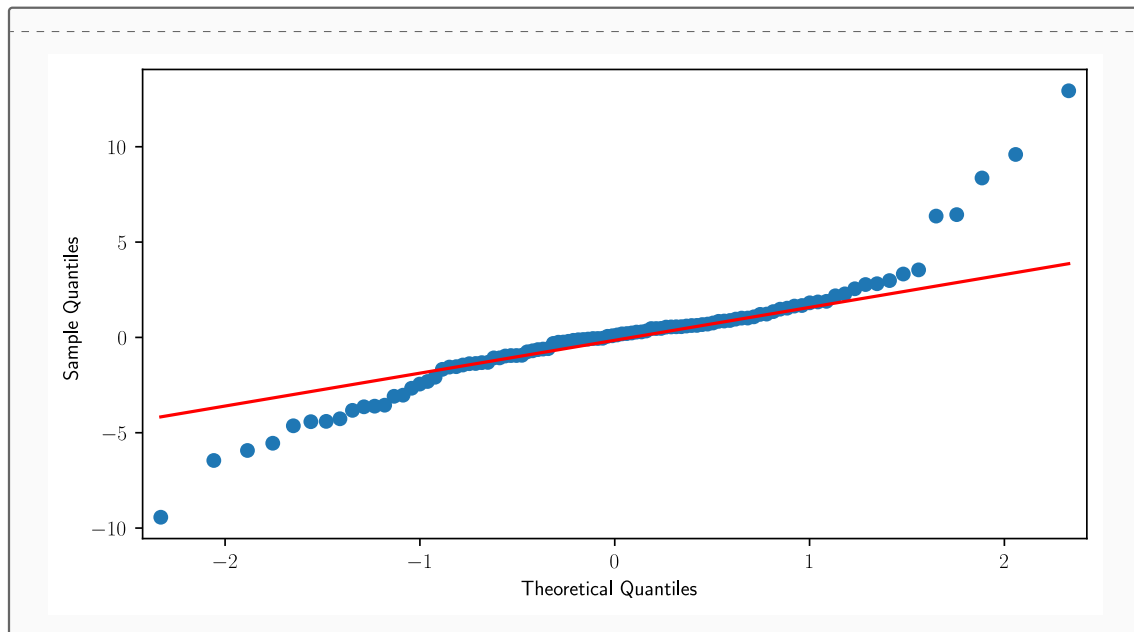
1.7109127008978512e-06
```

vemos que $p < \alpha$, con lo que podemos rechazar la hipótesis nula (es decir, podemos concluir que la suposición de que los residuos están normalmente distribuidos no se cumple). Una forma visual de verificar la normalidad de los residuos es a través de un gráfico Q-Q (*Quantile-Quantile plot*), que compara los cuantiles empíricos de la muestra con los valores teóricos, y que si los valores muestrales están normalmente distribuidos la figura debería aproximarse a una línea recta. Para nuestro caso, podemos generar el gráfico Q-Q mediante `smg.qqplot`:

CELDA 08

```
import matplotlib.pyplot as plt
import statsmodels.graphics.api as smg

fig, ax = plt.subplots(figsize=(8, 4))
smg.qqplot(resultado_1.resid, ax=ax, line='q');
```



Como se puede ver en la salida de la celda 8, los puntos se desvían notablemente de una recta, sugiriendo que es poco probable que los residuos observados sean una muestra de una variable aleatoria distribuida normalmente.

Por último, mencionamos que en la segunda tabla de la salida de `summary()` se muestran los intervalos de confianza para cada predictor. Un intervalo de confianza pequeño indica que el ajuste permite establecer con mayor precisión el valor de estos predictores, mientras que intervalos más grandes indican una mayor incerteza o varianza en estos parámetros. Al igual que los valores de AIC y BIC, los intervalos de confianza resultan útiles a la hora de comparar el grado de ajuste entre diferentes modelos.

Los resultados previos sugieren que el modelo propuesto no es suficiente y que necesitamos mejorarlo. Podemos entonces incluir el término de interacción en la fórmula de patsy y repetir los análisis:

CELDA 09

```

modelo_2 = smf.ols("y ~ x1 + x2 + x1 : x2", data)
resultado_2 = modelo_2.fit()
print(resultado_2.summary())

```

OLS Regression Results

Dep. Variable:		y	R-squared:	0.972
Model:		OLS	Adj. R-squared:	0.971
Method:		Least Squares	F-statistic:	1119.
Date:		jue, 17 oct 2024	Prob (F-statistic):	1.57e-74
Time:		18:56:51	Log-Likelihood:	-77.990
No. Observations:		100	AIC:	164.0
Df Residuals:		96	BIC:	174.4
Df Model:		3		
Covariance Type:		nonrobust		

	coef	std err	t	P> t	[0.025	0.975]

```

-----
Intercept    3.9800    0.055    72.494    0.000    3.871    4.089
x1           0.9716    0.058    16.666    0.000    0.856    1.087
x2          -1.9678    0.069   -28.610    0.000   -2.104   -1.831
x1:x2        2.9938    0.054    55.075    0.000    2.886    3.102
=====
Omnibus:                 1.217    Durbin-Watson:                 2.304
Prob(Omnibus):            0.544    Jarque-Bera (JB):              0.713
Skew:                     0.134    Prob(JB):                      0.700
Kurtosis:                 3.316    Cond. No.                      1.51
=====

```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Ahora vemos que varios indicadores muestran un mejor ajuste del modelo. Para empezar, R^2 está más cerca de la unidad:

CELDA 10	
resultado_2.rsquared	

0.972209883186191	

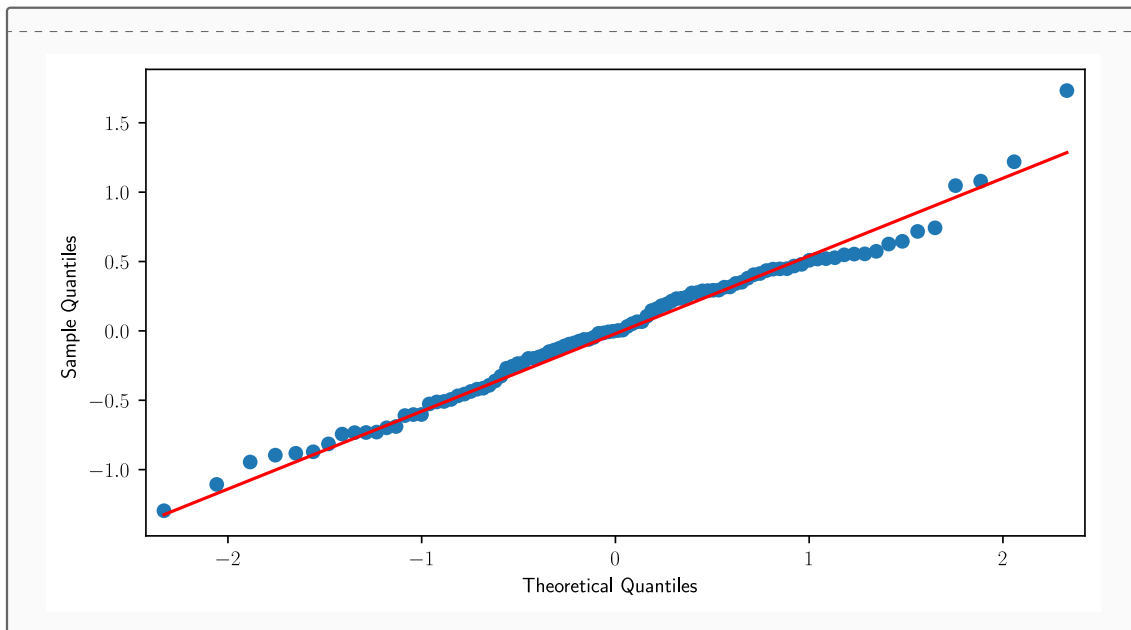
También los valores de AIC y BIC son menores que en el caso anterior, indicando una mejora en la calidad del modelo. Si realizamos un t -test sobre los residuos obtenemos:

CELDA 11	
z, p = stats.normaltest(resultado_2.resid)	
p	

0.5442003027882565	

y ahora el p -valor es mayor que nuestro umbral, con lo que no podemos rechazar la hipótesis nula que indica la normalidad de la distribución de residuos. Esto se visualiza con el gráfico Q-Q, que se aproxima mejor a una recta:

CELDA 12	
fig, ax = plt.subplots(figsize=(8, 4))	
smg.qqplot(resultado_2.resid, ax=ax, line='q');	



Los parámetros de ajustes del modelo mejorado son:

CELDA 13

```
resultado_2.params

Intercept    3.979993
x1           0.971593
x2          -1.967837
x1:x2        2.993805
dtype: float64
```

que son bastante cercanos a los modelos “verdaderos” con los cuales generamos los datos (sin el ruido agregado). Finalmente, podemos comparar los valores que se obtienen mediante la función correcta y los predichos por el modelo en forma gráfica:

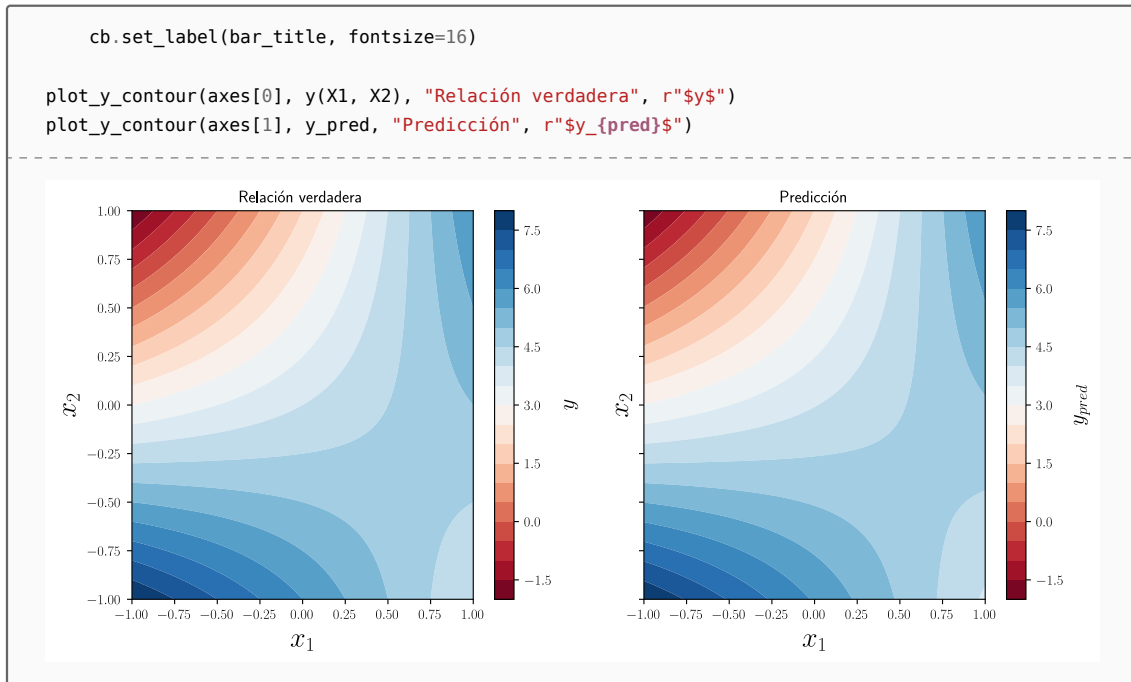
CELDA 14

```
N_new = 100
x = np.linspace(-1, 1, N_new)
X1, X2 = np.meshgrid(x, x)
datos_nuevos = pd.DataFrame({"x1": X1.ravel(), "x2": X2.ravel()})
y_pred = resultado_2.predict(datos_nuevos)
```

CELDA 15

```
y_pred = y_pred.values.reshape(N_new, N_new)
fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey = True)

def plot_y_contour(ax, Y, title, bar_title):
    c = ax.contourf(X1, X2, Y, 20, cmap=plt.cm.RdBu)
    ax.set_xlabel(r"$x_1$", fontsize=20)
    ax.set_ylabel(r"$x_2$", fontsize=20)
    ax.set_title(title)
    cb = fig.colorbar(c, ax=ax)
```



En el panel de la izquierda mostramos los valores obtenidos utilizando la relación inicial verdadera entre las variables independientes x_1 y x_2 , mientras que en el panel de la derecha visualizamos la predicción. De este modo podemos cuantificar en forma visual el grado de precisión de nuestro modelo.

17.5. Regresión discreta

Los modelos de regresión discreta son aquellos en los que la variable dependiente toma valores discretos. El más simple de los casos es el que la variable dependiente y es binaria (solo puede tomar dos valores, que por conveniencia y sin perder generalidad, denotamos por 0 y 1). Por ejemplo, podemos definir que $y = 1$ si un individuo es adulto y $y = 0$ si no lo es; 1 si hace deportes y 0 en caso contrario; 1 si un vehículo tiene motor eléctrico y 0 si no; etc.

Cuando la variable dependiente toma más de dos valores, podemos clasificar los diferentes casos en variables categóricas y no categóricas. Como ejemplo del primer caso, podemos categorizar individuos según su rango etario:

- $y = 1$ si la edad es menor a 20 años,
- $y = 2$ si la edad es entre 20 y 60 años,
- $y = 3$ si la edad es mayor a 60 años,

Como ejemplo de una variable no categórica podemos mencionar el número de unidades vendidas por una empresa durante un día. En este caso y puede tomar los valores $0, 1, 2, \dots$, es decir, valores discretos sin ser una variable categórica.

Los métodos usuales de regresión lineal que vimos hasta ahora no son suficientes, dado que en esos casos la variable de respuesta obtenida es continua, y no puede ser utilizada directamente para modelar una variable que solo puede tomar valores discretos. Sin embargo, es posible

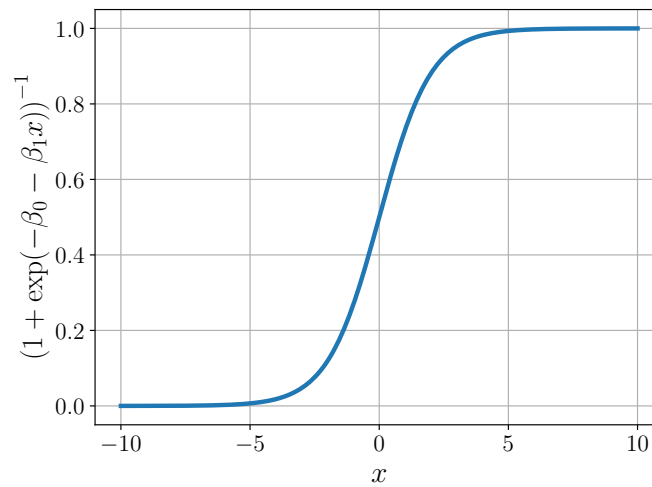


FIGURA 17.2: Representación gráfica de la función logística con los valores $\beta_0 = 0$ y $\beta_1 = 1$.

mapear un predictor lineal mediante una transformación adecuada a un intervalo que se puede interpretar como una probabilidad para diferentes resultados discretos. Para el caso de variables de respuesta binaria, una transformación muy utilizada es la función logística:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$$

o

$$p = (1 + \exp(-\beta_0 - \beta_1 x))^{-1}$$

que mapea $x \in (-\infty, \infty)$ a $p \in (0, 1)$, y cuya representación gráfica se muestra en la figura 17.2. Es decir, que la variable independiente (continua o discreta) x es mapeada a través de los parámetros del modelo lineal β_0 y β_1 , y posteriormente utilizando la función logística transformamos la salida del modelo lineal en una probabilidad p . Si $p < 0,5$ podemos decir que $y = 0$, y si $p \geq 0,5$, la predicción es que $y = 1$.

La biblioteca statsmodels ofrece varios métodos para regresiones discretas, incluyendo la clase `Logit` para regresiones binarias, la clase para regresión logística multinomial `MNLogit` (para más de dos categorías), y la clase `Poisson` para variables no categóricas con distribución de Poisson (enteros positivos). Vamos a desarrollar un ejemplo de regresión discreta binaria utilizando el famoso conjunto de datos de flor Iris de Fisher². Este conjunto contiene 50 muestras de cada una de las tres especies de Iris: setosa, virginica y versicolor. En cada muestra se midieron el largo y el ancho del sépalos y del pétalo (en centímetros). En el ejemplo, tomaremos un subconjunto de los datos pertenecientes solo a dos especies e intentaremos construir un modelo que prediga, en términos de los valores de las dimensiones del sépalos y del pétalo, a cuál de las dos especies corresponden esos valores.

En la primera celda utilizamos el submódulo `datasets` de statsmodels cuya función `get_rdataset` provee acceso a conjuntos de datos utilizados en el popular lenguaje R³.

² Se puede leer más sobre este conjunto de datos en la entrada de [Wikipedia](#).

³ Los conjuntos de datos están disponibles [aquí](#).

CELDA 1 DE DISCRETO.IPYNB

```
import statsmodels.api as sm

df = sm.datasets.get_rdataset("iris").data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype
---  ---
 0   Sepal.Length    150 non-null    float64
 1   Sepal.Width     150 non-null    float64
 2   Petal.Length    150 non-null    float64
 3   Petal.Width     150 non-null    float64
 4   Species         150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

Extrayendo los valores únicos en la columna Species de nuestro *dataset*, vemos que están las tres especies de Iris:

CELDA 02

```
df.Species.unique()

array(['setosa', 'versicolor', 'virginica'], dtype=object)
```



Siméon Denis Poisson - Físico y matemático francés (1781-1840) al que se le conoce por sus diferentes trabajos en el campo de la electricidad y por sus publicaciones acerca de la geometría diferencial y la teoría de probabilidades.

Construimos ahora un nuevo *dataframe* que constituye un subconjunto del original, incluyendo solo las especies versicolor y setosa, y reasignamos los valores de la columna Species a los valores 0 o 1 para versicolor y virginica respectivamente. Finalmente, antes de proceder con la construcción del modelo, es necesario renombrar las columnas del *dataframe* reemplazando los puntos por otro carácter de modo que Python pueda interpretar correctamente su nombre. Si pasamos como nombre de variable en expresiones Patsy "**Sepal.Length**", Python intentará encontrar una variable llamada Sepal que tiene un atributo Length. Evitamos esto reemplazando los puntos por, por ejemplo, el carácter "_":

CELDA 03

```
df_subconjunto = df[df.Species.isin(["versicolor", "virginica"])]
df_subconjunto.Species = df_subconjunto.Species.map({"versicolor": 0, "virginica": 1})
df_subconjunto.columns = [c.replace('.', '_') for c in df_subconjunto.columns]
df_subconjunto.head()
```

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Species
50	7.0	3.2	4.7	1.4	0
51	6.4	3.2	4.5	1.5	0
52	6.9	3.1	4.9	1.5	0
53	5.5	2.3	4.0	1.3	0
54	6.5	2.8	4.6	1.5	0

Podemos construir ahora el modelo utilizando la clase `Logit`, y realizamos el ajuste con su método `fit()` utilizando como variables independientes las medidas del pétalo (largo y ancho):

CELDA 04

```
import statsmodels.formula.api as smf

modelo = smf.logit("Species ~ Petal_Length + Petal_Width", data=df_subconjunto)
resultado = modelo.fit()
```

Optimization terminated successfully.
Current function value: 0.102818
Iterations 10

Como ya vimos, los resultados del ajuste se pueden mostrar con el método `summary()`:

CELDA 05

```
print(resultado.summary())
```

Logit Regression Results

```
=====
```

Dep. Variable:	Species	No. Observations:	100
Model:	Logit	Df Residuals:	97
Method:	MLE	Df Model:	2
Date:	jue, 17 oct 2024	Pseudo R-squ.:	0.8517
Time:	18:56:00	Log-Likelihood:	-10.282
converged:	True	LL-Null:	-69.315
Covariance Type:	nonrobust	LLR p-value:	2.303e-26

```
=====
```

	coef	std err	z	P> z	[0.025	0.975]
-----	-----	-----	-----	-----	-----	-----
Intercept	-45.2723	13.612	-3.326	0.001	-71.951	-18.594
Petal_Length	5.7545	2.306	2.496	0.013	1.235	10.274
Petal_Width	10.4467	3.756	2.782	0.005	3.086	17.808
-----	-----	-----	-----	-----	-----	-----

Possibly complete quasi-separation: A fraction 0.34 of observations can be perfectly predicted. This might indicate that there is complete quasi-separation. In this case some parameters will not be identified.

Al igual que en los ejemplos precedentes, no haremos un análisis exhaustivo de toda la información que muestra `summary()` sino que solo centraremos nuestra atención en los parámetros del ajuste (`Intercept`, `Petal_Length` y `Petal_Width`) ya que éstos nos permitirán realizar predicciones sobre nuevos valores de mediciones del largo y ancho del pétalo. Podemos simular nuevas mediciones generando valores con variables aleatorias dentro de los rangos de variación de estas magnitudes, los cuales registramos en un nuevo *dataframe* `df_nuevo`. Utilizando el ajuste del modelo podemos predecir a qué especie pertenece cada “nuevo” valor, tomando el criterio de asignar el valor 1 o 0 según la predicción sea mayor o menor a 0,5:

CELDA 06

```
import numpy as np
import pandas as pd
```

```

n_datos = 20
amplitud = 1.2
np.random.seed(1729)
df_nuevo = pd.DataFrame({
    "Petal_Length": np.random.randn(n_datos) * amplitud + 5,
    "Petal_Width": np.random.rand(n_datos) * amplitud + 1.1,
})
df_nuevo["Species_pred"] = resultado.predict(df_nuevo)
print(df_nuevo["Species_pred"].head(3))
df_nuevo["Species_pred"] = (df_nuevo["Species_pred"] > 0.5).astype(int)
df_nuevo["Species_pred"].head(3)

```

```

0    0.004483
1    0.078261
2    0.999989
Name: Species_pred, dtype: float64

```

```

0    0
1    0
2    1
Name: Species_pred, dtype: int64

```

Para tener una apreciación visual del desempeño del modelo en la clasificación de las especies y en la predicción de los resultados, podemos realizar un gráfico que muestre los valores originales (de los cuales conocemos a qué especie pertenecen), la recta que separa ambas clasificaciones, y las predicciones sobre los nuevos valores obtenidos aleatoriamente:

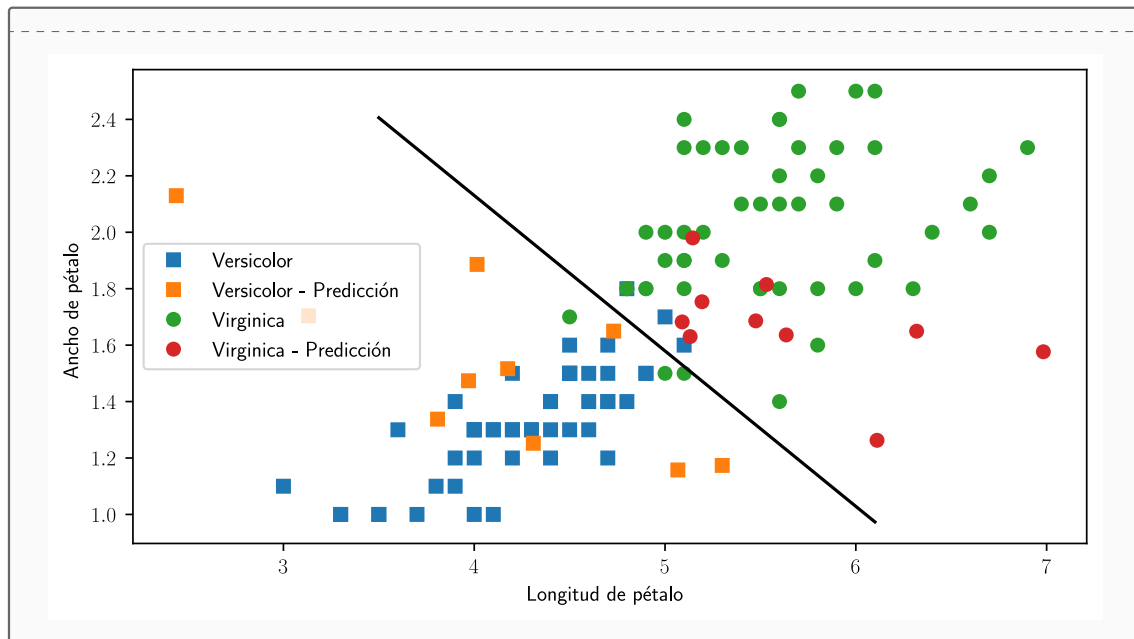
CELDA 07

```

import matplotlib.pyplot as plt

parametros = resultado.params
# Parámetros de la recta que define la separación de especies
a = -parametros['Intercept']/parametros['Petal_Width']
b = -parametros['Petal_Length']/parametros['Petal_Width']
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(df_subconjunto[df_subconjunto.Species == 0].Petal_Length.values,
        df_subconjunto[df_subconjunto.Species == 0].Petal_Width.values,
        's', label='Versicolor')
ax.plot(df_nuevo[df_nuevo.Species_pred == 0].Petal_Length.values,
        df_nuevo[df_nuevo.Species_pred == 0].Petal_Width.values,
        's', label='Versicolor - Predicción')
ax.plot(df_subconjunto[df_subconjunto.Species == 1].Petal_Length.values,
        df_subconjunto[df_subconjunto.Species == 1].Petal_Width.values,
        'o', label='Virginica')
ax.plot(df_nuevo[df_nuevo.Species_pred == 1].Petal_Length.values,
        df_nuevo[df_nuevo.Species_pred == 1].Petal_Width.values,
        'o', label='Virginica - Predicción')
x = np.array([3.5, 6.1])
ax.plot(x, a + b * x, 'k')
ax.set_xlabel('Longitud de pétalo')
ax.set_ylabel('Ancho de pétalo')
plt.legend()
plt.show()

```



Vemos que algunos valores originales próximos a la recta de separación se encuentran en el lado erróneo de la clasificación, pero las predicciones sobre los valores generados aleatoriamente son muy buenas.

Como último ejemplo de modelado sobre variables discretas, intentaremos ahora realizar un ajuste sobre un conjunto de datos que registra el número de visitas al médico en las últimas dos semanas para una muestra de 5.190 adultos de la Encuesta de Salud Australiana 1977-78⁴. Este y otros conjuntos de datos acerca del uso de servicios de salud fueron objeto de análisis por A. Cameron y P. Trivedi en el contexto de un modelo económico [12].

El modelo discreto más simple que podemos considerar para modelar datos que representan conteos es el de Poisson, que expresa la probabilidad de un número dado de eventos que ocurren en un intervalo de tiempo (o espacio) fijo, en el caso en que estos eventos ocurran con una frecuencia media constante e independientemente del intervalo de tiempo desde el último evento. Si suponemos que los eventos ocurren aleatoriamente en el tiempo de modo que se cumplan las siguientes condiciones:

- La probabilidad de ocurrencia de al menos un evento en un intervalo dado es proporcional a la longitud del intervalo.
- La probabilidad de dos o más ocurrencias de eventos en un intervalo muy pequeño es despreciable.
- El número de ocurrencias de eventos en intervalos de tiempos disjuntos son mutuamente independientes.

entonces la distribución de probabilidad del número de ocurrencias de eventos en un intervalo de tiempo fijo es una distribución de Poisson con media $\mu = \lambda t$, donde λ es la frecuencia de ocurrencia del evento por unidad de tiempo y t es la longitud del intervalo de tiempo. Un proceso

⁴ El conjunto de datos se puede descargar desde [este enlace](#).

que satisface las tres condiciones mencionadas se denomina proceso de Poisson. Ejemplos modelados como procesos de Poisson que podemos mencionar son: la cantidad de goles anotados en un partido de fútbol, potenciales de acción emitidos por una neurona o la cantidad de vehículos que pasan por una calle.

Formalmente, si Y_i denota el número de ocurrencias para el i -ésimo de N individuos, de un evento de interés en un intervalo dado de tiempo, la densidad de probabilidad está dada por:

$$P(Y_i = y_i) = \frac{\lambda_i^{y_i} e^{-\lambda_i}}{y_i!}, \quad y_i = 0, 1, 2, \dots, \quad i = 1, 2, \dots, N$$

donde y_i es el valor obtenido de la variable aleatoria. Esta es una distribución con un solo parámetro en la cual el valor medio y varianza de Y_i son ambos iguales a λ_i . Para incorporar un conjunto de variables independientes X_{ij} ($j = 1, \dots, K$), incluyendo un término constante, el parámetro λ resulta:

$$\lambda_i = \exp(\mathbf{X}_i \boldsymbol{\beta})$$

Volviendo al ejemplo, vemos en la celda 8 que leemos el archivo `DoctorVisits.csv` y almacenamos sus datos en el `dataframe` `df`. Invocando el método `info()` podemos ver los nombres de las columnas, la cantidad de valores registrados y los tipos de datos. En la celda siguiente podemos ver que la columna `'visits'` contiene 10 valores y la cantidad de veces que aparece cada uno de ellos en los datos.

CELDA 08	
<pre>df = pd.read_csv('DoctorVisits.csv', header=0, index_col=[0]) df.info()</pre>	
<pre><class 'pandas.core.frame.DataFrame'> Index: 5190 entries, 1 to 5190 Data columns (total 12 columns): # Column Non-Null Count Dtype --- --- 0 visits 5190 non-null int64 1 gender 5190 non-null object 2 age 5190 non-null float64 3 income 5190 non-null float64 4 illness 5190 non-null int64 5 reduced 5190 non-null int64 6 health 5190 non-null int64 7 private 5190 non-null object 8 freepoor 5190 non-null object 9 freerepat 5190 non-null object 10 nchronic 5190 non-null object 11 lchronic 5190 non-null object dtypes: float64(2), int64(4), object(6) memory usage: 527.1+ KB</pre>	
CELDA 09	
<pre>df['visits'].value_counts()</pre>	
<pre>visits 0 4141 1 782</pre>	

```

2    174
3     30
4     24
7     12
6     12
5      9
8      5
9      1
Name: count, dtype: int64

```

En la celda 10 instanciamos el objeto modelo especificando que es un modelo de Poisson en el cual la columna '*visits*' del *dataframe* constituye nuestra variable dependiente, a la que queremos explicar solo a través de una constante, sin relacionarla aún con otros campos del *dataframe*. A continuación realizamos el ajuste del modelo con el método `fit()` asignándolo al objeto resultado, y mostramos los resultados del dicho ajuste mediante el método `summary()`:

CELDA 10						
<pre> modelo = smf.poisson("visits ~ 1", data=df) resultado = modelo.fit() resultado.summary() </pre>						
Optimization terminated successfully. Current function value: 0.767475 Iterations 1						
Dep. Variable:	visits	No. Observations:	5190			
Model:	Poisson	Df Residuals:	5189			
Method:	MLE	Df Model:	0			
Date:	jue, 17 oct 2024	Pseudo R-squ.:	1.110e-16			
Time:	18:56:02	Log-Likelihood:	-3983.2			
converged:	True	LL-Null:	-3983.2			
Covariance Type:	nonrobust	LLR p-value:	nan			
	coef	std err	z	P> z 	[0.025	0.975]
Intercept	-1.1982	0.025	-47.416	0.000	-1.248	-1.149

El resultado que estamos buscando aquí es el valor de Intercept, que es el que podemos vincular con el parámetro λ de la distribución. En la celda siguiente calculamos este parámetro y generamos tres distribuciones de Poisson: una con el parámetro recién obtenido, y dos utilizando los valores extremos del intervalo de confianza del 95 % de Intercept informados en la celda 10:

CELDA 11						
<pre> from scipy import stats lam = np.exp(resultado.params) print(lam) X = stats.poisson(lam) X_ci_l = stats.poisson(np.exp(resultado.conf_int().values)[0,0]) </pre>						

```
X_ci_u = stats.poisson(np.exp(resultado.conf_int().values)[0,1])
```

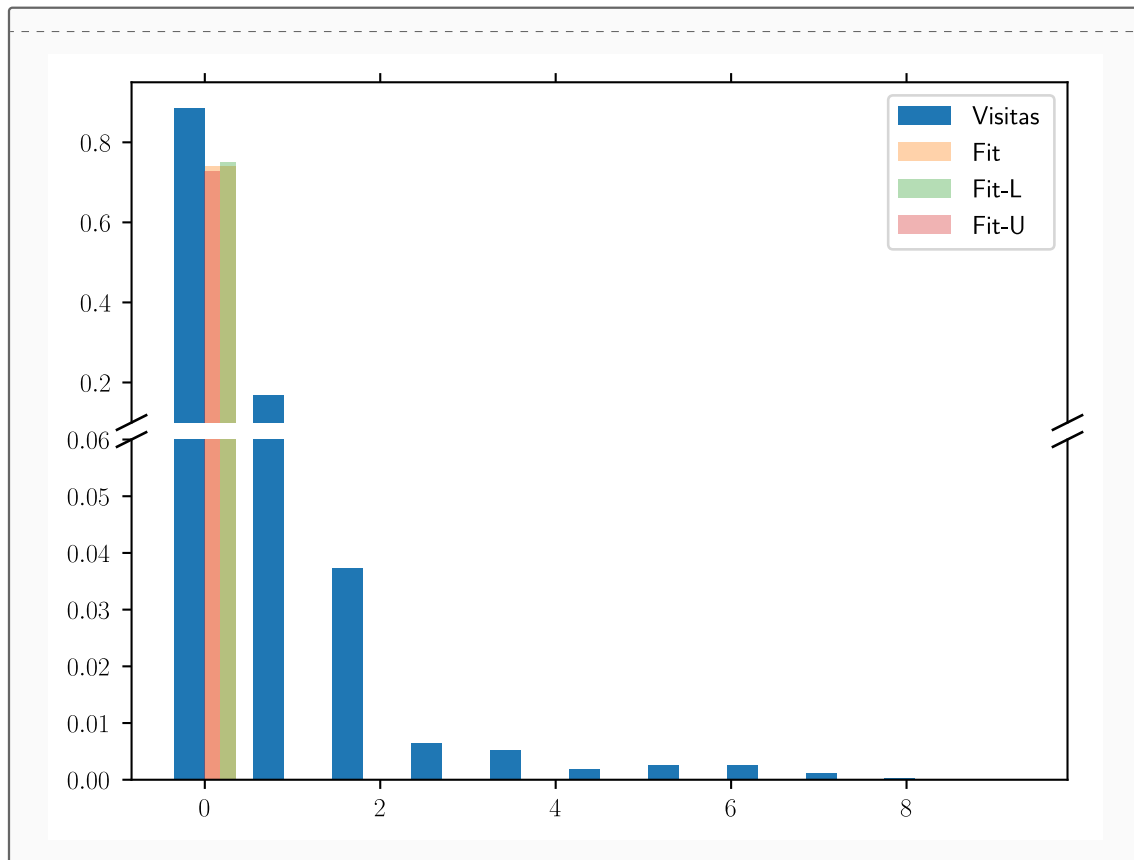
```
Intercept    0.301734
dtype: float64
```

Utilizando estas tres distribuciones, compararemos el histograma de visitas de los datos (normalizado con el número total de visitas) con las funciones de masa de probabilidad de las distribuciones, obtenidas mediante el método `pmf()`⁵. Generamos esta comparación en forma gráfica, como se ve en la celda a continuación:

CELDA 12

```
v, k = np.histogram(df.visits, density=True)
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
fig.subplots_adjust(hspace=0.05)
width = 0.35
for ax in (ax1, ax2):
    ax.bar(k[:-1]-width/2, v, width, align='center', label='Visitas', alpha=1)
    ax.bar(k+width/2, X.pmf(k), width, align='center', label='Fit', alpha=0.35)
    ax.bar(k+width/2, X_ci_l.pmf(k), width/2, align='edge', label='Fit-L', alpha=0.35)
    ax.bar(k+width/2, X_ci_u.pmf(k), -width/2, align='edge', label='Fit-U', alpha=0.35)
ax1.set_ylim(0.1, 0.95)
ax2.set_ylim(0, 0.06)
ax1.spines.bottom.set_visible(False)
ax2.spines.top.set_visible(False)
ax1.xaxis.tick_top()
ax1.tick_params(labeltop=False)
ax2.xaxis.tick_bottom()
d = .5 # proporción de extensión vertical a horizontal de la marca de separación
kwargs = dict(marker=[(-1, -d), (1, d)], markersize=12,
               linestyle="none", color='k', mec='k', mew=1, clip_on=False)
ax1.plot([0, 1], [0, 0], transform=ax1.transAxes, **kwargs)
ax2.plot([0, 1], [1, 1], transform=ax2.transAxes, **kwargs)
ax1.legend()
plt.show()
```

⁵ La función de masa de probabilidad devuelve la probabilidad de que una variable aleatoria discreta sea exactamente igual a un valor dado.



Podemos ver que los valores del histograma no están comprendidos entre las distribuciones con los límites superior e inferior del intervalo de confianza del parámetro ajustado, por lo que debemos rechazar la hipótesis de que el número de visitas al médico en las últimas dos semanas es un proceso de Poisson. No tener éxito en el ajuste de un modelo a un conjunto de datos es habitual durante el proceso de modelado estadístico, y tal vez podríamos obtener mejores resultados si intentamos explicar nuestra variable dependiente con algunas de las columnas del *dataframe*. Por ejemplo, podríamos reemplazar la expresión del modelo en la celda 10 por `"visits ~ 1 + age + illness * income"`. Dejamos aquí al lector la tarea de explorar estas posibilidades, que resultan bastante más complejas que el caso que usamos como ejemplo.

17.6. Series temporales

Una serie temporal consiste en el registro de datos sucesivos en el tiempo. Las técnicas utilizadas para analizar series temporales tratan de identificar patrones de los datos en el pasado e intentan anticipar (o predecir) los valores futuros. Estos datos se encuentran ordenados cronológicamente:

$$y_0, y_1, \dots, y_{t-2}, y_{t-1}, y_t$$

El análisis de series temporales presenta una característica distintiva respecto de los análisis realizados hasta aquí: no se puede considerar que un conjunto de observaciones de una serie temporal representen una muestra de valores aleatorios de una población, ya que existe una fuerte correlación entre observaciones que están cercanas en el tiempo. Por otra parte, se pueden

identificar los valores futuros de una serie temporal como las variables dependientes, que son predichas por los valores pasados de la serie (variables independientes). Esto se refuerza cuando existen autocorrelaciones como ciclos diarios o anuales, o fuertes tendencias que se mantienen en el tiempo. Ejemplos de series temporales son las observaciones climáticas, orográficas, poblacionales, precios de acciones, etc.

El modelo más simple que podemos considerar es el autoregresivo AR (“*autoregressive model*”), que utiliza una combinación lineal de p valores pasados para estimar el presente. La expresión matemática para el modelo AR(p) es:

$$y_t = \phi_0 + \sum_{n=1}^p \phi_n y_{t-n} + \varepsilon_t$$

Aquí, p es el orden del modelo y ε_t representa ruido blanco sin correlación. Los modelos autoregresivos se aplican a procesos estacionarios, es decir, aquellos en los cuales sus propiedades estadísticas no varían en el tiempo. Esta condición impone restricciones sobre los valores que pueden tener los coeficientes ϕ_i .

Otro algoritmo estadístico simple para modelar series temporales es el de la media móvil MA (“*moving average*”), que utiliza la dependencia entre una observación dada y los residuos calculados a partir de observaciones previas. Estos residuos consisten en la diferencia entre los valores registrados en el instante t y el promedio (o media móvil) de un subconjunto precedente de q datos. Matemáticamente el modelo MA(q) se expresa como:

$$y_t = \mu + \sum_{n=1}^q \theta_n \varepsilon_{t-n} + \varepsilon_t$$

donde θ_i son los parámetros del modelo, μ es la media de la serie y ε_i los residuos.

Estos abordajes pueden combinarse dando origen al modelo ARMA(p, q), pero tal como mencionamos, solo puede aplicarse a series que sean estacionarias. Vamos a intentar modelar ahora una serie temporal que registra mediciones atmosféricas de CO₂ desde 1958 en el Observatorio Mauna Loa [13], en partes por millón (ppm). Para ello utilizaremos las herramientas disponibles en el módulo statsmodels.

En la primera celda leemos los datos desde los *datasets* disponibles en statsmodels, podemos ver que obtenemos un *dataframe* de pandas con 2284 registros, ordenados por fecha:

CELDA 1 DE CO2.IPYNB

```
import statsmodels.api as sm

data = sm.datasets.co2.load_pandas()
co2 = data.data
print(co2.info())
print(co2.head())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2284 entries, 1958-03-29 to 2001-12-29
Freq: W-SAT
Data columns (total 1 columns):
#   Column  Non-Null Count  Dtype
---  -
0    co2      2225 non-null    float64
```

```

dtypes: float64(1)
memory usage: 35.7 KB
None
      co2
1958-03-29  316.1
1958-04-05  317.3
1958-04-12  317.6
1958-04-19  317.5
1958-04-26  316.4

```

Antes de comenzar el análisis, vamos a transformar nuestra serie temporal original en otra cuyos índices temporales sean el primer día de cada mes conteniendo el valor medio de los datos del mes correspondiente. Además buscamos fechas sin registros (encontramos 5) y completamos esos registros artificialmente con el valor precedente. Finalmente graficamos la serie temporal:

```

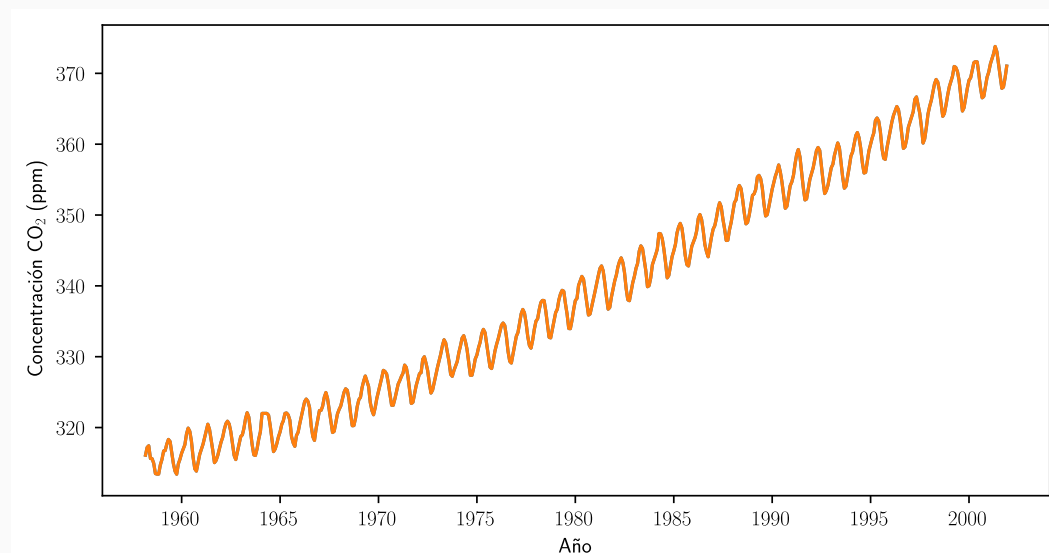
import matplotlib.pyplot as plt

y = co2['co2'].resample('MS').mean()
print(y.isnull().sum())
y = y.bfill()
fig, ax = plt.subplots(1, 1, figsize=(8, 4))
ax.plot(y)
plt.xlabel('Año')
plt.ylabel('Concentración CO2 (ppm)')
plt.plot(y)

```

5

[<matplotlib.lines.Line2D at 0x772663993560>]



De este modo, finalizamos con una serie temporal con 526 valores. En la figura se observa claramente que existe una tendencia creciente de la concentración de CO₂ con el tiempo, a la que se le superpone una oscilación estacional. Además de la inspección visual, es útil disponer

de una herramienta cuantitativa para determinar la estacionalidad de una serie temporal. En nuestro caso, haremos uso del *test* de Dickey-Fuller aumentado⁶ cuya hipótesis nula consiste en que la serie no es estacionaria, por lo que al obtener el *p*-valor del *test* por encima del valor 0,05 no podemos asegurar que la serie no es estacionaria con el 95 % de confianza. En la celda siguiente definimos una función que calcular este *p*-valor y nos informa para distintos valores críticos:

CELDA 03

```
from statsmodels.tsa.stattools import adfuller

def check_stationary(values):
    dfctest = adfuller(values, autolag='AIC')
    print("Test statistic = {:.3f}".format(dfctest[0]))
    print("P-value = {:.3f}".format(dfctest[1]))
    print("Critical values :")
    mensaje = "\t{}: {} - Los datos {} estacionarios con {}% de confianza"
    for k, v in dfctest[4].items():
        print(mensaje.format(k, v, "no son" if v < dfctest[0] else "son", 100-int(k[:-1])))
```

Aplicamos esta función con nuestros datos, y obtenemos obviamente lo que vimos en la figura:

CELDA 04

```
check_stationary(y)
```

```
Test statistic = 2.360
P-value = 0.999
Critical values :
    1%: -3.4432119442564324 - Los datos no son estacionarios con 99% de confianza
    5%: -2.8672126791646955 - Los datos no son estacionarios con 95% de confianza
   10%: -2.569791324979607 - Los datos no son estacionarios con 90% de confianza
```

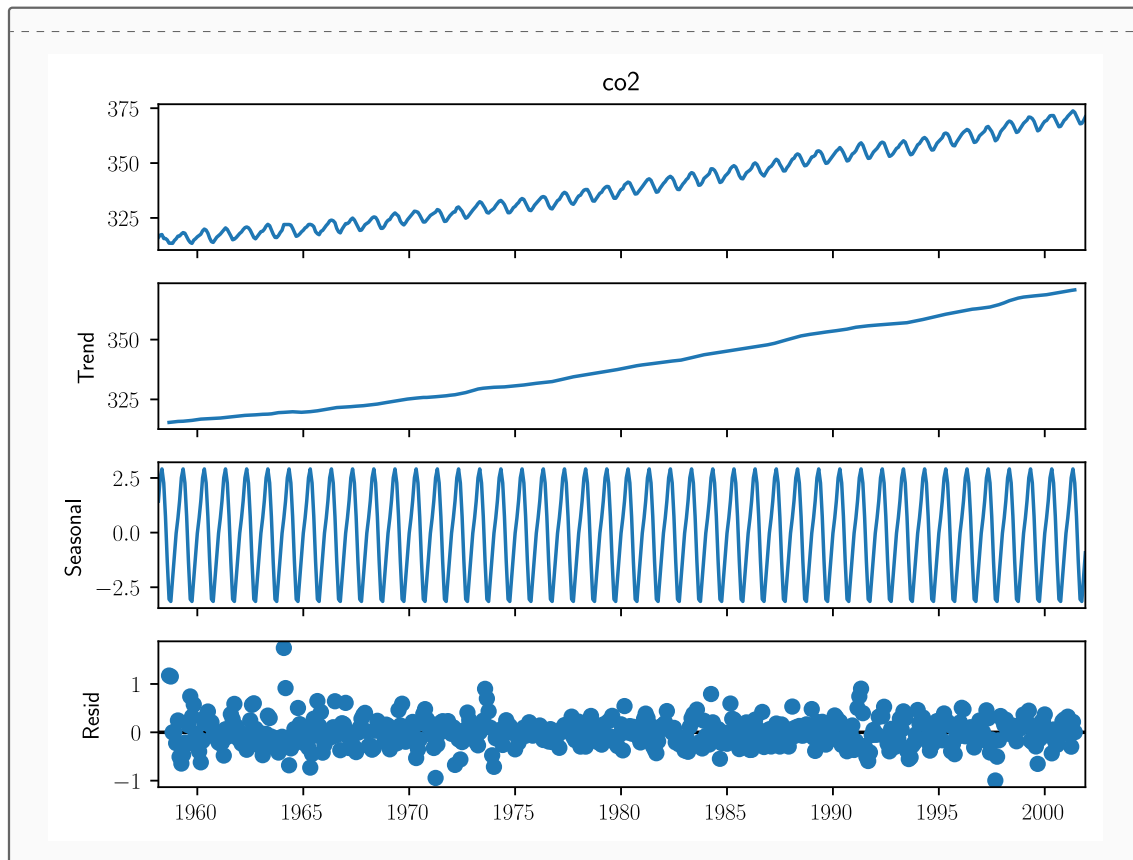
Una herramienta práctica para descomponer nuestra serie original en términos de tendencia, estacionales y residuos, es la función `seasonal_decompose()` de `statsmodels.tsa.seasonal`. Usando nuestros datos y como argumento, y utilizando un método aditivo para la descomposición, obtenemos:

CELDA 05

```
from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(y, model='additive', period=12)
result.plot();
```

⁶ Ver la siguiente [entrada](#) en Wikipedia.



Podemos entonces repetir nuestro *test* sobre la parte estacional para ver si es estacionario, lo que se confirma claramente:

CELDA 06

```
check_stationary(result.seasonal)
```

Test statistic = -43337046870601.086
P-value = 0.000
Critical values :

- 1%: -3.4432119442564324 - Los datos son estacionarios con 99% de confianza
- 5%: -2.8672126791646955 - Los datos son estacionarios con 95% de confianza
- 10%: -2.569791324979607 - Los datos son estacionarios con 90% de confianza

Es posible transformar una serie temporal no estacionaria en una que sí lo es a través de calcular diferencias entre los sucesivos valores. Si, por ejemplo, la línea de tendencia es lineal, generar una nueva serie temporal a través de la primera diferencia $\bar{y}_t = y_t - y_{t-1}$ produce una serie estacionaria, a la cual le podemos aplicar un modelo ARMA. Aplicando esto a nuestra serie temporal obtenemos:

CELDA 07

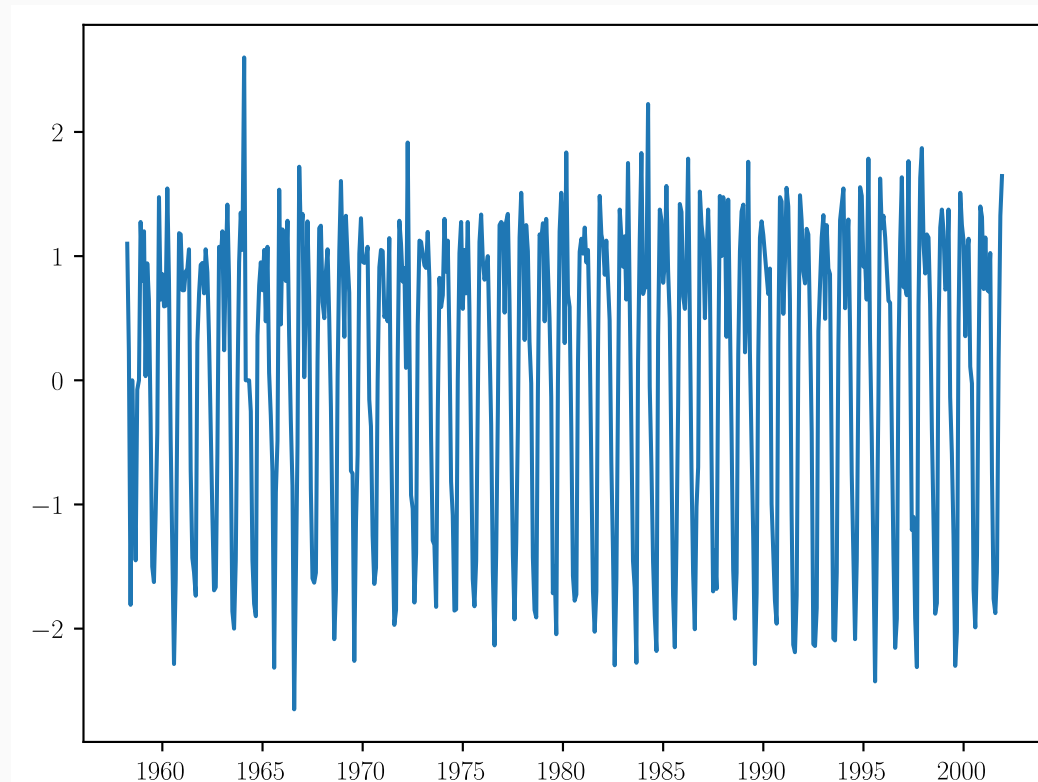
```
y_1 = y.diff()
check_stationary(y_1[1:])
plt.plot(y_1)
```

```

Test statistic = -5.063
P-value = 0.000
Critical values :
    1%: -3.4432119442564324 - Los datos son estacionarios con 99% de confianza
    5%: -2.8672126791646955 - Los datos son estacionarios con 95% de confianza
    10%: -2.569791324979607 - Los datos son estacionarios con 90% de confianza

[<matplotlib.lines.Line2D at 0x77268045ffb0>]

```



Podemos generar un modelo que contemple la “desestacionalización” mediante diferencias de orden d con un modelo $\text{ARIMA}(p, d, q)$, donde $I(d)$ es el preprocesamiento que realiza el cálculo de las diferencias.

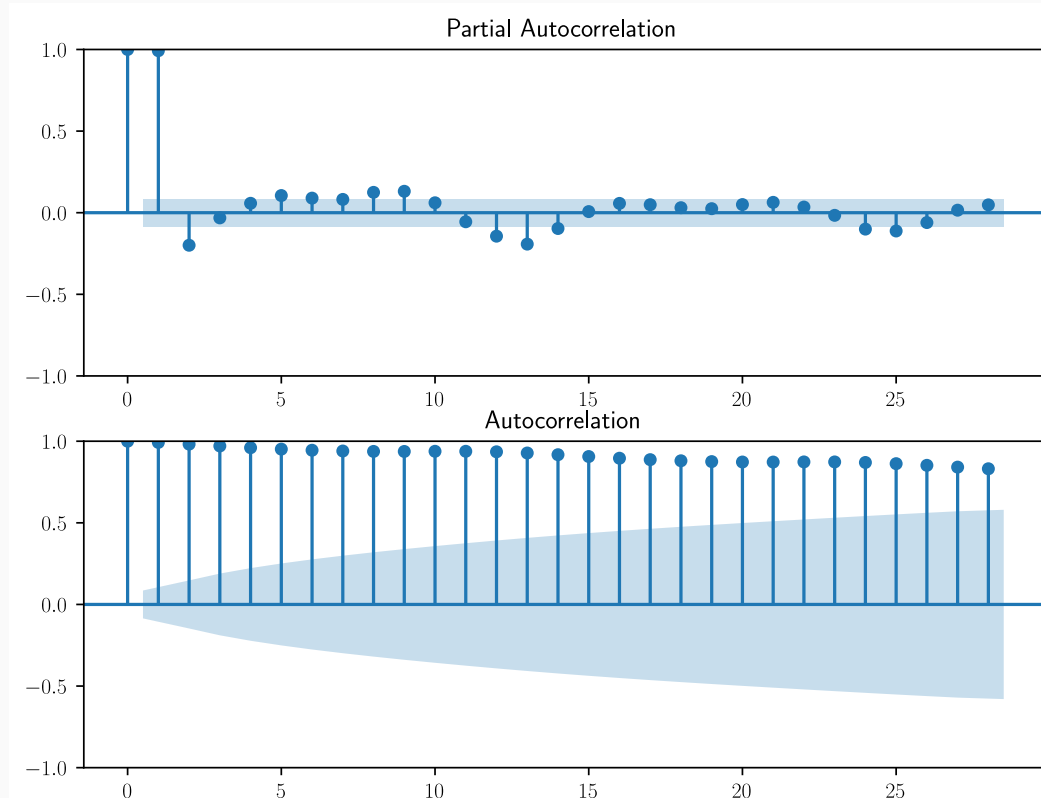
El tratamiento de series temporales con componentes estacionales se puede realizar directamente con un modelo SARIMAX , con considera además de la no estacionalidad, la posibilidad de incorporar otras variables exógenas al modelo. Intentaremos entonces trabajar sobre nuestra serie original ajustándola con un modelo SARIMAX que requiere, además de los parámetros p , d y q , parámetros adicionales para tratar con la componente estacional: P , Q , D y m , siendo este último parámetro el número de pasos temporales que comprende un período estacional.

Una forma de estimar estos parámetros es utilizando los gráficos de las funciones de autocorrelación (ACF) y autocorrelación parcial (PACF). La ACF es la correlación de una serie temporal con sí misma en función del retraso o *lag*, mientras que la PACF es la correlación entre la serie temporal y una versión retrasada de sí misma, sustrayendo el efecto de la correlación de *lags* menores. En el caso de nuestro ejemplo, podemos graficar estas funciones con las herramientas provistas por statsmodels:

CELDA 08

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 6))
plot_pacf(y, method='ywmm', ax=ax1);
plot_acf(y, ax=ax2);
```



En estas figuras, las zonas sombreadas representan el intervalo de confianza del 95 %, por lo que los valores que quedan dentro de estas regiones podrían considerarse nulos. Se puede ver que para la serie analizada existen fuertes correlaciones de período 12 (que representan un año en nuestros datos) en la PACF, y un decaimiento muy suave para la función de autocorrelación. Se suele utilizar como parámetro p el último valor del lag de la PACF antes del primero que se anula, y el correspondiente lag de la ACF para determinar q . La figura de la PACF sugiere entonces usar un valor de $p = 2$, pero no es concluyente para determinar d debido a que el decaimiento de la función es muy lento.

Entonces podemos recurrir a otros indicadores para determinar los parámetros del modelo. Al realizar un ajuste $\text{SARIMAX}(p, d, q)(P, D, Q)_m$, obtenemos los valores de AIC (*Akaike Information Criterion*) y BIC (*Bayesian Information Criterion*). En ambos casos, menores valores de estos indicadores sugieren mejores modelos. Tanto AIC como BIC penalizan modelos con muchos parámetros, aunque difieren en la forma en que penalizan la complejidad del mismo. BIC penaliza órdenes adicionales del modelo más que AIC, por lo que el criterio BIC suele sugerir modelos más simples que AIC. En muchas ocasiones, ambos sugieren el mismo modelo, pero cuando no lo hacen la elección depende de las prioridades: si queremos identificar un mejor mo-

delo predictivo, AIC es una mejor elección, mientras que si lo que buscamos identificar un buen modelo explicativo, entonces podemos basar nuestra decisión en el que produzca el menor valor de BIC.

Vamos a realizar entonces una búsqueda de los parámetros en una grilla, evaluando los mismos para los valores 0, 1 y 2, lo que nos da un total de $3^6 = 729$ modelos. Almacenaremos los valores de los parámetros, junto con los resultados obtenidos para AIC y BIC en una lista, y luego examinaremos las combinaciones de parámetros con menores valores de AIC y BIC.

Previamente, dividiremos nuestra serie temporal en un conjunto de datos para el ajuste del modelo (`y_train`) conteniendo el primer 85 % del total de datos, y reservaremos el restante 15 % (`y_valid`) para contrastar las predicciones obtenidas por nuestro mejor modelo con los datos registrados, de modo de tener una medida de cuán buenas son las predicciones:

CELDA 09

```
n_train = int(0.85 * y.size)
n_samples = y.size
n_valid = n_samples - n_train
y_train, y_valid = y[:n_train], y[n_train:]
print(f'Número de muestras: {n_samples}, entrenamiento: {n_train}, validación: {n_valid}')
```

Número de muestras: 526, entrenamiento: 447, validación: 79

A continuación generamos la búsqueda, iterando sobre todas las tuplas (p, d, q) y (P, D, Q) con valores entre 0 y 2 (dejamos fijo $m = 12$). Los resultados de cada ajuste los guardamos en la lista `model_evals`.

CELDA 10

```
import itertools
import warnings
from statsmodels.tsa.api import SARIMAX

warnings.filterwarnings("ignore")

p = d = q = range(0, 3)
# (0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), ...
pdq = itertools.product(p, d, q)
# (0, 0, 0, 12), (0, 0, 1, 12), (0, 0, 2, 12), (0, 1, 0, 12), ...
seasonal_PDQ = itertools.product(p, d, q, [12])

model_evals = []
for param in pdq:
    for param_seasonal in seasonal_PDQ:
        mod = SARIMAX(
            y_train, order=param, seasonal_order=param_seasonal,
            enforce_stationarity=False, enforce_invertibility=False)
        results = mod.fit(dispatch=0)
        model_evals.append([param, param_seasonal, results.aic, results.bic])
```

Mostramos ahora los parámetros obtenidos para los modelos con AIC y BIC menores a 900, siendo este valor sugerido por ajustes exploratorios para algunos pocos tuplas:

CELDA 11

```

for param, param_seasonal, aic, bic in model_evals:
    if aic < 900 or bic < 900:
        print(f"AIC: {aic}, BIC: {bic}, (p, d, q)={param}, (P, D, Q, 12)={param_seasonal}")

```

AIC: 850.0800066715967, BIC: 858.1123209909935, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(0, 2, 1, 12)
 AIC: 806.9674764810895, BIC: 818.9268324969429, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(0, 2, 2, 12)
 AIC: 859.3854459797975, BIC: 867.5361380419748, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 0, 0, 12)
 AIC: 858.2005034043332, BIC: 870.4196370066344, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 0, 1, 12)
 AIC: 831.1939496722154, BIC: 847.3739709283594, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 0, 2, 12)
 AIC: 856.4371945906423, BIC: 868.5722105327503, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 1, 1, 12)
 AIC: 823.7388931519195, BIC: 839.8035217907129, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 1, 2, 12)
 AIC: 841.9230576997115, BIC: 853.9715291788066, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 2, 1, 12)
 AIC: 807.7313884905143, BIC: 823.677196511652, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(1, 2, 2, 12)
 AIC: 831.4336726656362, BIC: 843.575789202775, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 0, 0, 12)
 AIC: 832.8510525948286, BIC: 849.0405413110137, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 0, 1, 12)
 AIC: 832.9819569262969, BIC: 853.2069834964769, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 0, 2, 12)
 AIC: 826.51149391015, BIC: 842.5858667681349, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 1, 1, 12)
 AIC: 826.6356659604376, BIC: 846.7164517589293, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 1, 2, 12)
 AIC: 879.1720972084877, BIC: 891.1389814591573, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 2, 0, 12)
 AIC: 801.073238333745, BIC: 817.0290840509339, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 2, 1, 12)
 AIC: 801.4018994251785, BIC: 821.3341594516007, (p, d, q)=(0, 0, 0), (P, D, Q, 12)=(2, 2, 2, 12)

Podemos ver que algunas combinaciones de parámetros generan valores muy bajos de AIC y BIC. Sin embargo, dichos modelos no ajustan bien a la serie temporal (esto se muestra en los *warnings* que desactivamos para evitar salidas muy largas, que informan sobre problemas de convergencia del método de ajuste o inestabilidades numéricas). Descartando estos valores, el conjunto $(p, d, q) = (0, 0, 0)$ y $(P, D, Q, 12) = (2, 2, 1, 12)$ coinciden en bajos valores tanto de AIC como BIC, por lo que construiremos un modelo SARIMAX basado en estos parámetros:

CELDA 12

```

model = SARIMAX(
    y_train, order=(0, 0, 0), seasonal_order=(2, 2, 1, 12),
    enforce_stationarity=True, enforce_invertibility=False)
model_fit = model.fit(dispatch=0, maxiter=200)
print(model_fit.summary())

```

SARIMAX Results

```

=====
Dep. Variable:          co2      No. Observations:          447
Model:                SARIMAX(2, 2, [1], 12)      Log Likelihood          -423.255
Date:                  lun, 14 oct 2024      AIC              854.511
Time:                  22:51:54      BIC              870.700
Sample:                03-01-1958      HQIC             860.908
                    - 05-01-1995
Covariance Type:      opg
=====

```

	coef	std err	z	P> z	[0.025	0.975]
ar.S.L12	-0.2795	0.067	-4.172	0.000	-0.411	-0.148
ar.S.L24	-0.2458	0.064	-3.835	0.000	-0.371	-0.120
ma.S.L12	-0.7492	0.055	-13.506	0.000	-0.858	-0.641
sigma2	0.4140	0.027	15.379	0.000	0.361	0.467

```

=====

```


Ljung-Box (L1) (Q):	229.89	Jarque-Bera (JB):	4.55
Prob(Q):	0.00	Prob(JB):	0.10
Heteroskedasticity (H):	1.06	Skew:	-0.01
Prob(H) (two-sided):	0.72	Kurtosis:	3.51

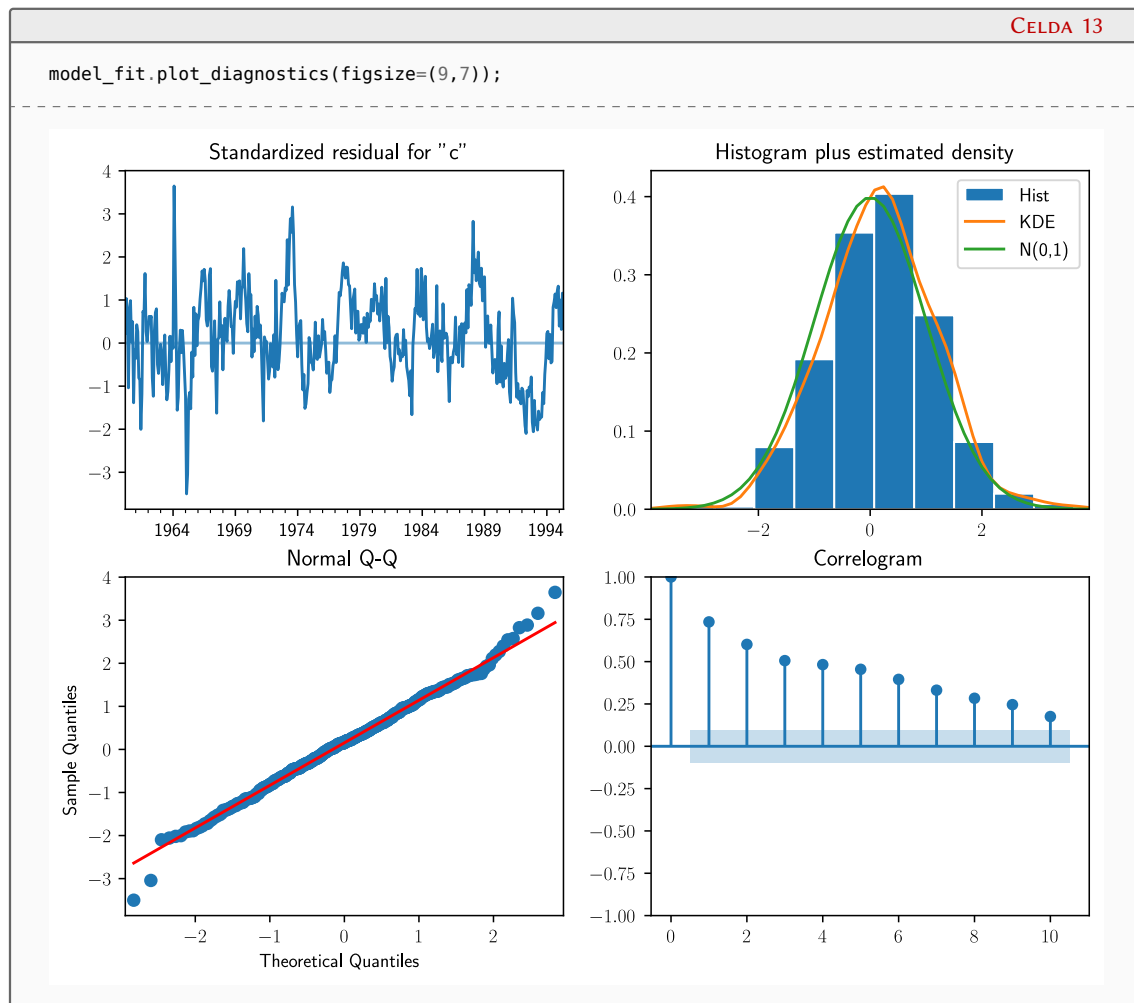
=====

Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Con el método `summary()` del objeto `model_fit` podemos obtener los parámetros del ajuste y sus correspondientes p -valores. En todos los casos, estos p -valores muestran que los coeficientes obtenidos son significativos.

También podemos examinar la calidad del ajuste a través del análisis de los residuos de los datos que utilizamos como ajuste (`y_train`). Estos residuos son la diferencia entre los valores generados por el modelo y los datos reales. El modelo ideal de ajuste debería tener residuos que sean ruido blanco gaussiano centrados en cero y sin correlación. El método `plot_diagnostics` genera cuatro gráficos que permiten evaluar estas características de los residuos:

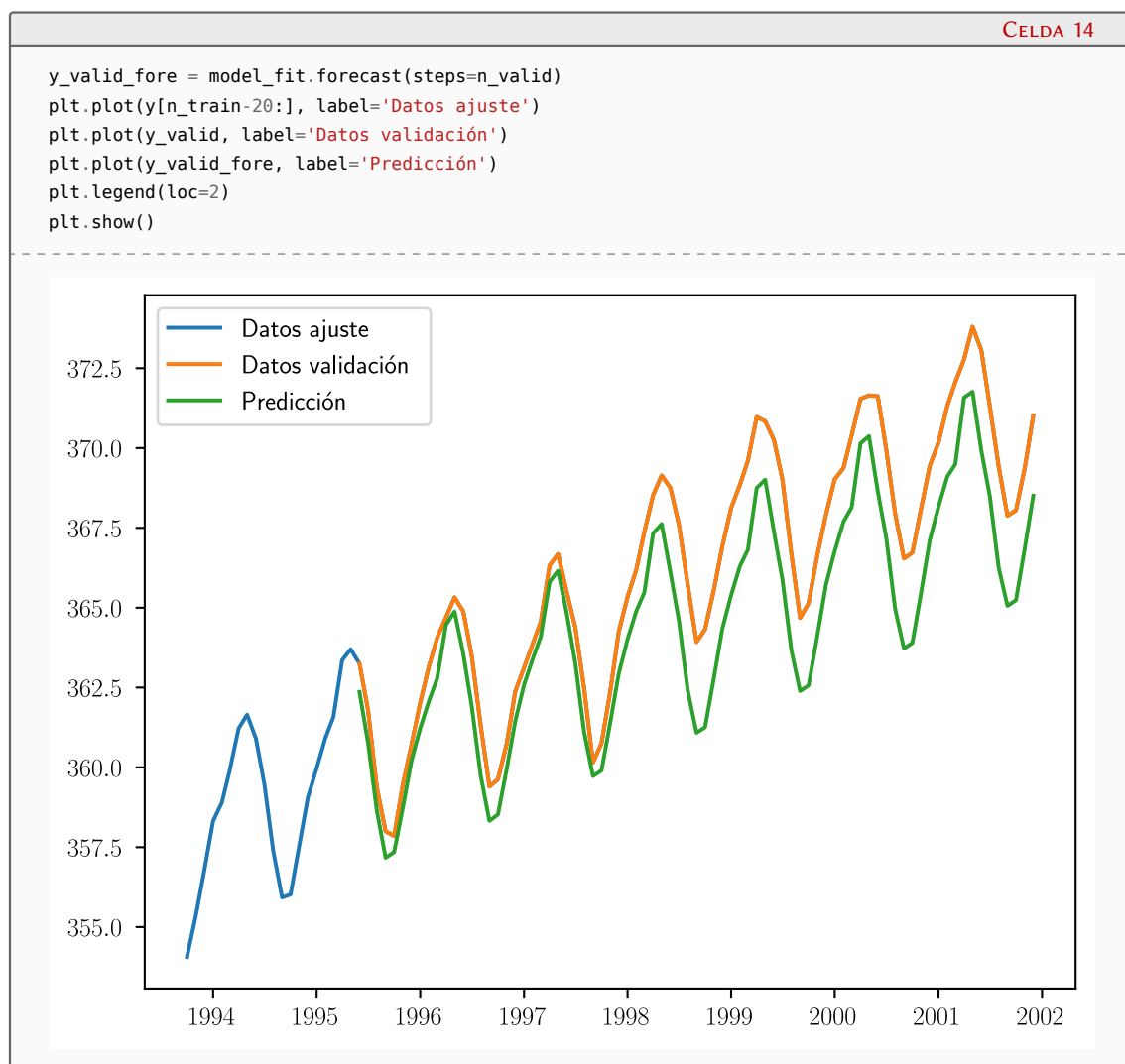


El panel superior izquierdo muestra los residuos para cada valor registrado en los datos de ajuste. Si nuestro modelo funciona adecuadamente, no se debería apreciar ningún patrón obvio

en estos residuos, tal como muestra la figura. El panel superior derecho muestra la distribución de dichos residuos, tanto en forma de histograma como su KDE, y muestra también para facilitar la comparación una distribución normal con media cero y varianza 1. Se puede ver que la KDE de los residuos es muy cercana a la distribución normal.

En el panel inferior izquierdo se muestra el diagrama Q-Q de los residuos. Este diagrama compara la distribución teórica de los residuos con la observada. En el caso ideal deberíamos obtener una línea recta como la roja. Esto sucede en casi todos los valores excepto en los extremos. Finalmente, en el panel inferior derecho se muestra un correlograma que representa la ACF de los residuos. El 95 % de las correlaciones para *lags* mayores que cero no deberían ser significativos (es decir, deben estar dentro de la región sombreada). Si hubiese una correlación en los residuos, podríamos interpretar que existe información en los datos que no fue capturada por el modelo. En nuestro caso, no parecen haber correlaciones para *lags* mayores que cero, lo que indica que el ajuste es bueno.

Finalmente, podemos utilizar el modelo ajustado para predecir valores de la serie temporal posteriores a los que utilizamos para el ajuste, y compararlos con los valores que reservamos con este `fin`, `y_valid`:



Podemos ver que el modelo es capaz de predecir con buena precisión los valores posteriores a los utilizados en el ajuste del modelo, pese a que alguna diferencia es notoria.

Hemos visto un ejemplo de análisis estadístico de una serie temporal no estacionaria, ajustándola con un modelo relativamente simple. Además de statsmodels, existen otras herramientas poderosas para el ajuste de modelos, entre las que podemos destacar [Darts](#) y muchas bibliotecas de redes neuronales. El proceso no es simple y hay que analizar con espíritu crítico cada resultado obtenido.

17.7. Lecturas recomendadas

- Peter McCullagh. «What is a statistical model?» En: *The Annals of Statistics* 30.5 (2002), págs. 1225-1310. DOI: [10.1214/aos/1035844977](https://doi.org/10.1214/aos/1035844977). URL: <https://doi.org/10.1214/aos/1035844977>.
- Galit Shmueli. «To Explain or to Predict?» En: *Statistical Science* 25.3 (2010), págs. 289-310. DOI: [10.1214/10-STS330](https://doi.org/10.1214/10-STS330). URL: <https://doi.org/10.1214/10-STS330>.
- Susan Holmes y Wolfgang Huber. *Modern statistics for modern biology*. Cambridge, England: Cambridge University Press, feb. de 2019. URL: <https://web.stanford.edu/class/bios221/book/introduction.html>. Se puede acceder [aquí](#).
- George E.P. Box, J. Stuart Hunter y William G. Hunter. *Statistics for experimenters*. 2.^a ed. Wiley Series in Probability and Statistics. Chichester, England: Wiley-Blackwell, mayo de 2005.
- Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.^a ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: [10.1007/978-3-319-28316-6](https://doi.org/10.1007/978-3-319-28316-6).
- A. Colin Cameron y Pravin K. Trivedi. *Regression Analysis of Count Data*. 2.^a ed. Econometric Society Monographs. Cambridge University Press, 2013. DOI: [10.1017/CB09781139013567](https://doi.org/10.1017/CB09781139013567).
- Terence Mills. *Applied time series analysis*. San Diego, CA: Academic Press, 2019.
- George E P Box et al. *Time Series Analysis*. en. 5.^a ed. Wiley Series in Probability and Statistics. Nashville, TN: John Wiley & Sons, jun. de 2015.



18 | Ecuaciones algebraicas

En este capítulo abordaremos la resolución de ecuaciones algebraicas. Es necesario distinguir entre ecuaciones univariadas o multivariadas (es decir, ecuaciones que contienen una variable desconocida o muchas), y también si las ecuaciones son lineales o no lineales. Esta clasificación es necesaria pues cada tipo de ecuación requiere la utilización de técnicas específicas de solución.

Comenzaremos con sistemas de ecuaciones lineales cuya utilidad es central en muchos campos científicos, y pueden resolverse mediante la aplicación del álgebra lineal. Por otra parte, las ecuaciones no lineales son más complicadas de resolver y usualmente demandan métodos que son computacionalmente intensivos.

Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1
Sympy	1.13.3

[Código disponible](#)

18.1. Sistema de ecuaciones lineales

Una de las aplicaciones más importantes del álgebra lineal es la resolución de sistemas de ecuaciones lineales. Tanto NumPy como SciPy poseen submódulos con rutinas de álgebra lineal. Sin embargo `scipy.linalg` contiene todas las funciones de `numpy.linalg` y algunas más avanzadas no disponibles en NumPy. Una diferencia importante entre ambos módulos es que `scipy.linalg` está compilado con soporte de las bibliotecas ATLAS LAPACK y BLAS, que son rutinas escritas en Fortran altamente optimizadas para realizar operaciones con vectores y matrices (BLAS) y para resolver sistemas de ecuaciones lineales (LAPACK), mientras que esto es opcional para NumPy. De este modo, la versión de `linalg` de `scipy` puede ser más rápida que la de NumPy, dependiendo de cómo se ha instalado Numpy. Por este motivo recomendamos el uso de `scipy.linalg`.

La forma general de un sistema de ecuaciones lineales es:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \end{aligned}$$

Este sistema representa un conjunto de m ecuaciones con n incógnitas x_j ($1 \leq j \leq n$), en las que a_{ij} ($1 \leq i \leq m$) y b_j son constantes o parámetros conocidos. Naturalmente, al trabajar con sistemas de ecuaciones lineales resulta muy práctico hacerlo en notación matricial:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \quad (18.1)$$

o, en forma compacta, $\mathbb{A}\mathbf{x} = \mathbf{b}$, en la que $\mathbb{A}$ es una matriz $m \times n$, $\mathbf{b}$ es un vector de m componentes (o matriz $m \times 1$) y $\mathbf{x}$ es el vector incógnita o solución de n componentes (o matriz $n \times 1$).

El sistema representado por la ecuación (18.1) (o su forma compacta) puede tener solución o no, y en caso de que exista dicha solución la misma puede no ser única, y esto depende de las propiedades de la matriz $\mathbb{A}$.

Cuando tenemos menos ecuaciones que incógnitas, esto es, $n < m$, el sistema está subdeterminado y por lo tanto no puede determinarse una solución única. Por el contrario, cuando existen más ecuaciones que incógnitas ($n > m$) el sistema está sobredeterminado, y en este caso, por lo general, se producen restricciones que no permiten hallar una solución.

Los sistemas cuadrados, esto es, cuando $n = m$, son de importancia debido a que puede existir una solución única, y para ello la matriz $\mathbb{A}$ debe ser *no singular* lo que implica que existe la inversa de $\mathbb{A}$, y la solución puede escribirse como:

$$\mathbf{x} = \mathbb{A}^{-1} \cdot \mathbf{b} \quad (18.2)$$

Si la matriz $\mathbb{A}$ es singular, esto es cuando el rango de la matriz es menor que n , $\text{rg}(\mathbb{A}) < n$, o equivalentemente el determinante es nulo ($\det(\mathbb{A}) = 0$), la ecuación (18.1) puede no tener solución o tener infinitas, dependiendo del vector $\mathbf{b}$. Cuando $\text{rg}(\mathbb{A}) < n$ hay filas o columnas que son

combinaciones lineales de otras filas o columnas, y por lo tanto corresponden a ecuaciones que no representan una nueva relación entre las incógnitas, y por lo tanto el sistema está subdeterminado. De este modo, calcular el rango de la matriz que define el sistema de ecuaciones lineales es un método útil para saber si la matriz es singular o no, y en consecuencia saber de antemano si existe la solución del sistema.

Al abordar el problema numéricamente pueden presentarse dos cuestiones adicionales que complican hallar la solución. Por un lado, aunque algunas ecuaciones no sean exactamente combinaciones lineales de otras, pueden estar tan cerca de ser linealmente dependientes que los errores de redondeo en el cálculo con representación de punto flotante las hacen linealmente dependientes en alguna etapa del cálculo de la solución, y en este caso la solución encontrada no será tal.

Por otro lado, particularmente cuando el número n es muy grande, los errores de redondeo acumulados pueden arruinar completamente la solución, aún cuando el proceso algorítmico es correcto. Este hecho se acentúa cuando $\mathbb{A}$ está cerca de ser singular.

18.1.1. Condicionamiento

Aún cuando $\text{rg}(\mathbb{A}) = n$ y por consiguiente la solución existe, puede no ser posible obtener dicha solución en forma precisa. En efecto, el número de condición de la matriz, $\text{cond}(\mathbb{A})$ determina si la matriz está bien o mal condicionada. Cuando $\text{cond}(\mathbb{A}) \approx 1$, la matriz está “bien condicionada”, pero cuando $\text{cond}(\mathbb{A}) \gg 1$ se dice que está “mal condicionada”. En este último caso, la solución del sistema de ecuaciones puede tener errores de gran magnitud. Una forma de interpretar cómo afecta el número de condición a los errores consiste en realizar un análisis del error. Supongamos que tenemos un sistema de ecuaciones lineales de la forma $\mathbb{A}\mathbf{x} = \mathbf{b}$ donde $\mathbf{x}$ es el vector solución. A una pequeña variación en $\mathbf{b}$, $\delta\mathbf{b}$, le corresponderá un cambio en la solución, $\delta\mathbf{x}$ dado por $\mathbb{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$, y dado que el sistema es lineal, resulta que $\mathbb{A}\delta\mathbf{x} = \delta\mathbf{b}$. Entonces, podemos determinar cuán grande es el cambio relativo en $\mathbf{x}$ comparado con el cambio relativo en $\mathbf{b}$ haciendo uso de los cocientes de las normas de estos vectores. Específicamente, comparamos $\|\delta\mathbf{x}\|/\|\mathbf{x}\|$ y $\|\delta\mathbf{b}\|/\|\mathbf{b}\|$, donde $\|\mathbf{x}\|$ denota la norma del vector $\mathbf{x}$. Usando la relación $\|\mathbb{A}\mathbf{x}\| \leq \|\mathbb{A}\|\|\mathbf{x}\|$, podemos escribir

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} = \frac{\|\mathbb{A}^{-1}\delta\mathbf{b}\|}{\|\mathbf{x}\|} \leq \frac{\|\mathbb{A}^{-1}\|\|\delta\mathbf{b}\|}{\|\mathbf{x}\|} = \frac{\|\mathbb{A}^{-1}\|\|\mathbf{b}\|}{\|\mathbf{x}\|} \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \leq \|\mathbb{A}^{-1}\|\|\mathbb{A}\| \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

Dado que la definición del número de condición es $\text{cond}(\mathbb{A}) = \|\mathbb{A}\|\|\mathbb{A}^{-1}\|$, la expresión anterior da una cota superior al error relativo en la solución dado el error relativo en $\mathbf{b}$. De aquí se infiere que si un sistema lineal de ecuaciones está definido por una matriz $\mathbb{A}$ que está mal condicionada, pequeñas variaciones del vector $\mathbf{b}$ pueden originar grandes errores en el vector solución $\mathbf{x}$. Por esto es que al resolver estos sistemas es necesario calcular el número de condición para estimar la precisión de las soluciones obtenida, particularmente cuando se usan números con representación de punto flotante, que tal como vimos en el Capítulo 12 son solo aproximaciones a los números reales.

En el caso en que estemos tratando simbólicamente con una matriz, el rango, el número de condición y la norma se pueden obtener con los métodos `rank`, `condition_number` y `norm` del objeto `Matrix` de `SimPy`, mientras que si solo trabajamos en forma numérica, las mismas magnitudes se obtienen de las funciones del submódulo `linalg` de `NumPy`: `matrix_rank`, y `cond` y `norm` (`SciPy`

solo tiene implementada la función `norm` en el submódulo `linalg`).

Como ejemplo veremos cómo calcular estas cantidades en una matriz de Hilbert de 2×2 , que se conoce como ejemplo de matriz mal condicionada. Los elementos de esta matriz son $H_{ij} = 1/(i + j - 1)$.

Definimos la matriz de Hilbert, utilizando el método Rational de SimPy para manter la expresión de los elementos de la matriz como fracciones (la llamamos `H_s` para destacar que es un objeto Matrix de SimPy):



David Hilbert - Matemático alemán (1862-1943) reconocido como uno de los más influyentes del siglo XIX y principios del XX. Estableció su reputación como gran matemático y científico inventando y desarrollando un gran abanico de ideas, como la teoría de invariantes, la axiomatización de la geometría y la noción de espacio de Hilbert, uno de los fundamentos del análisis funcional.

CELDA 1 DE SISTEMAS_LINEALES.IPYNB
<pre>from sympy import Matrix, Rational # Matriz de Hilbert 2x2 H_s = Matrix([[1, Rational(1, 2)], [Rational(1, 2), Rational(1, 3)],]) H_s</pre>
$\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{bmatrix}$

A continuación obtenemos el rango, el número de condición exacto y aproximado, y la norma de H

CELDA 02
<pre>H_s.rank()</pre>
2

CELDA 03
<pre>H_s.condition_number()</pre>
$\frac{\sqrt{\frac{2\sqrt{13}}{9} + \frac{29}{36}}}{\sqrt{\frac{29}{36} - \frac{2\sqrt{13}}{9}}}$

CELDA 04
<pre>from sympy import N N(_)</pre>
19,281470067904

CELDA 05
<code>H_s.norm()</code>
$\frac{\sqrt{58}}{6}$

Ahora pasamos a la representación en punto flotante de H , utilizando las rutinas de NumPy y SciPy para definirla (SciPy permite definir esta matriz indicando el rango de la misma) y para calcular su rango, número de condición y norma:

CELDA 06
<pre>import scipy.linalg as la # Matriz de Hilbert 2x2 (punto flotante) H = la.hilbert(2) H</pre>
<pre>array([[1. , 0.5], [0.5 , 0.33333333]])</pre>

CELDA 07
<pre>import numpy as np np.linalg.matrix_rank(H)</pre>
2

CELDA 08
<code>np.linalg.cond(H)</code>
19.28147006790397

CELDA 09
<code>la.norm(H)</code>
1.2692955176439846

CELDA 10
<code>np.linalg.norm(H)</code>
1.2692955176439846

18.1.2. Descomposición LU

La forma directa de resolver un sistema de ecuaciones lineales consiste en obtener la matriz inversa de $\mathbb{A}$, y multiplicarla por el vector $\mathbf{b}$. Sin embargo esta no es la forma más eficiente de hacerlo, particularmente cuando utilizamos el sistema caracterizado por $\mathbb{A}$ para resolver múltiples sistemas con distintos vectores $\mathbf{b}$: $\mathbb{A}\mathbf{x}_i = \mathbf{b}_i$.

La descomposición LU consiste en factorizar la matriz $\mathbb{A}$ de la siguiente forma:

$$\mathbb{L} \cdot \mathbb{U} = \mathbb{A} \quad (18.3)$$

donde $\mathbb{L}$ es una matriz diagonal inferior (es decir, tiene elementos no nulos en la diagonal y debajo de ella) y $\mathbb{U}$ es una matriz diagonal superior (con elementos no nulos en la diagonal y arriba de ella). Para el caso de una matriz $\mathbb{A}$ de 4×4 , esto se vería:

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{m1} & a_{m2} & a_{43} & a_{44} \end{bmatrix} \quad (18.4)$$

Utilizando esta factorización podemos resolver el sistema lineal

$$\mathbb{A} \cdot \mathbf{x} = (\mathbb{L} \cdot \mathbb{U}) \cdot \mathbf{x} = \mathbf{b} \quad (18.5)$$

resolviendo primero para un vector $\mathbf{y}$ tal que

$$\mathbb{L} \cdot \mathbf{y} = \mathbf{b} \quad (18.6)$$

y luego resolvemos

$$\mathbb{U} \cdot \mathbf{x} = \mathbf{y} \quad (18.7)$$

Este procedimiento de dividir el problema dado por la ecuación (18.5) en dos ecuaciones sucesivas resulta eficiente debido a que obtener la solución de sistemas triangulares es trivial: la ecuación (18.6) se resuelve con sustitución hacia adelante mientras que la (18.7) se resuelve con sustitución hacia atrás. De este modo, en el caso de resolver nuestro sistema original para múltiples términos independientes $\mathbf{b}_i$, solo una vez hay que obtener la factorización LU y luego simplemente se resuelve para cada caso mediante las sustituciones de las ecuaciones (18.6) y (18.7).

Tanto SymPy como SciPy poseen rutinas que realizan la factorización LU, veamos ejemplos de aplicación de esos métodos. En las celdas siguientes utilizamos el método `LUdecomposition()` aplicado a la matriz de Hilbert:

CELDA 11	
<pre>L_s, U_s, P_s = H_s.LUdecomposition() L_s</pre>	
$\begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix}$	
CELDA 12	
<pre>U_s</pre>	

$$\begin{bmatrix} 1 & \frac{1}{2} \\ 0 & \frac{1}{12} \end{bmatrix}$$

CELDA 13

L_s * U_s

$$\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{bmatrix}$$

Podemos ver que la matriz L posee unos en su diagonal, lo que indica que Sympy realiza una descomposición de Doolittle. En general es posible realizar la factorización de muchas formas, en el caso en que los unos forman la diagonal de la matriz U , por ejemplo, estaríamos ante una descomposición de Crout. Tanto Sympy como SciPy tienen implementada la descomposición de Doolittle.

Vemos además que el método devuelve también la matriz P , que está compuesta por filas con pares de índices de intercambio. Esto resulta de intercambiar el orden de las filas de modo eliminar el mal condicionamiento que puede ocurrir cuando un elemento a_{ij} de la matriz es mucho menor que uno. Las estrategias de pivoteo exceden el alcance de este capítulo, pero están implementadas en estos métodos para ofrecer soluciones robustas.

Por último, en la celda 13 podemos ver que el producto de L y U reconstruye la matriz de Hilbert, tal como establece su factorización.

En el caso de la factorización numérica con SciPy, la función `lu` del submódulo `linalg` devuelve las tres matrices anteriores pero en distinto orden, tal como se puede ver en las celdas siguientes:

CELDA 14

```
P, L, U = la.lu(H)
L
```

```
array([[1. , 0. ],
       [0.5, 1. ]])
```

CELDA 15

U

```
array([[1. , 0.5 ],
       [0. , 0.08333333]])
```

CELDA 16

```
np.allclose(H - P @ L @ U, np.zeros((2,2)))
```

```
True
```

En la celda 16 comparamos elemento por elemento los valores de $H - P @ L @ U$ con un array 2×2 con elementos nulos, lo que nos devuelve **True** comprobando de este modo que la descom-

posición LU recupera la matriz de Hilbert original al realizar el producto de las matrices que lo factorizan. Debemos notar en la celda 16 que denotamos el producto de matrices con el operador @ que es un alias de la función `np.matmul` sobre arrays.

Utilizando la descomposición LU mostraremos ahora un ejemplo de resolución de un sistema lineal de ecuaciones en forma analítica con Sympy y numérica con SciPy, y evaluaremos el efecto del condicionamiento en la precisión de la solución numérica. Por lo general, la solución analítica solo es útil para matrices muy pequeñas, ya que al aumentar el rango de la matriz las expresiones pueden resultar muy difíciles de leer, y finalmente se evalúan numéricamente.

Nuestro sistema de ejemplo será la matriz:

$$\mathbb{A}(p) = \begin{bmatrix} 1 & \frac{1}{p+1} \\ \frac{1}{p+1} & \frac{1}{p+2} \end{bmatrix}$$

Podemos ver que cuando $p = 1$, $\mathbb{A}$ se convierte en la matriz de Hilbert 2×2 . En la siguiente celda definimos esta matriz en forma simbólica con Sympy:

CELDA 17

```
import sympy as sy

# Especificación del problema simbólico
p = sy.symbols("p", positive=True)
A_s = Matrix([[1, 1/(p + 1)], [1/(p + 1), 1/(p + 2)]])
A_cond = A_s.condition_number() # Número de condición
A_s
```

$$\begin{bmatrix} 1 & \frac{1}{p+1} \\ \frac{1}{p+1} & \frac{1}{p+2} \end{bmatrix}$$

y también calculamos el número de condición `A_cond`. Definimos a continuación el vector de términos independientes y obtenemos la solución del sistema, que obviamente es función del parámetro p :

CELDA 18

```
# Término independiente y solución analítica
b_s = Matrix([5, 3])
x_s = A_s.solve(b_s)
x_s
```

$$\begin{bmatrix} \frac{2p^2+p-1}{p^2+p-1} \\ \frac{3p^3+7p^2-4}{p^2+p-1} \end{bmatrix}$$

A continuación definimos el problema numérico haciendo uso de funciones `lambda` de modo de poder resolver el problema especificando cada vez el valor del parámetro p .

CELDA 19

```
# Especificación del problema numérico
A_n = lambda p: np.array([[1, 1/(p + 1)], [1/(p + 1), 1/(p + 2)]])
b_n = np.array([5, 3])
x_n = lambda p: np.linalg.solve(A_n(p), b_n)
```

Para el caso en que $p = 1$ podemos obtener las soluciones del sistema determinado por la matriz de Hilbert tanto en forma exacta como numérica:

CELDA 20

```
# Solución exacta para p = 1 (matriz de Hilbert)
x_s.subs(p, 1)
```

$$\begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

CELDA 21

```
# Solución numérica para p = 1 (matriz de Hilbert)
x_n(1)
```

```
array([2., 6.])
```

También podemos encontrar la solución haciendo uso de la descomposición LU, lo que sería muy eficiente si quisiéramos resolver el sistema para múltiples términos independientes. En primer lugar realizamos la factorización de la matriz A para $p = 1$, luego obtenemos la solución numérica x_{lu} y finalmente verificamos que al reemplazar la solución obtenida en el sistema se verifica cada una de las ecuaciones (es decir, que $\mathbb{A}(1)x - b = 0$):

CELDA 22

```
# Solución numérica usando la factorización LU
lu, piv = la.lu_factor(A_n(1))
x_lu = la.lu_solve((lu, piv), b_n)
np.allclose(A_n(1) @ x_lu - b_n, np.zeros(2))
```

```
True
```

Para finalizar exploraremos el efecto que tiene el condicionamiento de la matriz $\mathbb{A}(p)$ en la precisión de los resultados obtenidos. Para ello notemos que el número de condición diverge para $p = (\sqrt{5} - 1)/2$ (dejamos la verificación de esto para quienes se entusiasmen con el cálculo). En consecuencia, construiremos un array de valores de p para los que resolveremos el sistema tanto en forma exacta como numérica, y evaluaremos el error relativo de la solución numérica.

En la celda 23 graficamos a la izquierda este error relativo para cada una de las componentes del vector solución, mientras que el panel de la derecha mostramos el número de condición. Podemos apreciar que donde diverge el número de condición ($p \approx 0,618$), el error relativo de la solución numérica aumenta significativamente. Pese a que el método de solución es muy robusto para este caso, dado que el error relativo es del orden de 4×10^{-13} , este valor es tres órdenes de magnitud mayor que el épsilon de la máquina donde ejecutamos el ejemplo (ver Sección 12.1), como muestra la celda 24, con lo cual es un error apreciable que puede magnificarse al resolver

sistemas grandes de ecuaciones (o al resolver sucesivamente sistemas lineales de ecuaciones en un algoritmo).

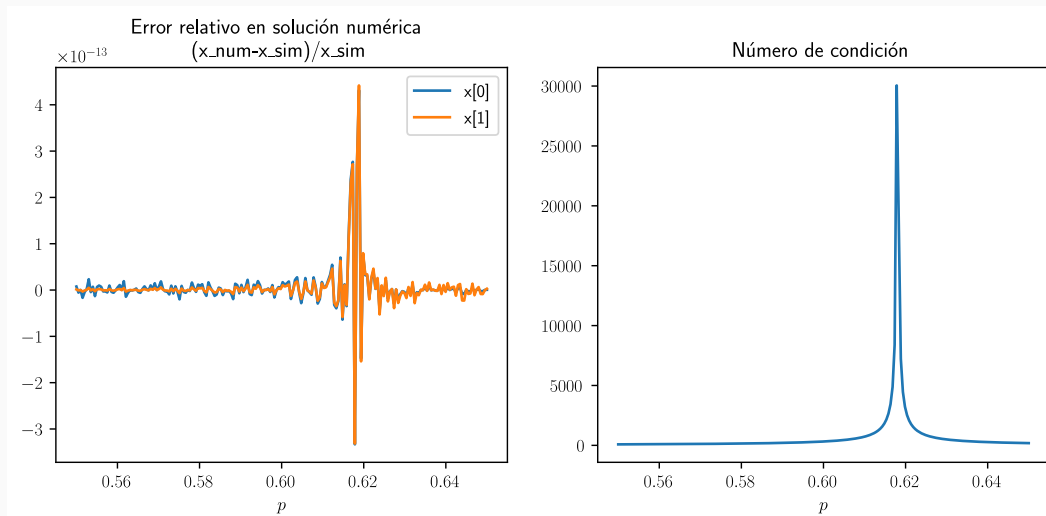
CELDA 23

```
import matplotlib.pyplot as plt

# Gráfico de la diferencia entre las soluciones analítica (exacta) y numérica
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
p_valores = np.linspace(0.55, 0.65, 200)
for n in range(2):
    x_sim = np.array([x_s[n].subs(p, pp).evalf() for pp in p_valores])
    x_num = np.array([x_n(pp)[n] for pp in p_valores])
    ax[0].plot(p_valores, (x_num - x_sim)/x_sim, label=f'x[{n}]')

ax[0].set_title("Error relativo en solución numérica\n(x_num-x_sim)/x_sim")
ax[0].set_xlabel(r'$p$')
ax[0].legend()

ax[1].plot(p_valores, [A_cond.subs(p, pp).evalf() for pp in p_valores])
ax[1].set_title("Número de condición")
ax[1].set_xlabel(r'$p$')
plt.show()
```



CELDA 24

```
import sys

print(sys.float_info.epsilon)
```

2.220446049250313e-16

18.1.3. Métodos iterativos

Los métodos directos para la resolución del sistema lineal $\mathbb{A} \mathbf{x} = \mathbf{b}$, donde $\mathbb{A} \in \mathbb{R}^{n \times n}$, requieren una cantidad de operaciones que es del orden n^3 . Esto es un gran esfuerzo de cálculo cuando

n es grande, tal como ocurre en diversos problemas científicos y de ingeniería, particularmente en la solución de ecuaciones en derivadas parciales que se resuelven con diferencias finitas o elementos finitos (ver el Capítulo 21).

Este costo computacional se puede reducir si es posible aprovechar la estructura de $\mathbb{A}$. Tal reducción es considerable si la matriz es «rala», esto es, cuando una gran cantidad de elementos de $\mathbb{A}$ es cero (lo opuesto de rala es «densa»). En estos casos incluso el almacenamiento de los elementos no nulos de la matriz puede hacerse mediante estructuras de datos que resultan sumamente eficientes comparados con el almacenamiento de la matriz completa.

Los métodos iterativos consisten en la generación de una secuencia de aproximaciones $\{\mathbf{x}^{(k)}\}_{k=1}^{\infty}$ a la solución a partir de una estimación inicial $\mathbf{x}^{(0)}$, con el propósito de que $\mathbf{x}^{(k)} \rightarrow \mathbf{x} = \mathbb{A}^{-1} \mathbf{b}$ a medida que $k \rightarrow \infty$. En la práctica, el proceso iterativo se detiene en el menor valor de k tal que $\|\mathbf{x}^{(k)} - \mathbf{x}\| < \varepsilon$, donde ε es una tolerancia fija y $\|\cdot\|$ es cualquier norma que resulte conveniente.

En esta sección describiremos brevemente los métodos de Jacobi y Gauss-Seidel, considerados como «clásicos» pero que no se utilizan realmente en aplicaciones que resuelven sistemas de gran dimensión. Para completar la sección incluiremos un breve ejemplo de un método más moderno como el GMRES.

Si los elementos diagonales de $\mathbb{A}$ son no nulos, podemos escribir explícitamente el sistema de ecuaciones y dejar en cada una de ellas la correspondiente incógnita

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j \right], \quad i = 1, \dots, n$$

En el método de Jacobi, una vez que establecemos una estimación inicial $\mathbf{x}^{(0)}$, podemos iniciar un proceso iterativo para calcular $\mathbf{x}^{(k+1)}$ mediante las siguientes fórmulas:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^{(k)} \right], \quad i = 1, \dots, n \quad (18.8)$$

Si la matriz $\mathbb{A}$ es diagonalmente dominante ($|a_{ii}| > \sum_{j \neq i} |a_{ij}|$) la convergencia a la solución está garantizada.

Formalmente es posible desarrollar el método utilizando la descomposición aditiva de la matriz $\mathbb{A}$ de la siguiente forma: $\mathbb{A} = \mathbb{D} + \mathbb{L} + \mathbb{U}$, donde $\mathbb{D}$ es una matriz diagonal (que contiene los elementos diagonales de $\mathbb{A}$) y las matrices $\mathbb{L}$ y $\mathbb{U}$ son las partes triangulares inferior y superior de $\mathbb{A}$, respectivamente, con sus correspondientes elementos diagonales nulos. De este modo,

$$\mathbb{A} = \mathbb{D} + \mathbb{L} + \mathbb{U}$$

Si llamamos $\mathbb{K} = \mathbb{L} + \mathbb{U}$, el sistema de ecuaciones puede escribirse:

$$\mathbb{D} \mathbf{x} = \mathbf{b} - \mathbb{K} \mathbf{x}$$

lo que da origen la ecuación iterativa:

$$\mathbf{x}^{(k+1)} = \mathbb{D}^{-1} (\mathbf{b} - \mathbb{K} \mathbf{x}^{(k)}) \quad (18.9)$$

donde se puede aprovechar el hecho que la inversa de una matriz diagonal está compuesta simplemente por los elementos recíprocos de $\mathbb{A}$. De este modo, podemos escribir una función que implemente este método matricial que resulta más eficiente al utilizar operaciones vectorizadas de NumPy y evitando el uso de bucles, tal como mostramos en el *notebook* siguiente:

CELDA 1 DE METODOS_ITERATIVOS.IPYNB

```
import numpy as np
import scipy.sparse as sp

def jacobi(A, b, tol, max_iter):
    """Resuelve  $Ax = b$  por el método de Jacobi."""
    x = np.zeros_like(b)
    D = sp.diags(A.diagonal()) # Extraemos la diagonal de A
    K = A - D #  $K = L + U$ 
    errores = []
    for k in range(max_iter):
        x_nuevo = (b - K @ x) / D.diagonal()
        error = np.linalg.norm(x_nuevo - x, ord=np.inf)
        errores.append(error)
        if error < tol:
            print(f"Convergencia en {k} iteraciones (Jacobi).")
            return x_nuevo, errores
        x = x_nuevo
    print("No hubo convergencia (Jacobi).")
    return x, errores
```

Luego de importar los módulos necesarios, definimos la función `jacobi` que recibe como argumentos la matriz A , el vector de términos independientes b , la tolerancia `tol` que nos indicará cuándo estamos suficientemente próximos a la solución buscada (y en consecuencia, detener el algoritmo), y el máximo número de iteraciones a generar `max_iter`.

Continuamos con la definición un vector x con todos sus elementos cero y de la misma dimensión que b , obtenemos la matriz D que es diagonal y contiene los elementos diagonales de A , y calculamos la matriz K . Adicionalmente, almacenaremos en la lista `errores` la progresión de la norma de la diferencia entre dos iteraciones sucesivas, lo que nos servirá para analizar la convergencia del método.

Entramos entonces en el bucle de las iteraciones, calculando el nuevo valor de la secuencia utilizando la ecuación (18.9), y su diferencia con la iteración anterior según la norma L_∞ ¹. Finalmente, detenemos las iteraciones si alcanzamos la tolerancia requerida enviando un mensaje y devolviendo la solución obtenida y la secuencia de errores. En caso de alcanzar el máximo número de iteraciones, informamos que no hubo convergencia y devolvemos el estado alcanzado.

Para poner a prueba este método de cálculo, resolveremos la ecuación de Laplace en dos dimensiones

$$\frac{\partial u(x, y)}{\partial x} + \frac{\partial u(x, y)}{\partial y} = 0$$

sobre un dominio en dos dimensiones cuadrado de lado 1, con las condiciones de borde $u(0, y) = 1$, $u(x, 0) = 1$, $u(1, y) = 0$ y $u(x, 1) = 0$, discretizando este dominio en una malla de $N \times N$ nodos con el método de las diferencias finitas, tal como describimos en el Capítulo 21. Sin entrar en detalles que mostramos en el capítulo mencionado, destacamos que se forma una matriz rara

¹ La definición usual de la norma L_∞ es $\|x\|_\infty = \max\{|x_i| : i = 1, \dots, n\}$.

con estructura tridiagonal cuya construcción hace uso del método especializado para matrices con esta estructura `scipy.linalg.diags()`, tal como se ve en la celda siguiente:

CELDA 02

```
def set_A_b(N):
    """
    Construye la matriz A y el vector b para la ecuación de Laplace en 2D.

    Parámetros:
    - N: número de nodos interiores en cada dimensión (sin contar los bordes).

    Retorna:
    - A: matriz dispersa rala del sistema.
    - b: vector independiente.
    """
    # Tamaño del paso
    h = 1 / (N + 1)

    # Número total de nodos interiores
    num_nodes = N * N

    # Coeficientes para las diagonales de la matriz A
    main_diag = 4 * np.ones(num_nodes) # Diagonal principal
    side_diag = -1 * np.ones(num_nodes - 1) # Vecinos en la misma fila
    up_down_diag = -1 * np.ones(num_nodes - N) # Vecinos en filas adyacentes

    # Ajustar la diagonal secundaria (side_diag) para evitar conexiones entre filas
    for i in range(1, N):
        side_diag[i * N - 1] = 0

    # Construcción de A usando scipy.sparse.diags
    A = sp.diags([main_diag, side_diag, side_diag, up_down_diag, up_down_diag],
                 [0, -1, 1, -N, N], format="csc")

    # Construcción del vector b
    b = np.zeros(num_nodes)

    # Condiciones de borde no nulas
    for j in range(N): # Filas (y)
        for i in range(N): # Columnas (x)
            idx = j * N + i
            # Borde izquierdo (x = 0): u(0, y) = 1
            if i == 0:
                b[idx] += 1
            # Borde inferior (y = 0): u(x, 0) = 1
            if j == 0:
                b[idx] += 1

    return A, b
```

Ahora usamos esta función para construir una matriz cuadrada de N^2 filas (en el ejemplo generamos una matriz de 2.500 filas por 2.500 columnas). Mostramos que efectivamente es rala ya que menos del 5% de los elementos son no nulos, y obtenemos la solución con el método de Jacobi mostrando la cantidad de iteraciones que fueron necesarias para alcanzar la tolerancia establecida (y los primeros cuatro elementos de la solución):

CELDA 03

```
# Configuración del problema
N = 50 # Cantidad de puntos de la malla N x N
tol = 1.0e-6 # Tolerancia para la convergencia
max_iter = 10000 # Máximos número de iteraciones

A, b = set_A_b(N)
print(f"Elementos no nulos de A: {A.nnz}, {A.nnz / (N ** 2)} % del total.")

# Solución por el método de Jacobi
x_sol_jacobi, errores_jacobi = jacobi(A, b, tol=tol, max_iter=max_iter)
x_sol_jacobi[:4]

-----

Elementos no nulos de A: 12300, 4.92 % del total.
Convergencia en 3865 iteraciones (Jacobi).

array([0.99915622, 0.99831244, 0.99746866, 0.99662484])
```

La velocidad de convergencia del método iterativo puede aumentarse si notamos que, en la ecuación (18.8), para un valor de i determinado ya tenemos nuevas estimaciones para los componentes x_j^{k+1} para $j < i$. Esto da origen al método de Gauss-Seidel, que en forma explícita se puede escribir como

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right], \quad i = 1, \dots, n \quad (18.10)$$

Al igual que hicimos en el método de Jacobi, podemos expresar este sistema de ecuaciones lineales en forma matricial, descomponiendo la matriz $\mathbb{A}$ como la suma de una matriz diagonal $\mathbb{D}$, una triangular inferior $\mathbb{L}$ y una triangular superior $\mathbb{U}$. Utilizando estas matrices el sistema queda:

$$(\mathbb{D} + \mathbb{L}) \mathbf{x} = \mathbf{b} - \mathbb{U} \mathbf{x}$$

que origina el esquema iterativo

$$\mathbf{x}^{(k+1)} = (\mathbb{D} + \mathbb{L})^{-1} [\mathbf{b} - \mathbb{U} \mathbf{x}^{(k)}] \quad (18.11)$$

En este caso, el cálculo de la inversa de $\mathbb{D} + \mathbb{L}$ es apenas más complejo que el cálculo para el caso del método de Jacobi, debido a que por ser una matriz triangular deben realizarse solo sustituciones hacia adelante por lo que no es tan costoso computacionalmente. Aprovechando nuevamente esta representación matricial del método iterativo, podemos escribir una función que implemente este proceso:

CELDA 04

```
import scipy.sparse.linalg as spla

def gauss_seidel(A, b, tol=1e-6, max_iter=1000):
    """Resuelve A x = b por el método de Gauss-Seidel."""
    n = len(b)
    x = np.zeros(n) # Solución inicial
    errores = []
```

```

# Extraemos L, D y U de A
D = sp.diags(A.diagonal(), format='csc') # Matriz diagonal
L = sp.tril(A, k=-1, format='csc')      # Parte estrictamente inferior
U = sp.triu(A, k=1, format='csc')      # Parte estrictamente superior

# Descomposición: (L + D)^(-1)
LD_inv = spla.inv(L + D)

for k in range(max_iter):
    # Actualización vectorizada
    x_new = LD_inv @ (b - U @ x)
    error = np.linalg.norm(x_new - x, ord=np.inf)
    errores.append(error)
    # Comprobar convergencia
    if error < tol:
        print(f"Convergencia en {k + 1} iteraciones (Gauss-Seidel)")
        return x_new, errores

    x = x_new

print("No convergió (Gauss-Seidel)")
return x, errores

```

La estructura de la función es análoga a la correspondiente al método de Jacobi, solamente cambiando la forma de extraer las matrices diagonales y el esquema matricial de iteración. Si aplicamos ahora este método a la solución del sistema anterior, obtenemos:

CELDA 05

```

# Solución por el método de Gauss-Seidel
x_sol_gs, errores_gs = gauss_seidel(A, b, tol=tol, max_iter=max_iter)
x_sol_gs[:4]

```

Convergencia en 2105 iteraciones (Gauss-Seidel)

```
array([0.99915712, 0.99831425, 0.99747138, 0.99662845])
```

Podemos ver que se necesitan menos iteraciones que con el método de Jacobi para alcanzar la tolerancia requerida. A continuación visualizamos la evolución del error a medida que avanzan las iteraciones hasta alcanzar la tolerancia establecida.

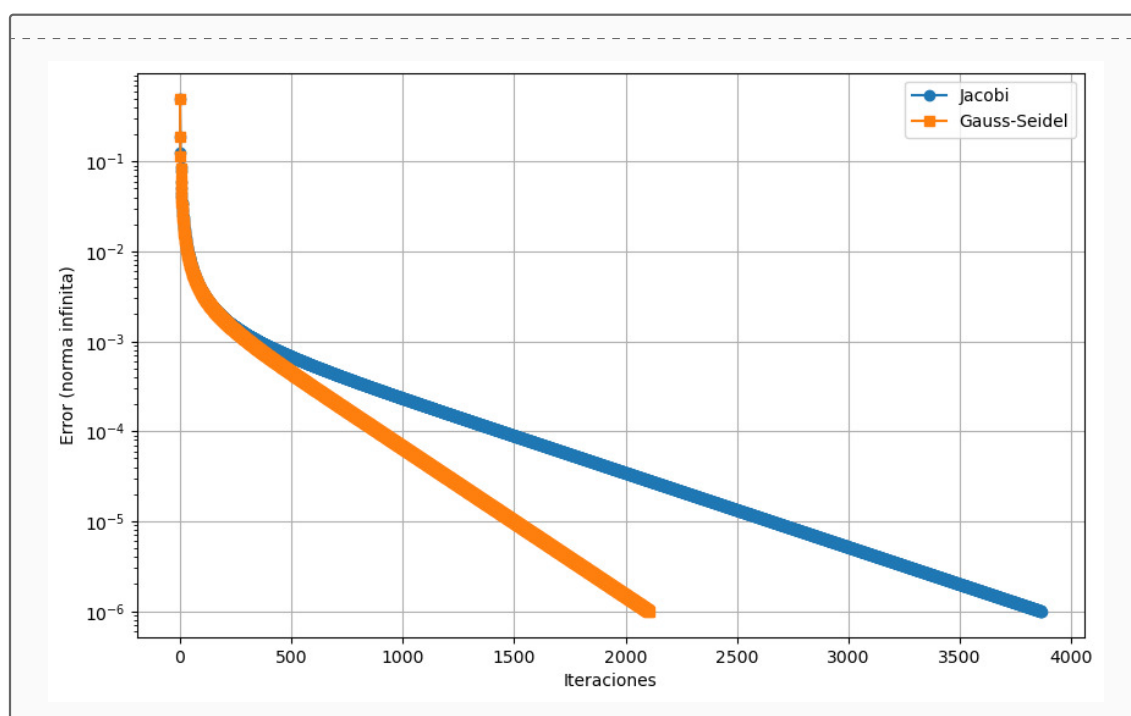
CELDA 06

```

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.semilogy(errores_jacobi, label='Jacobi', marker='o')
plt.semilogy(errores_gs, label='Gauss-Seidel', marker='s')
plt.xlabel('Iteraciones')
plt.ylabel('Error (norma infinita)')
plt.legend()
plt.grid()
plt.show()

```



Al igual que para el método de Jacobi, el método de Gauss-Seidel es convergente si la matriz $\mathbb{A}$ es diagonalmente dominante. En términos más generales, la convergencia de estos métodos se puede analizar en función de las matrices que componen el operador iterativo, esto es, $\mathbb{D}^{-1}(\mathbb{L}+\mathbb{U})$ para el caso de Jacobi (ver la ecuación (18.9)) y $(\mathbb{D} + \mathbb{L})^{-1}\mathbb{U}$ para Gauss-Seidel (ver (18.11)). En ambos casos, la convergencia de ambos métodos está asegurada cuando los radios espectrales de estas matrices es menor que 1.

El método de Jacobi y el de Gauss-Seidel pueden considerarse «iterativos puros», y existe una amplia variedad de estos métodos que mejoran la velocidad de convergencia. Sin embargo, para problemas efectivamente grandes se utilizan actualmente métodos basados en los espacios de Krylov (cuya base contiene todas las combinaciones de $\mathbf{b}$, $\mathbb{A}\mathbf{b}$, $\mathbb{A}^2\mathbf{b}$, $\mathbb{A}^3\mathbf{b}$, ...). Los métodos de Krylov buscan un subespacio con la mejor combinación de los elementos de la base. Cuando esta base alcanza todo el espacio $\mathbb{R}^n$, los métodos de Krylov dan la solución exacta $\mathbb{A}^{-1}\mathbf{b}$, pero por lo general los algoritmos se detienen mucho antes de completar los n pasos. Dejaremos para las lecturas recomendadas los detalles técnicos de estos métodos, pero veremos un ejemplo de cómo se utiliza uno de ellos (GMRES, por *Generalized Minimal RESidual*) provisto por `scipy.sparse.linalg`.

En la celda siguiente realizamos la llamada a la función `spla.gmres()` pasando como argumentos la matriz $\mathbf{A}$, el vector $\mathbf{b}$ y la tolerancia `tol`. El método devuelve la solución obtenida si hubo convergencia (y en este caso `num_it = 0`) o el número realizado de iteraciones (`num_it > 0`) si no hubo convergencia.

CELDA 07

```
x_gmres, num_it = spla.gmres(A, b, rtol=tol)
if num_it:
    print(f"No hubo convergencia en {num_it} iteraciones.")
else:
```

```
print(x_gmres[:4])

[0.99915818 0.99831629 0.99747435 0.99663237]
```

Para concluir esta sección mencionamos que existen bibliotecas especializadas para trabajar en sistemas de ecuaciones lineales muy grandes (e incluso no lineales), entre las cuales podemos destacar [PETSc for Python](#).

18.2. Problema de autovalores

Así como los sistemas de ecuaciones lineales suelen originarse a partir de problemas en estado estacionario, los problemas de autovalores son de gran importancia en la representación de problemas dinámicos, por lo que aparecen frecuentemente en muchas áreas de la ciencia y de la ingeniería.

Si $\mathbb{A}$ es una matriz cuadrada $n \times n$, el problema de autovalores de esta matriz se define como

$$\mathbb{A} \cdot \mathbf{x} = \lambda \mathbf{x} \quad (18.12)$$

donde λ un escalar denominado “autovalor” y $\mathbf{x}$ es el “autovector” correspondiente, ambos desconocidos. Obviamente, cualquier múltiplo de un autovector es también un autovector, por lo que no se consideran autovectores distintos (el autovector cero no se considera un autovector). Resolver el problema de autovalores (18.12) consiste en determinar este par autovalor/autovector.

Para abordar el problema, podemos introducir la matriz identidad $\mathbb{I}$ (matriz cuyos elementos diagonales son unos y el resto son ceros), y reescribir la ecuación (18.12) como

$$(\mathbb{A} - \lambda \mathbb{I}) \cdot \mathbf{x} = \mathbf{0} \quad (18.13)$$

Esta expresión nos dice que la matriz $\mathbb{A} - \lambda \mathbb{I}$ multiplicada por el vector $\mathbf{x}$ nos da el vector nulo, es decir, los autovectores construyen el subespacio nulo de $\mathbb{A} - \lambda \mathbb{I}$. Para que esta expresión sea útil, el subespacio nulo debe contener otros vectores distintos del vector cero, lo que implica que $\mathbb{A} - \lambda \mathbb{I}$ debe ser *singular*, y esto ocurre si y solo si

$$\det(\mathbb{A} - \lambda \mathbb{I}) = 0 \quad (18.14)$$

que da origen a una ecuación polinómica en λ denominada “ecuación característica” de grado n . El Teorema Fundamental del Álgebra asegura que la matriz $\mathbb{A}$, de $n \times n$ elementos, siempre tiene n autovalores, aunque éstos pueden no ser reales ni tampoco todos distintos (autovalores iguales que provienen de raíces múltiples se denominan “degenerados”). Los autovalores complejos de una matriz real siempre ocurren en pares conjugados, esto es, si $\alpha + i\beta$ es un autovalor de una matriz real, también lo es $\alpha - i\beta$, siendo $i = \sqrt{-1}$ la unidad imaginaria. Una vez determinados los autovalores se pueden obtener los autovectores resolviendo la ecuación (18.13).

Tanto SymPy como SciPy tienen rutinas para calcular autovalores y autovectores. En el caso de SymPy podemos resolver un problema de autovalores con elementos simbólicos en la matriz $\mathbb{A}$. Por ejemplo, para la siguiente matriz simétrica:

CELDA 1 DE AUTOVALORES.IPYNB

```
import sympy as sy

sy.init_printing()
a, b = sy.symbols("alpha, beta")
A = sy.Matrix([[a, b],[b, -a]])
A
```

$$\begin{bmatrix} \alpha & \beta \\ \beta & -\alpha \end{bmatrix}$$

los autovalores se obtienen con `eigenvals()`:

CELDA 02

```
A.eigenvals()
```

$$\left\{ -\sqrt{\alpha^2 + \beta^2} : 1, \sqrt{\alpha^2 + \beta^2} : 1 \right\}$$

La salida de `eigenvals()` es un diccionario, cuya clave es el autovalor y su valor es la multiplicidad del autovalor. Para obtener los autovectores de la matriz, utilizamos el método `eigenvects()`:

CELDA 03

```
A.eigenvects()
```

$$\left[\left(-\sqrt{\alpha^2 + \beta^2}, 1, \left[\begin{bmatrix} \frac{\alpha}{\beta} - \frac{\sqrt{\alpha^2 + \beta^2}}{\beta} \\ 1 \end{bmatrix} \right] \right), \left(\sqrt{\alpha^2 + \beta^2}, 1, \left[\begin{bmatrix} \frac{\alpha}{\beta} + \frac{\sqrt{\alpha^2 + \beta^2}}{\beta} \\ 1 \end{bmatrix} \right] \right) \right]$$

En este caso, la salida es una lista cuyos elementos son tuplas. A su vez, cada tupla está compuesta por el autovalor, la multiplicidad del mismo y el correspondiente autovector. Podemos desempacar esta información (notando que el autovector es una lista de un solo elemento) para verificar que los autovectores son ortogonales²:

CELDA 04

```
(val1, _, vec1), (val2, _, vec2) = A.eigenvects()
sy.simplify(vec1[0].T * vec2[0])
```

$$\begin{bmatrix} 0 \end{bmatrix}$$

Por supuesto que también podemos usar Sympy para resolver el problema de autovalores con matrices con elementos fraccionales, por ejemplo nuestra conocida matriz de Hilbert 2×2 :

² Los autovectores de una matriz normal, como la simétrica del ejemplo, con autovalores no degenerados (es decir, todos distintos) forman la base de un espacio vectorial n -dimensional.

CELDA 05

```
H_s = sy.Matrix([[1, sy.Rational(1,2)], [sy.Rational(1,2), sy.Rational(1,3)]])
H_s.eigenvals()
```

$$\left\{ \frac{2}{3} - \frac{\sqrt{13}}{6} : 1, \frac{\sqrt{13}}{6} + \frac{2}{3} : 1 \right\}$$

CELDA 06

```
H_s.eigenvects()
```

$$\left[\left(\frac{2}{3} - \frac{\sqrt{13}}{6}, 1, \begin{bmatrix} \frac{2}{3} - \frac{\sqrt{13}}{3} \\ 1 \end{bmatrix} \right), \left(\frac{\sqrt{13}}{6} + \frac{2}{3}, 1, \begin{bmatrix} \frac{2}{3} + \frac{\sqrt{13}}{3} \\ 1 \end{bmatrix} \right) \right]$$

El abordaje simbólico se hace impracticable para matrices mayores a 3×3 debido a la dificultad de encontrar las raíces de la ecuación característica. De echo, el teorema de Abel-Rufini³ establece que no existe una solución general para la solución de ecuaciones polinómicas de grado cinco o mayor. Por esto tienen especial importancia los métodos numéricos para la solución del problema de autovalores.

Pese a que los problemas de autovalores son en cierta forma similares a los sistemas de ecuaciones lineales que vimos en la sección anterior, no podemos utilizar los mismos métodos para resolverlos: si a una fila de $\mathbb{A}$ le sumamos otra fila, o intercambiamos filas, los autovalores generalmente cambian. Es por esto que existen rutinas especializadas que resuelven eficientemente los problemas de autovalores (cuya descripción excede el alcance de este libro), y que por lo general obtienen una solución parcial. Por ejemplo, pueden obtener todos los autovalores y ningún autovector, o solo algunos autovectores, o solo los primeros autovalores mayores, o solo los negativos. Estas especializaciones permiten obtener resultados útiles sobre matrices grandes ahorrando tiempo de cálculo y espacio de almacenamiento.

SciPy tiene varias funciones para calcular autovalores y autovectores en el submódulo `linalg`, especializadas en abordar matrices con determinadas estructuras. Para matrices generales disponemos de `eigvals` (para autovalores) y `eig` para obtener autovalores y autovectores. Por ejemplo, para la matriz de Hilbert 4×4 podemos calcular sus autovalores y autovectores con `eig`:

CELDA 07

```
import scipy.linalg as la
```

```
H = la.hilbert(4)
H
```

```
array([[1.         , 0.5        , 0.33333333, 0.25       ],
       [0.5        , 0.33333333, 0.25       , 0.2        ],
       [0.33333333, 0.25       , 0.2        , 0.16666667],
       [0.25       , 0.2        , 0.16666667, 0.14285714]])
```

³ Ver entrada en [Wikipedia](#).

CELDA 08
<pre>vals, vects = la.eig(H) vals</pre>
<pre>array([1.50021428e+00+0.j, 1.69141220e-01+0.j, 6.73827361e-03+0.j, 9.67023040e-05+0.j])</pre>

CELDA 09
<pre>vects</pre>
<pre>array([[0.79260829, 0.5820757 , 0.17918629, -0.02919332], [0.45192312, -0.37050219, -0.74191779, 0.32871206], [0.3224164 , -0.50957863, 0.10022814, -0.79141115], [0.25216117, -0.51404827, 0.63828253, 0.51455275]])</pre>

Cada uno de los autovectores aparece como una fila del array mostrado en la celda 9. A diferencia de los valores que se obtienen con Sympy, estos autovalores se encuentran normalizados. En la celda 10 podemos verificar esta normalización (con precisión cercana al épsilon de la máquina), y también que los cuatro autovectores son ortogonales:

CELDA 10
<pre>for i in range(len(vects)): for j in range(i, len(vects)): print(f'v[{i}] . v[{j}] = {vects[i] @ vects[j]:+.15f}')</pre>
<pre>v[0] . v[0] = +1.000000000000000 v[0] . v[1] = -0.000000000000000 v[0] . v[2] = +0.000000000000000 v[0] . v[3] = -0.000000000000000 v[1] . v[1] = +1.000000000000001 v[1] . v[2] = -0.000000000000001 v[1] . v[3] = +0.000000000000000 v[2] . v[2] = +1.000000000000001 v[2] . v[3] = +0.000000000000001 v[3] . v[3] = +0.999999999999999</pre>

En la celda 8 vemos que los autovalores se muestran en formato de número complejo (SciPy utiliza la letra *j* para denotar la parte compleja), aunque por ser la matriz de Hilbert una matriz real simétrica, todos sus autovalores son reales. `scipy.linalg` tiene una rutina específica para trabajar con matrices simétricas o hermiticas, `eigh`, que devuelve un array con los autovalores en formato de punto flotante real:

CELDA 11
<pre>vals_real, vects_real = la.eigh(H) vals_real</pre>
<pre>array([9.67023040e-05, 6.73827361e-03, 1.69141220e-01, 1.50021428e+00])</pre>

Podemos verificar dos propiedades de los autovalores, la que establece que la suma de los mismos es igual a la traza de la matriz:

CELDA 12

```
import numpy as np
vals_real.sum() - np.trace(H)
```

$-4,44089209850063 \cdot 10^{-16}$

y que el producto de los autovalores dan el determinante de la matriz:

CELDA 13

```
np.prod(vals_real) - la.det(H)
```

$1,96670462440608 \cdot 10^{-20}$

Además de la función para matrices simétricas/hermíticas, `linalg` tiene también rutinas para calcular autovalores y autovectores en matrices banda (`eig_banded`, `eigvals_banded` para solo autovalores), y tridiagonales (`eigh_tridiagonal`, `eigvalsh_tridiagonal` para obtener unicamente los autovalores). En el caso de trabajar con matrices ralas, el submódulo de SciPy `sparse.linalg` tiene las funciones `eigs` para obtener autovalores y autovectores de una matriz arbitraria, y `eigsh` para matrices simétricas o hermíticas.

Cabe destacar que tanto `linalg.eig` como `sparse.linalg.eig`, y sus versiones para matrices simétricas/hermíticas, permiten resolver también el problema de autovalores general

$$\mathbb{A} \cdot \mathbf{v} = \lambda \mathbb{B} \cdot \mathbf{v} \quad (18.15)$$

para las matrices cuadradas $\mathbb{A}$ y $\mathbb{B}$, que es una generalización del problema de autovalores usual con $\mathbb{A} = \mathbb{I}$.

18.3. Ecuaciones no lineales

El problema de resolver ecuaciones no lineales aparece frecuentemente en todas las áreas de la ciencia y la tecnología. A diferencia de los sistemas lineales, las soluciones de ecuaciones no lineales no pueden obtenerse fácilmente por sustitución u otro método directo, por lo cual es necesario recurrir a métodos iterativos en los cuales comenzamos con una estimación inicial de la solución y luego nos acercamos a ella mediante aproximaciones sucesivas hasta satisfacer algún criterio de convergencia.

La forma general del problema puede enunciarse simplemente como

$$f(x) = 0 \quad (18.16)$$

donde f es una función no lineal de x . Cuando hay solo una variable dependiente, el problema es unidimensional o univariado, y resolver el problema consiste en hallar la raíz o raíces de la función. Un conjunto de n ecuaciones no lineales puede representarse en notación vectorial, y se trata de encontrar uno o más vectores solución $\mathbf{x}$ (de dimensión n) tal que

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad (18.17)$$

donde $\mathbf{f}$ es una función vectorial n -dimensional cuyas componentes son las ecuaciones que deben satisfacerse simultáneamente. Pese a la aparente similitud que existe entre las ecuaciones

(18.16) y (18.17), resolver sistemas de ecuaciones no lineales es *mucho* más difícil que el caso unidimensional, fundamentalmente porque en este último caso es posible encerrar la solución en un intervalo (*bracketing*), y mediante un algoritmo reducir este intervalo hasta dar con la raíz, mientras que en el caso multidimensional esto no es tan simple.

Una complejidad adicional proviene de la existencia de raíces múltiples, especialmente si la multiplicidad de la raíz es un número par. Esto puede cuantificarse mediante el número de condición que, del mismo modo que vimos en la subsección 18.1.1, establece una medida de cuánto afecta un cambio en el valor de la función al cambio del valor de su argumento. Para establecer este número de condición, llamemos x^* a la raíz de la ecuación (18.16), y $\tilde{x}^*$ a nuestra aproximación a la raíz. Expandiendo en serie de Taylor $f(\tilde{x}^*)$ alrededor de x^* , obtenemos

$$f(\tilde{x}^*) = f(x^* + \delta x^*) = f(x^*) + f'(x^*)\delta x^* + O(\delta x^{*2}) \quad (18.18)$$

en donde usamos $\delta x^* = \tilde{x}^* - x^*$. Al asumir que δx^* es un valor pequeño, podemos despreciar los términos de orden superior en (18.18), y entonces resulta:

$$\delta x^* = \tilde{x}^* - x^* \approx \frac{1}{f'(x^*)} [f(\tilde{x}^*) - f(x^*)] \quad (18.19)$$

lo que nos permite definir el “número de condición” de nuestro problema:

$$\kappa_f = \frac{1}{f'(x^*)} \quad (18.20)$$

Esta magnitud cuantifica el efecto de un pequeño cambio en el valor de la función sobre la evaluación de la raíz. En efecto, una función que cruza “rápidamente” el eje x tiene un valor grande de $f'(x^*)$, y por lo tanto un número de condición κ_f pequeño, lo que define un problema *bien condicionado*. Por otra parte, si la función es bastante “plana” alrededor de x^* , cruzando el eje x suavemente, esto corresponde a un valor pequeño de $f'(x^*)$ y en consecuencia a un valor grande de κ_f , lo que caracteriza a un problema “mal condicionado”. Este condicionamiento afecta a la dificultad de encontrar la raíz de f , ya que puede ser difícil distinguirla de valores vecinos.



Isaac Newton - Físico, teólogo, inventor, alquimista y matemático inglés (1643-1727). Describió la ley de la gravitación universal y estableció las bases de la mecánica clásica mediante las leyes que llevan su nombre. Entre sus otros descubrimientos científicos destacan los trabajos sobre la naturaleza de la luz y la óptica, y en matemáticas el desarrollo del cálculo infinitesimal.

18.3.1. Ecuaciones no lineales unidimensionales

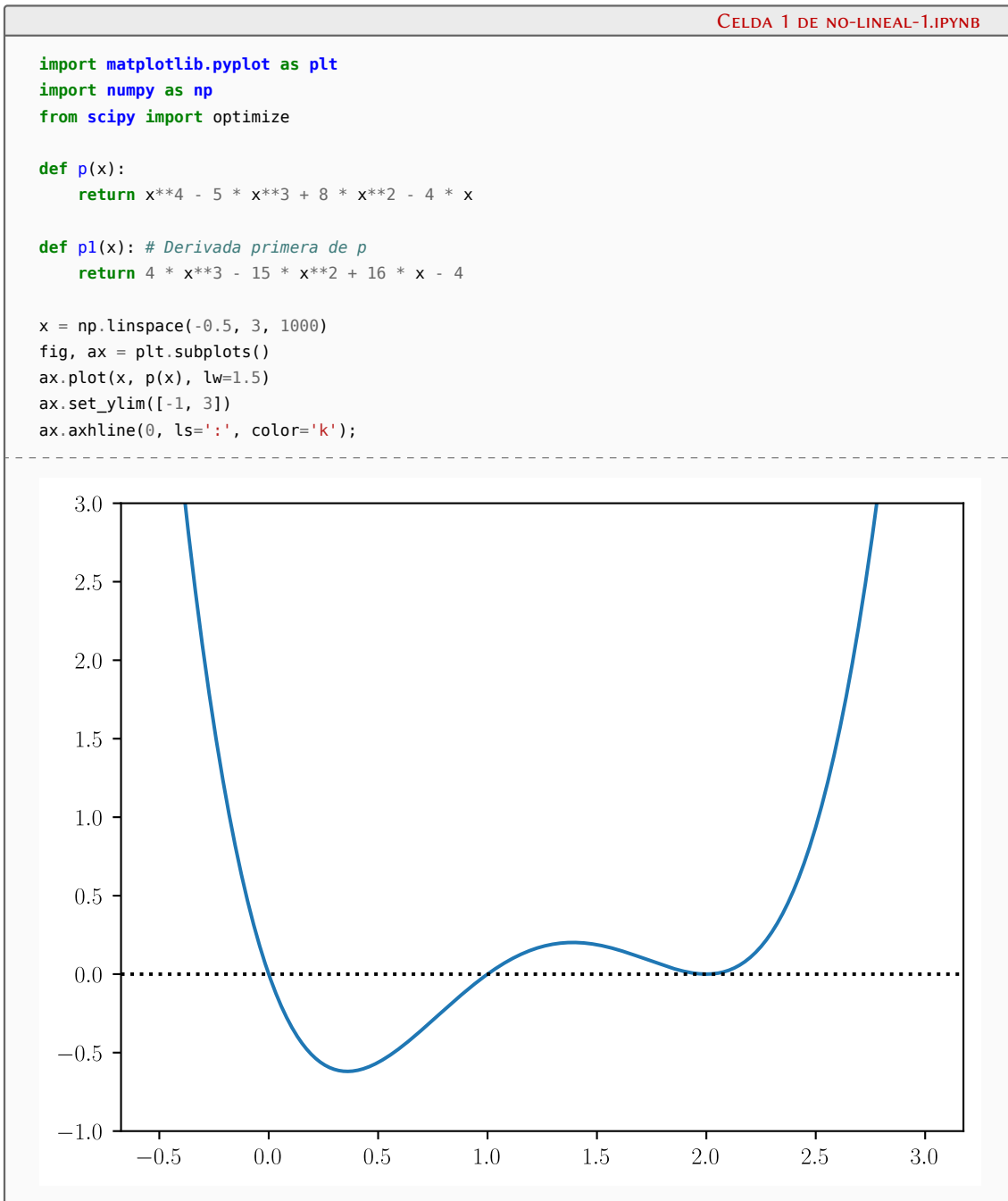
Para abordar estos problemas siempre es conveniente “encerrar” la raíz en un intervalo y para ello es útil visualizar el comportamiento de la función. Los métodos cerrados, o de *bracketing*, consisten en determinar un intervalo inicial donde la función cambia de signo, y si dicha función es continua en el intervalo entonces los algoritmos de *bracketing* garantizan la convergencia sobre la raíz (aunque pueden hacerlo con diferente velocidad).

Por otra parte, cuando la raíz se encuentra en un intervalo en que la función no cambia de signo (como en el caso de raíces con multiplicidad par), es necesario utilizar métodos abiertos que no garantizan la convergencia a la solución. El submódulo `optimize` de SciPy ofrece una variedad de métodos tanto cerrados como abiertos. Veremos aquí algunos ejemplos de uso.

Comenzaremos analizando un polinomio de cuarto grado, cuyas raíces pueden obtenerse fácilmente por medio de su factorización, de modo que sabemos de antemano cuáles son para evaluar el desempeño de los algoritmos. El polinomio

$$p(x) = x^4 - 5x^3 + 8x^2 - 4x$$

tiene raíces simples en $x_1 = 0$ y $x_2 = 1$, y una raíz doble en ($x_{34} = 2$) Si graficamos esta función, tal como vemos en la celda 1 del *jupyter-notebook* siguiente



vemos que $p(x)$ está bien condicionada en x_1 ($\kappa_p = -1/4$). En el submódulo encontramos la función `root_scalar` que permite hallar la raíz de una función mediante diversos métodos. Utili-

haremos para el ejemplo el método de Brent, que combina un método de *bracketing* como el de bisección con el método de la secante y una interpolación cuadrática inversa, lo que brinda un algoritmo robusto y de rápida convergencia.

CELDA 02
<pre>sol_1 = optimize.root_scalar(p, method='brentq', bracket=[-0.5, 0.5]) sol_1</pre>
<pre>converged: True flag: converged function_calls: 11 iterations: 10 root: 1.2352144989557884e-14 method: brentq</pre>
CELDA 03
<pre>sol_2 = optimize.root_scalar(p, method='brentq', bracket=[0.5, 1.5]) sol_2</pre>
<pre>converged: True flag: converged function_calls: 10 iterations: 9 root: 1.0000000000000004 method: brentq</pre>

En la celda 2 vemos que llamamos a la función `root_scalar` pasando como primer argumento a la función `p`, luego seleccionamos el método de solución `'brentq'`, y por último establecemos el intervalo que encierra la raíz buscada, que cumple con la condición que $p(x)$ tiene signos opuestos en los extremos del intervalo. Como resultado de la función `root_scalar`, obtenemos un objeto `RootResults` que almacenamos en la variable `sol_1`, y que contiene diversos atributos que dan cuenta del resultado de la aplicación del algoritmo como el motivo de la finalización (`flag`), si hubo convergencia (`converged`), la cantidad de iteraciones y llamadas a la función, y por supuesto, la raíz encontrada. En la celda 3 volvemos a aplicar el método de Brent para hallar x_2 , encerrando esta raíz en el intervalo $[1/2, 3/2]$.

La situación es diferente para la raíz doble en x_{34} , donde la función presenta un mal condicionamiento ya que también la derivada de $p(x)$ se anula en ese punto. Si queremos determinar la raíz, no podemos usar un método de *bracketing* dado que la función no cambia de signo a ambos lados de la raíz. Podemos acudir entonces a métodos abiertos, como el de Newton, que no nos garantiza la convergencia a la solución pero si partimos de una estimación inicial suficientemente cercana, tenemos buenas posibilidades de éxito. La celda 4 muestra el uso de la función `root_scalar` seleccionando el método de Newton. A diferencia del caso anterior, no especificamos un intervalo que encierra a la raíz, pero si un valor inicial x_0 . Además, el método de Newton requiere especificar también la derivada de la función en el argumento `fprime`, al que le pasamos la función `p1(x)` que definimos en la celda 1 (si no disponemos de una expresión para la derivada, es posible utilizar el método de la secante, en el que la derivada se aproxima por diferencia finita):

CELDA 04

```
sol_3 = optimize.root_scalar(p, method='newton', x0=2.5, fprime=p1)
sol_3
```

```
converged: True
flag: converged
function_calls: 61
iterations: 30
root: 1.9999999814380152
method: newton
```

Puede verse que en comparación con el método de Brent, el método de Newton es más costoso computacionalmente debido a la complejidad misma del método (que requiere evaluar dos funciones). Además en este caso la convergencia es más lenta debido al mal condicionamiento de la función en x_{34} .

Para mostrar la importancia de la elección de la estimación inicial, notemos que $p(x)$ tiene un máximo relativo en $x_m = (\sqrt{17} + 7)/8 \approx 1,3904$. Si el valor de x_0 se encuentra a la derecha de x_m , el método de Newton convergerá a la solución buscada, tal como muestra la celda 5, mientras que si nuestra estimación inicial está a la izquierda de x_m , el método converge a x_1 como puede verse en la celda 6. Una variación de apenas 0.072 % respecto del valor de x_m produce un cambio muy grande en la raíz encontrada, incluso “saltando” la raíz en x_2 . Si nuestro valor inicial se encuentra más próximo al valor de x_m , el método no converge a ninguna raíz en el número máximo de iteraciones por defecto (50). Aumentando el número máximo de iteraciones vemos que converge a x_{34} en 70 iteraciones (pero no siempre tenemos esa suerte).

CELDA 05

```
sol_md = optimize.root_scalar(p, method='newton', x0=1.391, fprime=p1)
sol_md
```

```
converged: True
flag: converged
function_calls: 87
iterations: 43
root: 2.00000002189259
method: newton
```

CELDA 06

```
sol_mi = optimize.root_scalar(p, method='newton', x0=1.390, fprime=p1)
sol_mi
```

```
converged: True
flag: converged
function_calls: 50
iterations: 25
root: -5.9164567891575885e-31
method: newton
```

CELDA 07

```
sol_m = optimize.root_scalar(p, method='newton', x0=1.3904, fprime=p1)
sol_m
```

```
converged: False
flag: convergence error
function_calls: 100
iterations: 50
root: 2.000002582731664
method: newton
```

CELDA 08

```
sol_m71 = optimize.root_scalar(p, method='newton', x0=1.3904, fprime=p1, maxiter=71)
sol_m71
```

```
converged: True
flag: converged
function_calls: 141
iterations: 70
root: 2.000000047778081
method: newton
```

Se puede suponer que podemos encontrar x_{34} utilizando un método de *bracketing* utilizando un intervalo más grande que contenga a esta raíz. Sin embargo, esto no siempre funciona, tal como muestra el resultado de la celda 9 en el que el método de Brent utilizando el intervalo $[1/2, 3/2]$ encuentra la raíz en x_1 , pero no la raíz doble.

CELDA 09

```
sol_x = optimize.root_scalar(p, method='brentq', bracket=[0.5, 2.5])
sol_x
```

```
converged: True
flag: converged
function_calls: 10
iterations: 9
root: 1.00000000000002822
method: brentq
```

18.3.2. Sistema de ecuaciones no lineales

Tal como anticipamos al comienzo de la sección 18.3, resolver un sistema de ecuaciones no lineales es mucho más complejo que el caso unidimensional, en parte debido a la gran variedad de escenarios posibles. Para mencionar solo un aspecto de esta complejidad, suele ser habitual no saber si existe solución del problema, o en caso de existir puede que no sea posible determinar cuántas hay. En consecuencia no existe un método que garantice la convergencia a una solución, tal como el método de bisección para el caso unidimensional. Por otra parte, los métodos disponibles son mas costosos computacionalmente que para el caso multidimensional, y este costo crece aceleradamente al aumentar el número de incógnitas.

En este contexto, SciPy ofrece un conjunto de métodos para abordar la solución de estos sistemas. Estos métodos pueden agruparse según el número de ecuaciones a resolver. El método

híbrido de Powell (`hibr`) y la implementación del método de Levenberg-Marquardt (`lm`) de MINPACK son eficientes para un sistema de pocas ecuaciones. Cuando el número de ecuaciones n aumenta, la aplicación de estos métodos se vuelve ineficiente debido a que requieren la inversión de una matriz jacobiana densa $n \times n$ en cada iteración. Para estos casos, SciPy ofrece funciones que implementan el método inexacto de Newton, en los que la matriz jacobiana se evalúa en forma aproximada. Estos métodos son `anderson`, `broyden2` y `krylov`.

Como ejemplo (sencillo) de resolución de un sistema no lineal de ecuaciones, determinaremos las raíces del sistema

$$\begin{cases} f_1(x, y) = y - x^2 + x \cos(\pi x) = 0 \\ f_2(x, y) = y - x^3 + 4x - \ln(y + 1)^4 = 0 \end{cases} \quad (18.21)$$

Por ser un sistema con dos incógnitas, cada función define una superficie tridimensional que cruza el plano xy . El siguiente *script* genera la figura 18.1 en donde visualizamos para cada función la curva de nivel cero, es decir, las intersecciones de las superficies generadas por $f_1(x, y)$ y $f_2(x, y)$ con el plano xy (ver las líneas 16 y 18 del *script*). Esto significa que sobre cada una de esas curvas las funciones se anulan individualmente. Y en los puntos donde las curvas se cruzan, ambas funciones se anulan *simultáneamente*. Esto puede orientarnos a proponer valores iniciales para los algoritmos de solución que estén cercanos a los puntos que son solución del sistema.

```

1 #!/usr/bin/env python3
2
3 from math import pi
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 n_puntos = 300
9 xs = np.linspace(-3, 3, n_puntos)
10 ys = np.linspace(-5, 15, n_puntos)
11 X, Y = np.meshgrid(xs, ys)
12 Z1 = Y - X**2 + X * np.cos(pi * X)
13 Z2 = Y - (X - 2) * (X + 2) * X - np.log((Y + 1) ** 4)
14 fig, ax = plt.subplots()
15 CS1 = ax.contour(X, Y, Z1, levels=[0], colors=['royalblue'])
16 ax.clabel(CS1, levels=[0], fmt={0: r'$f_1$'}, fontsize=12)
17 CS2 = ax.contour(X, Y, Z2, levels=[0], colors=['maroon'])
18 ax.clabel(CS2, levels=[0], fmt={0: r'$f_2$'})
19 plt.savefig('f1_f2_ceros.pdf')

```

En el siguiente programa determinaremos las soluciones del sistema no lineal de ecuaciones (18.21), y exploraremos las regiones de convergencia a cada solución. En las líneas 3–6 importamos los módulos NumPy, Matplotlib, las funciones coseno, logaritmo (natural) y la constante π del módulo `math`, y el submódulo `optimize` de SciPy. En la línea 8 especificamos que el renderizado de la figura se haga utilizando $\text{\LaTeX}$ por imprescindibles cuestiones estéticas.

```

1 #!/usr/bin/env python3
2
3 from math import cos, log, pi

```

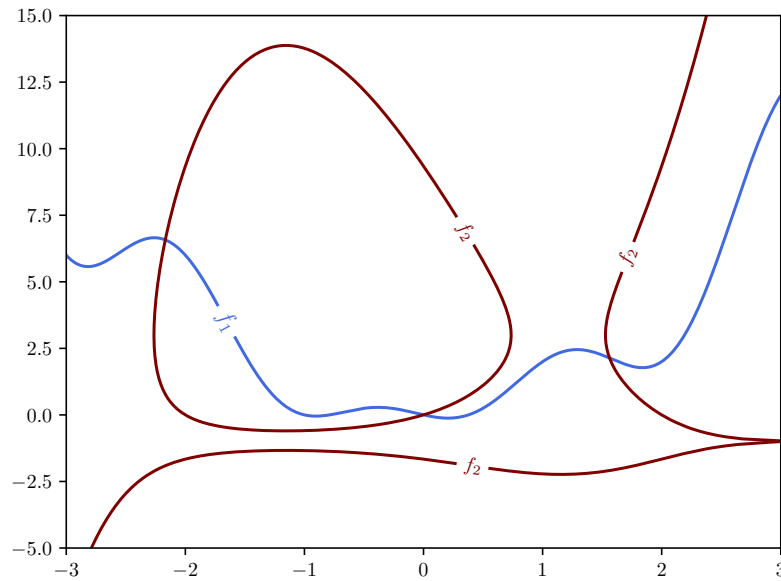


FIGURA 18.1: Ceros de las funciones del sistema no lineal de ecuaciones.

```

4
5 import numpy as np
6 import matplotlib.pyplot as plt
7 from scipy import optimize
8
9 n_puntos = 300
10 xs = np.linspace(-3, 3, n_puntos)
11 ys = np.linspace(-5, 15, n_puntos)
12 X, Y = np.meshgrid(xs, ys)
13 Z1 = Y - X**2 + X * np.cos(pi * X)
14 Z2 = Y - (X - 2) * (X + 2) * X - np.log((Y + 1)**4)
15 fig, ax = plt.subplots()
16 CS1 = ax.contour(X, Y, Z1, levels=[0], colors=['royalblue'])
17 ax.clabel(CS1, levels=[0], fmt={0: r'$f_1$'}, fontsize=12)
18 CS2 = ax.contour(X, Y, Z2, levels=[0], colors=['maroon'])
19 ax.clabel(CS2, levels=[0], fmt={0: r'$f_2$'})
20
21
22 def f(x):
23     return [
24         x[1] - x[0]**2 + x[0] * cos(pi * x[0]),
25         x[1] - x[0]**3 + 4 * x[0] - log((x[1] + 1)**4)
26     ]
27
28
29 x_guess = [[-2.0, 5.0], [0, 0], [1.5, 2.0]]
30 sols = []
31 for x_g in x_guess:
32     sol = optimize.root(f, x_g, method='hybr')
33     print(sol.message)

```

```

34     print(f'x: [{sol.x[0]:.3f}, {sol.x[1]:.3f}] -- root: {f(sol.x)}')
35     sols.append(sol.x)
36
37     colors = ['r', 'b', 'g']
38     for i, s in enumerate(sols):
39         ax.plot(s[0], s[1], colors[i] + '*', markersize=15)
40
41     n_trial = 100
42     xs = np.linspace(-3, 3, n_trial)
43     ys = np.linspace(-5, 15, n_trial)
44     n_div = 0
45     for xc in xs:
46         for yc in ys:
47             ss = optimize.root(f, [xc, yc], method='hybr')
48             if ss.success:
49                 idx = (abs(sols - ss.x)**2).sum(axis=1).argmin()
50                 ax.plot(xc, yc, marker='.', color=colors[idx], alpha=0.15)
51             else:
52                 n_div += 1
53     print(f'Proporción de no convergencia: {n_div / n_trial**2:.4f}')
54     plt.ylim([-5, 15])
55     plt.savefig('sol_hybr.pdf')

```

Entre las líneas 10 y 20 generamos nuevamente el mismo gráfico que mostramos en la figura 18.1, que usaremos como base para mostrar encima las raíces y las cuencas de convergencia. Implementamos la definición del sistema de ecuaciones en la función $f(x)$ de las líneas 23–25, que tiene como argumento una lista de dos elementos ($x \mapsto x[0]$, $y \mapsto x[1]$), y devuelve también una lista con dos elementos que representa cada función evaluada en los argumentos.

En la línea 28 definimos la lista `x_guess` que contiene tres pares de valores iniciales con los que vamos a iniciar la búsqueda de soluciones. Estos valores fueron elegidos inspeccionando visualmente la figura 18.1 de modo que estén “cerca” de los lugares donde se cruzan las curvas de nivel cero de f_1 y f_2 . Si encontramos soluciones, las iremos almacenando en la lista `sols`.

En el bucle de las líneas 29–33 utilizamos cada valor inicial como argumento de la función `root` del submódulo `optimize`, al que le pasamos también la función f que representa las ecuaciones y la cadena `'hybr'` al argumento `method`, que es el método por defecto de `root` (y que por lo tanto puede ser omitido). `root` devuelve un objeto que guardamos en `sol` del tipo `OptimizeResult` que contiene varios atributos, y del que imprimimos `sol.message` (que describe la causa de la finalización), `sol.x` que muestra la solución del sistema y `sol.f` que nos permite verificar el valor de las funciones en la solución encontrada. Al ejecutarse este programa la salida es:

```

> ./sistema_no_lineal.py
The solution converged.
x: [-2.169, 6.574] -- root: [2.5308866113959994e-11, 9.906742093335197e-12]
The solution converged.
x: [0.000, 0.000] -- root: [0.0, 0.0]
The solution converged.
x: [1.562, 2.137] -- root: [9.992007221626409e-16, -1.4210854715202004e-14]
Proporción de no convergencia: 0.2510

```

El objeto `sol` del tipo `OptimizeResult` tiene otros atributos que no mostramos en este programa, pero que resulta útil inspeccionar (se puede ver la lista de atributos en este [enlace](#)). Particu-

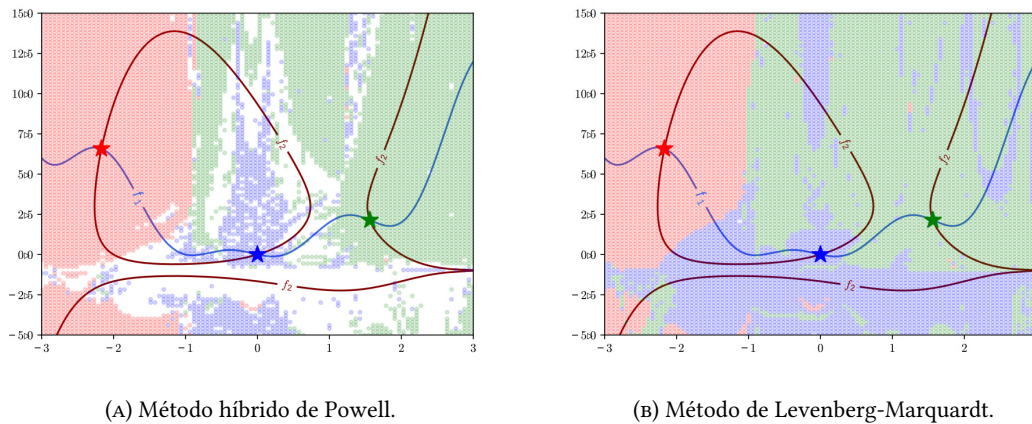


FIGURA 18.2: Regiones de convergencia.

larmente el atributo booleano `success` que es `True` cuando el algoritmo converge a una solución. A continuación, en las líneas 35–37 marcamos en nuestra figura en construcción las raíces encontradas, con estrellas de colores rojo, azul y verde respectivamente. Utilizaremos este código de colores asignado a cada solución para representar las regiones de convergencia hacia cada una.

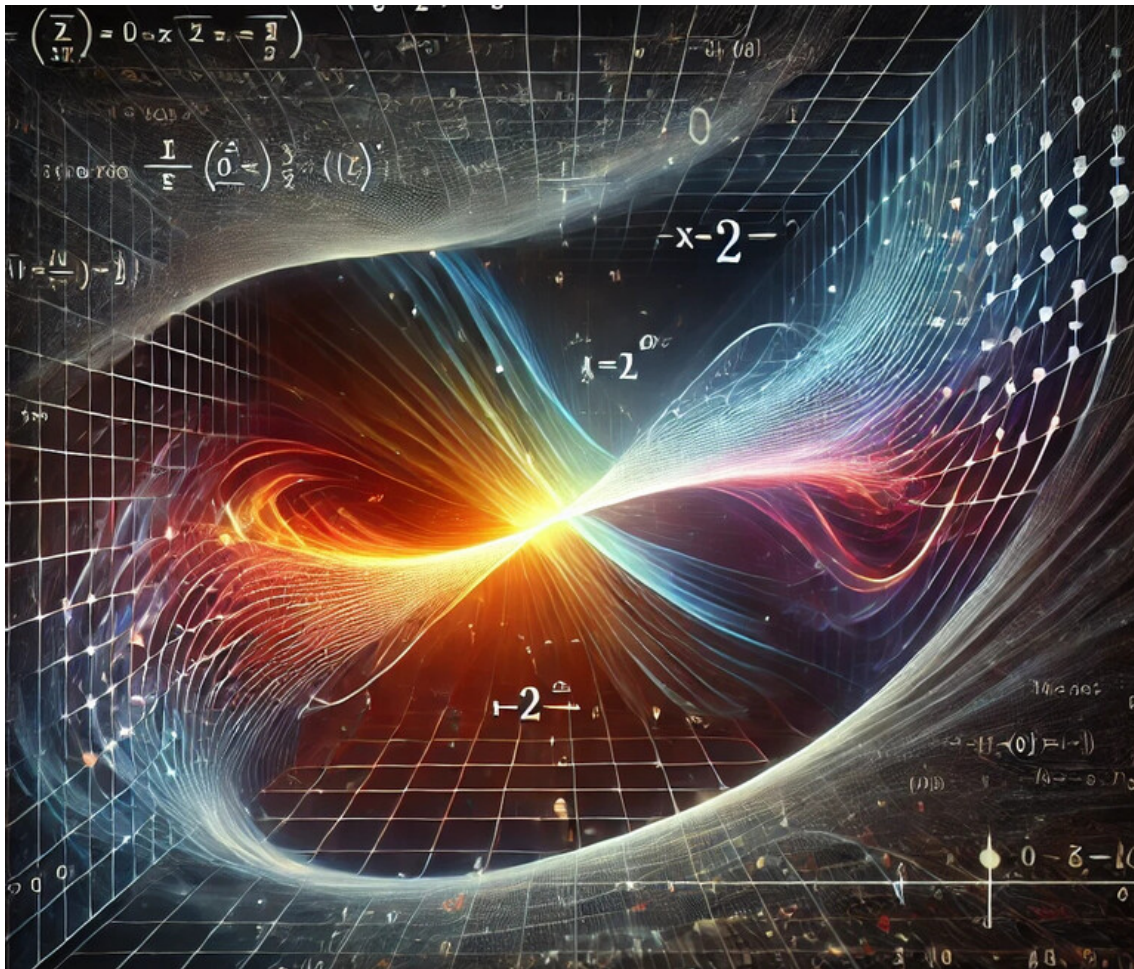
Para determinar estas regiones de convergencia armamos una grilla de `n_trial = 100` puntos en cada variable (representada por los arrays `xs` y `ys`), y para cada punto de esta grilla (que recorreremos en el bucle doble de las líneas 44 y 45) intentamos obtener una solución utilizando el método híbrido de Powell (`'hybr'`). Si hay convergencia, determinamos a cuál de las soluciones llegó el algoritmo a partir del valor inicial (línea 47), y marcamos este valor inicial con un pequeño círculo con el color correspondiente (línea 48). En caso de no haber convergencia, incrementamos el valor de `n_div` que dará cuenta del número total de valores iniciales que no convergieron a ninguna solución.

Tal como muestra la última línea de la salida anterior mostrada, al ejecutar el algoritmo con el método híbrido de Powell hay un 25,1 % de puntos que no convergen a ninguna solución, al menos con el máximo número de llamadas a la función o tolerancia establecidos por defecto. Esto se visualiza en la figura 18.2a como zonas blancas. Por el contrario, si ejecutamos nuevamente el programa pero cambiando el argumento `method` de `root` en la línea 45 por el de Levenberg-Marquardt `'lm'`, la proporción de no convergencia se reduce al 0,03 %, mostrando que este método es más robusto. Se puede ver en la figura 18.2b que casi no hay zonas en blanco.

18.4. Lecturas recomendadas

- Gilbert Strang. *Linear Algebra and Its Applications, 4th Edition*. 4th. Brooks Cole, 2006.
- Stephen H. Friedberg, Arnold J. Insel y Lawrence E. Spence. *Linear Algebra*. 4th ed. Pearson, 2014.
- R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.

- A. Quarteroni, R. Sacco y F. Saleri. *Numerical Mathematics*. New York, United States: Springer-Verlag, 2000.
- Gilbert Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007.
- Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.



19 | Integración numérica

Muchas aplicaciones científicas y tecnológicas requieren la integración de funciones, en una gran variedad de contextos. La integración analítica de funciones es mucho más compleja que la diferenciación, y solo se puede realizar en los pocos casos en los que existe la primitiva del integrando. Por esta razón es necesario recurrir frecuentemente a la integración numérica, denominada “cuadratura” por razones históricas.

El cálculo de una integral definida en forma analítica consiste en evaluar

$$\int_a^b f(x) dx = F(b) - F(a)$$

donde

$$f(x) = \frac{dF(x)}{dx}$$

El problema se reduce a encontrar una función primitiva F (o “antiderivada”) correspondiente a una dada $f(x)$. El intervalo de integración $[a, b]$ puede ser finito, semi infinito, (cuando $a = -\infty$

Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1
SymPy	1.13.3
mpmath	1.3.0
vegas	6.1.3

[Código disponible](#)

o $b = \infty$), o infinito (ambos $a = -\infty$ y $b = \infty$). Gráficamente, calcular la integral corresponde a evaluar el área entre la curva de $f(x)$ y el eje x , tal como muestra la figura 19.1, donde el área se considera positiva cuando $f(x) > 0$ (verde) y negativa si $f(x) < 0$ (roja).

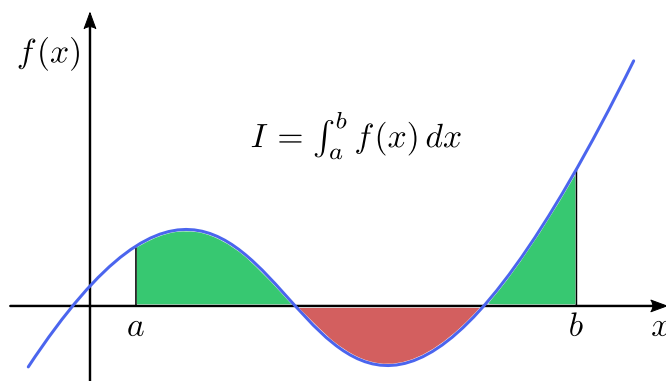


FIGURA 19.1: Interpretación gráfica de la integración.

El cálculo de las cuadraturas numéricas consiste en establecer reglas para aproximar el valor de la integral mediante el cálculo de áreas a partir de la información disponible de f , ya sea porque se conoce su valor en un número establecido de puntos en el intervalo $[a, b]$ o porque es posible evaluarla numéricamente en ese intervalo. Veremos en la sección siguiente cómo obtener expresiones analíticas (cuando es posible) con SymPy, y las opciones que ofrece SciPy para el cálculo de cuadraturas, en la sección 19.3.

19.1. Integración simbólica

SymPy permite calcular integrales definidas e indefinidas de funciones a través del método `integrate()`, que utiliza el poderoso algoritmo Risch-Norman extendido, junto con algoritmos heurísticos y reconocimiento de patrones. De esta forma, SymPy es capaz de integrar funciones polinómicas, racionales, productos de polinomios y exponenciales, senos y cosenos. En el siguiente *jupyter-notebook* vemos algunos ejemplos:

CELDA 1 DE INTEGRACION.IPYNB

```
import sympy as sym

sym.init_printing()
x = sym.Symbol('x')
```

CELDA 02

```
# Función polinómica
sym.integrate(3* x**2 + x - 1, x)
```

$$x^3 + \frac{x^2}{2} - x$$

CELDA 03

```
# Función racional
sym.integrate(x**2/(x**2 + 2 * x + 1), x)
```

$$x - 2 \log(x + 1) - \frac{1}{x + 1}$$

CELDA 04

```
# Función polinómica-exponencial
sym.integrate(x**2 * sym.exp(x) * sym.sin(x), x)
```

$$\frac{x^2 e^x \sin(x)}{2} - \frac{x^2 e^x \cos(x)}{2} + x e^x \cos(x) - \frac{e^x \sin(x)}{2} - \frac{e^x \cos(x)}{2}$$

CELDA 05

```
# Integral definida
f = 4 * sym.sqrt(1 - x**2)
a, b = -1, 1
valor_simb = sym.integrate(f, (x, a, b))
valor_simb
```

$$2\pi$$

Tal como muestran las celdas 2–4, SymPy puede obtener las primitivas de algunas funciones, ahorrándonos, en algunos casos, la tediosa aplicación de la integración por partes. Incluso es capaz de resolver en forma analítica la integral definida

$$\int_{-1}^1 4\sqrt{1-x^2} dx = 2\pi \quad (19.1)$$

como se ve en la celda 5. No obstante, esta facilidad de obtener la primitiva en forma analítica es la excepción más que la regla, y en general no será posible obtener la expresión analítica en forma cerrada. En la gran mayoría de los casos debemos recurrir a cuadraturas numéricas, como las provistas por SciPy que veremos en la sección 19.3, pero es oportuno mencionar acá que es posible integrar numéricamente funciones simbólicas definidas con SymPy pasando estas funciones como argumento de `quad` de la biblioteca `mpmath`¹, la cual permite realizar cuadraturas numéricas con precisión arbitraria. Las versiones de SymPy previas a 1.0 incluían `mpmath` como un submódulo, pero actualmente es una biblioteca externa a SymPy.

Con `mpmath.quad()` podemos evaluar integrales definidas con cualquier precisión, sin la restricción de las limitaciones de la representación de punto flotante. Por supuesto, esta virtud se obtiene con el costo de ser un cálculo mucho más lento que el que ofrece SciPy, por lo que es conveniente usar `mpmath.quad()` para cuadraturas numéricas cuando se requiere evaluar pocas integrales con más precisión que las que brinda SciPy.

Por ejemplo, si queremos evaluar numéricamente la integral de la ecuación (19.1) con una precisión dada utilizando `mpmath`, podemos crear la función integrando usando `sympy.lambdify`

¹ `mpmath` es una biblioteca para aritmética real o compleja con punto flotante de precisión arbitraria. Recomendamos ver su [documentación](#).

pasando `mpmath` como tercer argumento para establecer que usaremos una función compatible con esa biblioteca, tal como vemos a continuación:

CELDA 06
<pre>import mpmath mpmath.mp.dps = 50 f_mpmath = sym.lambdify(x, f, 'mpmath')</pre>
CELDA 07
<pre>valor_num = mpmath.quad(f_mpmath, (a, b)) sym.sympify(valor_num)</pre>
6,2831853071795864769252867665590057683943387987502
CELDA 08
<pre>sym.N(valor_simb, mpmath.mp.dps + 1) - valor_num</pre>
$-2,00457353256914673460540235036361140911139766007879 \cdot 10^{-51}$

En la celda 6 importamos el módulo `mpmath`, definimos la precisión con la que vamos a trabajar, y construimos la función `f_mpmath` utilizando la función `lambdify` de SymPy, la cual permite transformar expresiones de SymPy a funciones lambda para poder evaluarlas en forma numérica muy rápido. Luego, en la celda 7 asignamos a la variable `valor_num` el resultado de la cuadratura numérica calculada con `mpmath`, y obtenemos su representación numérica con `sympify`, que convierte cualquier expresión arbitraria en un tipo de dato específico que usa internamente SymPy. Finalmente, en la celda 8 calculamos la diferencia entre el valor numérico resultante y el resultado exacto que obtuvimos en la celda 5. Vemos que esta diferencia es un orden de magnitud menor al establecido en la celda 6 (50).

19.1.1. Transformaciones integrales (Laplace y Fourier)

Existen varias clases de problemas que son difíciles de resolver en sus representaciones originales, pero que a partir de una transformación integral que mapea una ecuación en su dominio original a otro, su manipulación y solución pueden ser más convenientes que en el dominio original. Una vez resuelto el problema en el nuevo dominio, es posible volver a la representación original con la operación inversa de la transformación integral. Un problema típico es la transformación de una ecuación diferencial en una algebraica, que se puede realizar mediante una transformada de Laplace, o la transformación de un problema en el dominio temporal al dominio en frecuencia a través de una transformada de Fourier.

En general, la transformada de una función f se puede escribir como

$$F(u) = (Tf)(u) = \int_{t_1}^{t_2} K(u, t) f(t) dt \quad (19.2)$$

Aquí, la función f es transformada en una función F mediante el operador T , que se define mediante la función núcleo o *kernel* de dos variables $K(u, t)$. Algunos kernels tienen asociados

un kernel inverso $K^{-1}(u, t)$ que permite la transformación inversa

$$f(t) = \int_{u_1}^{u_2} F(u) K^{-1}(t, u) du \quad (19.3)$$

SymPy dispone de varias funciones para realizar transformaciones integrales. Veremos ejemplos de transformaciones de Laplace y de Fourier.

19.1.1.1. Transformada de Laplace

La transformada y antitransformada de Laplace se definen como

$$L(s) = \int_0^{\infty} \exp(-st) f(t) dt \quad (19.4)$$

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} \exp(st) F(s) ds \quad (19.5)$$

donde $i = \sqrt{-1}$ y c es lo suficientemente grande como para que $F(s)$ no tenga singularidades en el semiplano $\text{Re}(s) > c - \epsilon$. SymPy puede obtener la transformada de Laplace de muchas funciones elementales y combinaciones de ellas. Por ejemplo, calculemos la transformada de $f(t) = \exp(-at)$:

CELDA 09

```
s = sym.symbols("s")
a, t = sym.symbols("a, t", positive=True)
f = sym.exp(-a*t)
```

CELDA 10

```
sym.laplace_transform(f, t, s)
```

$$\left(\frac{1}{a + s}, -a, \text{True} \right)$$

En la celda 9 definimos los símbolos y la función `f`, y en la 10 obtenemos la transformada de Laplace. Como resultado obtenemos una tupla con la función transformada, y las condiciones de convergencia dado que la transformación requiere de una integración indefinida. Si lo que necesitamos es solamente la función, podemos pasar el argumento `noconds=True` y la asignamos a `F`:

CELDA 11

```
F = sym.laplace_transform(f, t, s, noconds=True)
F
```



Pierre-Simon Laplace - Astrónomo, físico, matemático y filósofo francés (1749-1827). Continuador de la mecánica newtoniana, hizo un gran aporte para el posterior descubrimiento de la transformada de Laplace, y descubrió la ecuación de Laplace; como estadístico sentó las bases de la teoría analítica de la probabilidad; y como astrónomo planteó la teoría nebular sobre la formación del sistema solar.

$$\frac{1}{a+s}$$

Podemos recuperar la función original a partir de la antitransformada de F:

CELDA 12

```
sym.inverse_laplace_transform(F, s, t, noconds=True)
```

$$e^{-at}$$

En las celdas 13 y 14 podemos ver un ejemplo en el que combinamos funciones simples en f:

CELDA 13

```
f = sym.exp(-a*t)*(1 + t)**2
F = sym.laplace_transform(f, t, s, noconds=True)
F
```

$$\frac{1}{a+s} + \frac{2}{(a+s)^2} + \frac{2}{(a+s)^3}$$

CELDA 14

```
sym.inverse_laplace_transform(F, s, t, noconds=True)
```

$$t^2 e^{-at} + 2te^{-at} + e^{-at}$$

19.1.1.2. Transformada de Fourier

Las transformada y antitransformada de Fourier son calculadas por SymPy de un modo análogo a las de Laplace. Las definiciones de estas transformaciones son:

$$\mathcal{F}(k) = \int_{-\infty}^{\infty} f(x) \exp(-2\pi i x k) dx \quad (19.6)$$

$$f(x) = \int_{-\infty}^{\infty} \mathcal{F}(k) \exp(-2\pi i x k) dk \quad (19.7)$$

Veamos como ejemplo la transformada de Fourier de $f(x) = \exp(-ax^2)$ (celda 8):

CELDA 15

```
x, k = sym.symbols("x, k")
f = sym.exp(-a * x**2)
F = sym.fourier_transform(f, x, k)
F
```

$$\frac{\sqrt{\pi} e^{-\frac{\pi^2 k^2}{a}}}{\sqrt{a}}$$

Al calcular la transformación inversa recuperamos la función f original:

CELDA 16	
<code>sym.inverse_fourier_transform(F, k, x)</code>	
e^{-ax^2}	

En caso que las transformadas no puedan obtenerse en forma analítica cerrada, estas funciones devuelven un objeto no evaluado.

19.2. Integración numérica en una dimensión

El abordaje central para la integración numérica (definida) de una función consiste en aproximar la integral por una suma:

$$I = \int_a^b f(x) dx = \sum_{i=1}^n \omega_i f(x_i) + r_n \quad (19.8)$$

donde x_i son las “abscisas nodales”, ω_i son los “pesos” y r_n es el residuo producto de la aproximación. Es importante tener una estimación del término r_n para saber con qué precisión estamos estimando la integral, pero en general asumiremos que es una cantidad pequeña.

En la suma de la expresión (19.8) tenemos n abscisas, por lo que estamos considerando una cuadratura de n -puntos. Según cómo elijamos la ubicación de las abscisas y los pesos correspondientes tendremos diferentes esquemas de cuadratura, con diferencias en el grado de precisión alcanzado y la complejidad en el cálculo. En este aspecto, podemos distinguir dos grandes grupos:

- **Métodos de Newton-Cotes:** consisten en considerar que la integral se puede aproximar por la suma de áreas de formas elementales (rectángulos, trapecios). Por lo general, este método utiliza abscisas igualmente espaciadas que se utilizan para interpolar la función integrando, lo que resulta útil (y a veces necesario) cuando $f(x)$ ha sido evaluada en puntos de una grilla.
- **Cuadraturas gaussianas:** utilizan abscisas que no están equiespaciadas, eligiendo los valores x_i de modo que pueda obtenerse un aumento en la precisión. Como resultado se utilizan menos abscisas que en las fórmulas de Newton-Cotes y por lo tanto un número menor de evaluaciones de f , lo cual es una ventaja cuando estas evaluaciones son costosas computacionalmente.

Por otra parte, se puede considerar otra categorización teniendo en cuenta si los límites del intervalo de integración (a y b) pertenecen al grupo de abscisas x_i en donde se evalúa la función. Los métodos que incluyen las evaluaciones $f(a)$ y $f(b)$ son “cerrados”, mientras que los que los excluyen son “abiertos”. Las cuadraturas gaussianas forman parte de los métodos abiertos, al igual que la regla del punto medio, mientras que las fórmulas de la regla trapezoidal y Simpson de Newton-Cotes constituyen ejemplos de métodos cerrados.

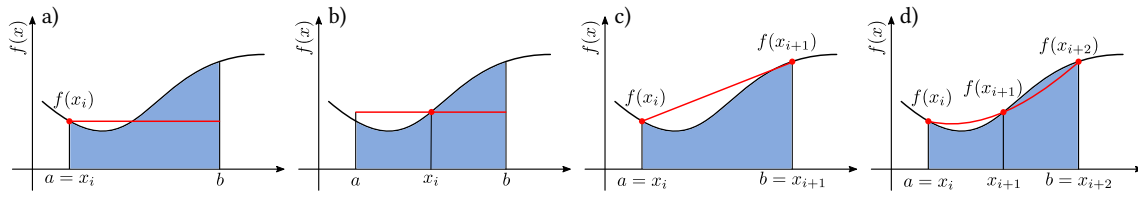


FIGURA 19.2: Esquemas de Newton-Cotes: los puntos rojos representan los valores de la función evaluada en las abscisas x_i . a) Regla del rectángulo; b) regla del punto medio, c) regla del trapecio y d) regla de Simpson.

19.2.1. Métodos de Newton-Cotes

Tal como adelantamos, para explorar estos métodos asumiremos que conocemos los valores de f en un conjunto discreto de n puntos, es decir, en una tabla de n pares $(x_i, f(x_i))$.

Las diversas reglas de integración de Newton-Cotes se derivan del orden de interpolación para la función integrando. Por ejemplo, si aproximamos $f(x)$ con un polinomio de grado cero (es decir, un valor constante) utilizando el valor medio de f en el intervalo $[a, b]$, obtenemos:

$$I = \int_a^b f(x) dx \approx f\left(\frac{a+b}{2}\right) \int_a^b dx = (b-a)f\left(\frac{a+b}{2}\right) \quad (19.9)$$

que se conoce como la “regla del punto medio”. En este caso, la integral se aproxima con el área de un rectángulo cuya base es $(b-a)$ y la altura es $f[(b-a)/2]$, y es capaz de integrar polinomios hasta primer orden (es decir, funciones lineales), y por este motivo se dice que es de grado polinomial uno. Con la misma aproximación (polinomio de grado cero) se puede utilizar uno de los extremos del intervalo dando origen de esta forma a la “regla del rectángulo”. Este abordaje es muy común en las reglas de integración compuestas (de la que hablaremos más adelante).

Aproximando la función $f(x)$ por un polinomio de primer grado, evaluada en los puntos extremos del intervalo de integración, obtenemos la “regla del trapecio”:

$$I = \int_a^b f(x) dx \approx \frac{(b-a)}{2} [f(a) + f(b)] \quad (19.10)$$

que también es polinomial de grado uno. Si utilizamos, en cambio, un polinomio de interpolación de segundo grado, resulta la “regla de Simpson”:

$$I = \int_a^b f(x) dx \approx \frac{(b-a)}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \quad (19.11)$$

que utiliza la evaluación en los extremos del intervalo de integración y en el punto medio del mismo. Este método es polinomial de segundo grado, lo que significa que podemos integrar exactamente polinomios de grado menor o igual a tres.

La figura 19.2 muestra los distintos esquemas mencionados. En todos los casos la integral (área azul) se aproxima por el área de la figura delimitada por el eje x por debajo, la línea roja por arriba y las líneas verticales en $x = a$ y $x = b$. Utilizaremos las capacidades de álgebra simbólica e integración de SymPy para determinar los valores de los pesos en la regla de Simpson.

Si consideramos la regla de Simpson como una cuadratura de tres puntos, es muy simple utilizar la condición de integración exacta de un polinomio de segundo grado para obtener los

pesos ω_i de la fórmula (19.8). Siguiendo con el *jupyter-notebook* de la sección anterior, en la celda 17 definimos los símbolos que vamos a utilizar y declaramos que f será una función, y en la celda 18 definimos el conjunto de las abscisas en los que se evalúa la función, y también declaramos la lista w conteniendo los pesos respectivos:

CELDA 17

```
a, b, x = sym.symbols("a, b, x")
f = sym.Function("f")
```

CELDA 18

```
# Abscisas (x) y pesos (w) de la regla de Simpson
abs_x = a, (a + b)/2, b
w = [sym.symbols(f"w_{i}") for i in range(len(abs_x))]
```

Dadas las abscisas y los pesos, podemos construir la regla de Simpson en forma simbólica:

CELDA 19

```
simpson = sum(w[i] * f(abs_x[i]) for i in range(len(abs_x)))
simpson
```

$$w_0 f(a) + w_1 f\left(\frac{a}{2} + \frac{b}{2}\right) + w_2 f(b)$$

Ahora haremos uso del hecho que con la regla de Simpson podemos integrar en forma exacta hasta polinomios de tercer grado. Dado que tenemos que obtener tres pesos ω_i , nos será suficiente con construir una base de polinomios hasta segundo grado, con los cuales podemos interpolar $f(x)$. Usaremos `sympy.Lambda` para crear las representaciones simbólicas de esta base:

CELDA 20

```
phi = [sym.Lambda(x, x**n) for n in range(len(abs_x))]
phi
```

$$[(x \mapsto 1), (x \mapsto x), (x \mapsto x^2)]$$

Con esta base podemos construir un sistema de ecuaciones para determinar los pesos. Para cada una de las funciones de la base tenemos:

$$\sum_{i=0}^2 \omega_i \phi_n(x_i) - \int_a^b \phi_n(x) dx = 0$$

La lista `ecs` de la celda 21 muestra cómo construimos el sistema de ecuaciones, cuya solución obtenemos en la celda 24. En la celda 25 sustituimos los ω_i obtenidos en la regla de Simpson, y podemos ver que recuperamos la expresión dada por (19.11).

CELDA 21

```
ecs = [simpson.subs(f, phi[n]) - sym.integrate(phi[n](x), (x, a, b)) for n in range(len(abs_x))]
ecs[0]
```

$$a - b + w_0 + w_1 + w_2$$

CELDA 22

ecs[1]

$$\frac{a^2}{2} + aw_0 - \frac{b^2}{2} + bw_2 + w_1 \left(\frac{a}{2} + \frac{b}{2} \right)$$

CELDA 23

ecs[2]

$$\frac{a^3}{3} + a^2w_0 - \frac{b^3}{3} + b^2w_2 + w_1 \left(\frac{a}{2} + \frac{b}{2} \right)^2$$

CELDA 24

```
ecs_sol = sym.solve(ecs, w)
ecs_sol
```

$$\left\{ w_0 : -\frac{a}{6} + \frac{b}{6}, w_1 : -\frac{2a}{3} + \frac{2b}{3}, w_2 : -\frac{a}{6} + \frac{b}{6} \right\}$$

CELDA 25

```
simpson.subs(ecs_sol).simplify()
```

$$\frac{(a-b) \left(-f(a) - f(b) - 4f\left(\frac{a}{2} + \frac{b}{2}\right) \right)}{6}$$

Las reglas de cuadratura de mayor orden se pueden obtener mediante el abordaje general de aproximar la función $f(x)$ mediante un polinomio de interpolación de grado n , $P_n(x)$, con $n+1$ puntos de interpolación. En efecto, para cada n natural las fórmulas de Newton-Cotes

$$\int_a^b P_n(x) dx = h \sum_{i=0}^n \omega_i f_i \quad (19.12)$$

proveen valores aproximados para la integral de $f(x)$ en $[a, b]$, siendo $f_i = f(a + ih)$ y $h = (b - a)/n$. Los pesos ω_i se encuentran tabulados² y son números racionales con la propiedad

$$\sum_{i=0}^n \omega_i = n \quad (19.13)$$

Para funciones $f(x)$ suficientemente suaves en el intervalo cerrado $[a, b]$, se puede demostrar

² Algunos casos se pueden ver en la entrada de [Wikipedia](#).

que el error en la aproximación de la integral se puede expresar como:

$$\int_a^b P_n(x) dx - \int_a^b f(x) dx = Kh^{p+1} f^{(p)}(\xi), \quad \xi \in (a, b) \quad (19.14)$$

donde los valores de K y p dependen solo de n pero no del integrando f . En particular, para el caso de la regla de Simpson la estimación del error es

$$\varepsilon = \frac{h^5}{90} f^{(iv)}(\xi) \quad (19.15)$$

donde $f^{(iv)}$ denota la derivada cuarta de f . Podemos observar que esta estimación del error contiene una derivada cuarta, lo que implica que la regla de Simpson es exacta hasta polinomios de tercer grado.

Las reglas de cuadratura de mayor orden se pueden obtener aumentando el número de abscisas (y pesos) en el intervalo $[a, b]$. Sin embargo, la interpolación con polinomios de mayor orden pueden introducir efectos indeseables entre los puntos de interpolación, por lo que resulta aconsejable dividir el intervalo $[a, b]$ en subintervalos $[a, x_1]$, $[x_1, x_2]$, $\dots$, $[x_{n-1}, x_n = b]$, y usar cuadraturas de orden bajo en cada uno de ellos. Este abordaje da origen a las fórmulas compuestas de Newton-Cotes.

19.2.2. Fórmulas de Newton-Cotes compuestas

El uso de las fórmulas de Newton-Cotes es inapropiado en intervalos largos de integración. Por un lado, esto requeriría usar interpolaciones de grado superior y los pesos correspondientes son engorrosos de calcular. Por otro, dado que estas fórmulas utilizan abscisas equiespaciadas, la naturaleza oscilatoria de los polinomios de interpolación de alto orden no necesariamente convergen al valor exacto de la integral.

El procedimiento para obtener las fórmulas compuestas de Newton-Cotes, por ejemplo para la regla de Simpson, consiste en seleccionar un entero par n y subdividir el intervalo $[a, b]$ en n subintervalos, aplicando la regla de Simpson en cada par consecutivo de ellos. Definiendo el “paso” $h = (b - a)/n$, y $x_i = a + ih$ para cada $i = 0, 1, \dots, n$ tenemos:

$$\begin{aligned} \int_a^b f(x) dx &= \sum_{i=1}^{n/2} \int_{x_{2i-2}}^{x_{2i}} f(x) dx \\ &= \sum_{i=1}^{n/2} \frac{h}{3} [f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})] \end{aligned}$$

Dado que para cada $i = 1, 2, \dots, n/2 - 1$ aparece $f(x_{2i})$ en el intervalo $[x_{2i-2}, x_{2i}]$ y también en el $[x_{2i}, x_{2i+2}]$, podemos agrupar estos términos y reducir la suma:

$$\int_a^b f(x) dx = \frac{h}{3} \left[f(x_0) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + f(x_n) \right] \quad (19.16)$$

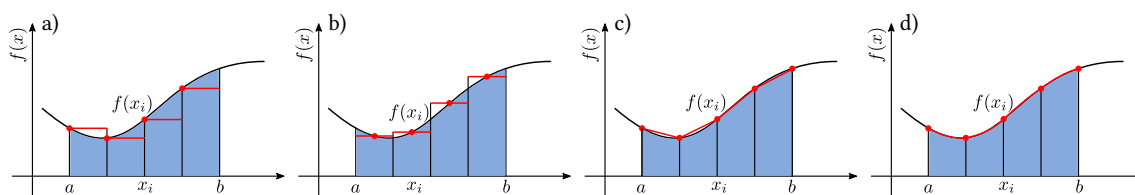


FIGURA 19.3: Fórmulas compuestas de Newton-Cotes: los puntos rojos representan los valores de la función evaluada en las abscisas x_i . a) Regla del rectángulo; b) regla del punto medio, c) regla del trapecio y d) regla de Simpson.

Del mismo modo se pueden construir las fórmulas compuestas para la regla del rectángulo y del trapecio. La estimación del error de las fórmulas compuestas se obtiene simplemente sumando los errores en cada subintervalo. Por ejemplo, para el caso de la fórmula compuesta de la regla de Simpson, la suma de los errores dados por la expresión (19.15) para cada par de subintervalos consecutivos resulta:

$$\varepsilon_C = \frac{b-a}{180} h^4 f^{(iv)}(\xi) \quad \xi \in (a, b)$$

siempre que $f \in C^4[a, b]$, es decir, f sea continua y con todas sus derivadas hasta orden 4 continuas en $[a, b]$. Por lo tanto este método es de orden 4.



Johann Carl Friedrich Gauss - Matemático, astrónomo y físico alemán (1777-1855) que contribuyó significativamente en muchos ámbitos, incluida la teoría de números, el análisis matemático, la geometría diferencial, la estadística, el álgebra, la geodesia, el magnetismo y la óptica. Fue de los primeros en extender el concepto de divisibilidad a otros conjuntos además de los números enteros.

19.2.3. Cuadraturas gaussianas

En las secciones anteriores hemos visto que dado un conjunto de n abscisas x_i , podemos encontrar los pesos ω_i de modo tal que la fórmula resultante sea exacta para la integración de polinomios de grado menor o igual a $n - 1$. En algunos casos, la fórmula de Simpson de tres puntos es exacta para polinomios hasta tercer grado. En las fórmulas de Newton-Cotes tenemos fijos los valores de las abscisas, y obtenemos los pesos para que las integrales sean exactas en el contexto mencionado.

Podemos aumentar la precisión de las aproximaciones si tenemos más parámetros que podamos utilizar aumentando de este modo los “grados de libertad” disponibles para este propósito. De esta manera, si podemos *elegir* los valores de las abscisas y de los pesos, podemos obtener expresiones que dan valores exactos para la integración de polinomios hasta de grado $2n - 1$, dando origen de este modo a las “cuadraturas gaussianas”.

Ilustraremos el caso de derivación de las abscisas y los pesos de la fórmula de Gauss-Legendre, de modo de poder integrar exactamente polinomios de tercer grado en el intervalo $[-1, 1]$ solo con dos evaluaciones de la función integrando. Usaremos la misma metodología que para derivar la regla de Simpson.

CELDA 26

```
abs_x = [sym.symbols(f"x_{i}") for i in range(2)]
w = [sym.symbols(f"w_{i}") for i in range(len(abs_x))]
```

CELDA 27

```
GL = sum(w[i] * f(abs_x[i]) for i in range(len(abs_x)))
GL
```

$$w_0 f(x_0) + w_1 f(x_1)$$

En la celda 26 definimos la lista de abscisas y pesos considerando que usaremos solo dos valores para cada caso, y luego, en la celda 27, la representación de la integral según la cuadratura de Gauss-Legendre de dos puntos:

$$I = \int_{-1}^1 f(x) dx \approx \omega_0 f(x_0) + \omega_1 f(x_1) \quad (19.17)$$

A continuación definimos en phiGL la base del espacio de los polinomios de grado menor o igual a 3:

CELDA 28

```
phiGL = [sym.Lambda(x, x**n) for n in range(4)]
phiGL
```

$$[(x \mapsto 1), (x \mapsto x), (x \mapsto x^2), (x \mapsto x^3)]$$

Ahora construimos el sistema de ecuaciones que establecen la igualdad entre la integración de Gauss-Legendre y la integral exacta para cada una de las funciones base, es decir:

$$\omega_0 \phi_n^{GL}(x_0) + \omega_1 \phi_n^{GL}(x_1) - \int_{-1}^1 \phi_n^{GL}(x) dx = 0$$

donde $\phi_n^{GL}(x) = x^n$ (que son las funciones base almacenadas en phiGL). Las ecuaciones resultantes para cada $n = 0, 1, 2, 3$ se muestran en las celdas 29–32.

CELDA 29

```
ecsGL = [GL.subs(f, phiGL[n]) - sym.integrate(phiGL[n](x), (x, -1, 1)) for n in range(4)]
ecsGL[0]
```

$$w_0 + w_1 - 2$$

CELDA 30

```
ecsGL[1]
```

$$w_0 x_0 + w_1 x_1$$

CELDA 31
ecsGL[2]
$w_0 x_0^2 + w_1 x_1^2 - \frac{2}{3}$

CELDA 32
ecsGL[3]
$w_0 x_0^3 + w_1 x_1^3$

En la celda 33 construimos la lista `y` que concatena las listas `x` y `w` para tener en un único contenedor todas las incógnitas del sistema de ecuaciones, que en la misma celda resolvemos y mostramos las soluciones obtenidas:

CELDA 33
<pre>y = abs_x + w ecsGL_sol = sym.solve(ecsGL, y) ecsGL_sol</pre>
$\left[\left(-\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, 1, 1 \right), \left(\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, 1, 1 \right) \right]$

Estas soluciones muestran que:

$$\omega_0 = \omega_1 = 1$$

$$x_0, x_1 = \pm \frac{1}{\sqrt{3}}$$

De este modo, podemos substituir en la ecuación (19.17) que resulta en la fórmula de Gauss-Legendre de dos puntos:

$$I \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \quad (19.18)$$

con lo que llegamos al resultado interesante que con la simple suma de los valores de la función en $x = -1/\sqrt{3}$ y $x = 1/\sqrt{3}$ resulta en la estimación de una integral que es precisa hasta tercer orden.

Se puede observar que en la ecuación (19.17) estamos integrando en el intervalo $[-1, 1]$, lo que hacemos para simplificar la matemática y hacer la formulación tan general como es posible. Si la integración se realiza sobre otro intervalo, se puede reescalar el intervalo de integración para llevarlo al $[-1, 1]$ mediante un simple cambio de variable lineal.

Es posible extender la metodología para obtener fórmulas de Gauss-Legendre de mayor orden aumentando el número de abscisas. Existen tablas³ en donde figuran las abscisas y los pesos para cuadraturas de Gauss-Legendre con más de dos puntos. En la sección 19.3.2 mostramos cómo realizar la evaluación numérica de cuadraturas gaussianas utilizando el módulo `integrate` de SciPy.

³ Ver tablas 3.5.1-5 en [este sitio web](#)

El análisis del error en las cuadraturas gaussianas excede el alcance de este libro (invitamos a las personas interesadas en consultar la referencia [27] para ver una demostración rigurosa), pero podemos mencionar que en este caso, si $f \in C^{(2n)}[a, b]$, el error en la cuadratura es

$$\varepsilon_{CG} = K \frac{f^{(2n)}(\xi)}{(2n!)} \quad (19.19)$$

donde K es una constante y ξ algún valor en (a, b) (desconocido). Comparando con las fórmulas de Newton-Cotes, vemos que para el mismo esfuerzo computacional (es decir, cantidad de evaluaciones de f), la integración gaussiana ofrece una precisión mayor. Sin embargo, dado que las cuadraturas gaussianas requieren la evaluación de las funciones en un conjunto no equiespaciados de puntos dentro del intervalo de integración, no son apropiadas para los casos en los que no se conoce la expresión analítica de la función, como en los casos en que ésta se presenta en forma de tabla sobre una grilla de abscisas. No obstante, cuando es posible evaluar la función en puntos arbitrarios, la eficiencia de las cuadraturas gaussianas es una ventaja, que resulta particularmente útil cuando deben realizarse numerosas evaluaciones de integrales.



Adrien-Marie Legendre - Matemático francés (1752-1833). Otorgó importantes contribuciones a la estadística, a la teoría de números, al álgebra abstracta y al análisis matemático. Intervino en geodesia y en la comisión que estableció el metro como unidad de medida internacional.

19.2.4. Integración de Monte Carlo

Una forma alternativa de estimar integrales en forma numérica es mediante la integración *Monte Carlo*, que utiliza números aleatorios. No es un método eficiente para evaluar integrales unidimensionales, pero lo mencionamos aquí porque su interpretación gráfica es sencilla, y el método es muy poderoso para evaluar integrales multidimensionales.

La situación se representa en la figura 19.4. La función $f(x)$ a ser integrada está contenida en un rectángulo cuya base es el intervalo de integración, y su altura supera con cierto margen el máximo de f en ese intervalo. Naturalmente, el valor de la integral es el área bajo la curva que representa a $f(x)$.

Para estimar el valor del área bajo la curva, generamos a partir de una distribución uniforme una colección n_{MC} de puntos (x_i, y_i) dentro del área que contiene a la función f . La estimación de la integral corresponde, entonces, a la proporción de puntos que caen bajo la función, multiplicado por el área total de la figura. Cuanto mayor sea n_{MC} , mas precisa será la estimación de la integral.

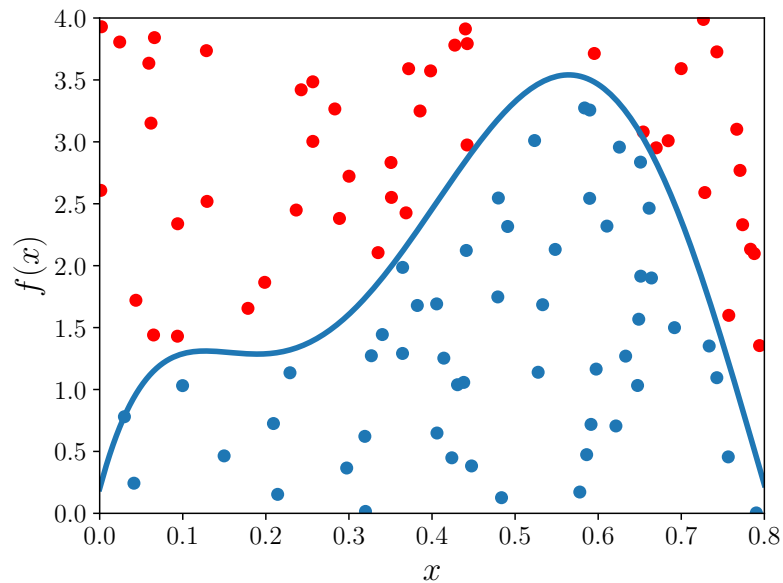
En el código siguiente vemos un ejemplo de cómo realizar la integración de Monte Carlo de la función polinómica:

$$f(x) = 400x^5 - 900x^4 + 675x^3 - 200x^2 + 25x + 0,2$$

en el intervalo $[0, 0,8]$, que está contenida en un rectángulo de altura 4,0.

1 `#!/usr/bin/env python3`

2

FIGURA 19.4: Estimación de la integral mediante integración de Monte Carlo con $n_{MC} = 100$.

```

3 import matplotlib.pyplot as plt
4 import numpy as np
5 import sympy as sym
6 plt.rcParams.update({
7     "text.usetex": True,
8     "axes.labelsize": 18,
9     "xtick.labelsize": 14,
10    "ytick.labelsize": 14})
11
12 def f(x):
13     return 400 * x**5 - 900 * x**4 + 675 * x**3 - 200 * x**2 + 25 * x + 0.2
14
15 a, b, c = 0, 0.8, 4 # Límites de integración y escala vertical
16 xs = sym.Symbol('x')
17 fs = f(xs)
18 int_sym = sym.integrate(fs, (xs, a, b))
19 print(f'Integral sympy = {int_sym}')
20
21 n_mc = 100 # Puntos de Monte Carlo
22 rng = np.random.default_rng(seed=13)
23 x = rng.uniform(a, b, n_mc)
24 y = rng.uniform(a, c, n_mc)
25
26 mask = y < f(x) # Selección de puntos de MC debajo de la curva
27 no_mask = ~mask # Selección de puntos de MC arriba de la curva
28 int_MC = y[mask].size / n_mc * (b - a) * c
29 print(fr' Integral MC = {int_MC}')
30 print(fr'Diferencia porcentual = {(int_MC - int_sym)/int_sym * 100:.3f} %')
31
32 x_plot = np.linspace(a, b, 200) # Coordenadas para graficar f(x)
33 plt.plot(x_plot, f(x_plot), c='tab:blue', lw=3)
34 plt.scatter(x[mask], y[mask], c='tab:blue')
35 plt.scatter(x[no_mask], y[no_mask], c='tab:red')

```

```

36
37 plt.xlabel(r'$x$')
38 plt.ylabel(r'$f(x)$')
39 plt.xlim([a, b])
40 plt.ylim([a, c])
41 plt.savefig('mc-1d.pdf', bbox_inches='tight')

```

Inicialmente importamos los módulos necesarios usuales (`matplotlib.pyplot` y `numpy`) y también `sympy`, ya que usaremos su capacidad de integración para comparar con el resultado que da Monte Carlo. En las líneas 6–10 realizamos alguna configuración de aspectos gráficos de la figura producida, que se muestra en la Figura 19.4.

En la línea 15 definimos el intervalo de integración (a, b) así como la altura del rectángulo que contiene la función (c). En las líneas 16–17 definimos $f(x)$ que utilizaremos para las evaluaciones numéricas, y luego definimos en `xs` y `fs` el argumento y la función simbólicas (líneas 18–19) para evaluar la integral con SymPy, que es almacenada en la variable `int_sym` y mostrada en pantalla en las líneas 17–18.

A continuación generamos los puntos de Monte Carlo que usaremos para la evaluación de la integral. Primero definimos en `n_mc` la cantidad de puntos a generar (línea 21), luego inicializamos el generador de números aleatorios con una semilla arbitraria⁴, y finalmente creamos dos arrays a partir de dicho generador, con distribución uniforme en (a, b) para x y (a, c) para y , determinando que evaluaremos `n_mc` puntos (líneas 21–24). Estos arrays contienen las coordenadas cartesianas de los puntos en el rectángulo $(a, b) \times (a, c)$ que contiene a $f(x)$.

Ahora realizamos la estimación de la integral. Para ello creamos una máscara⁵ que tiene `True` en los valores del array `y` que son menores que los valores de `f` evaluados para los valores de `x` correspondientes. Para poder distinguir en el gráfico los valores que están por arriba de la función, generamos la máscara `no_mask` que es la negación de la máscara `mask` (lo que convierte los `True` en `False` y viceversa). La estimación del área bajo la curva se obtiene multiplicando la proporción de los puntos bajo la curva por el área total que contiene la función (línea 28). A continuación imprimimos esta estimación y también la diferencia porcentual obtenida con SymPy. El resto del código genera la Figura 19.4.

Al ejecutar este programa obtenemos, además del archivo PDF con la figura, el cálculo numérico de la integral por ambos métodos y su comparación:

```

$ ./mc-1d.py
Integral sympy = 1.6405333333333333
Integral MC = 1.6960000000000002
Diferencia porcentual = 3.381 %
$

```

Se puede apreciar que con apenas 100 puntos al azar, la precisión del método (considerando como “exacto” el valor que da SymPy) es de poco más del 3 %. Si repetimos el cálculo con 1000 puntos la diferencia es menor al 1 %. En este caso se obtiene una precisión relativamente buena con pocos puntos dado que en el área de integración, la función ocupa una buena proporción de

⁴Para este ejemplo elegimos el [número de la suerte](#) 13, que es la suma y la diferencia de dos cuadrados consecutivos: $13 = 2^2 + 3^2 = 7^2 - 6^2$, además de ser uno de los tres [números primos de Wilson](#) conocidos.

⁵Para ver como se usan las máscaras en NumPy, invitamos a visitar la sección 11.3.

ésta. No obstante, si la función presenta un “pico” en un intervalo pequeño del dominio de integración, el muestreo con una distribución uniforme no resulta eficiente. Existen diversas técnicas que permiten hacer integraciones Monte Carlo mas “eficientes estadísticamente”, entendiendo por esto que podemos alcanzar una mayor precisión para el mismo número de muestras, u obtener la misma precisión con menos muestras. Podemos mencionar, como ejemplos de métodos de reducción de varianza, el muestreo de importancia o los muestreos estratificado y condicional. La descripción de estas técnicas están fuera del alcance de este libro, por lo que sugerimos a quienes estén interesados las lecturas recomendadas al final del capítulo.

19.3. Integración numérica con SciPy

El submódulo `integrate` de SciPy ofrece varias técnicas de integración que pueden agruparse según la forma en que disponemos de la función a integrar. Si dicha función se conoce solo en un número dado de valores los métodos disponibles son `trapezoid`, `simpson` y el de Romberg `romb` (extensión recursiva de la regla del trapecio que utiliza la extrapolación de Richardson). Por otra parte, si disponemos del integrando como una función que podemos evaluar en valores arbitrarios del argumento, `integrate` dispone de `quad`, `fixed_quad`, `quadrature` y `romberg`. `integrate` también dispone de rutinas para el cálculo de integrales de varias variables, lo que veremos en la sección [19.4](#).

19.3.1. Fórmulas de Newton-Cotes

La implementación de las fórmulas de Newton-Cotes comprende la del trapecio (`trapezoid`), la de Simpson (`simpson`) y la fórmula de Romberg (`romb`). En estos últimos dos casos existen requerimientos sobre la partición del intervalo de integración utilizado. Para el caso de Simpson, es necesario que el número de intervalos sobre los que se aplica la fórmula compuesta sea par, por lo que el número de abscisas debe ser impar. Para el caso del método de Romberg, el número de abscisas debe ser $n = 2^k + 1$, con k entero.

Como ejemplos de uso calculamos la integral

$$I = \int_0^5 2 \exp\left(-\frac{x^2}{2}\right) dx \quad (19.20)$$

de la cual obtenemos primero, haciendo uso de SymPy, su valor exacto y una aproximación numérica con una precisión de 20 cifras significativas (celdas 1–2), de modo de poder estimar la precisión de las cuadraturas de Newton-Cotes. Luego definimos la función en la celda 3, y la evaluamos en el array de abscisas `x` que tiene 17 valores (celda 4), cumpliendo de este modo los requisitos de Simpson y Romberg. En la celda 5 obtenemos los valores de las integraciones numéricas para cada método, notando que en el caso de Romberg es necesario indicar en `dx` el paso de integración (o longitud de cada subintervalo).

CELDA 1 DE INTEGRACION-SCIPY.IPYNB

```
import sympy as sym

sym.init_printing()
X = sym.Symbol('X')
I_exacta = sym.integrate(2 * sym.exp(-X**2/2), (X, 0, 5))
```

I_exacta

$$\sqrt{2}\sqrt{\pi} \operatorname{erf}\left(\frac{5\sqrt{2}}{2}\right)$$

CELDA 02

```
I_ex_num = sym.N(I_exacta, 20)
I_ex_num
```

2,5066268375731304228

CELDA 03

```
import numpy as np

def f(x, A, μ):
    return A * np.exp(-μ * x**2)
```

CELDA 04

```
a, b = 0, 5 # Intervalo de integración
A, μ = 2.0, 0.5 # Parámetros de la función f
n = 2**4 + 1 # número de abscisas (n = 17) con k = 4
x = np.linspace(a, b, n)
y = f(x, A, μ)
```

CELDA 05

```
import operator
import scipy.integrate as spi

I_trap = spi.trapezoid(y, x=x)
I_simp = spi.simpson(y, x=x)
I_romb = spi.romb(y, dx=x[1]-x[0])
I_NC = [("Trapezio", I_trap), ("Simpson", I_simp), ("Romberg", I_romb)]
I_NCerrores = [(name, value, (value-I_ex_num) / I_ex_num * 100) for (name, value) in I_NC]
for name, value, error in sorted(I_NCerrores, key=operator.itemgetter(2), reverse=True):
    print(f'{name:10s}: {value} ({error:.3e} %)')
```

```
Simpson    : 2.5066268019880233 (-1.420e-6 %)
Trapezio   : 2.5066265447230225 (-1.168e-5 %)
Romberg    : 2.5065305814372403 (-3.840e-3 %)
```

En la salida de la celda 5 mostramos la diferencia relativa con el valor “exacto” de la celda 2. Vemos que la fórmula de Simpson da el resultado más preciso, con $n = 17$, seguido por la fórmula del trapecio y luego por la de Romberg. Sin embargo, si repetimos el cálculo para $n = 257$ (duplicando el valor de k), la precisión dada por la fórmula de Romberg mejora significativamente, superando en cuatro órdenes de magnitud a la de Simpson, y en siete a la del trapecio.

En caso que querramos usar la fórmula de Simpson con un número impar de subintervalos (n par), es posible indicarle al método `simpson` que utilice una fórmula mixta usando la regla de Simpson en un número par de intervalos y la del trapecio en el restante. Esto se configura

en el parámetro `even` que admite las cadenas `'first'` para el método del trapecio en el primer subintervalo, `'last'` en el último y `'avg'` haciendo un promedio entre los dos casos anteriores (siendo esta la opción por defecto).

19.3.2. Cuadraturas con funciones evaluables

Si tenemos la posibilidad de evaluar la función integrando en argumentos arbitrarios, podemos utilizar los métodos de SciPy `quad`, `fixed_quad`, `quadrature` y `romberg`.

El método `quad` utiliza la biblioteca [QUADPACK](#) desarrollada en Fortran, que consiste en rutinas automáticas de integración que intentan obtener el resultado con un error (relativo o absoluto) menor al establecido como argumento. En la celda 6 vemos el cálculo de la integral (19.20) pasando la función `f`, los límites de integración `a` y `b`, y los parámetros adicionales `A` y `μ` en `args` (notar que el primer argumento de `f`, `x`, es asumido implícitamente como variable de integración). En este caso usamos el valor por defecto del error absoluto máximo `epsabs` de $1.49\text{e-}08$:

CELDA 06
<pre> I_quad = spi.quad(f, a, b, args=(A, μ), full_output=1) print(f'I = {I_quad[0]:.9f} ± {I_quad[1]:.3e} ({(I_quad[0] - I_ex_num) / I_ex_num * 100:.3e} %)') print(f"Evaluaciones de f: {I_quad[2]['neval']}") </pre> <pre> I = 2.506626838 ± 2.173e-09 (9.087e-15 %) Evaluaciones de f: 21 </pre>

En la salida de la celda 6 vemos, entre paréntesis, la diferencia porcentual con el valor obtenido a partir de la integración exacta con 20 cifras significativas (celda 2). En este caso, la estimación obtenida por `quad` se aproxima al valor “exacto” en varios órdenes comparados con las estimaciones dadas por las fórmulas de Newton-Cotes. Al pasar el argumento `full_output=1` obtenemos como tercer elemento de la tupla de salida de `quad` un diccionario con diversa información sobre la ejecución del método. En la salida de la celda 6 imprimimos el número de evaluaciones de la función `f` efectuadas durante la integración (21), lo que muestra la eficiencia de este método.

En las celdas 7–8 vemos el cálculo de la integral (19.20) utilizando cuadraturas gaussianas. En el primer caso, el método `fixed_quad` utiliza un número establecido de abscisas (en el caso mostrado usamos el valor por defecto, cinco), mientras que en el segundo `quadrature` utiliza el número de abscisas necesario para obtener el resultado con un error menor que el requerido (que por defecto es $1.49\text{e-}08$). En las salidas de ambas celdas mostramos también entre paréntesis la diferencia con el valor “exacto” (con 20 cifras significativas), mostrando que con pocos puntos se alcanza una precisión aceptable (el más preciso con apenas 11 puntos).

CELDA 07
<pre> I_fix_gauss = spi.fixed_quad(f, a, b, args=(A, μ), n=5)[0] print(f'I = {I_fix_gauss:.9f} ({(I_fix_gauss - I_ex_num) / I_ex_num * 100:.3e} %)') </pre> <pre> I = 2.508871444 (8.955e-2 %) </pre>

CELDA 08

```
I_gauss = spi.quad(f, a, b, args=(A, μ))[0]
print(f'I = {I_gauss:.9f} ( {(I_gauss - I_ex_num)/I_ex_num * 100:.3e} % )')
```

```
I = 2.506626838 ( 9.087e-15 % )
```

La función `quad` permite calcular integrales impropias con límite de integración infinitos. Estos límites se representan con la representación de punto flotante de infinito `float('inf')`, que en NumPy está disponible como `np.inf`. Por ejemplo, para evaluar la integral

$$\int_{-\infty}^{\infty} e^{-x^2} dx$$

podemos hacerlo con:

CELDA 09

```
def f_imp(x):
    return np.exp(-x**2)
```

```
val, err = spi.quad(f_imp, -np.inf, np.inf)
print(f'{val = } ± {err}')
```

```
val = 1.7724538509055159 ± 1.4202636780944923e-08
```

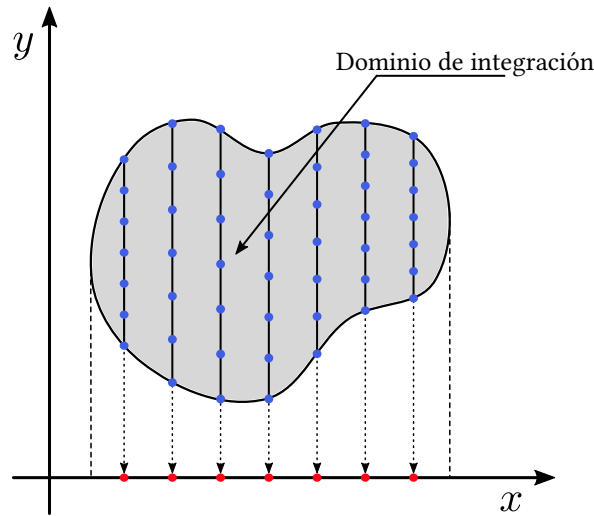
`quad` también es capaz de evaluar integrales con singularidades en el integrando, aunque en estas condiciones suele ser necesario “ayudar” al método indicando dónde no evaluar la función, ya que puede suceder que la cuadratura gaussiana intente evaluar la función en estos puntos singulares (por ejemplo, cuando la singularidad está en $x = 0$).

19.4. Integración múltiple

No es tarea sencilla evaluar una integral de varias variables. Esto se debe a dos motivos: en primer lugar, el número de evaluaciones de funciones necesarias para realizar un muestreo de un espacio N -dimensional crece como la potencia N -ésima del número de evaluaciones necesarias para una integral unidimensional. Por ejemplo, si utilizamos 50 puntos para obtener una cuadratura en una dimensión, serán necesarias 125.000 evaluaciones en un espacio tridimensional. En segundo lugar, a diferencia del caso unidimensional donde el dominio de integración queda definido por dos números (los límites del intervalo de integración), una región de integración multidimensional requiere de un contorno de integración $(N - 1)$ -dimensional que puede ser un objeto muy complejo.

En algunos casos es posible reducir la dimensionalidad de la integración, por ejemplo, aprovechando alguna simetría de la función respecto de sus variables. No obstante, estos suelen ser casos especiales y no la generalidad del problema.

Existen dos abordajes básicos para la integración multidimensional. Si la función a integrar es suficientemente suave y el contorno de integración es simple, es posible descomponer la integración múltiple en sucesivas integraciones unidimensionales utilizando algún esquema de Newton-Cotes o gaussiano. También este enfoque es necesario si se necesita obtener el valor de la integral con alta precisión.

FIGURA 19.5: Evaluaciones de la función $f(x, y)$ en una integración bidimensional.

El segundo abordaje, cuando el dominio de integración es complejo o no se requiere de un alto grado de precisión, es utilizando el método de Monte Carlo. Veremos a continuación algunos ejemplos utilizando estas técnicas.

19.4.1. Integración en dos, tres y n dimensiones con SciPy

SciPy dispone de los métodos `dblquad`, `tplquad` y `nquad` para la integración de funciones en dos, tres y n dimensiones, respectivamente. Todas utilizan el método de integración unidimensional `quad` descrito en la sección 19.3, que se invoca sucesivamente en cada dimensión de la integral. Por ejemplo, en el caso de la integral bidimensional

$$I = \int_a^b \int_{g(x)}^{h(x)} f(x, y) dx dy \quad (19.21)$$

el esquema de integración se muestra en la figura 19.5. En esta integral doble sobre el dominio sombreado en gris, la rutina externa de integración establece los valores de x (puntos rojos), para los cuales se selecciona un conjunto de valores de y que están delimitados por $g(x)$ y $h(x)$ (puntos azules) y que constituyen la rutina interna de evaluación. La rutina interna evalúa la integral sobre y en valores fijos de x , y finalmente la rutina externa obtiene el valor final de la integral sobre los valores de x .

Como un ejemplo simple de uso de las rutinas de SciPy, evaluaremos la integral

$$I = \int_{-1}^1 \int_{-1}^1 \exp(-x^2 - y^2) dx dy \quad (19.22)$$

En la celda 10 definimos la función integrando `f2`, y en la 11 definimos los límites de integración para x (`a` y `b`), así como las funciones $g(x)$ y $h(x)$, y finalmente invocamos el método `dblquad` de `scipy.integrate`, cuidando que el orden en el que aparecen los argumentos debe respetar el orden de los argumentos de `f2`. El resultado de `dblquad` muestra una tupla con el valor de la integral y una estimación del error.

CELDA 10

```
def f2(x, y):
    return np.exp(-x**2 - y**2)
```

CELDA 11

```
def g(x):
    return -1

def h(x):
    return 1

a, b = -1, 1
spi.dblquad(f2, a, b, g, h)
```

(2,23098514140413, $2,47689107158348 \cdot 10^{-14}$)

Podemos extender el ejemplo a tres dimensiones fácilmente. En este caso definimos la función f_3 agregando el argumento z , así como los límites de integración para esta variable que en este caso deben ser funciones de las variables de las rutinas “mas internas” que integran en x y y . Nuevamente definimos funciones asignando los límites en z a q y r :

CELDA 12

```
def f3(x, y, z):
    return np.exp(-x**2 - y**2 - z**2)

def q(x, y):
    return -1

def r(x, y):
    return 1

spi.tplquad(f3, a, b, g, h, q, r)
```

(3,3323070870931, $3,69960405321235 \cdot 10^{-14}$)

Para integrales de dimensiones mayores, SciPy ofrece el método `nquad`. Este método recibe como argumentos la función a ser integrada y los rangos de integración para cada variable como una secuencia de dos números. Naturalmente, `nquad` devuelve los mismos resultados que `dblquad` y `tplquad` para dos y tres dimensiones, respectivamente:

CELDA 13

```
spi.nquad(f3, [(-1, 1), (-1, 1), (-1, 1)])
```

(3,3323070870931, $3,69960405321235 \cdot 10^{-14}$)

Tal como anticipamos al comienzo de esta sección, la complejidad computacional crece rápidamente al incrementar las dimensiones de la integral. Para cuantificar esta tendencia usaremos `nquad` con una función generalizada del integrando de los ejemplos anteriores que acepta un número arbitrario de argumentos:

CELDA 14

```
def fn(*args):
    """
    f(x1, x2, ..., x2) = exp(-x1^2 - x2^2 ... -xn^2)
    """
    return np.exp(-np.sum(np.array(args)**2))
```

Evaluaremos entonces la integral con un número creciente de dimensiones (desde 1 hasta 5) y estimaremos el tiempo de ejecución utilizando el comando “mágico” `%time`⁶:

CELDA 15

```
%time spi.nquad(fn, [(-1, 1)] * 1) # Dimensión 1
```

```
CPU times: user 557 µs, sys: 0 ns, total: 557 µs
Wall time: 566 µs
```

(1,49364826562485, 1,65828269518814 · 10⁻¹⁴)

CELDA 16

```
%time spi.nquad(fn, [(-1, 1)] * 2) # Dimensión 2
```

```
CPU times: user 1.64 ms, sys: 0 ns, total: 1.64 ms
Wall time: 1.66 ms
```

(2,23098514140413, 2,47689107158348 · 10⁻¹⁴)

CELDA 17

```
%time spi.nquad(fn, [(-1, 1)] * 3) # Dimensión 3
```

```
CPU times: user 47.6 ms, sys: 2.79 ms, total: 50.4 ms
Wall time: 48.8 ms
```

(3,3323070870931, 3,69960405321235 · 10⁻¹⁴)

CELDA 18

```
%time spi.nquad(fn, [(-1, 1)] * 4) # Dimensión 4
```

```
CPU times: user 649 ms, sys: 0 ns, total: 649 ms
Wall time: 648 ms
```

(4,97729470116603, 5,52590717757931 · 10⁻¹⁴)

CELDA 19

```
%time spi.nquad(fn, [(-1, 1)] * 5) # Dimensión 5
```

⁶ Estos cálculos se realizaron en un procesador Intel i7 1355U.

```

CPU times: user 13.1 s, sys: 0 ns, total: 13.1 s
Wall time: 13.1 s

```

$(7,43432759790041, 8,25376167179527 \cdot 10^{-14})$

Podemos ver que los tiempos de ejecución aumentan desde algunas centenas de microsegundos para una dimensión, hasta varios segundos para 5. Es decir, aumentando el número de dimensiones 5 veces, el tiempo aumenta cuatro órdenes de magnitud.

19.4.2. Integración Monte Carlo

En la sección anterior vimos que el tiempo de evaluación de integrales múltiples crece rápidamente con el número de dimensiones, por lo que utilizar cuadraturas unidimensionales “anidadas” puede resultar impracticable incluso con un número bajo de dimensiones. Si no se requiere una gran precisión en el resultado, la integración Monte Carlo ofrece una alternativa poderosa y simple que consiste en la evaluación del integrando en puntos del dominio seleccionados aleatoriamente.

El principio básico de la integración Monte Carlo consiste en aproximar la integral de una función f que depende de d variables:

$$I = \int_{\Omega} f(\mathbf{x}) dV \quad (19.23)$$

donde $\mathbf{x} = (x_1, x_2, \dots, x_d)$, $dV = dx_1 dx_2 \cdots dx_d$ y Ω es un subconjunto de $\mathbb{R}^d$ con volumen V

$$V = \int_{\Omega} dV$$

Si hacemos un muestreo de N puntos aleatorios uniformemente distribuidos sobre Ω , que denotamos $\mathbf{x}_n$ con $n = 1, 2, \dots, N$, la estimación Monte Carlo de I es

$$I_N = \langle f \rangle V \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \quad (19.24)$$

Aquí, $\langle \cdot \rangle$ representa la media aritmética en los N puntos de la muestra:

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i) \quad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^N f^2(\mathbf{x}_i) \quad (19.25)$$

La ley de los grandes números asegura que la estimación Monte Carlo converge al valor exacto de la integral:

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i) = I$$

El segundo término de (19.24) es una estimación del error con una desviación estándar, no una cota rigurosa. El teorema del límite central indica que el error en la integración Monte Carlo escala como $1/\sqrt{N}$, independiente de la dimensión d . Este hecho hace que esta técnica de integración

resulte muy eficiente para el cálculo de integrales multidimensionales, aunque generalmente la convergencia es lenta.

Estimaremos mediante Monte Carlo la versión generalizada de la integral (19.21) (fn de la celda 13), en cinco dimensiones, utilizando para ello el módulo **vegas** para la evaluación de integrales multidimensionales utilizando un algoritmo de Monte Carlo adaptativo [28, 29]. Implementar computacionalmente un algoritmo propio de integración Monte Carlo no es demasiado complicado (como vimos en la sección 19.2.4 para el caso unidimensional). Sin embargo, este módulo especializado tiene rutinas que permiten un muestreo más eficiente, uso de múltiples procesadores y otras características que están fuera del alcance de este libro, y que para el ejemplo de un cálculo simple es muy conveniente ya que en pocas líneas tenemos el resultado buscado.

En la celda 20 importamos el módulo **vegas**, y a continuación creamos un objeto **Integrator** que almacenamos en **integ** y donde especificamos el volumen de integración (una lista con cinco elementos `[-1, 1]`). A continuación aplicamos el objeto **integ** a nuestro integrando estableciendo que realice **nitrn=10** iteraciones con un máximo de **neval=1000** puntos en cada una de ellas. Cada iteración produce una estimación independiente de la integral, pero adaptando el muestreo de los puntos dentro del volumen a partir de la precisión obtenida en iteraciones previas. El resultado de la integración se almacena en **result**, y la ejecución la hacemos con el comando **%time** para obtener el tiempo requerido de la misma. El resultado de cada una de las iteraciones se muestra imprimiendo **result.summary()**, donde se puede ver que en cada nueva iteración se alcanza una mejor precisión debido a que el nuevo muestreo “aprende” de los resultados obtenidos en las iteraciones anteriores. El promedio pesado que se muestra en la tercer columna se realiza mediante la inversa de la varianza, por lo que le asigna menos importancia a las primeras iteraciones. Las cuarta y quinta columna muestran los valores de la estimación estadística de los errores (según una distribución χ^2 y el correspondiente *p*-valor o *Q*). Finalmente mostramos el resultado de la estimación por Monte Carlo, con su correspondiente incerteza entre paréntesis.

CELDA 20				
<pre>import vegas integ = vegas.Integrator([-1, 1] * 5) %time result = integ(fn, nitrn=10, neval=1000) print(result.summary()) print(f'I_MC = {result}')</pre>				
CPU times: user 64.6 ms, sys: 0 ns, total: 64.6 ms Wall time: 65.2 ms				
itr	integral	wgt average	chi2/dof	Q
1	7.54(14)	7.54(14)	0.00	1.00
2	7.41(10)	7.455(82)	0.63	0.43
3	7.354(86)	7.407(59)	0.68	0.51
4	7.439(63)	7.421(43)	0.50	0.69
5	7.415(55)	7.419(34)	0.37	0.83
6	7.501(43)	7.451(27)	0.75	0.59
7	7.374(36)	7.424(21)	1.12	0.35
8	7.488(28)	7.448(17)	1.44	0.18
9	7.436(23)	7.443(14)	1.28	0.25
10	7.428(19)	7.438(11)	1.19	0.30
I_MC = 7.438(11)				

Lo primero que podemos notar es que el tiempo requerido para evaluar la integral (en la misma computadora que en los casos anteriores) es de poco menos de 65 ms, esto es, unas 200 veces más rápido que utilizando `nquad` de `scipy.integrate` para la misma dimensionalidad del dominio de integración. Como contrapartida, la precisión obtenida es de apenas el 0,16 % frente al $1,1 \times 10^{-12}$ % de `nquad`. Podemos mejorar la precisión obtenida con el método de Monte Carlo incrementando el número de puntos muestreados `neval`, o la cantidad de iteraciones `nitn`, aunque es generalmente mejor aumentar `neval`. Si utilizamos un orden de magnitud mayor (`neval=1e5`), la precisión puede disminuir un poco sin casi incrementar el tiempo de ejecución. Para más detalles sobre el paquete **vegas** y métodos avanzados de integración, recomendamos leer [su documentación](#).

19.5. Lecturas recomendadas

Fórmulas de Newton-Cotes y cuadraturas gaussianas:

- R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.
- Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, ago. de 2020. DOI: [10.1017/9781108772310](https://doi.org/10.1017/9781108772310). URL: <https://doi.org/10.1017/9781108772310>.
- Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.
- R. Bulirsch J. Stoer. *Introduction to Numerical Analysis*. 3rd ed. Texts in Applied Mathematics 12. Springer New York, 2002.
- Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, ago. de 2020. DOI: [10.1017/9781108772310](https://doi.org/10.1017/9781108772310). URL: <https://doi.org/10.1017/9781108772310>.
- S. Weinzierl. «Introduction to Monte Carlo methods». En: *arXiv: High Energy Physics - Phenomenology* (2000). URL: <https://arxiv.org/abs/hep-ph/0006269>.
- Malvin H. Kalos y Paula A. Whitlock. *Monte Carlo Methods*. Weinheim: John Wiley & Sons, 2009.
- Dirk Kroese, Thomas Taimre y Zdravko I. Botev. *Handbook of Monte Carlo Methods*. Hoboken, New Jersey: John Wiley & Sons, 2011.



20 | Ecuaciones diferenciales ordinarias

Las ecuaciones diferenciales ordinarias (EDOs) constituyen una de las herramientas más poderosas para la comprensión y predicción del comportamiento de sistemas dinámicos en la naturaleza, en ingeniería y en la sociedad. Un sistema dinámico es un sistema en algún estado, generalmente descrito por un conjunto de variables, que evoluciona en el tiempo. Por ejemplo, un cuerpo a una dada temperatura en un ambiente que está a otra temperatura, la evolución de una población, la propagación de un virus o el precio de un producto en un mercado de oferta y demanda. Las EDOs son ubicuas en todos los ámbitos de la ciencia y la tecnología¹.

Para representar un sistema dinámico necesitamos expresar las reglas matemáticas que gobiernan su evolución en el tiempo, y para ello podemos usar las leyes de la física, o simplemente la intuición. Estas reglas, cuando las variables que representan

Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1
Sympy	1.13.3

[Código disponible](#)

¹ Tal vez la ecuación diferencial de mayor impacto en el desarrollo científico y tecnológico sea la Segunda Ley de Newton [34]

el estado del sistema son continuas, toman la forma de ecuaciones diferenciales.

En las ecuaciones diferenciales la incógnita es una función, y para determinarla es necesario resolver dicha ecuación que contiene sus derivadas. Una función diferencial “ordinaria” es un caso especial en cual esta función desconocida tiene una sola variable independiente con respecto de la cual se deriva la función. Si por el contrario, la ecuación contiene derivadas de más de una variable, entonces es una ecuación en “derivadas parciales”, y esto se trata en el capítulo 21.

Por lo general, no existen soluciones analíticas de las EDOs excepto en algunos pocos casos, en los cuales se puede determinar la función desconocida utilizando métodos simbólicos. Si esto no es posible, existen diversos métodos numéricos que permiten obtener soluciones prácticas. En este capítulo exploraremos ambos abordajes para resolver EDOs.

20.1. Ecuaciones diferenciales ordinarias

La forma más simple de una EDO es

$$\frac{dy(x)}{dx} = f(x, y(x)) \quad (20.1)$$

donde $y(x)$ es la función desconocida y $f(x, y(x))$ se conoce. Esta ecuación es de primer orden porque solo aparece la derivada primera de f (omitiremos por comodidad la dependencia de la variable independiente en las expresiones de las derivadas). Más generalmente, una EDO de orden n se puede representar en forma “explícita” como

$$\frac{d^n y}{dx^n} = f\left(x, y, \frac{dy}{dx}, \dots, \frac{d^{n-1}y}{dx^{n-1}}\right) \quad (20.2)$$

o en forma “implícita” como

$$F\left(x, y, \frac{dy}{dx}, \dots, \frac{d^{n-1}y}{dx^{n-1}}\right) = 0 \quad (20.3)$$

donde f y F son funciones conocidas.

Cuando aparecen derivadas de órdenes superiores, siempre se puede reducir al estudio de un conjunto de EDOs de primer orden. Por ejemplo, la ecuación de segundo orden:

$$\frac{d^2 y}{dx^2} + f(x) \frac{dy}{dx} = g(x) \quad (20.4)$$

puede reescribirse como dos ecuaciones de primer orden:

$$\frac{dy}{dx} = z(x) \quad (20.5)$$

$$\frac{dz}{dx} = g(x) - f(x)z(x) \quad (20.6)$$

donde z es una nueva variable. Esto ejemplifica la forma usual de reducción de una ecuación de orden superior a un sistema lineal de EDOs, simplemente eligiendo las nuevas variables como derivadas de las otras (y de la variable original).

El problema entonces de las EDOs se reduce al estudio de un sistema de n ecuaciones diferenciales de primer orden acopladas, para las funciones y_i , $i = 1, 2, \dots, n$, con la forma general

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \dots, y_n), \quad i = 1, \dots, n \quad (20.7)$$

en la que introducimos las nuevas n funciones $y_1 = y$, $y_2 = dy/dx$, ..., $y_n = d^n y/dx^n$. Este sistema de EDOs de primer orden se puede escribir en una forma matricial:

$$\frac{d}{dx} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} y_2 \\ y_3 \\ \vdots \\ y_n \\ f(x, y_1, \dots, y_n) \end{bmatrix} \quad (20.8)$$

y en una forma vectorial compacta:

$$\frac{d}{dx} \mathbf{y}(x) = \mathbf{f}(x, \mathbf{y}(x)) \quad (20.9)$$

que resulta práctica para obtener soluciones numéricas.

Un problema que involucra EDOs no está completamente determinado por sus ecuaciones, es necesario también especificar las condiciones de borde del problema. Las condiciones de borde son expresiones algebraicas para los valores y_i en la ecuación 20.7. En general pueden ser satisfechas en un conjunto discreto de puntos, pero no se satisfacen en todos los puntos intermedios. Las condiciones de borde pueden ser tan simple como requerir que ciertas variables tengan valores numéricos dados, o tan complicadas como un conjunto de ecuaciones algebraicas no lineales entre las variables.

Generalmente, la naturaleza de las condiciones de borde determina el método de solución. Se dividen en dos categorías principales:

- **Problemas de valores iniciales:** todos los y_i tienen valores establecidos en algún valor inicial x_i , y se requiere determinar los valores de y_i en algún valor final x_f , o en alguna lista discreta de puntos (por ejemplo, a intervalos equiespaciados).
- **Problemas de contorno de dos puntos:** en este caso las condiciones de borde se especifican en más de un valor de x . Típicamente, algunas condiciones se especifican para x_i y las restantes para x_f .

En este capítulo abordaremos la solución de problemas de valores iniciales, dejando los problemas de contorno para el capítulo 21 referido a ecuaciones en derivadas parciales.

20.2. Solución analítica de EDOs

En 1838, Pierre-François Verhulst propuso un modelo de crecimiento poblacional en el que la tasa de reproducción es proporcional tanto a la población existente como a la cantidad de recursos

disponibles. Si llamamos P a la cantidad de individuos en la población en función del tiempo t , el modelo queda formalizado por la ecuación:

$$\frac{dP(t)}{dt} = rP(t) \left[1 - \frac{P(t)}{K} \right] = rP(t) - \frac{rP(t)^2}{K} \quad (20.10)$$

donde r define la tasa de crecimiento y K la capacidad de carga del sistema.

En este modelo, el crecimiento inicial para valores pequeños de P (omitimos en lo que sigue la dependencia explícita del tiempo) está dominado por el primer término rP , donde el valor de r representa el incremento proporcional a la población P por unidad de tiempo. A medida que la población crece, lo hace también el segundo término $-rP^2/K$, disminuyendo de este modo la velocidad de crecimiento, debido a que los miembros de la población compiten entre sí por los recursos disponibles tales como espacio o alimentos, que están representados por la constante K . Esta competencia entre dos términos de signo opuesto limita el crecimiento, que se detiene cuando $dP/dt = 0$.

Resolveremos la EDO representada por la ecuación 20.10 utilizando la función `dsolve` de SymPy, que es capaz de encontrar soluciones analíticas a un conjunto de EDOs elementales. Es necesario enfatizar aquí que la mayoría de las EDO no pueden resolverse en forma analítica. Es posible obtener dichas soluciones para EDOs de primer o segundo orden, o sistemas lineales de primer orden que contienen unas pocas funciones conocidas.

Utilizaremos un *Jupyter-notebook* para codificar la solución al problema y encontrar la solución utilizando SymPy. Para ello es necesario primero definir símbolos para las variables P , t , r , K y P_0 , siendo esta última la población inicial $P_0 = P(0)$; y también necesitamos representar la función desconocida $P(t)$. Para ello en la celda 1 definimos tales cantidades, la función y almacenamos en `edo` la función diferencial dada por la ecuación 20.10.

CELDA 1 DE ODE.IPYNB
<pre>import sympy t, K, r, P0, C1 = sympy.symbols('t, K, r, P_0, C_1') P = sympy.Function('P') edo = P(t).diff(t) - r * P(t) * (1 - P(t)/K) edo</pre> $-r \left(1 - \frac{P(t)}{K} \right) P(t) + \frac{d}{dt} P(t)$

Junto con las demás variables, definimos también en la primera celda a la constante `C1`, que resultará la constante de integración que se determina con la condición inicial.

A continuación, `sympy.odesolve` recibe la ecuación `ode` y la función $P(t)$ como argumentos y devuelve la solución en la variable `edo_sol`.

CELDA 02
<pre>edo_sol = sympy.odesolve(edo, P(t)) edo_sol</pre>

$$P(t) = \frac{Ke^{C_1 K + rt}}{e^{C_1 K + rt} - 1}$$

En la celda 3 definimos las condiciones iniciales mediante un diccionario, y finalmente en la celda 4 reemplazamos estas condiciones iniciales en la solución guardada en `edo_sol`, lo que genera una expresión (que en este ejemplo guardamos en `C_eq`) que permite resolver la constante de integración C_1 en función de la condición inicial P_0 y de la capacidad de carga K .

CELDA 03

```
ini_cond = {P(0): P0}
ini_cond
{P(0): P_0}
```

CELDA 04

```
C_eq = edo_sol.subs(t,0).subs(ini_cond)
C_eq
```

$$P_0 = \frac{Ke^{C_1 K}}{e^{C_1 K} - 1}$$

Con esta última expresión, podemos despejar C_1 en función de P_0 y K , con lo cual tenemos completamente resuelto el problema para la condición inicial dada. En este caso, es sencillo determinar el valor de C_1 utilizando álgebra elemental. Definiendo

$$A = e^{C_1 K}$$

el valor de esta constante resulta

$$A = \frac{P_0}{P_0 - K}$$

En las últimas dos celdas del *notebook* implementamos la solución en la forma usual de las funciones en Python, para poder graficar soluciones según los valores de los parámetros. En particular, visualizamos dos soluciones del problema con valores de la población inicial P_0 por debajo y por encima de la capacidad de carga K del sistema.

CELDA 05

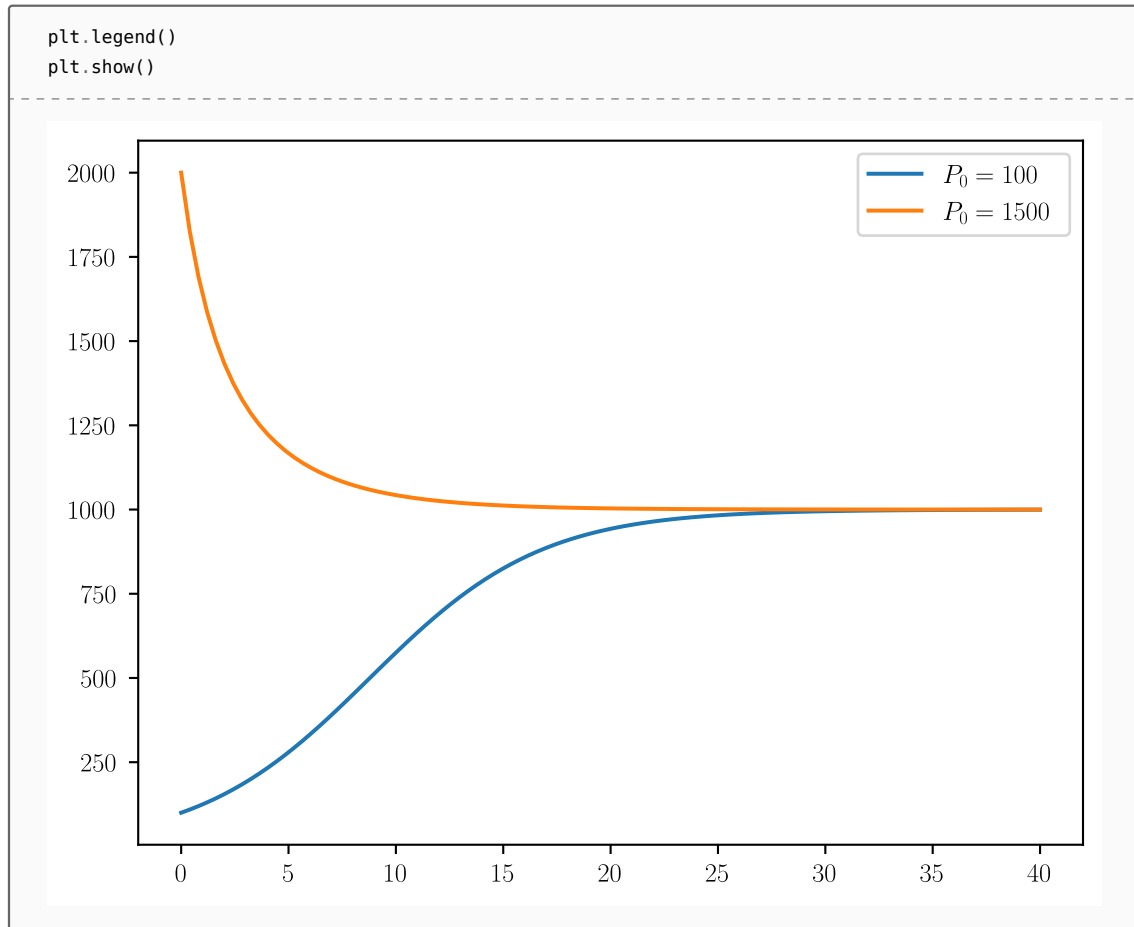
```
import numpy as np

def logistica(t, P0=100, K=1000, r=0.25):
    A = P0 / (P0 - K)
    return K / (1 - np.exp(-r*t) / A)
```

CELDA 06

```
import matplotlib.pyplot as plt

t = np.linspace(0, 40, 100) # Intervalo de tiempo en que se obtiene la solución
p1 = plt.plot(t, logistica(t, P0 = 100), label=r'$P_0 = 100$')
p1 = plt.plot(t, logistica(t, P0 = 2000), label=r'$P_0 = 1500$')
```



20.3. Métodos numéricos para resolver EDOs

Aunque existen problemas representados por EDOs que pueden resolverse analíticamente, por lo general es común que tal solución analítica no exista. Por lo tanto, en la práctica los problemas suelen ser resueltos utilizando métodos numéricos.

Existen muchos abordajes para resolver EDOs numéricamente, y muchos de ellos están diseñados para problemas formulados como un sistema de ecuaciones de primer orden en su forma estándar como se representa en la ecuación 20.9. SciPy ofrece funciones para resolver este tipo de problemas, que veremos en forma práctica más adelante, ya que primero haremos una revisión de los conceptos principales y de la terminología utilizada para la resolución numérica de EDOs.

El método de Euler constituye la base de muchos otros métodos para EDOs, y puede derivarse a partir de la expansión en serie de Taylor de $y(x)$ alrededor de x :

$$y(x+h) = y(x) + \frac{dy(x)}{dx}h + \frac{1}{2} \frac{d^2y(x)}{dx^2}h^2 + O(h^3) \quad (20.11)$$

donde por simplicidad de la notación hemos considerado que $y(x)$ es una función escalar. Si despreciamos términos de segundo orden y mayores, obtenemos la ecuación aproximada

$$y(x+h) \approx y(x) + f(x, y(x))h \quad (20.12)$$

que es precisa hasta el primer orden en el paso h . Esta ecuación se puede transformar en una fórmula iterativa discretizando la variable x en el intervalo de interés: $x_0, x_1, \dots, x_n$, eligiendo el paso $h = x_{k+1} - x_k$ y denotando $y_k = y(x_k)$, lo que da origen a la fórmula del método de Euler “hacia adelante”:

$$y_{k+1} = y_k + hf(x_k, y_k) \quad (20.13)$$

que avanza una solución desde x_k hasta $x_{k+1} \equiv x_k + h$. Esta fórmula es asimétrica: avanza la solución sobre un intervalo h utilizando la información de la derivada solo al comienzo del mismo. La fórmula de Euler hacia adelante es una forma explícita debido a que dado el valor de y_k podemos calcular directamente el de y_{k+1} .

El objetivo de obtener la solución numérica de un problema con valores iniciales es el de calcular $y(x)$ en un conjunto de puntos x_n dada la condición inicial $Y(x_0) = y_0$. La fórmula del método de Euler hacia adelante permite calcular los valores sucesivos de y_k comenzando en y_0 . Sin embargo, este método no es recomendado para su uso práctico por dos razones: (a) el método no es preciso comparado con otros métodos más elegantes con el mismo paso, y (b) no es muy estable.

Una fórmula alternativa que puede derivarse en forma similar es el método de Euler “hacia atrás”, dado por la iteración

$$y_{k+1} = y_k + f(x_{k+1}, y_{k+1}) \quad (20.14)$$

que resulta en un método “implícito” debido a que y_{k+1} aparece en ambos lados de la ecuación y por lo tanto es necesario resolver una ecuación algebraica, que generalmente es no lineal. Los métodos implícitos resultan más complicados de implementar que los explícitos, y cada iteración requiere un mayor esfuerzo computacional. Sin embargo, tienen la ventaja de tener mejor precisión y una mayor región de estabilidad. La eficacia de cada abordaje (implícito o explícito) depende en definitiva de cada problema, aunque en términos generales los métodos implícitos son útiles en problemas “rígidos”, esto es, en los cuales existen dinámicas con tiempos característicos muy dispares (por ejemplo, dinámicas que incluyen oscilaciones rápidas y lentas).

Existen varios métodos que permiten mejorar los métodos de primer orden de Euler hacia adelante y hacia atrás. Por ejemplo, podemos mantener términos de mayor orden en la expansión en serie de Taylor dada por la ecuación 20.11 que brindan fórmulas iterativas de mayor orden, y que por lo tanto tienen mayor precisión. No obstante, estos métodos requieren evaluar derivadas superiores de $y(x)$, lo que puede ser un problema si $f(x, y(x))$ no se conoce de antemano (y no está dada en forma simbólica).

Una forma de evitar este problema es el de aproximar las derivadas de orden superior utilizando diferencias finitas de las derivadas, o evaluando la función $f(x, y(x))$ en puntos intermedios del intervalo $[x_k, x_{k+1}]$. Por lejos, el esquema más utilizado constituye la fórmula de Runge-Kutta



Leonhard Paul Euler - Matemático y físico suizo (1707-1783). Principal matemático del siglo XVIII y uno de los más grandes y prolíficos de todos los tiempos, muy conocido por el número de Euler (e). Realizó importantes descubrimientos en áreas tan diversas como el cálculo o la teoría de grafos, introdujo gran parte de la terminología moderna y notación matemática, y se le conoce por sus trabajos en los campos de la mecánica, la óptica y la astronomía.

de cuarto orden:

$$\begin{aligned}
 k_1 &= h(f(x_k, y_k)) \\
 k_2 &= hf\left(x_k + \frac{h}{2}, y_k + \frac{k_1}{2}\right) \\
 k_3 &= hf\left(x_k + \frac{h}{2}, y_k + \frac{k_2}{2}\right) \\
 k_4 &= hf(x_k + h, y_k + k_3) \\
 y_{k+1} &= y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)
 \end{aligned} \tag{20.15}$$

Aquí se realizan cuatro evaluaciones diferentes de la función $f(x, y(x))$ que son utilizadas en la fórmula explícita 20.15, que es precisa hasta el cuarto orden, con un error de quinto orden. Es posible estimar también el error en la aproximación mediante la combinación de dos métodos de órdenes diferentes. Una combinación muy utilizada es la de esquemas de Runge-Kutta de cuarto y quinto orden, que resulta en un método preciso de cuarto orden con estimación del error, conocido como el método RK45 o Runge-Kutta-Fehlberg. Otro ejemplo es el método Dormand-Prince, que utiliza un control de paso adaptativo y que está disponible en SciPy.

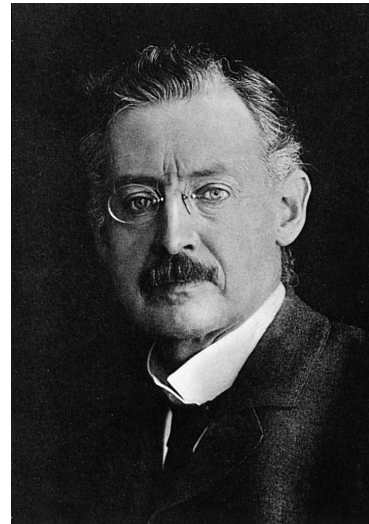
Un método alternativo consiste en usar más de un valor previo y_{n-k} para calcular y_n . Estos métodos se denominan multipaso y se pueden expresar en forma general como

$$y_n = \sum_{j=1}^k a_j y_{n-j} + h \sum_{j=0}^k b_j f(x_{n-j}, y_{n-j}) \tag{20.16}$$

De este modo, para calcular y_n se utilizan s valores previos de y_{n-j} y $f(x_{n-j}, y_{n-j})$. A partir de esta fórmula general se derivan diferentes esquemas según la elección de los coeficientes a_j y b_j . Por ejemplo, los métodos multipaso más importantes para problemas “no rígidos” son los del tipo Adams, que consisten en la elección de los coeficientes para la solución en x_n

tomando $a_1 = 1$ y $a_2 = a_3 = \dots = a_k = 0$. Aquí, las constantes b_j se elijen de forma tal que se obtiene el máximo orden posible. Debe notarse que si $b_0 = 0$ el método es explícito y se conoce como métodos de Adams-Bashforth, mientras que si $b_0 \neq 0$ los métodos resultan implícitos y se denominan Adams-Moulton. Por ejemplo, los métodos de un paso Adams-Bashforth y Adams-Moulton se reducen a los métodos de Euler hacia adelante y hacia atrás, respectivamente.

Los métodos de Adams se implementan, generalmente, en la forma “predictor-corrector”. En este esquema se realiza un cálculo preliminar usando la forma Bashforth del método (explícito). Esta solución aproximada en x_n se utiliza para evaluar una aproximación al valor de la derivada en este nuevo punto, que luego es utilizada en la fórmula de Moulton como valor inicial para resolver el método implícito.



Carl Runge - Matemático, físico y espectroscopista alemán (1856-1927). Sus intereses incluían la matemática, la espectroscopia, la geodesia y la astrofísica. Fue desarrollador del método de Runge-Kutta en el campo conocido actualmente como análisis numérico.

Una cuestión que es necesaria considerar al utilizar métodos multipaso es la forma de inicializar el proceso. Para un método de k -pasos, el método ofrece un algoritmo para calcular y_k en términos de $y_0, y_1, \dots, y_{k-1}$, sin embargo se debe pensar en cómo obtener estos valores previos. Una posibilidad es evaluar los primeros $k - 1$ valores usando una secuencia de métodos de orden bajo, sin embargo, esto introduce errores que anularían las ventajas de un uso posterior de métodos de mayor orden. También es posible utilizar un método de Runge-Kutta para los primeros $k - 1$ pasos, siempre que el método de Runge-Kutta sea del mismo orden que el método de k -pasos utilizado posteriormente, de modo que no haya una pérdida de orden.

Una técnica simple que se utiliza para no utilizar métodos alternativos y al mismo tiempo mantener el orden durante el proceso de inicialización es el siguiente. Introducimos, como incógnitas a ser calculadas, aproximaciones a los valores de $f(x_i, y_i)$ para $i = -(k - 1), -(k - 2), \dots, -1$. Los valores iniciales para estas cantidades se eligen como $f(x_{i-1}, y_{i-1}) = f(x_0, y_0)$. A partir de estos valores es posible calcular los valores de y_i y $f(x_i, y_i)$ para $i = 1, 2, \dots, k - 1$. Luego de esto invertimos el orden de la integración cambiando el signo de h y recalculamos los valores iniciales y_i y $f(x_i, y_i)$ para $i = -1, -2, \dots, -(k - 1)$. Este proceso de alternar integraciones hacia adelante y hacia atrás se repite hasta que se alcanza convergencia, y entonces hemos encontrado valores iniciales aceptables que permitirán avanzar en la integración calculando los pasos $i = k, i = k + 1, \dots$.

Para finalizar esta sección, queremos mencionar que muchos métodos avanzados de solución de EDOs utilizan un control de paso, o paso adaptativo. La razón de esto es que la precisión y estabilidad de los métodos dependen fuertemente del paso de integración. Entonces, se presentan dos alternativas para ajustar el paso en cada iteración de modo de reducir el costo computacional manteniendo una precisión establecida. Una de estas alternativas surge cuando es posible obtener junto con y_n una estimación del error. De este modo es posible ajustar el tamaño de h_n para utilizar valores grandes cuando es posible, y pasos pequeños cuando sea necesario. La otra posibilidad es ajustar el orden del método de modo de utilizar un orden bajo (de menor costo computacional) cuando el error es pequeño, y un método de mayor orden cuando el error estimado no alcanza la precisión requerida con un método de orden menor. Para esta última alternativa, los métodos de Adams son apropiados ya que se pueden cambiar de orden fácilmente.

Existe una gran variedad de métodos para resolver numéricamente EDOs, y no es aconsejable implementar alguno de los métodos descritos en esta sección ya que para fines prácticos es posible utilizar muchas de las bibliotecas que están muy optimizadas y extensamente verificadas, disponibles en paquetes de código abierto y libre como SciPy. Por supuesto, para poder utilizar estas librerías es necesario conocer las ideas básicas y las metodologías para poder elegir correctamente el método apropiado para cada problema.



Martin Wilhelm Kutta - Físico y matemático alemán (1867-1944). Codesarrolló el método de Runge-Kutta para resolver ecuaciones diferenciales ordinarias. Será recordado también por el teorema de Zhukovsky-Kutta así como por la condición de Kutta, ambas fundamentales en la aerodinámica potencial.

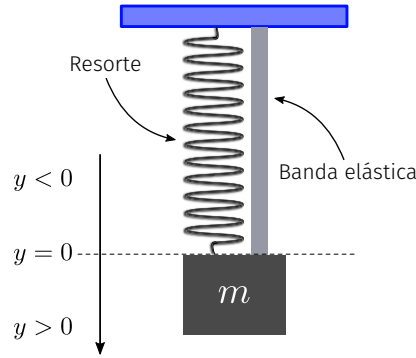


FIGURA 20.1: sistema masa - resorte - banda elástica representada por la ecuación 20.17.

20.4. Solución de EDOs con valores iniciales

El módulo `integrate` de SciPy ofrece la función `integrate.solve_ivp` para resolver numéricamente un sistema de ecuaciones diferenciales con valores iniciales con diferentes métodos de integración: variantes de Runge-Kutta explícito de varios órdenes, un método implícito multipaso basado en una fórmula de diferenciación hacia atrás, y también una implementación de uno de los códigos más robustos y confiables, LSODA, que forma parte del paquete ODEPACK desarrollado en FORTRAN por el Lawrence Livermore National Laboratory². Este integrador alterna automáticamente entre rutinas de integración para métodos rígidos (Adams predictor-corrector) y no rígidos (Gear, método de diferenciación hacia atrás).

Resolveremos la dinámica de un sistema compuesto por una masa suspendida mediante la combinación de un resorte y una banda elástica [35] tal como se esquematiza en la figura 20.1. En este sistema, una masa m está sujeta a un resorte vertical junto a una banda elástica que provee un sostén adicional. Mientras que el resorte produce una fuerza restauradora tanto en las direcciones ascendentes como descendentes, la banda elástica solo ejerce una fuerza hacia arriba al estirarse cuando la masa desciende. Llamemos $y(t)$ al desplazamiento de m de su posición de equilibrio $y = 0$ al tiempo t .

Tres fuerzas actúan sobre la masa: la fuerza restauradora de la banda elástica, la fuerza lineal restauradora del resorte, y el peso. Sin la banda elástica la fuerza restauradora del resorte (en el sistema de referencia indicado en la figura) está dada por $-k_r y$. La banda elástica aporta una fuerza restauradora $-k_b y^+$ debido a que solo actúa cuando la masa desciende ($y > 0$), pero no en compresión ($y < 0$). Por lo tanto, la fuerza restauradora combinada resultante es $-ay^+ + by^-$, donde $a > b$ y $y^+ = y$ si $y > 0$, mientras que es 0 si $y < 0$, y $y^- = -y$ si $y < 0$ y 0 en otro caso.

Queremos estudiar la respuesta de este sistema al someterlo a una pequeña fuerza periódica $f(t) = \lambda \sin(\mu t)$, donde λ es la amplitud de la fuerza y μ es la frecuencia angular. Combinando estas fuerzas y añadiendo un pequeño término de amortiguación viscosa (proporcional a la velocidad de la masa) el siguiente modelo:

$$\ddot{y} + 0,01\dot{y} + ay^+ - by^- = 10 + \lambda \sin(\mu t) \quad (20.17)$$

² Se puede ver más información sobre ODEPACK en su [web](#).

donde como es usual, la derivada de y respecto del tiempo se denota $\dot{y}$, y

$$y^+ = \begin{cases} y, & y \geq 0 \\ 0, & y < 0 \end{cases} \quad y^- = \begin{cases} 0, & y \geq 0 \\ -y, & y < 0 \end{cases} \quad (20.18)$$

o simplemente $y^+ = \max(y, 0)$ y $y^- = \max(-y, 0)$. Para algunos valores de los parámetros a , b , λ y μ , el sistema muestra un comportamiento senoidal, pero para otros, la no linealidad que aporta la banda elástica genera un comportamiento caótico. El código siguiente integra numéricamente este sistema.

```

1  #!/usr/bin/env python3
2  """Programa que integra el sistema masa-resorte-banda elástica."""
3
4  import matplotlib.pyplot as plt
5  import numpy as np
6  from scipy.integrate import solve_ivp
7
8  # Configure matplotlib
9  plt.rcParams.update({
10     'text.usetex': True,
11     'font.size': 14,
12     'axes.labelsize': 'large',
13 })
14
15
16  def pos(x):
17     """Devuelve x si x > 0, 0 en otro caso."""
18     return max(x, 0)
19
20
21  def neg(x):
22     """Devuelve -x si x < 0, 0 en otro caso."""
23     return max(-x, 0)
24
25
26  # Parámetros del sistema
27  a, b = 17, 1 # Combinación de constantes elásticas
28  λ = 15.4 # Amplitud de la excitación periódica
29  μ = 0.75 # Frecuencia angular de la excitación periódica
30
31
32  def sistema(t, z):
33     """Definición del sistema de ecuaciones diferenciales de primer orden.
34
35     Argumentos:
36         t: escalar que representa el tiempo
37         z: lista con [y, dy/dt]
38
39     Devuelve:
40         [dy(t)/dt, d^2y(t)/dt^2]
41     """
42     y, yp = z # yp = dy/dt
43     return [yp, 10 + λ * np.sin(μ * t) - 0.01 * yp - a * pos(y) + b * neg(y)]
44
45

```

```

46 # Array con la discretización del intervalo de tiempo en el que se calcula
47 # la solución del sistema de EDOs
48 t = np.linspace(0, 100, 10000)
49
50 # Solución del sistema de EDOs
51 sol = solve_ivp(sistema, [0, 100], [1, 0], t_eval=t)
52
53 # Gráfico de la solución y del diagrama de fase
54 fig, (ax0, ax1) = plt.subplots(1, 2)
55 ax0.plot(sol.t, sol.y[0])
56 ax0.set_ylim(8, -19)
57 ax0.set_xlabel(r'$t$')
58 ax0.set_ylabel(r'$y$')
59 ax0.text(0.05, 0.9, 'a)', transform=ax0.transAxes, fontsize=18)
60 ax1.plot(sol.y[0], sol.y[1])
61 ax1.set_xlabel(r'$y$')
62 ax1.set_ylabel(r'$dy/dt$')
63 ax1.text(0.05, 0.9, 'b)', transform=ax1.transAxes, fontsize=18)
64 plt.show()

```

Importamos el método `solve_ivp` de `scipy.integrate` en la línea 7. Las funciones 20.18 están definidas en las líneas 11 y 15, respectivamente, mientras que los parámetros del sistema para una integración particular están definidos en las líneas 20–22.

Dado que el sistema representado en la ecuación 20.17 es de segundo orden, es necesario primero convertirlo en un sistema de dos ecuaciones de primer orden. Esto se realiza en la función `sistema()`, que recibe como argumentos un escalar t que representa a la variable tiempo t , y una lista con los valores de y y $\dot{y}$. La función devuelve entonces una lista con los valores calculados para $\dot{y}$ y $\ddot{y}$, siguiendo el mismo esquema que mostramos en las ecuaciones 20.8 y 20.9.

La solución del sistema se calcula en la línea 39, en la que usamos el método `solve_ivp` al que se le pasa como argumentos la función que representa al sistema de EDOs (`sistema()`), el intervalo de integración en forma de lista con el valor inicial y final $[t_0, t_f]$, que en nuestro caso es $[0, 100]$, una lista con los valores iniciales $y(0) = 1$ y $\dot{y}(0) = 0$, y finalmente un argumento con nombre (`t_eval`) que contiene un array con los instantes de tiempo en los que guardamos los valores de la solución calculada. En este ejemplo usamos el array definido en la línea 36. Es requisito que los valores del array se encuentren dentro del intervalo definido en el intervalo de integración.

El método `solve_ivp` devuelve un objeto que contiene numerosos campos con la solución del sistema de EDOs e información sobre el proceso de integración, y que nosotros guardamos en `sol`. En este ejemplo, solamente utilizaremos los atributos `t` y `y` de `sol` que contienen el array de valores de tiempo donde se calculó la solución, y una lista con los valores calculados de y y $\dot{y}$ correspondientes, respectivamente, con el propósito de visualizar la trayectoria y el espacio de fase, tal como se muestra en la figura 20.2.

El método `solve_ivp` admite la selección de diversos métodos de integración. Por defecto utiliza RK45, como hicimos en el ejemplo, pero se pueden pasar con el argumento `method` los métodos RK23, DOP853, Radau, BDF y LSODA, tal como se indica en la [documentación](#).

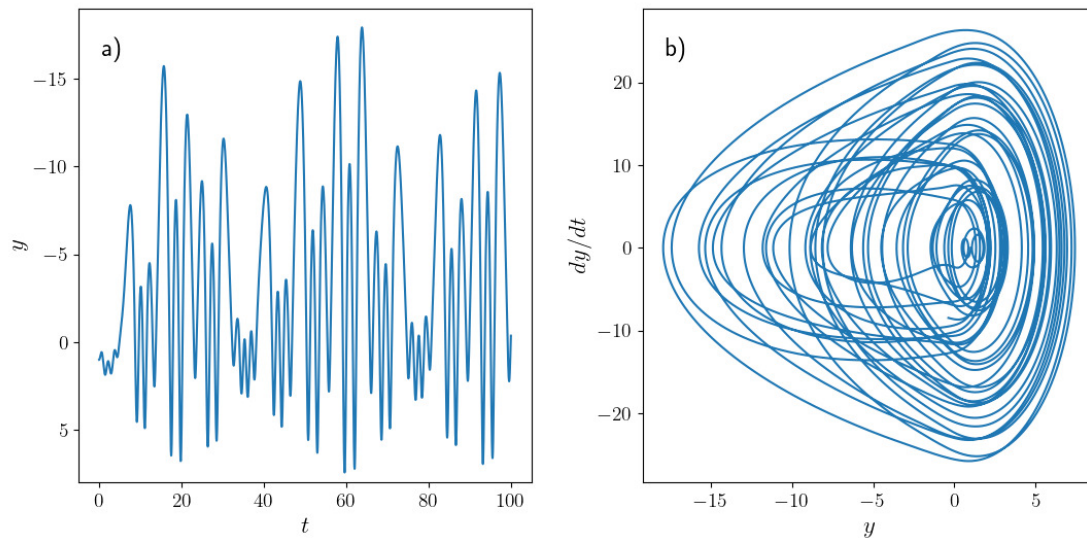


FIGURA 20.2: solución de la ecuación 20.17 En el panel a) se muestra la trayectoria de la masa m , mientras que en el b) se visualiza el espacio de fase del sistema.

20.5. Lecturas recomendadas

- John Charles Butcher. *Numerical methods for ordinary differential equations*. Chichester, England Hoboken, NJ: Wiley, 2008.
- William H Press et al. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge, UK New York: Cambridge University Press, 2007.



21 | Ecuaciones en derivadas parciales

La mayoría de los problemas presentados en la naturaleza, que son de interés en para físicos y matemáticos, se representan por medio de una o de un sistema de ecuaciones diferenciales. En general, los sistemas físicos involucran más de una variable independiente, por lo que en ese caso el modelo matemático contiene ecuaciones en derivadas parciales (EDP).

Las EDP juegan un papel fundamental en muchas ramas de las ciencias aplicadas y las ingenierías, por ejemplo, en transferencia de calor, elasticidad, dinámica de fluidos, electromagnetismo, óptica, mecánica cuántica, matemática financiera, etc. En particular, fenómenos como el flujo de calor en una varilla

metálica u ondas en una cuerda se pueden representar por una EDP de segundo orden del tipo

$$R(x, y) \frac{\partial^2 u}{\partial x^2} + S(x, y) \frac{\partial^2 u}{\partial x \partial y} + T(x, y) \frac{\partial^2 u}{\partial y^2} + L \left(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right) = 0 \quad (21.1)$$

en la que u es la función desconocida de x y y (omitimos por simplicidad la dependencia explícita en las expresiones), las funciones R , S y T son funciones continuas de las variables x y y , mientras

Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1

[Código disponible](#)

que L es una función continua de $x, y, u, \partial u/\partial x$ y $\partial u/\partial y$, de esta manera, la ecuación es lineal.

En lo que sigue, con el propósito de simplificar la escritura, adoptaremos la notación:

$$u_x = \frac{\partial u}{\partial x}, \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}, \quad u_{xy} = \frac{\partial^2 u}{\partial x \partial y}$$

La ecuación (21.1) se clasifica en tres clases diferentes según el valor de $S^2 - 4RT$:

- a) Elíptica, si $S^2 - 4RT < 0$.
- b) Parabólica, si $S^2 - 4RT = 0$.
- c) Hiperbólica, si $S^2 - 4RT > 0$.

Esta clasificación es útil por dos razones. Por un lado, cada clase está asociada a diferentes problemas específicos, y por otro, cada grupo requiere técnicas de solución analítica especiales. La terminología utilizada para clasificar las ecuaciones surge por analogía con la utilizada en la clasificación de ecuaciones generales de segundo orden en la geometría analítica. Es importante notar que según como sea la dependencia de R, S y T de x y y , la ecuación puede estar en una clase diferente dependiendo del dominio para el cual se quiere obtener la solución.

Para determinar completamente una solución particular de una EDP es necesario definir las condiciones de borde, que son valores conocidos de la función o una combinación de sus derivadas sobre la frontera del dominio Ω del problema. Usualmente, la frontera del dominio se denota como Γ o $\partial\Omega$, y por lo general las condiciones de borde pueden especificarse de forma diferente en partes distintas de la frontera. Dos condiciones de borde importantes son las de Dirichlet, en las que se especifica el valor de la función en la frontera, $u(\mathbf{x}) = h(\mathbf{x})$ para $\mathbf{x} \in \Gamma_D$, y las de Neumann, que especifica la derivada normal en la frontera:

$$\frac{\partial u(\mathbf{x})}{\partial \mathbf{n}} = g(\mathbf{x})$$

para $\mathbf{x} \in \Gamma_N$, donde $\mathbf{n}$ es el vector normal a la frontera con sentido hacia afuera. Aquí, $h(\mathbf{x})$ y $g(\mathbf{x})$ son funciones arbitrarias.

Del mismo modo que sucede con las ecuaciones diferenciales ordinarias, solo se puede hallar soluciones analíticas de un problema representado mediante EDP en muy pocos y especiales casos [38]. En casi todos los problemas de interés es necesario buscar soluciones mediante técnicas numéricas.

Muchas de las técnicas numéricas para la solución de EDP se basan en la idea de discretizar el dominio en cada una de las variables independientes que aparecen en las ecuaciones, transformando de este modo el problema en un sistema de ecuaciones algebraicas. Este abordaje suele ser muy demandante de recursos computacionales, en parte debido a que el número de puntos requerido para discretizar una región crece exponencialmente con el número de dimensiones. Por ejemplo, si en un problema unidimensional discretizamos su dominio en 100 puntos, un problema bidimensional con resolución similar requiere $100^2 = 10^4$ puntos, y uno tridimensional necesita $100^3 = 10^6$ puntos. Dado que cada punto en el espacio discretizado corresponde a una variable desconocida, un problema en tres dimensiones puede resultar en un sistema muy grande de ecuaciones.

En este capítulo abordaremos dos de estas técnicas. En primer lugar, el método de las diferencias finitas (MDF) que consiste en aproximar las derivadas por sus expresiones en diferencias

finitas, y posteriormente veremos el método de los elementos finitos (MEF), en el que la función desconocida se escribe como combinación lineal de una base de funciones simples que pueden ser diferenciadas e integradas fácilmente. La función desconocida se describe entonces por un conjunto de coeficientes en esta representación, y mediante una reformulación del problema se pueden obtener ecuaciones algebraicas para estos coeficientes.

21.1. Método de las diferencias finitas

El método de las diferencias finitas es conceptualmente simple y el más utilizado para resolver ecuaciones diferenciales con condiciones de borde. Consiste, básicamente, en discretizar el dominio y aproximar las ecuaciones diferenciales con sus expresiones en diferencias finitas, calculadas en los puntos discretos del dominio.

Estas ecuaciones en diferencias finitas proceden de las expansiones en series de Taylor de funciones multivariadas, que dan origen a diferentes esquemas. Por ejemplo, la expansión en serie de Taylor de la función $u = u(x, y)$ es:

$$\begin{aligned}
 u(x+h, y) &= u(x, y) + hu_x + \frac{h^2}{2!}u_{xx} + \frac{h^3}{3!}u_{xxx} + \cdots \\
 u(x-h, y) &= u(x, y) - hu_x + \frac{h^2}{2!}u_{xx} - \frac{h^3}{3!}u_{xxx} + \cdots \\
 u(x+2h, y) &= u(x, y) + 2hu_x + \frac{(2h)^2}{2!}u_{xx} + \frac{(2h)^3}{3!}u_{xxx} + \cdots \\
 u(x-2h, y) &= u(x, y) - 2hu_x + \frac{(2h)^2}{2!}u_{xx} - \frac{(2h)^3}{3!}u_{xxx} + \cdots \\
 u(x, y+k) &= u(x, y) + ku_y + \frac{k^2}{2!}u_{yy} + \frac{k^3}{3!}u_{yyy} + \cdots \\
 u(x, y-k) &= u(x, y) - ku_y + \frac{k^2}{2!}u_{yy} - \frac{k^3}{3!}u_{yyy} + \cdots \\
 &\dots \\
 u(x+h, y+k) &= u(x, y) + (hu_x + ku_y) + \frac{1}{2!}(h^2u_{xx} + 2hku_{xy} + k^2u_{yy}) + \cdots
 \end{aligned}$$

A partir de estas expresiones es sencillo obtener las aproximaciones para las derivadas parciales de primer y segundo orden. Para las de primer se pueden obtener diferencias hacia adelante:

$$\begin{aligned}
 u_x &= \frac{u(x+h, y) - u(x, y)}{h} + O(h) \\
 u_y &= \frac{u(x, y+k) - u(x, y)}{k} + O(k)
 \end{aligned}$$

también diferencias hacia atrás:

$$\begin{aligned}
 u_x &= \frac{u(x, y) - u(x-h, y)}{h} + O(h) \\
 u_y &= \frac{u(x, y) - u(x, y-k)}{k} + O(k)
 \end{aligned}$$

y diferencias centrales:

$$u_x = \frac{u(x+h, y) - u(x-h, y)}{2h} + O(h^2)$$

$$u_y = \frac{u(x, y+k) - u(x, y-k)}{2k} + O(k^2)$$

Del mismo modo se pueden obtener expresiones en diferencias para las derivadas parciales de segundo orden. En el caso de diferencias hacia adelante resultan:

$$u_{xx} = \frac{u(x+2h, y) - 2u(x+h, y) + u(x, y)}{h^2} + O(h)$$

$$u_{yy} = \frac{u(x, y+2k) - 2u(x, y+k) + u(x, y)}{k^2} + O(k)$$

$$u_{xy} = \frac{u(x+h, y+k) - u(x, y+k) - u(x+h, y) + u(x, y)}{hk} + O(h, k)$$

donde la última derivada mixta se obtiene a partir de aproximaciones en diferencias finitas de las derivadas de primer orden. Para el caso de las diferencias hacia atrás las aproximaciones son

$$u_{xx} = \frac{u(x, y) - 2u(x-h, y) + u(x-2h, y)}{h^2} + O(h)$$

$$u_{yy} = \frac{u(x, y) - 2u(x, y-k) + u(x, y-2k)}{k^2} + O(k)$$

$$u_{xy} = \frac{u(x, y) - u(x, y-k) - u(x-h, y) + u(x-h, y-k)}{hk} + O(h, k)$$

y por último, para las diferencias centrales, se obtiene

$$u_{xx} = \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h^2} + O(h^2)$$

$$u_{yy} = \frac{u(x, y+k) - 2u(x, y) + u(x, y-k)}{k^2} + O(k^2)$$

$$u_{xy} = \frac{u(x+h, y+k) - u(x+h, y-k) - u(x-h, y+k) + u(x-h, y-k)}{4hk} + O(hk)$$

Debe notarse que las expresiones en diferencias centrales son mejores aproximaciones que las demás, dado que estamos despreciando términos que son de orden h^2 en diferencias centrales comparados con términos de orden h en las diferencias hacia adelante y hacia atrás. Por este motivo, es preferible utilizar expresiones en diferencias centrales.

Para utilizar las expresiones en diferencias finitas obtenidas en la solución de una EDP es necesario discretizar el dominio. Esto se hace introduciendo una malla particionada uniformemente. Los puntos de la malla son $x_i = ih$ y $y_j = jk$, donde $i = 0, 1, \dots, N_x$, siendo $N_x + 1$ el número de puntos en los que particionamos el dominio en la dimensión x . Si $x \in [0, L_x]$, resulta que $h = L_x/N_x$. Análogamente, $j = 0, 1, \dots, N_y$, y si $y \in [0, L_y]$ el valor de k es L_y/N_y .

Una vez discretizado el dominio, procederemos a la obtención de una aproximación a la solución de la EDP en los puntos de la malla. Consideremos la función $u(x_i, y_j) = u_{i,j}$. Las expresiones anteriores para las derivadas parciales de primer y segundo orden en el punto (x_i, y_j) se pueden establecer para cada caso. Por ejemplo, para las derivadas de primer orden en diferencias

finitas hacia adelante resultan:

$$\begin{aligned}\frac{\partial u(x_i, y_j)}{\partial x} &= u_x \Big|_{i,j} = \frac{u(x_i + h, y_j) - u(x_i, y_j)}{h} + O(h) \\ &= \frac{u(x_{i+1}, y_j) - u(x_i, y_j)}{h} + O(h) \\ &= \frac{u_{i+1,j} - u_{i,j}}{h} + O(h)\end{aligned}$$

Análogamente para u_y :

$$\frac{\partial u(x_i, y_j)}{\partial y} = u_y \Big|_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{k} + O(k)$$

Del mismo modo, las expresiones para las ecuaciones en diferencias hacia atrás:

$$\begin{aligned}\frac{\partial u(x_i, y_j)}{\partial x} &= u_x \Big|_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{h} + O(h) \\ \frac{\partial u(x_i, y_j)}{\partial y} &= u_y \Big|_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{k} + O(k)\end{aligned}$$

y para las diferencias centrales:

$$\begin{aligned}\frac{\partial u(x_i, y_j)}{\partial x} &= u_x \Big|_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2) \\ \frac{\partial u(x_i, y_j)}{\partial y} &= u_y \Big|_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2k} + O(k^2)\end{aligned}$$

Las expresiones de las derivadas de segundo orden, aproximadas por ecuaciones en diferencias finitas, calculadas sobre los puntos de la malla se obtienen de manera análoga que las de primer orden. Para las ecuaciones en diferencias hacia adelante se obtienen:

$$\begin{aligned}\frac{\partial^2 u(x_i, y_j)}{\partial x^2} &= u_{xx} \Big|_{i,j} = \frac{u(x_i + 2h, y_j) - 2u(x_i + h, y_j) + u(x_i, y_j)}{h^2} + O(h) \\ &= \frac{u(x_{i+2}, y_j) - 2u(x_{i+1}, y_j) + u(x_i, y_j)}{h^2} + O(h) \\ &= \frac{u_{i+2,j} - 2u_{i+1,j} + u_{i,j}}{h^2} + O(h)\end{aligned}$$

Del mismo modo se obtienen:

$$\begin{aligned}\frac{\partial^2 u(x_i, y_j)}{\partial y^2} &= u_{yy} \Big|_{i,j} = \frac{u_{i,j+2} - 2u_{i,j+1} + u_{i,j}}{k^2} + O(k) \\ \frac{\partial^2 u(x_i, y_j)}{\partial x \partial y} &= u_{xy} \Big|_{i,j} = \frac{u_{i+1,j+1} - u_{i+1,j} - u_{i,j+1} + u_{i,j}}{hk} + O(h, k)\end{aligned}$$

Procediendo de forma análoga se obtienen las expresiones para las diferencias hacia atrás:

$$\begin{aligned}\frac{\partial^2 u(x_i, y_j)}{\partial x^2} &= u_{xx} \Big|_{i,j} = \frac{u_{i,j} - 2u_{i-1,j} + u_{i-2,j}}{h^2} + O(h) \\ \frac{\partial^2 u(x_i, y_j)}{\partial y^2} &= u_{yy} \Big|_{i,j} = \frac{u_{i,j} - 2u_{i,j-1} + u_{i,j-2}}{k^2} + O(k) \\ \frac{\partial^2 u(x_i, y_j)}{\partial x \partial y} &= u_{xy} \Big|_{i,j} = \frac{u_{i,j} - u_{i-1,j} - u_{i,j-1} + u_{i-1,j-1}}{hk} + O(h, k)\end{aligned}$$

y finalmente las expresiones para las diferencias centrales:

$$\begin{aligned}\frac{\partial^2 u(x_i, y_j)}{\partial x^2} &= u_{xx} \Big|_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + O(h^2) \\ \frac{\partial^2 u(x_i, y_j)}{\partial y^2} &= u_{yy} \Big|_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} + O(k^2) \\ \frac{\partial^2 u(x_i, y_j)}{\partial x \partial y} &= u_{xy} \Big|_{i,j} = \frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{4hk} + O(hk)\end{aligned}$$

21.1.1. Ecuación de conducción de calor en 1D con un método explícito

Aplicaremos ahora el método de las diferencias finitas a la resolución de la EDP que representa la conducción de calor en una varilla (1D):

$$u_t = cu_{xx}, \quad x \in (0, L), \quad t \in (0, T) \quad (21.2)$$

En esta ecuación, del tipo parabólico, $u(x, t)$ es la temperatura en función de la localización en la barra y el tiempo, y $c > 0$ es la difusividad térmica del medio:

$$c = \frac{k}{c_p \rho}$$

donde k es la conductividad térmica (cuyas unidades son $\text{J m}^{-1} \text{s}^{-1} \text{K}^{-1}$), c_p es el calor específico a presión constante (en $\text{J kg}^{-1} \text{K}^{-1}$) y ρ es la densidad del material que compone la barra (en kg m^{-3}), por lo que las unidades de c resultan $\text{m}^2 \text{s}^{-1}$. Consideramos que dicha barra es unidimensional (una de sus dimensiones es mucho mayor que las otras dos) y se encuentra térmicamente aislada en su superficie lateral, intercambiando calor solo por sus extremos que se mantienen a temperatura constante.

Para obtener una solución única necesitamos condiciones de borde. Con solo una derivada temporal en el tiempo, necesitamos solamente una condición inicial, mientras que al haber una derivada de segundo orden espacial, son necesarias dos condiciones de borde. En el caso de la variable temporal, especificar la solución a $t = 0$ representa una condición inicial que podemos formalizar como $u(x, 0) = I(x)$ donde I es una función conocida. En la variable espacial las condiciones de borde se establecen en cada punto de la frontera del dominio, en los que debemos conocer el valor de u , u_x o una combinación de ambos.

Para resolver un problema simple, adoptaremos las condiciones de borde en x estableciendo el valor de u . Entonces, el sistema completo es:

$$u(x, 0) = I(x), \quad x \in [0, L] \quad (21.3)$$

$$u(0, t) = 0, \quad t > 0 \quad (21.4)$$

$$u(L, t) = 2, \quad t > 0 \quad (21.5)$$

El primer paso en el procedimiento de discretización es reemplazar el dominio $[0, L] \times [0, T]$ por un conjunto de puntos de una malla. Tomaremos los puntos equiespaciados:

$$x_i = i\Delta x, \quad i = 0, 1, \dots, N_x$$

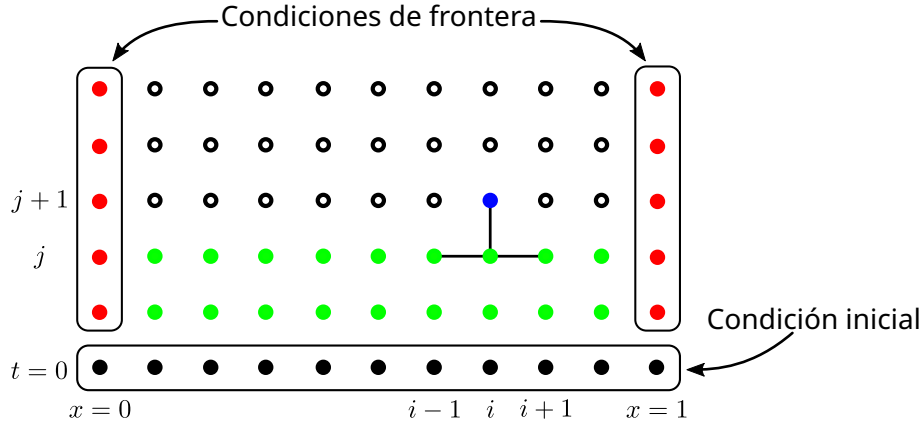


FIGURA 21.1: Esquema de solución por diferencias finitas de la ecuación de calor en 1D. Puntos negros: valores conocidos de $u(x, 0)$ (condición inicial). Puntos rojos: valores conocidos de $u(0, t)$ y $u(1, t)$ (condiciones de frontera). Puntos verdes, valores ya calculados de $u(x_i, t_j)$. Puntos blancos: valores por calcular sobre la grilla. Punto azul: valor de $u(x_i, t_{j+1})$ que se obtiene al aplicar el estencil representado por las líneas negras.

y

$$t_j = j\Delta t, \quad j = 0, 1, \dots, N_t$$

donde tomamos $\Delta x = h = L/N_x$ y $\Delta t = k = T/N_t$. Como hicimos antes, u_{ij} denota la función en los puntos de la malla que aproxima a $u(x_i, t_j)$. Dado que la ecuación (21.2) debe cumplirse en el punto (x_i, t_j) se obtiene la ecuación:

$$u_t|_{i,j} = cu_{xx}|_{i,j} \quad (21.6)$$

El próximo paso consiste en reemplazar las derivadas por las aproximaciones en diferencias finitas. Esto da origen a diversos esquemas según la representación elegida. El método computacionalmente más simple resulta de utilizar una diferencia hacia adelante en el tiempo (esquema de Euler hacia adelante) y una diferencia central en el espacio:

$$\frac{u_{i,j+1} - u_{i,j}}{k} = c \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \quad (21.7)$$

De este modo hemos convertido una EDP en ecuaciones algebraicas (o discretas), que son fáciles de resolver. Esto se hace a partir de una ecuación de recurrencia en la que asumimos que $u_{i,j}$ es conocida, mientras que $u_{i,j+1}$ es la única incógnita en (21.7):

$$u_{i,j+1} = u_i + F(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \quad (21.8)$$

donde hemos introducido al número de malla de Fourier $F = ck/h^2$. F es un número adimensional que agrupa el parámetro físico c del problema, junto con los parámetros de la discretización h y k . Las propiedades del método numérico dependen en forma crítica del valor de F .

Podemos ver una representación gráfica de la aplicación de este método en la figura 21.1. A la ecuación de recurrencia (21.8) se la puede representar como un estencil sobre la grilla, que al ir desplazando sobre puntos en los que ya conocemos el valor aproximado de la solución, nos permite obtener los valores sobre los puntos aún no calculados de la malla.

Analizaremos ahora el código que permite resolver la ecuación (21.2) con las condiciones de borde (21.3)–(21.5). En las primeras líneas del programa importamos, como es usual, los módulos necesarios para acceder a las rutinas especializadas para graficar (líneas 7–9), numpy para utilizar arrays (línea 10) y algunos módulos de scipy para disponer de diversos métodos de álgebra lineal. A continuación configuramos algunos parámetros de los gráficos que vamos a generar (líneas 18–22). Debe notarse que este bloque inicial de código (líneas 1–22) es común al resto del código analizado en esta sección.



El código completo se encuentra en [calor1D-completo.py](#)

Este programa incluye los tres métodos tratados a continuación para resolver el problema en diferencias finitas, y también la visualización de los resultados.

```

1  #!/usr/bin/env python3
2  """Programa que resuelve la ecuación de calor en 1D."""
3
4  import argparse
5  from contextlib import contextmanager
6
7  import matplotlib.pyplot as plt
8  import matplotlib.cm as cmx
9  from matplotlib import colors
10 import numpy as np
11 from scipy import (
12     linalg, # Módulo de álgebra lineal, para el método implícito
13     sparse, # Paquete para matrices ralas
14 )
15 from scipy.sparse import linalg as sparse_linalg # Solvers para matrices ralas
16
17 # Configure matplotlib
18 plt.rcParams.update({
19     'text.usetex': True,
20     'font.size': 14,
21     'axes.labelsize': 'large',
22 })
23
24
25 def I(x):
26     """Representa la curva de temperatura (en función de x) inicial (para t=0)
27
28     Entonces  $u(x, 0) = I(x)$ :
29          $u(0, 0) = 0$ 
30          $u(1, 0) = 2$ 
31     """
32     return 4 * x * (3 / 2 - x)
33
34
35 # Parámetros globales del problema
36 L = 1 # Longitud de la barra
37 T = 1 # Tiempo máximo de la solución
38 c = 2 # Difusividad térmica
39

```

```

40
41 @contextmanager
42 def graficar(x, t):
43     """Muestra la temperatura en función de x y t."""
44
45     # Semántica de tonos
46     cm = plt.get_cmap('Greens_r')
47     c_norm = colors.PowerNorm(gamma=0.25, vmin=t[0], vmax=t[-1])
48     scalar_map = cmx.ScalarMappable(norm=c_norm, cmap=cm)
49
50     # La curva en sí
51     fig, ax = plt.subplots()
52
53     def agregar_curva(point, u):
54         cval = scalar_map.to_rgba(point)
55         ax.plot(x, u, color=cval)
56
57     # Pasamos la función para agregar las curvas
58     yield agregar_curva
59
60     # Dibujamos
61     fig.colorbar(cmx.ScalarMappable(norm=c_norm, cmap=cm))
62     plt.xlabel(r'$x$')
63     plt.ylabel(r'$u(x)$')
64     plt.show()

```

En la línea 25 definimos la condición inicial. Dado el array x como argumento, devuelve un nuevo array con los valores de una función que en este caso es una parábola que abre hacia abajo, manteniendo los valores de borde establecidos. Luego, en las líneas 36–38 definimos los parámetros físicos del problema: longitud de la barra, el valor máximo de tiempo sobre el que vamos a calcular la solución, y el coeficiente de difusividad térmica.

A continuación (líneas 41–64) definimos la función `graficar(x, t)` que se ocupará de visualizar las curvas que obtengamos de la solución a intervalos de tiempo regulares. Esta función trabaja como un administrador de contexto o *context manager* que nos permite utilizarla “alrededor” de las diferentes rutinas de resolución de las ecuaciones en diferencias, evitando de este modo la repetición de código para cada caso.

```

67 def resolver_explicito():
68     """Método explícito."""
69     # Puntos de resolución del problema
70     Nx = 10 # Número de puntos de la malla en x
71     Nt = 500 # Número de puntos de la malla en t
72     h = L / Nx # Paso en x
73     k = T / Nt # Paso en t
74     F = c * k / h ** 2 # Número de malla de Fourier
75     print(f'{h=} | {k=} | {F=}')
76
77     # Malla en x y t
78     x = np.linspace(0, L, Nx + 1)
79     t = np.linspace(0, T, Nt + 1)
80
81     # Valores de u(x,t)
82     u = np.zeros(Nx + 1) # u en j
83

```

```

84  # Iteración en t (j)
85  with graficar(x, t) as agregar_curva:
86      # Condición inicial (y la dibujamos)
87      up = I(x)
88      agregar_curva(0, up)
89
90      for j in range(1, Nt):
91          # estencil (sin los bordes)
92          u[1:Nx] = up[1:Nx] + F * (up[0:Nx - 1] - 2 * up[1:Nx] + up[2:Nx + 1])
93
94          # Condiciones de borde
95          u[0] = 0
96          u[Nx] = 2
97
98          # Trazamos una solución cada 10 pasos en j
99          if not j % 10:
100             agregar_curva(j * k, u)
101
102             # Cambiamos las variable antes del próximo paso
103             up = u

```

La implementación del método explícito de solución queda delimitada por la función `resolver_explicito()` entre las líneas 67 y 103. En dicha función, definimos inicialmente los parámetros del algoritmo: los números de puntos en que dividimos la malla en x (Nx) y en t (Nt), y calculamos el paso en ambas direcciones (h y k), así como el valor del número de malla de Fourier. Estos valores se imprimen en pantalla en la línea 75, especialmente para verificar la condición de estabilidad de la solución $F \leq 0,5$.

Representamos los valores de la malla en cada variable mediante los arrays x y t , definidos en las líneas 78 y 79, mientras que los valores de la función $u(x_i, t_j)$ serán almacenados en el array u para cada valor de t_j , al cual dimensionamos con $Nx + 1$ valores nulos en la línea 82.

En la línea 85 inicializamos el administrador de contexto y a continuación almacenamos en el array up los valores de la temperatura de la barra para $t = 0$, graficando este estado inicial (líneas 87 y 88). A continuación iniciamos el bucle de la línea 90 en el que iremos iterando sobre los valores de j que representan los distintos instantes de tiempo en que dividimos la malla para la variable t . En la línea 92 calculamos la solución para j conociendo los valores de la solución para $(j - 1)$ que tenemos almacenados en up con la ecuación de recurrencia (21.8). Este cálculo se realiza para los puntos interiores de la malla en x , dado que conocemos los valores de borde que se agregan al array u en las líneas 95–96. Cada 10 pasos de j agregamos la curva que representa la solución al gráfico (líneas 99–100) y finalmente actualizamos los valores en el array up con los calculados recientemente almacenados en u para avanzar en la iteración en el cálculo del siguiente valor de j .

Como salida del programa generamos un gráfico que representa el valor de $u(x)$ para distintos valores de t , y representamos la evolución temporal con una secuencia de tonos, desde el más oscuro para $t = 0$ hasta el más claro para $t = T$. Esta semántica de tonos está definida en las líneas 46–48 del código. Primero elegimos un mapa de color ('Greens_r') entre los disponibles en Matplotlib. Luego, en la línea 47, escalamos los valores del array t en el intervalo $[0, 1]$ en forma exponencial, ya que la dinámica es rápida al inicio y luego se hace muy lenta a medida que se aproxima a la solución de equilibrio, y esta forma de escalar permite visualizar ese cambio rápido. El resultado obtenido puede verse en la figura 21.2.

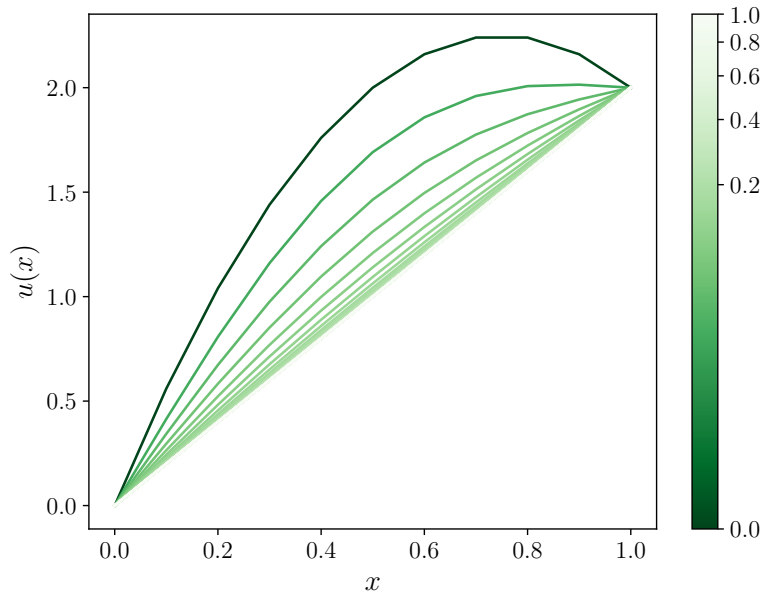


FIGURA 21.2: Solución obtenida para el problema de transmisión de calor en 1D. La escala de colores indica la evolución temporal de la solución.

En esta solución en particular, con la malla establecida obtenemos valores de $h = 0,1$ y $k = \Delta t = 0,002$, lo cual da un valor de $F \approx 0,4 < 0,5$. Las soluciones obtenidas son suaves, a costa de usar un paso temporal pequeño de modo de obtener acotado el valor de F . Invitamos al lector a modificar el código del programa para verificar que con valores de F mayores a $0,5$ se obtienen soluciones que presentan inestabilidades no físicas.

21.1.2. Ecuación de conducción de calor en 1D con un método implícito

La solución mediante el esquema de Euler hacia adelante impone la restricción $F \leq 0,5$ para obtener soluciones numéricamente estables. Si fijamos la separación espacial h de la malla, esto significa que debemos tomar un paso temporal de integración $k = \delta t = h^2/(2c)$. En procesos representados por la ecuación (21.2) (conducción de calor, difusión, etc.) la solución varía rápido al inicio, pero a medida que transcurre el tiempo el proceso se hace más lento a medida que se acerca al equilibrio (en condición estacionaria obtenemos que $u_t = 0$) y en esta situación tener un paso temporal pequeño puede resultar inconveniente.

Los métodos implícitos dan origen a un sistema acoplado de ecuaciones que se deben resolver en cada paso de tiempo, en los cuales es posible utilizar cualquier paso δt , sin obtener inestabilidades numéricas, aunque naturalmente la precisión disminuye al aumentar δt . El método implícito más simple de implementar es el esquema de Euler hacia atrás, que consiste en utilizar nuevamente una diferencia central como aproximación a la derivada parcial espacial, pero una diferencia hacia atrás en el tiempo:

$$\frac{u_{i,j} - u_{i,j-1}}{k} = c \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \quad (21.9)$$

En este caso asumimos que conocemos la solución $u_{i,j-1}$ en el tiempo $j - 1$, pero todas las cantidades en el tiempo j son ahora desconocidas, con lo cual no es posible resolver con respecto

de $u_{i,j}$ porque este valor está acoplado a los vecinos en la malla $u_{i-1,j}$ y $u_{i+1,j}$, que también son desconocidos. Esta expresión da origen a un sistema de $(N_x - 1) \times (N_t - 1)$ ecuaciones algebraicas acopladas.

Multiplicando ambos términos de la ecuación (21.9) por k y reacomodando obtenemos:

$$-Fu_{i-1,j} + (1 + 2F)u_{i,j} - Fu_{i+1,j} = u_{i,j-1} \quad (21.10)$$

para $i = 1, \dots, N_x - 1$, ya que por las condiciones de borde (21.4) y (21.5) conocemos los valores en $i = 0$ y $i = N_x$. Entonces, dado un valor de j , tenemos $N_x - 1$ ecuaciones dadas por (21.10) sobre los puntos internos de la malla en x . Una vez obtenida la solución para este sistema, es necesario iterar sobre j para obtener las sucesivas soluciones que representan la evolución temporal del sistema.

El sistema de ecuaciones acopladas dadas por la ecuación (21.10), junto con las condiciones de borde (21.4) y (21.5), pueden representarse en forma matricial:

$$\mathbb{A} \cdot \mathbf{u} = \mathbf{b} \quad (21.11)$$

donde $\mathbf{u} = (u_{1,j}, u_{1,j}, \dots, u_{N_x-1,j})$ y la matriz $\mathbb{A}$ tiene $N_x - 1$ filas y $N_x - 1$ columnas con la siguiente estructura tridiagonal:

$$\mathbb{A} = \begin{pmatrix} (1 + 2F) & -F & 0 & 0 & \dots & \dots & 0 & 0 \\ -F & (1 + 2F) & -F & 0 & \dots & \dots & 0 & 0 \\ 0 & -F & (1 + 2F) & -F & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \dots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & -F & (1 + 2F) & -F & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 & -F & (1 + 2F) & -F \\ 0 & \dots & \dots & \dots & 0 & 0 & -F & (1 + 2F) \end{pmatrix}$$

mientras que el vector $\mathbf{b}$ es:

$$\mathbf{b} = \begin{pmatrix} u_{1,j-1} + Fu_{0,j} \\ u_{2,j-1} \\ \vdots \\ u_{N_x-2,j-1} \\ u_{N_x-1,j-1} + Fu_{N_x,j} \end{pmatrix}$$

La forma de calcular los valores de la matriz $\mathbb{A}$ puede representarse desplazando el estencil mostrado en la figura 21.3 a lo largo de la fila j , y resolver este sistema nos permite obtener la solución para el tiempo correspondiente a ese valor de j . Una vez resuelto el sistema, se construye uno nuevo para la fila $j + 1$ a partir de los valores recién calculados, avanzando de este modo la solución temporal.

Debe notarse que la matriz $\mathbb{A}$ no posee elementos que cambien con el tiempo, por lo que una vez construida es la misma para todos los soluciones de la progresión en j , mientras que el vector $\mathbf{b}$ depende de los valores de u calculados en $j - 1$. Por lo tanto, en cada iteración que resuelva el sistema en su evolución temporal, debemos actualizar los valores de $\mathbf{b}$.

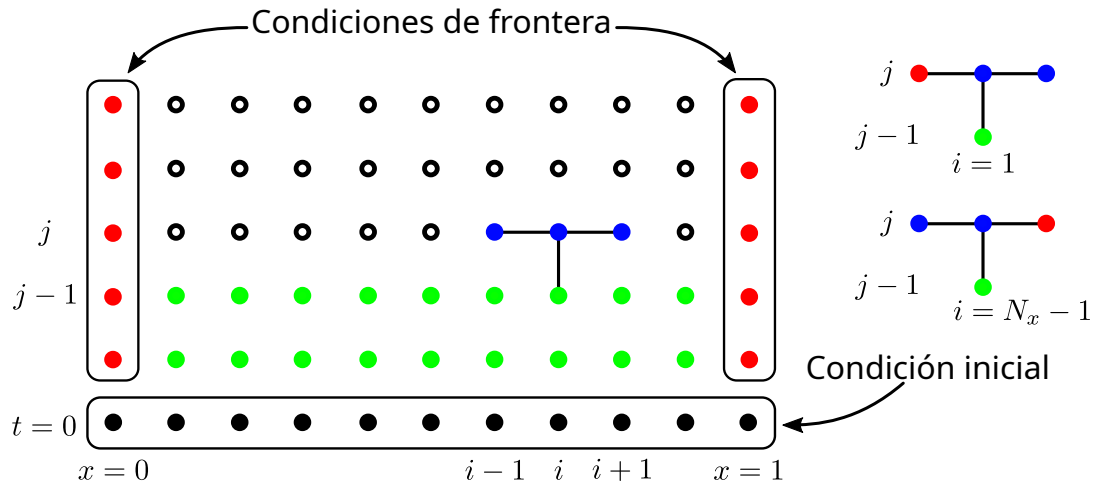


FIGURA 21.3: Esquema de solución por diferencias finitas de la ecuación de calor en 1D usando un esquema implícito. Puntos negros: valores conocidos de $u(x, 0)$ (condición inicial). Puntos rojos: valores conocidos de $u(0, t)$ y $u(1, t)$ (condiciones de frontera). Puntos verdes, valores ya calculados de $u(x_i, t_j)$. Puntos blancos: valores por calcular sobre la grilla. Puntos azules: valor de $u(x_i, t_{j+1})$ que se calculan al aplicar el estencil representado por las líneas negras. A la derecha se esquematizan las modificaciones en el estencil al evaluar las condiciones de borde.

La implementación del algoritmo para resolver la ecuación de calor en una dimensión usando el esquema implícito se encuentra integrada al código que presentamos en la subsección anterior (21.1.1), esta vez en la definición de la función `resolver_implicito()` en la línea 106. Inicialmente definimos los parámetros del algoritmo: cantidad de puntos en que dividimos la malla en cada dimensión x y t (N_x y N_t , los pasos en cada malla (h y k) y el número de malla de Fourier (F), tal como se puede ver en las líneas 109–113.

```

106 def resolver_implicito():
107     """Método implícito."""
108     # Puntos de resolución del problema
109     Nx = 100 # Número de puntos de la malla en x
110     Nt = 1000 # Número de puntos de la malla en t
111     h = L / Nx # Paso en x
112     k = T / Nt # Paso en t
113     F = c * k / h ** 2 # Número de malla de Fourier
114     print(f'{h=} | {k=} | {F=}')
115
116     # Malla en x y t
117     x = np.linspace(0, L, Nx + 1)
118     t = np.linspace(0, T, Nt + 1)
119
120     # Valores de u(x,t)
121     u = np.zeros(Nx + 1) # u en j
122
123     # Estructura de datos para el sistema lineal
124     A = np.zeros((Nx - 1, Nx - 1))
125     b = np.zeros(Nx - 1)
126     for i in range(1, Nx - 2):
127         A[i, i - 1] = -F
128         A[i, i] = 1 + 2 * F

```

```

129     A[i, i + 1] = -F
130     A[0, 0] = A[Nx - 2, Nx - 2] = 1 + 2 * F
131     A[0, 1] = A[Nx - 2, Nx - 3] = -F
132
133     with graficar(x, t) as agregar_curva:
134         # Condición inicial (y la dibujamos)
135         up = I(x)
136         agregar_curva(0, up)
137
138         # Iteración en t (j)
139         for j in range(1, Nt):
140             # Calculamos b
141             b = up[1:Nx]
142             b[Nx - 2] += F * 2 # b[0] += F * 0 (condición de borde)
143
144             # Resolvemos el sistema
145             u[1:Nx] = linalg.solve(A, b)
146             u[Nx] = 2
147
148             # Trazamos una solución cada 10 pasos en j
149             if not j % 10:
150                 agregar_curva(j * k, u)
151
152             # Actualizamos los valores de up antes del próximo paso
153             up = u

```

En las líneas siguientes (117–131) generamos las estructuras en arrays para representar la malla en las dimensiones x y t , la solución $u(x_i, t_j)$, la matriz $\mathbb{A}$ y el vector $\mathbf{b}$.

En la línea 133 inicializamos el administrador de contexto y a continuación generamos el valor inicial de la solución, que almacenamos en el array `up`, trazando la solución inicial en la línea 136.

En el bucle `for` de la línea 139 iteramos sobre los valores de j que representan la progresión temporal de la solución, calculando en cada paso de la iteración el valor del vector $\mathbf{b}$. La solución se obtiene en la línea 145 con el método `linalg.solve` que importamos previamente del módulo `scipy`, y establecemos el valor conocido en el borde en la línea 146. Completamos el bloque del `for` trazando la curva de la solución cada 10 pasos de iteración y finalmente actualizamos el valor del array `up` antes de dar el próximo paso.

Tal como hicimos al final de la sección anterior, proponemos ahora al lector resolver el sistema utilizando pasos temporales mayores (es decir con menos divisiones de la malla en la dimensión temporal) para observar que, a diferencia del esquema implícito, no se generan inestabilidades numéricas.

21.1.3. Implementación con matriz rara

Hemos visto que la matriz $\mathbb{A}$ de la sección anterior es tridiagonal, aunque no explotamos este hecho en el código de resolución con el abordaje implícito, ya que almacenamos en el array `A` una representación completa (o “densa”) que incluye muchos elementos que sabemos desde el inicio que son nulos, y el algoritmo implementado en `solve` resuelve el sistema con todos los ceros. Considerando que tenemos $N_x - 1$ incógnitas, el método `solve` realiza operaciones del orden de $(N_x - 1)^3$ y requiere el almacenamiento de $(N_x - 1)^2$ elementos.

Por otra parte, utilizando el hecho que $\mathbb{A}$ es tridiagonal y utilizando las herramientas de *software* correspondientes, las demanda de cálculo y almacenamiento son proporcionales solamente a N_x . Esto conduce a una mejora significativa en el rendimiento del código, tanto por el uso de memoria como por la velocidad de ejecución. Veremos ahora cómo resolver la ecuación de calor en 1 dimensión con un esquema implícito tomando ventaja de que $\mathbb{A}$ es tridiagonal.

La idea es utilizar por un lado una representación de $\mathbb{A}$ como matriz rala (o *sparse*, como se usa en inglés), y para ello usaremos el paquete `scipy.sparse`, de modo de almacenar solamente los elementos no nulos de $\mathbb{A}$. Por otro lado, utilizaremos algoritmos de solución que operan sobre estructuras de matrices ralas, contenidos también en `scipy.sparse`. Esta implementación está contenida en la función `resolver_ralas` del mismo código común a toda esta sección:

```

156 def resolver_ralas():
157     """Método implícito utilizando matrices ralas."""
158     # Puntos de resolución del problema
159     Nx = 100 # Número de puntos de la malla en x
160     Nt = 1000 # Número de puntos de la malla en t
161     h = L / Nx # Paso en x
162     k = T / Nt # Paso en t
163     F = c * k / h ** 2 # Número de malla de Fourier
164     print(f'{h=} | {k=} | {F=}')
165
166     # Malla en x y t
167     x = np.linspace(0, L, Nx + 1)
168     t = np.linspace(0, T, Nt + 1)
169
170     # Valores de u(x,t)
171     u = np.zeros(Nx + 1) # u en j
172
173     # Calculamos la matriz rala
174     diagonal = 1 + 2 * F
175     superior = -F
176     inferior = -F
177
178     A = sparse.diags(
179         diagonals=[diagonal, superior, inferior],
180         offsets=[0, 1, -1],
181         shape=(Nx - 1, Nx - 1),
182         format='csr')
183     print(A.todense())
184
185     with graficar(x, t) as agregar_curva:
186         # Condición inicial (y la dibujamos)
187         up = I(x)
188         agregar_curva(0, up)
189
190         # Iteración en t (j)
191         for j in range(1, Nt):
192             # Calculamos b
193             b = up[1:Nx]
194             b[Nx - 2] += F * 2 # b[0] += F * 0 (condición de borde)
195
196             # Resolvemos el sistema
197             u[1:Nx] = sparse.linalg.spsolve(A, b)
198             u[Nx] = 2
199

```

```

200     # Trazamos una solución cada 10 pasos en j
201     if not j % 10:
202         agregar_curva(j * k, u)
203
204     # Actualizamos los valores de up antes del próximo paso
205     up = u

```

Al igual que en los casos de las subsecciones anteriores, las primeras líneas de esta función determinan los parámetros del algoritmo y los arrays que almacenan la grilla y la solución (ver líneas 159–171).

En las líneas 174 a 176 calculamos los elementos no nulos de $\mathbb{A}$ que componen la diagonal principal, superior e inferior de $\mathbb{A}$, almacenándolos en las variables `diagonal`, `superior` e `inferior`, respectivamente. Con estos valores construimos la matriz rara `A` en la línea 178. Este método requiere varios argumentos: `diagonals` es una lista conteniendo los valores que definen la matriz; `offsets` establece la ubicación en la matriz (0 es la diagonal principal, 1 es la primer diagonal superior, -1 es la primer diagonal inferior); `shape` establece la cantidad de elementos en la matriz y finalmente `format` da cuenta de cómo se almacenan internamente los elementos de `A`. En particular, el método `'csr'` (formato *Compressed Sparse Row*) es el apropiado para realizar las operaciones de manipulación y multiplicación requeridas para resolver el sistema lineal de ecuaciones.¹

En la línea 183 imprimimos en pantalla la representación densa de `A`, para verificar que fue construida correctamente. Cuando establecemos el valor `Nx = 10` en la línea 159 la salida en pantalla es:

```

[[ 1.8 -0.4  0.  0.  0.  0.  0.  0.  0. ]
 [-0.4  1.8 -0.4  0.  0.  0.  0.  0.  0. ]
 [ 0.  -0.4  1.8 -0.4  0.  0.  0.  0.  0. ]
 [ 0.  0.  -0.4  1.8 -0.4  0.  0.  0.  0. ]
 [ 0.  0.  0.  -0.4  1.8 -0.4  0.  0.  0. ]
 [ 0.  0.  0.  0.  -0.4  1.8 -0.4  0.  0. ]
 [ 0.  0.  0.  0.  0.  -0.4  1.8 -0.4  0. ]
 [ 0.  0.  0.  0.  0.  0.  -0.4  1.8 -0.4 ]
 [ 0.  0.  0.  0.  0.  0.  0.  -0.4  1.8]]

```

Del mismo modo que hicimos en los métodos explícito e implícito, en la línea 185 iniciamos el administrador de contexto, seguido del cálculo del valor inicial que almacenamos en `up` y graficamos en la línea 188. El bucle `for` de la línea 191 es casi idéntico al mostrado en la subsección anterior (21.1.2), con la única diferencia de que utilizamos el método `spsolve` del submódulo `linalg` de `scipy.sparse` (ver línea 15) al que nos referimos con el nombre `sparse_linalg` para diferenciarlo del método `linalg` de `scipy`, que resuelve el sistema lineal de ecuaciones, especializado en la utilización de matrices raras.

Ejecutando el código con una malla fina en x (usando `Nx = 100` en la línea 159) obtenemos la figura 21.4, en la que se puede apreciar que con una malla tan fina la curva se aprecia muy suave, a diferencia de la figura 21.2. También en este caso hemos incrementado la resolución de la malla temporal, usando `Nt = 1000` en la línea 160.

¹ Ver entrada en Wikipedia.

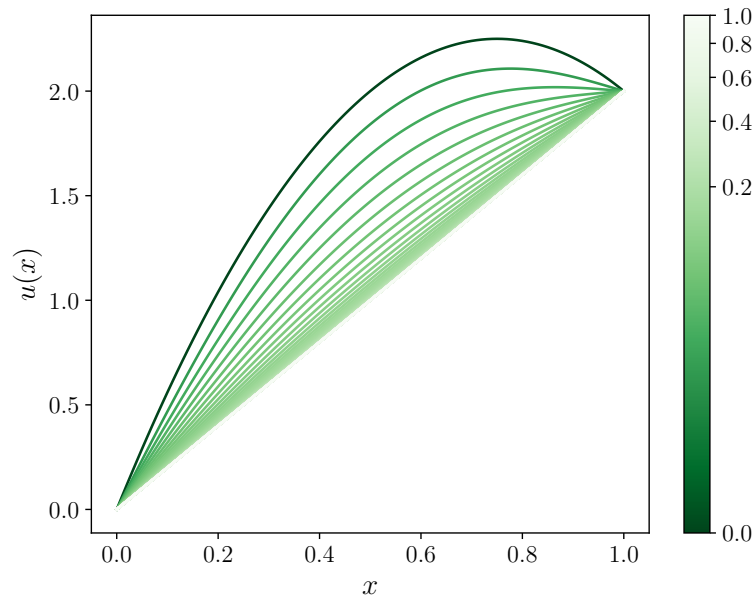


FIGURA 21.4: Solución obtenida para el problema de transmisión de calor en 1D con el método implícito y usando matrices rasas. La escala de colores indica la evolución temporal de la solución.

Podemos comparar la eficiencia de usar matrices rasas, al menos en cuanto a su velocidad de ejecución y uso de memoria, utilizando el comando

```
/usr/bin/time -v [programa ejecutable]
```

y comparando la salida al ejecutar ambos códigos en condiciones similares. Para ello comentamos las salidas a pantalla con `print` y todas las líneas relativas a la generación de gráficos. Para realizar la comparación, en ambos casos usamos una grilla temporal con $N_t = 1000$ y variamos el tamaño de la grilla espacial (N_x). En la Tabla 21.1 podemos ver la salida de `time` extrayendo solo los valores de `System time` (tiempo de ejecución del sistema) y `Maximum resident set size` que es el máximo tamaño que ocupa la ejecución del programa en memoria. Los valores corresponden a la ejecución de ambos códigos en un procesador Intel® Core™ i7-2600.

Vemos que el uso de matrices rasas reduce dramáticamente el tiempo de ejecución del programa, debido a la eficiencia en el manejo de los pocos valores que contienen las estructuras de datos de las matrices rasas en comparación con las densas. Por otra parte, la diferencia en memoria no es muy grande para sistemas pequeños, pero la demanda de almacenamiento se reduce a casi el 20 % para la malla más grande analizada.

21.2. Método de elementos finitos

El método de diferencias finitas descrito en la sección anterior tiene más interés académico que práctico, solo problemas con geometrías simples pueden ser abordados eficientemente de esta forma. Para problemas reales, los métodos de elementos finitos (MEF) o volúmenes finitos (MVF) son los que se usan comúnmente en ciencias e ingenierías.

Aunque ambos métodos (MEF y MVF) son aplicables a cualquier sistema de ecuaciones en derivadas parciales multi-física, el uso de MEF está más consolidado en problemas de análisis

Nx	Tiempo (s)			Memoria (kb)		
	Densas	Ralas	%	Densas	Ralas	%
500	20.44	0.43	2.10	82548	66144	80.13
1000	88.15	0.52	0.59	97212	66260	68.16
2000	343.49	0.59	0.17	145908	67692	46.39
4000	651.98	0.82	0.13	342296	68332	19.96

TABLA 21.1: Rendimientos con diferentes tamaños de malla. Se comparan los tiempos de ejecución y el máximo uso de memoria. Los porcentajes se calculan como fracción de los valores correspondientes a matrices densas.

estructural, elasticidad y transferencia de calor, mientras que el MVF utiliza el teorema de Gauss para simplificar ecuaciones integrales siempre que sea posible, y tiene amplia difusión en problemas de mecánica de fluidos para resolver la ecuación de Navier-Stokes. Para problemas vinculados al electromagnetismo (ecuaciones de Maxwell), campos gravitacionales, etc., ambos métodos presentan formulaciones útiles. Una descripción detallada de estos métodos excede el alcance de este libro, en esta sección daremos solo un abordaje introductorio al MEF, resolviendo un problema simple con esta técnica y utilizando un *software* específico que implementa los algoritmos necesarios.

Existen numerosos entornos de trabajo, o *frameworks*, para resolver EDP mediante elementos finitos. Entre ellos podemos mencionar [scikit-fem](#), [SfePy](#), [polyfem](#), [femPy](#) y [FEniCS](#). Estos *frameworks* incorporan diversas herramientas para resolver una variedad de problemas expresados mediante EDP, especialmente métodos eficientes para resolver los grandes sistemas de ecuaciones usando matrices ralas. El ejemplo que desarrollaremos a continuación lo implementaremos con FEniCS, ya que provee una práctica interfaz en Python a un poderoso código que implementa el MEF (utilizaremos el tutorial de FEniCS[39] como guía en esta sección).

21.2.1. Ecuación de Poisson

Tal vez el problema más simple expresado en EDP sea resolver la ecuación de Poisson, que aparece en muchos problemas físicos como la conducción de calor, la difusión de sustancias, electrostática o flujos de fluidos inmiscibles, entre otros. Dicha ecuación se expresa:

$$-\nabla^2 u = f \quad \text{en } \Omega \quad (21.12)$$

$$u = u_D \quad \text{en } \partial\Omega \quad (21.13)$$

donde $u = u(\mathbf{x})$ es la función desconocida mientras que $f = f(\mathbf{x})$ está determinada. ∇^2 es el operador de Laplace (denotado a menudo Δ), Ω es el dominio espacial y $\partial\Omega$ su frontera. No tenemos en esta ecuación una derivada temporal, por lo que estamos representando una condición estacionaria, a diferencia del caso abordado en la sección anterior.

En un espacio cartesiano bidimensional con coordenadas x y y , la ecuación (21.12) se escribe:

$$-\frac{\partial^2 u}{\partial^2 x} - \frac{\partial^2 u}{\partial^2 y} = f(x, y) \quad (21.14)$$

en la cual ahora u es función de dos variables, $u(x, y)$, definida sobre el dominio bidimensional Ω .

Resolver un problema con condiciones de borde como la ecuación de Poisson (21.14) con las condiciones de borde dadas por la ecuación (21.13) consiste en transitar las siguientes etapas:

1. Identificar el dominio computacional Ω , la EDP, sus condiciones de frontera y los términos fuente f .
2. Reformular la EDP como un problema variacional de elementos finitos.
3. Escribir un código en Python que defina el dominio computacional, el problema variacional, las condiciones de contorno.
4. Utilizar las rutinas de FEniCS para resolver el problema con condiciones de contorno y, opcionalmente, extender el programa para calcular cantidades derivadas como flujos y promedios, y visualizar los resultados.

21.2.2. Formulación variacional

La idea central del método de elementos finitos consiste en representar el dominio sobre el que está definido la EDP con un conjunto finito de regiones discretas, o “elementos”, y aproximar la función desconocida por medio de una combinación lineal de funciones base con soporte local sobre estos elementos (o sobre un grupo de elementos vecinos).

El núcleo del procedimiento para convertir una EDP en un problema variacional es el de multiplicar la EDP por una función v , integrar esa función sobre el dominio Ω mediante integración por partes de los términos con derivadas de segundo orden. A la función v utilizada para multiplicar la EDP se la denomina función de prueba o *test*. Por otra parte, la función desconocida u que se va a aproximar se denomina función ensayo o *trial*. Mantendremos en adelante los nombres en inglés debido a que es como se las llama en el código Python de FEniCS.

Multiplicamos entonces la ecuación de Poisson por la función *test* v e integramos:

$$-\int_{\Omega} (\nabla^2 u) v \, dx = \int_{\Omega} f v \, dx \quad (21.15)$$

Luego aplicamos integración por partes al integrando con derivadas de segundo orden:

$$-\int_{\Omega} (\nabla^2 u) v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds \quad (21.16)$$

donde $\partial u / \partial n$ es la derivada de u en la dirección normal a la frontera con sentido hacia afuera del dominio. Un requerimiento que debe cumplir la función v es que se anule en las partes de la frontera donde se conoce u , lo que para este problema implica que $v = 0$ en toda la frontera $\partial\Omega$. Esta condición anula el segundo término del lado derecho de la ecuación (21.16). De las ecuaciones (21.15) y (21.16) obtenemos:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad (21.17)$$

Esta ecuación es válida para toda v en algún espacio de funciones $\hat{V}$, mientras que la función *trial* u pertenece al (posiblemente diferente) espacio de funciones V . La ecuación (21.17) representa la “formulación débil” del problema con condiciones de borde original (o “formulación fuerte”) dado por las ecuaciones (21.12) – (21.13).

Para resolver numéricamente la ecuación de Poisson necesitamos transformar el problema variacional continuo (21.17) en un problema variacional discreto. Esto se consigue introduciendo espacios de dimensión finita para las funciones *test* y *trial*, que denotaremos como $\hat{V}_h \subset \hat{V}$ y $V_h \subset V$, respectivamente. El problema variacional discreto puede enunciarse entonces como: encontrar $u_h \in V_h \subset V$ tal que:

$$\int_{\Omega} \nabla u_h \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad \forall v \in \hat{V}_h \subset \hat{V} \quad (21.18)$$

La elección de V_h y $\hat{V}_h$ dependen del tipo de elementos finitos que queremos usar en nuestro problema, y para la mayoría de los casos de interés estos espacios son bien conocidos. Por ejemplo, si elegimos elementos lineales triangulares con tres nodos, V_h y $\hat{V}_h$ son los espacios de todas las funciones lineales por tramos sobre una malla de triángulos, en las que las funciones de $\hat{V}_h$ son cero en la frontera y las de V_h son iguales a u_D . Para ilustrar estas funciones en una dimensión, la figura 21.5 muestra al intervalo $[0, 6]$ discretizado en cinco puntos interiores, y una función continua (línea negra) se aproxima como una función lineal a tramos (en líneas de puntos verde) mediante una combinación lineal de funciones triangulares base (en líneas azules).

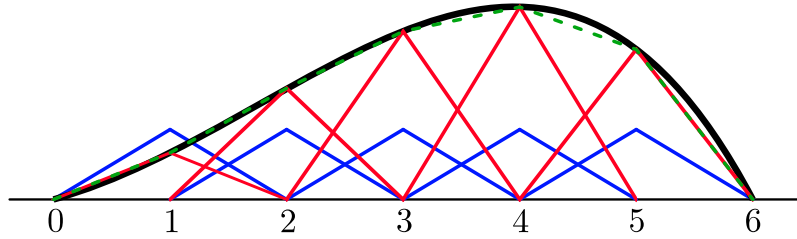


FIGURA 21.5: Ejemplo de aproximación de una función (línea negra) mediante funciones base lineales por tramos (líneas azules).

Es usual en la literatura matemática denotar a u_h como la solución del problema discreto y u la del problema continuo. Sin embargo utilizaremos u para representar la solución discreta para mantener la nomenclatura de FEniCS. La secuencia es, entonces, representar con u a la función desconocida de la EDP, derivar la formulación variacional con $u \in V$ y $v \in \hat{V}$, y luego discretizar el problema eligiendo espacios de dimensión finita para V y $\hat{V}$, con lo cual u se convierte en una función de elemento finito discreto. En la práctica esto ocurre a partir de crear una malla para discretizar el dominio Ω y la elección del tipo de elemento, y hacer entonces corresponder V a esta malla y elemento elegido. Dependiendo de si V es de dimensión finita o infinita, u será la solución aproximada o la exacta.

Resulta conveniente aquí introducir una notación unificada para representar la formulación débil del problema (21.18):

$$a(u, v) = L(v) \quad (21.19)$$

Para la solución de la ecuación de Poisson, las expresiones explícitas en la expresión anterior

son

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \quad (21.20)$$

$$L(v) = \int_{\Omega} f v \, dx \quad (21.21)$$

En la literatura matemática, $a(u, v)$ se conoce como una “forma bilineal” ($V \times \hat{V} \mapsto \mathbb{R}$), mientras que $L(v)$ es una “forma lineal” ($\hat{V} \mapsto \mathbb{R}$). De este modo, para cada problema lineal que queremos resolver tenemos que identificar los términos con la función incógnita u y agruparlos en $a(u, v)$, y del mismo modo agrupar los términos que dependen solo de las funciones conocidas en $L(v)$. Estas fórmulas para a y L se codifican directamente en el programa.

En síntesis, antes de escribir el código de Python para que FEniCS resuelva una EDP, debemos realizar dos pasos:

1. Convertir la EDP en un problema variacional discreto, esto es, encontrar $u \in V$ tal que:

$$a(u, v) = L(v) \quad \forall v \in \hat{V} \quad (21.22)$$

2. Elegir los espacios V y $\hat{V}$, lo que significa especificar la malla y el tipo de elementos finitos.

21.2.3. Implementación en FEniCS

Veamos ahora cómo resolvemos la ecuación de Poisson en un problema concreto. Queremos determinar la condición estacionaria de la temperatura de una placa (es decir, en dos dimensiones) cuyos bordes se mantienen a temperaturas fijas. Elegiremos un problema sencillo, del cual conocemos la solución exacta, para poder verificar la precisión de la solución. Para ello, vamos a elegir la solución exacta:

$$u_e(x, y) = 1 + x + 2y^2 \quad (21.23)$$

en un dominio en dos dimensiones. Si reemplazamos u_e en la ecuación de Poisson (21.14), vemos que $u_e(x, y)$ es solución si:

$$f(x, y) = -4, \quad u_D(x, y) = u_e(x, y) = 1 + x + 2y^2 \quad (21.24)$$

independientemente de la forma del dominio. Elegiremos por simplicidad un dominio cuadrado $\Omega = [0, 1] \times [0, 1]$. El MEF, sobre un dominio rectangular uniformemente particionado en elementos triangulares lineales, reproduce exactamente un polinomio de segundo grado en los vértices de las celdas, independientemente del tamaño de los elementos. De este modo, la solución aproximada obtenida deberá ser igual a la exacta (con la precisión de la representación de punto flotante) en los nodos.

El código para resolver la ecuación de Poisson en 2D usando FEniCS, dadas las elecciones de u_D , f y Ω es el siguiente:

```
1 #!/usr/bin/env python3
2 """
3 Solución de la ecuación de Poisson en 2D con condiciones de Dirichlet
4 para resolver un problema cuya solución es conocida.
```

```

5
6     -Δ(u) = f    en un cuadrado unitario
7         θ = u_D en la frontera
8
9     u_D = 1 + x + 2 y^2
10    f = -4
11    """
12
13    import fenics as fe
14    import numpy as np
15
16    # Creamos la malla y definimos el espacio de funciones
17    mesh = fe.UnitSquareMesh(10, 10)
18    V = fe.FunctionSpace(mesh, 'P', 1)
19
20    # Definimos las condiciones de borde
21    u_D = fe.Expression("1 + x[0] + 2 * x[1]*x[1]", degree=2)
22
23
24    def boundary(x, on_boundary):
25        return on_boundary
26
27
28    bc = fe.DirichletBC(V, u_D, boundary)
29
30    # Definimos el problema variacional
31    u = fe.TrialFunction(V)
32    v = fe.TestFunction(V)
33    f = fe.Constant(-4.0)
34    a = fe.dot(fe.grad(u), fe.grad(v)) * fe.dx
35    L = f * v * fe.dx
36
37    # Calculamos la solución
38    u = fe.Function(V)
39    fe.solve(a == L, u, bc)
40
41    # Guardamos la solución en un archivo con formato VTK
42    vtk_file = fe.File('poisson.pvd')
43    vtk_file << u
44
45    # Calculamos el error en la norma L2
46    error_L2 = fe.errornorm(u_D, u, 'L2')
47
48    # Calculamos los errores en los vértices
49    vertex_values_u_D = u_D.compute_vertex_values(mesh)
50    vertex_values_u = u.compute_vertex_values(mesh)
51    error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))
52
53    print(f'    Error L2 = {error_L2}')
54    print(f'Error máximo = {error_max}')

```

Iniciamos el programa importando todo lo que está definido en el módulo `fenics`². En particular nos permite usar las funciones `UnitSquareMesh`, `FunctionSpace`, `Expression` y demás defi-

² Este módulo de Python no está disponible en PyPI, hay que instalarlo en el sistema, [aquí hay instrucciones](#) para hacerlo.

niones. También importamos numpy para realizar alguna operación sobre arrays.

Lo primero que hacemos es crear una malla que almacenamos en la variable `mesh`, en la línea 17. Para eso usamos `UnitSquareMesh` que define una malla uniforme sobre el cuadrado unitario $[0, 1] \times [0, 1]$. Esta malla está compuesta de celdas que en 2D son triángulos. Los parámetros 10 y 10 especifican que el cuadrado debe dividirse primero en 10 divisiones en cada dimensión (lo que genera 100 cuadrados de lado 0,1 que cubren todo el cuadrado unitario). Cada uno de estos cuadrados interiores se divide en un par de triángulos (incluso se puede definir en qué dirección trazar la diagonal que divide el cuadrado en dos triángulos). En total se generan $2 \times 10 \times 10 = 200$ triángulos, con un total de $(10+1)(10+1) = 121$ vértices o nodos. FEniCS dispone de rutinas que realizan mallados sobre geometrías simples en una, dos y tres dimensiones, pero en problemas con geometrías complejas es necesario utilizar programas especializados en esta tarea (podemos destacar [Gmsh](#) como un excelente generador de mallas). En estos casos, las mallas generadas por programas externos pueden importarse en FEniCS para ser usadas en los cálculos de MEF.

A continuación creamos el espacio V de funciones de elemento finito, almacenado en la variable `V` de la línea 18. Este espacio se crea sobre `mesh`, que es el primer argumento. El segundo, `'P'`, declara la utilización de la familia estándar Lagrange de elementos (también se puede usar `'Lagrange'` para esta familia. FEniCS soporta todos los elementos de la [Tabla Periódica de los Elementos Finitos](#) [40] con su correspondiente notación. El tercer argumento (1) especifica el grado del elemento finito. En este caso, el elemento lineal de Lagrange estándar P_1 es un triángulo con nodos en los tres vértices (nombrado en algunas referencias como “triángulo lineal”).

Luego tenemos que definir las condiciones de borde $u = u_D$ en $\partial\Omega$. Esto se hace en la línea 26 usando la función `DirichletBC`. En los argumentos de esta función debe especificarse el espacio de funciones (`V`), la expresión matemática que define a u_D y la frontera donde ésta función aplica.

La definición de la función u_D se hace en la línea 21. La cadena que es el primer argumento de esta función debe ser escrita con la sintaxis de C++, ya que la evaluación de las expresiones se realiza por medio de código especializado en ese lenguaje (todas las funciones que se pueden incluir se encuentran definidas en el archivo de cabecera `cmath` de C++, como `sin`, `exp`, `log`, etc.), y en este caso depende de las variables `x[0]` y `x[1]` correspondientes a las coordenadas x y y . Como tercer argumento usamos un 2, lo que significa que u_D puede representar la solución cuadrática exacta de nuestro problema.

Para finalizar la definición de `bc` debemos indicar en el tercer argumento de la línea 26 la región $\partial\Omega$ que define el borde del dominio. Para eso definimos la función `boundary` en la línea 23, que devuelve `True` si x está en el borde del dominio. El argumento `on_boundary` es `True` si x está en el borde físico del dominio, por lo que simplemente podemos devolver ese valor en la función `boundary`. Esta función será llamada para cada punto discreto de la malla, lo que significa que podemos definir bordes donde u es conocida incluso dentro de la malla, si es necesario.

Con estas definiciones previas podemos empezar a resolver el problema. En las líneas 29 y 30 declaramos las variables u y v como funciones *trial* y *test* definidas sobre el espacio V . En la matemática del método distinguimos los espacios V y $\hat{V}$, pero en este problema la única diferencia ocurre en las condiciones de borde. FEniCS no especifica estas condiciones como parte de un espacio de funciones, por lo que es suficiente trabajar con un espacio común V para las funciones *test* y *trial*.

Antes de definir las formas bilineal y lineal del problema es necesario definir la función f . Podemos hacerlo del mismo modo que la variable `u_D` de la línea 21, pero como en este caso la función es constante, se representa en forma más eficiente usando la función `Constant`. Con

todas las definiciones en su lugar, podemos ahora especificar $a(u, v)$ y $L(v)$ tal como lo hacemos en las líneas 32 y 33 del programa. Podemos destacar aquí la similitud con las contrapartes matemáticas dadas por las ecuaciones (21.20) y (21.21). Esta característica de poder expresar la formulación variacional con un código Python muy similar constituye una de las fortalezas de FEniCS, ya que hace simple representar problemas en EDP más complejos que el de este ejemplo.

En las líneas 36 y 37 se resuelve el problema. Notemos que primero definimos la variable u como `TrialFunction` y la usamos para representar la función desconocida de la forma bilineal a . Ahora la redefinimos como un objeto `Function` que representa la solución. Si bien son objetos diferentes, ambos representan el mismo objeto matemático por lo que es natural usar la misma variable.

Utilizaremos en este ejemplo un programa externo para visualizar la solución: [ParaView](#). Esta es una herramienta poderosa para la representación gráfica de campos escalares y vectoriales, incluyendo los calculados por FEniCS. Para ello es necesario guardar la información de estos campos, contenida en la variable u , en el popular [formato VTK](#), lo que hacemos en las líneas 40 (definiendo el nombre del archivo que escribiremos en disco) y 41 (escribiendo en el archivo el contenido de u). Debe notarse que la forma de esta operación de escritura no es muy “pythónica”, ya que hace uso del operador de inserción “<<” propio de C++.

Finalmente, dado que empezamos este ejemplo a partir de la solución exacta de la ecuación de Poisson, podemos calcular el error obtenido con el MEF. Este cálculo lo realizaremos de dos maneras: mediante la norma L^2 y el error en los vértices de la malla.

Para el primer caso, almacenamos en la variable `error_L2` el resultado de la función `errornorm`, que calcula la norma L^2 definida como:

$$E = \sqrt{\int_{\Omega} (u_D - u)^2 dx} \quad (21.25)$$

Dado que la solución exacta es cuadrática, mientras que la solución por MEF es lineal por tramos, el error será distinto de cero.

Para el segundo caso, debemos recordar que el error en todos los vértices de la malla debería ser nulo con precisión de punto flotante de la máquina. Para calcularlo evaluamos la función exacta y la obtenida por FEniCS en todos los vértices de la malla (usando el método `compute_vertex`, obtenemos el valor absoluto de la diferencia en cada punto, y luego buscamos el máximo de estos valores. Estas operaciones se realizan en las líneas 47 – 49.

Ejecutando el programa obtenemos:

```
> ./fem.py
Solving linear variational problem.
Error L2 = 0.003651483716703487
Error máximo = 6.661338147750939e-15
```

Podemos ver que, efectivamente, el máximo error en los nodos es del orden del error de máquina.

En la figura 21.6 podemos ver la solución obtenida por MEF usando FEniCS. Si la ecuación de Poisson representa un problema térmico, los valores de $u(x, y)$ muestran la distribución de temperaturas en la placa cuadrada. En la vista tridimensional puede apreciarse que se cumplen

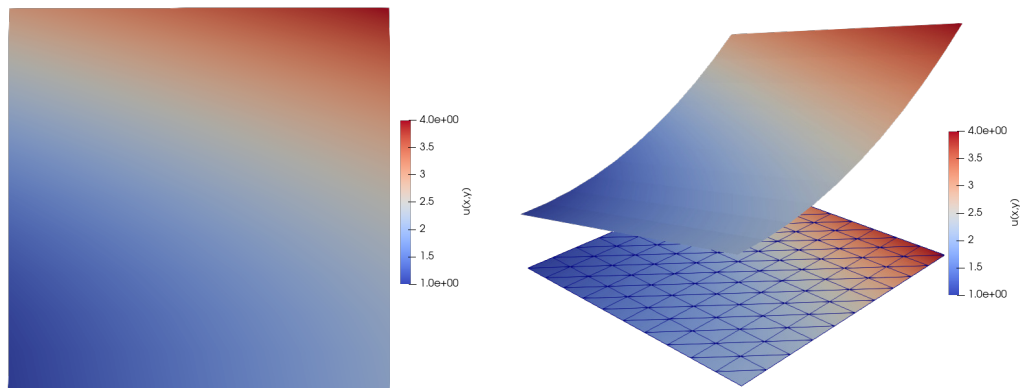


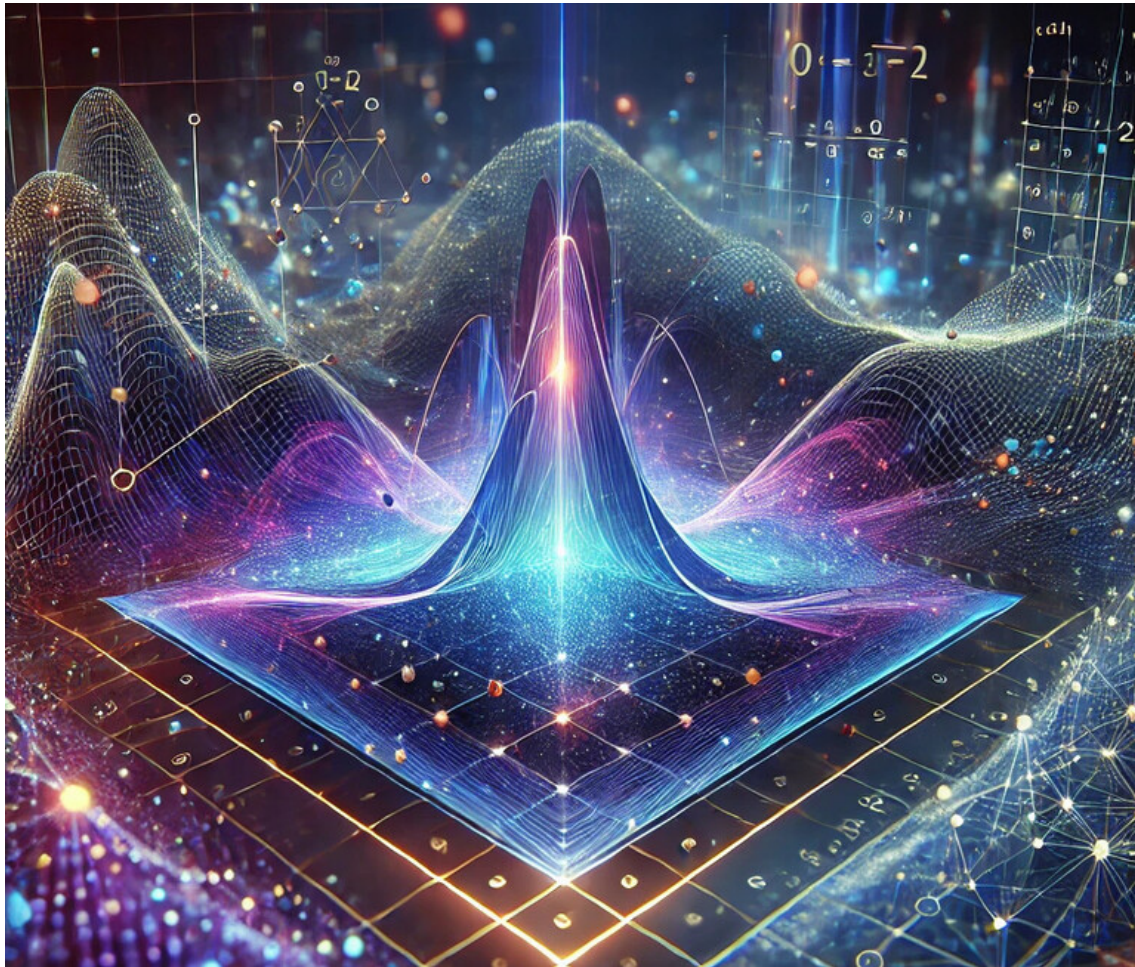
FIGURA 21.6: Visualización de la solución del problema de Poisson con condiciones de borde. A la izquierda una vista superior de la placa con la distribución de valores de $u(x, y)$. A la derecha una representación tridimensional de la solución, con la visualización de la malla generada por FEniCS.

las condiciones de borde, al presentar una variación lineal de la temperatura en los lados $(x, 0)$ y $(x, 1)$, un una variación cuadrática en los lados $(y, 0)$ y $(y, 1)$.

El ejemplo que mostramos aquí es solo una introducción básica a la resolución de EDP usando el método de elementos finitos con el *framework* FEniCS. Queremos destacar que este software ofrece la posibilidad de representar un problema enunciado en términos de EDP de una forma muy cercana a la formulación matemática. Por supuesto que los problemas reales son significativamente más complejos que el que mostramos aquí, incluyendo geometrías complejas, multifísica, diferentes condiciones de borde, etc. Sugerimos al lector interesado las lecturas de la sección siguiente.

21.3. Lecturas recomendadas

- A.N. Tijonov y A.A. Samarsky. *Ecuaciones de la Física Matemática*. Editorial Mir, Moscú, 1983.
- Hans Petter Langtangen y Svein Linge. *Finite Difference Computing with PDEs*. Springer International Publishing, 2017. DOI: [10.1007/978-3-319-55456-3](https://doi.org/10.1007/978-3-319-55456-3). URL: <https://doi.org/10.1007/978-3-319-55456-3>.
- M.G. Larson y F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2013.
- Hans Petter Langtangen y Anders Logg. *Solving PDEs in Python*. Springer, 2017. DOI: [10.1007/978-3-319-52462-7](https://doi.org/10.1007/978-3-319-52462-7).
- Anders Logg, Kent-Andre Mardal, Garth N. Wells et al. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. por Anders Logg, Kent-Andre Mardal y Garth N. Wells. Springer, 2012. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8).



22 | Optimización

Los problemas en los cuales se pretende encontrar la mejor solución aparecen frecuentemente en casi todos los ámbitos de la ciencia, la ingeniería, la tecnología y por supuesto, también en la vida cotidiana. Diariamente, para ir desde nuestro hogar al trabajo, tenemos que decidir si queremos ir por el camino más corto, más rápido, menos costoso, menos transitado, más lindo, etc. A menudo, a este requerimiento de obtener la “mejor solución” sobre la base de un criterio dado, le agregamos algunas restricciones adicionales que se deben cumplir también: tenemos que llegar a destino antes de una determinada hora, cruzar la menor cantidad de puentes, que nuestra ruta incluya un lugar donde tomar un desayuno.

En general, la optimización es un proceso por el cual queremos encontrar y seleccionar el elemento óptimo de un conjunto de candidatos posibles. Utilizando el formalismo matemático, este problema se expresa como la determinación del valor extremo de una función en un dominio dado. Tal valor extremo, o valor óptimo, puede referirse al máximo o mínimo de una función, dependiendo de la aplicación y el problema específico. En este contexto, un problema de optimi-

Módulo	Versión
Matplotlib	3.9.2
SciPy	1.14.1
SymPy	1.13.3
Código disponible	

zación tiene dos ingredientes:

- Una **función objetivo** que debe ser maximizada o minimizada. Por ejemplo, minimizar la distancia en el recorrido de un vehículo de entrega de productos.
- Un **conjunto de restricciones** (posiblemente vacío) que se deben satisfacer. Por ejemplo, no cruzar más de dos veces un río.

Sin perder generalidad podemos considerar la optimización como un problema de minimización¹. La forma estándar de un problema de optimización continua es:

$$\begin{aligned} &\text{minimizar} && f(\mathbf{x}) \\ &\text{sujeto a} && g_i(\mathbf{x}) \leq b_i, \quad i = 1, \dots, m \end{aligned} \tag{22.1}$$

Aquí, el vector $\mathbf{x} = (x_1, \dots, x_n)$ es la variable de optimización del problema (también denominadas usualmente “variables de decisión”), la función $f : \mathbb{R}^n \rightarrow \mathbb{R}$ es la función objetivo, las funciones $g_i : \mathbb{R}^n \rightarrow \mathbb{R}$, $i = 1, \dots, m$ son las funciones de restricción, y las constantes $b_1, \dots, b_m$ son los límites de las restricciones. Un vector $\mathbf{x}^*$ se denomina óptimo, o solución del problema (22.1), si tiene el menor valor objetivo entre todos los vectores que satisfacen las restricciones: para todo $\mathbf{z}$ con $g_1(\mathbf{z}) \leq b_1, \dots, g_m(\mathbf{z}) \leq b_m$ tenemos que $f(\mathbf{x}^*) \leq f(\mathbf{z})$.

Las formas particulares que presentan las funciones objetivo y de restricción caracterizan familias o clases de problemas de optimización muy variados. Un abordaje matemático general es difícil de tratar y no hay métodos eficientes para resolver completamente problemas genéricos de optimización. Sin embargo, existen métodos muy eficientes para muchos casos de importancia teórica o práctica, en los cuales resulta valioso conocer todo lo posible sobre las funciones objetivo y de restricciones para resolver con éxito el problema.

Los problemas de optimización se clasifican según las propiedades de $f(\mathbf{x})$ y $g_i(\mathbf{x})$. En primer lugar, si x es un escalar ($x \in \mathbb{R}$) el problema es unidimensional o univariado, mientras que si $\mathbf{x}$ es un vector ($\mathbf{x} \in \mathbb{R}^n$), el problema es multidimensional o multivariado. Naturalmente, cuanto mayor es n , más difícil es resolver el problema de optimización, y requiere más potencia de cálculo.

Cuando la función objetivo y las funciones de restricción son lineales, es decir, satisfacen que

$$f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$$

se suele llamar al problema de optimización (22.1) como **programación lineal**². Si la función objetivo o las de restricción son no lineales, el problema de optimización se denomina **programación no lineal**. Cuando se requiere que el vector $\mathbf{x}$ solo tome valores enteros, se trata de un problema de **programación entera**, mientras que si la restricción consiste en que solo tome dos valores (por ejemplo, 0 y 1), estamos frente a un problema de **programación binaria**. En relación con las funciones de restricción g_i , los problemas de optimización pueden ser con o sin restricciones, y en este último caso dichas restricciones pueden expresarse en forma de igualdades o desigualdades.

Los problemas no lineales presentan una complejidad mayor que los lineales, debido a que tienen una variedad muy amplia de posibles comportamientos. En general, estos problemas pueden tener mínimos globales y locales, lo que hace difícil identificar al mínimo global.

¹ Dado que es válido: $f(x_0) \geq f(x) \Leftrightarrow -f(x_0) \leq -f(x)$ es suficiente presentar el formalismo para problemas de minimización.

² Pese al nombre, no tiene que ver con programación de computadoras.

22.1. Optimización unidimensional

La forma más obvia de encontrar el mínimo de una función es diferenciarla y encontrar el valor de la variable independiente que hace la derivada igual a cero. Sin embargo, en muchas situaciones no es práctico encontrar la derivada directamente. El submódulo `optimize` de SciPy provee dos métodos para encontrar el mínimo de una función unidimensional: `brent` y `golden`, aunque en la documentación de SciPy sugieren no utilizar este último, incluido solo con fines académicos. Estos métodos solo utilizan evaluaciones de la función a optimizar, sin requerir del cálculo de la derivada, y se aproximan al valor óptimo mediante técnicas de *bracketing* o acotamiento. Para funciones con un único mínimo en un intervalo dado, estas metodologías de *bracketing* garantizan la convergencia al punto óptimo, pero esto no está asegurado para funciones más complejas.

Como ejemplo de optimización unidimensional consideraremos el ejemplo clásico de minimizar el área de un cilindro con un volumen V_0 dado. Las variables a considerar son el radio r y la altura h del cilindro, y la función objetivo es $f(r, h) = 2\pi r^2 + 2\pi r h$, con la condición de restricción $g(r, h) = \pi r^2 h - V_0 = 0$. El problema así planteado presenta dos variables, pero a partir de la restricción podemos reducirlo a una sola dimensión usando $h = V_0/\pi r^2$ y substituyendo en $f(r, h)$, lo que conduce a minimizar $f(r) = 2\pi r^2 + 2V_0/r$.

Comenzaremos solucionando el problema en forma analítica aprovechando las capacidades de cálculo simbólico de SymPy. En la primera celda declaramos las variables y ecuaciones que representan el problema, obteniendo el valor exacto de r que minimiza el área para un volumen V_0 dado:

CELDA 1 DE OPTIMIZACION.IPYNB

```

from sympy import symbols, pi, solve

r, h, V_0 = symbols("r h V_0")
Area = 2 * pi * r**2 + 2 * pi * r * h
Volumen = pi * r**2 * h
h_r = solve(Volumen - V_0, h)[0] # hallamos h en función de r
Area_r = Area.subs({h: h_r})      # expresamos el área como función de r
r_sol = solve(Area_r.diff(r), r)[0] # solución de r con significado físico
r_sol

```

$$\frac{2^{\frac{2}{3}} \sqrt[3]{V_0}}{2 \sqrt[3]{\pi}}$$

Podemos verificar que es un mínimo evaluando las derivadas primera y segunda de `Area_r` en el valor obtenido para `r_sol`:

CELDA 02

```

print(Area_r.diff(r, 1).subs(r, r_sol), Area_r.diff(r, 2).subs(r, r_sol))

```

0 12*pi

dando un valor positivo (12π) para la derivada segunda confirmando que `r_sol` es un mínimo. El área que corresponde a esta solución es:

CELDA 03
Area_r.subs(r, r_sol)
$3\sqrt[3]{2}\sqrt[3]{\pi}V_0^{\frac{2}{3}}$

Utilizaremos el valor exacto del mínimo obtenido para r (r_{sol}) en la celda 1 para compararlo con el que obtendremos mediante una optimización numérica. Para ello evaluamos r_{sol} para el caso en que $V_0 = 1$:

CELDA 04
<code>r_sol.subs({V_0: 1}).evalf()</code>
0,541926070139289

Para problemas simples la resolución analítica es posible, pero en problemas reales es común tener que recurrir a métodos numéricos (por ejemplo cuando no es posible resolver la ecuación $f'(x) = 0$). Para abordar este problema con el submódulo `optimize` de SciPy, primero debemos definir la función `f()` que implementa la función objetivo (en este caso asumimos que $V_0 = 1$), y luego pasamos esta función como argumento de `minimize_scalar` para realizar la minimización numérica, junto con un intervalo de *bracketing* y la selección del método de optimización. El método de Brent utiliza una combinación de interpolación parabólica y el método de sección de oro. En la celda siguiente realizamos la optimización de nuestro problema con este método:

CELDA 05
<pre>import numpy as np from scipy.optimize import minimize_scalar def f(r): return 2 * np.pi * r**2 + 2 / r solucion = minimize_scalar(f, bracket=[0.01, 1], method='brent') print(solucion) print(f"r_min = {solucion.x}")</pre>
<pre>message: Optimization terminated successfully; The returned value satisfies the termination criteria (using xtol = 1.48e-08) success: True fun: 5.535810445932086 x: 0.5419260710334868 nit: 12 nfev: 15 r_min = 0.5419260710334868</pre>

Como resultado de la optimización, el método devuelve un objeto (asignado en este caso a la variable `solucion`) que en sus atributos contiene información acerca de cómo fue la ejecución de la optimización, tal como un mensaje que describe el motivo de la finalización, una variable booleana que indica si hubo convergencia (`success`), el valor de la función evaluada en el valor óptimo (`fun`), el valor óptimo hallado (`x`), el número de iteraciones (`nit`) y las llamadas a la función

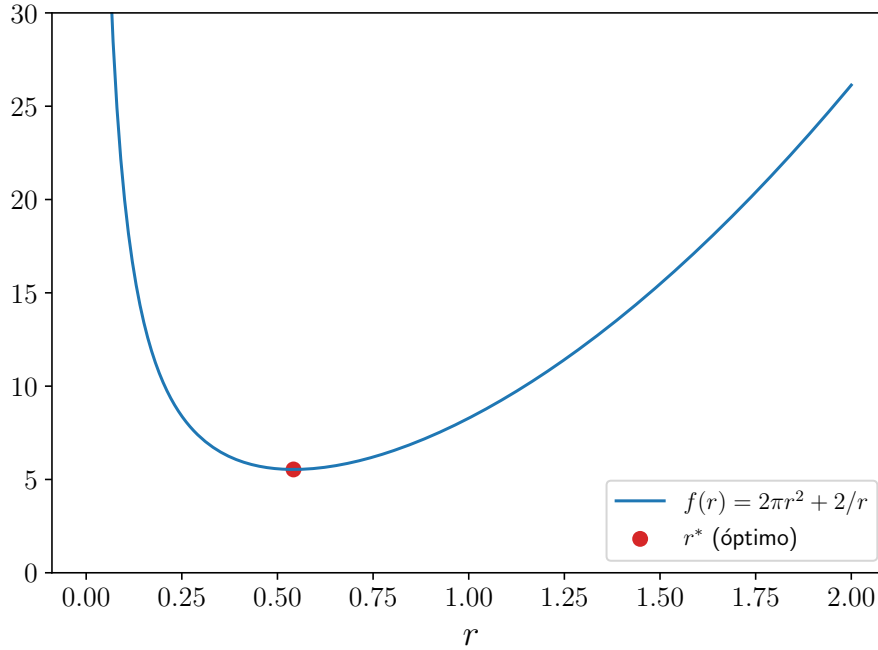


FIGURA 22.1: Optimización de la superficie de un cilindro con volumen dado.

objetivo (nfev). Podemos ver que el resultado numérico coincide con el valor “exacto” obtenido con SymPy hasta la precisión informada. La figura 22.1 muestra la función y el valor óptimo hallado. Siempre es recomendable visualizar la función antes de aceptar el valor obtenido como válido.

La función `minimize_scalar` permite establecer, además, la tolerancia requerida para la convergencia o el número máximo de iteraciones. Sugerimos ver las opciones en [su documentación](#).

22.2. Optimización multidimensional sin restricciones

La optimización multidimensional es más compleja que la correspondiente a la de una sola dimensión que vimos en la sección anterior. Los métodos de *bracketing* no son aplicables y en general se deben aplicar métodos que comienzan en algún punto en el espacio de coordenadas y mediante alguna estrategia se buscan nuevos puntos que sucesivamente se aproximen al mínimo.

Como abordaremos el caso de una minimización sin restricciones, podemos expresar nuestro problema como

$$\text{Minimizar } f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n$$

donde $f(\mathbf{x})$ es una función no lineal de $\mathbf{x} = (x_1, \dots, x_n)$. Estos problemas son frecuentes en muchas aplicaciones, por ejemplo en problemas de redes neuronales donde es necesario encontrar los pesos que minimizan la diferencia entre la salida de la red y la respuesta conocida a la entrada.

Iniciamos entonces asumiendo una aproximación inicial $\mathbf{x}_0$, y procedemos a generar una

mejor aproximación utilizando una fórmula iterativa de la forma

$$\mathbf{x}_{k+1} = \mathbf{x}_k + s\mathbf{d}_k, \quad \text{para } k = 0, 1, 2, \dots$$

Para utilizar esta fórmula es necesario determinar el vector $\mathbf{d}_k$, que representa la dirección en la que buscamos el nuevo valor $\mathbf{x}_{k+1}$, y el escalar s que representa la distancia en que nos movemos en esa dirección. Se pueden implementar muchas estrategias para definir estas cantidades. Tal vez la elección más simple consiste en tomar la dirección $\mathbf{d}_k$ como el negativo del vector gradiente de f en $\mathbf{x}_k$. Para un paso d suficiente pequeño, esta estrategia garantiza una reducción en el valor de la función. Esto conduce al algoritmo del descenso del gradiente:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - s\nabla f(\mathbf{x}_k)$$

donde $\nabla f(\mathbf{x}) = (\partial f/\partial x_1, \partial f/\partial x_2, \dots, \partial f/\partial x_n)$, y s es un valor constante pequeño. El mínimo se alcanza cuando el gradiente se anula. Aunque en cada paso nos movemos hacia un valor menor de la función, utilizando valores chicos de s el algoritmo es muy lento. Por otra parte, un valor grande de s pone en riesgo la convergencia hacia el mínimo.

El valor de s se puede optimizar considerando que el valor que produce la máxima reducción de f es el que se obtiene de minimizar la función a lo largo de la dirección establecida por $-\nabla f(\mathbf{x}_k)$. Formalmente, esto significa que para cada k debemos hallar el valor de s que minimiza

$$f[\mathbf{x}_k - s\nabla f(\mathbf{x}_k)] \quad (22.2)$$

que también constituye un problema de minimización denominado “búsqueda lineal”, dado que en definitiva es una minimización en una sola variable, s , a lo largo de la dirección del gradiente, ya que $\mathbf{x}_k$ es conocido. Claramente, el valor que minimiza $f[\mathbf{x}_k - s\nabla f(\mathbf{x}_k)]$ es tal que la derivada de $f[\mathbf{x}_k - s\nabla f(\mathbf{x}_k)]$ es cero. Entonces, diferenciando $f[\mathbf{x}_k - s\nabla f(\mathbf{x}_k)]$ respecto de s tenemos:

$$\frac{df[\mathbf{x}_k - s\nabla f(\mathbf{x}_k)]}{ds} = -(\nabla f(\mathbf{x}_{k+1}))^\top \nabla f(\mathbf{x}_k) = 0$$

lo que muestra que las direcciones sucesivas de búsqueda son ortogonales. Si bien este método genera una secuencia que se aproxima al mínimo de f , no presenta la mejor forma de hacerlo dado que los cambios en las direcciones de búsqueda son grandes.

Una mejora consiste en tomar una combinación de la dirección previa y de la nueva dirección para aproximarnos al valor óptimo en forma más directa, lo que da origen al método del gradiente conjugado. Utiliza el mismo paso s obtenido en (22.2), y, llamando $\mathbf{g}_{k+1} = \nabla f(\mathbf{x}_{k+1})$, la nueva dirección resulta de

$$\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta\mathbf{d}_k$$

Esto es, la nueva dirección es una combinación del negativo del gradiente en la nueva posición mas el escalar β por la dirección previa de búsqueda. Ahora el problema es establecer el valor de β , y el criterio utilizado es que las sucesivas direcciones de búsqueda sean conjugadas, esto es, que $(\mathbf{d}_{k+1})^\top \mathbf{A}\mathbf{d}_k = 0$ para alguna matriz dada $\mathbf{A}$.

Esta elección posee propiedades adecuadas de convergencia para el método del gradiente conjugado. Se puede mostrar que conduce a un valor de β dado por:

$$\beta = \frac{(\mathbf{g}_{k+1})^\top \mathbf{g}_{k+1}}{(\mathbf{g}_k)^\top \mathbf{g}_k}$$

El método de Newton para optimización multidimensional se puede considerar como una modificación del descenso del gradiente que puede mejorar la convergencia. Del mismo modo que para el caso unidimensional, el método de Newton consiste en una aproximación cuadrática local de la función, que cuando se minimiza da origen a un esquema iterativo. Escribiendo una expansión en serie de Taylor de segundo orden para $f(\mathbf{x})$ alrededor de $\mathbf{x} = \mathbf{x}_i$:

$$f(\mathbf{x}) = f(\mathbf{x}_i) + \nabla f^\top(\mathbf{x}_i)(\mathbf{x} - \mathbf{x}_i) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_i)^\top H_i(\mathbf{x} - \mathbf{x}_i)$$

donde H_i es la matriz hessiana³. En el mínimo,

$$\frac{\partial f(\mathbf{x})}{\partial x_j} = 0, \quad \text{para } j = 1, 2, \dots, n$$

Entonces,

$$\nabla f = \nabla f(\mathbf{x}_i) + H_i(\mathbf{x} - \mathbf{x}_i) = 0$$

Si H_i no es singular,

$$\mathbf{x}_{i+1} = \mathbf{x}_i - H_i^{-1} \nabla f(\mathbf{x}_i)$$

que se puede mostrar converge cuadráticamente cerca del mínimo. Este procedimiento altera, por lo general, tanto la dirección como la longitud del paso, por lo que no es estrictamente un método del descenso del gradiente, y puede no converger si el punto de partida no está suficientemente cerca del mínimo. Tal como está planteado aquí, el método de Newton requiere tanto el cálculo como el almacenamiento del gradiente y del hessiano de la función. Esto puede resultar impracticable para funciones de muchas variables (particularmente para optimización de la función de pérdida durante el entrenamiento de redes neuronales), por lo que existen métodos más apropiados, como los algoritmos cuasi-Newton, siendo uno de los más populares el BFGS (Broyden - Fletcher - Goldfarb - Shanno).

El submódulo `optimize` de SciPy contiene la función `minimize`, que permite utilizar, entre otros, los métodos mencionados aquí. Vamos a mostrar un ejemplo de optimización de la función de dos variables:

$$f(x_1, x_2) = \frac{1}{2} \sum_{i=1}^2 (x_i^4 - 16x_i^2 + 5x_i)$$

que tomamos de Styblinski y Tang [44] y tiene cuatro mínimos locales, con el mínimo global localizado en $x_i^* = -2,903534, i = 1, 2$, y un máximo local ubicado cerca del origen. Para utilizar el método de Newton, necesitamos calcular el gradiente de la función y también el hessiano. En el ejemplo que estamos analizando es simple realizar estos cálculos manualmente, pero aprovecharemos la oportunidad para utilizar la potencia de cálculo simbólico de SymPy, y veremos cómo utilizar las expresiones simbólicas obtenidas en el contexto numérico de NumPy.

Comenzamos entonces definiendo los símbolos a utilizar, la función f (simbólica) y sus derivadas parciales, tal como se muestra en la celda 6:

³ La matriz hessiana de un campo escalar $f : \mathbb{R}^n \rightarrow \mathbb{R}$ es la matriz cuadrada de tamaño $n \times n$ que tiene como entradas las derivadas parciales de segundo orden. Ver [entrada](#) en Wikipedia.

CELDA 06

```
x1, x2, x, y = symbols("x_1 x_2 x y")
f_sim = (x1**4 - 16 * x1**2 + 5 * x1) / 2 + (x2**4 - 16 * x2**2 + 5 * x2) / 2
f_d1 = [f_sim.diff(x_) for x_ in (x1, x2)]
```

Podemos ver el gradiente representando en forma matricial (aunque estrictamente es un vector columna en este caso) utilizando la función `Matrix` de SymPy:

CELDA 07

```
from sympy import Matrix

# Gradiente:
Matrix(f_d1)
```

$$\begin{bmatrix} 2x_1^3 - 16x_1 + \frac{5}{2} \\ 2x_2^3 - 16x_2 + \frac{5}{2} \end{bmatrix}$$

Análogamente podemos obtener la matriz hessiana:

CELDA 08

```
f_hess = [[f_sim.diff(x, y) for x in (x1, x2)] for y in (x1, x2)]
```

CELDA 09

```
# Hessiano:
Matrix(f_hess)
```

$$\begin{bmatrix} 2(3x_1^2 - 8) & 0 \\ 0 & 2(3x_2^2 - 8) \end{bmatrix}$$

Habiendo realizado estos cálculos, necesitamos ahora obtener la representación vectorizada de estas funciones, de modo de que puedan ser manipuladas por NumPy y SciPy. Para ello invocamos la función `lambdify` de SciPy:

CELDA 10

```
from sympy import lambdify

f_num = lambdify((x1, x2), f_sim, 'numpy')
f_d1_num = lambdify((x1, x2), f_d1, 'numpy')
f_hess_num = lambdify((x1, x2), f_hess, 'numpy')
```

con lo que ahora `f_num`, `f_d1_num` y `f_hess_num` representan a f , su gradiente y su matriz hessiana, respectivamente. Sin embargo, las funciones generadas por `lambdify` toman un argumento por cada variable en la correspondiente expresión, mientras que las funciones de optimización de SciPy esperan funciones vectorizadas en las que todos los argumentos se encuentran empaquetados en un array. Por lo tanto, para obtener funciones que sean compatibles con las rutinas de optimización de SciPy, debemos “envolver” las funciones generadas por `lambdify` con funciones de Python que reacomoden los argumentos, tal como mostramos en la celda 11:

CELDA 11

```
def func_XY_a_X_Y(f):
    return lambda X: np.array(f(X[0], X[1]))

f = func_XY_a_X_Y(f_num)
f1 = func_XY_a_X_Y(f_d1_num)
fh = func_XY_a_X_Y(f_hess_num)
```

Finalmente tenemos todo listo para proceder con la optimización. Esto lo hacemos con la función `minimize` de `scipy.optimize`, pasándole como argumentos la función a minimizar (`f`), su gradiente (`f1`, argumento `jac`) y la matriz hessiana (`fh`, argumento `hess`), así como el punto de partida de la búsqueda: `(1, 1)`:

CELDA 12

```
from scipy.optimize import minimize

solucion = minimize(f, (1, 1), method='Newton-CG', jac=f1, hess=fh)
solucion

-----

message: Optimization terminated successfully.
success: True
status: 0
      fun: -50.058893310567896
         x: [ 2.747e+00  2.747e+00]
        nit: 4
         jac: [ 2.301e-04  2.301e-04]
        nfev: 7
        njev: 7
        nhev: 4
```

La función encuentra un mínimo en $(x_1, x_2) = (2,747, 2,747)$, y muestra información sobre cómo ha resultado la ejecución del algoritmo, incluyendo el número de iteraciones, el valor de la función en el mínimo, el valor del gradiente (`jac`) y la cantidad de evaluaciones de la función, del gradiente y del hessiano. Podemos verificar el valor de la función en el mínimo hallado:

CELDA 13

```
f(solucion.x)

-----

array(-50.05889331)
```

Como es usual en estos casos, resulta útil visualizar el problema. Para ello realizamos un mapa de calor donde el color representa el valor de la función y marcamos con una estrella roja el óptimo hallado:

CELDA 14

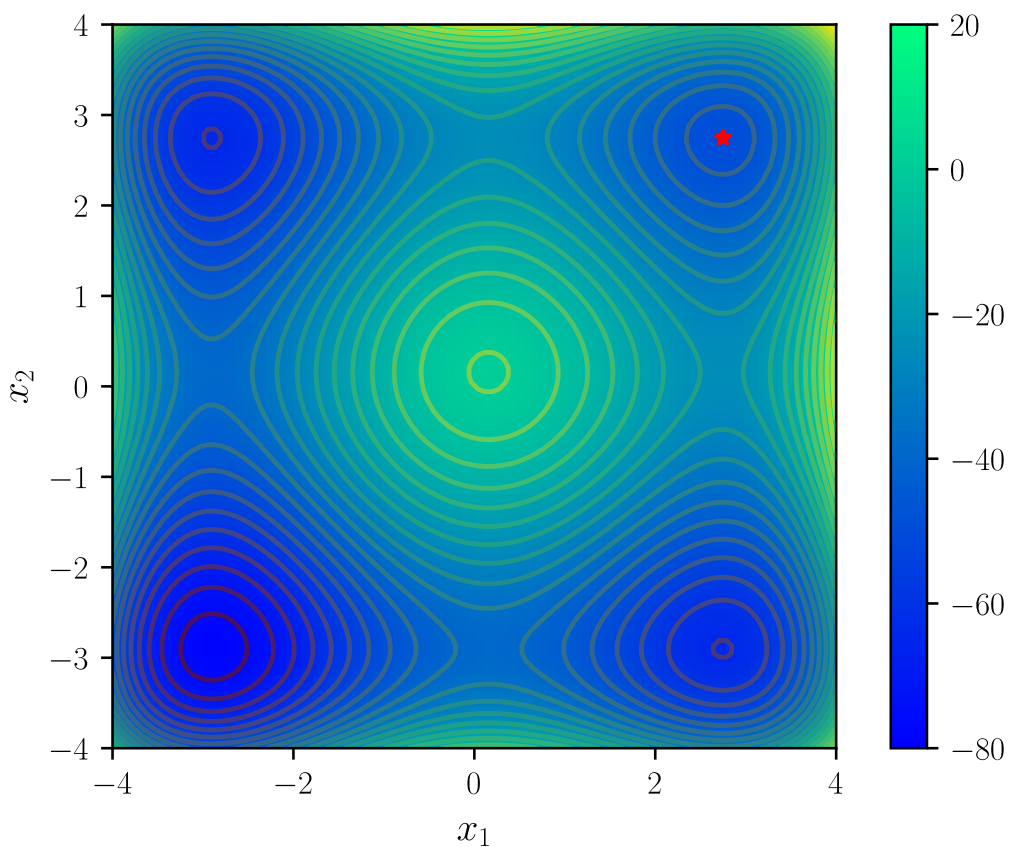
```
import matplotlib.pyplot as plt
import matplotlib.colors as colors

fig, ax = plt.subplots(figsize=(6, 4))
xLim = yLim = 4
x = y = np.linspace(-xLim, xLim, 100)
X, Y = np.meshgrid(x, y)
```

```

x_min = solucion.x
palette = plt.cm.winter
im=ax.imshow(f_num(X, Y), extent=[-xLim, xLim, -yLim, yLim],
             cmap=palette, norm=colors.Normalize(vmin=-80.0, vmax=20),
             origin='lower')
c = ax.contour(X, Y, f_num(X, Y), 30)
ax.plot(x_min[0], x_min[1], 'r*', markersize=5)
ax.set_xlabel(r"$x_1$", fontsize=12)
ax.set_ylabel(r"$x_2$", fontsize=12)
plt.colorbar(im, ax=ax)
plt.show()

```



En la práctica no siempre suele ser factible calcular el gradiente o el hessiano de la función, por lo que pueden utilizarse otros métodos que estiman estas funciones numéricamente. Un método popular mencionado anteriormente es el BFGS. Para utilizar este método podemos invocar la función `optimize` con el argumento `method='BFGS'`:

CELDA 15

```

sol_bfgs = minimize(f, (1, 1), method='BFGS')
sol_bfgs

```

```

message: Optimization terminated successfully.
success: True
status: 0
  fun: -50.058893310566276
   x: [ 2.747e+00  2.747e+00]
  nit: 6
  jac: [-6.676e-06 -6.676e-06]
hess_inv: [[ 4.741e-01 -4.802e-01]
            [-4.802e-01  5.582e-01]]
 nfev: 33
 njev: 11

```

Vemos que en este caso no es necesario suministrar el gradiente ni la matriz hessiana, y también podemos notar que, en consecuencia, el número de evaluaciones de la función aumenta significativamente (30) respecto del método de Newton (7 evaluaciones).

Otro método del que dispone SciPy es el del gradiente conjugado. En este caso podemos usar el gradiente de la función, mientras que el hessiano no es requerido:

CELDA 16

```

sol_cg = minimize(f, (1, 1), method='CG', jac=f1)
sol_cg

```

```

message: Optimization terminated successfully.
success: True
status: 0
  fun: -50.05889331056788
   x: [ 2.747e+00  2.747e+00]
  nit: 5
  jac: [-2.170e-09 -2.170e-09]
 nfev: 17
 njev: 16

```

Si no se conocen ni el gradiente ni el hessiano, el método BFGS suele ser un buen punto de partida para la búsqueda del óptimo. En caso de poder calcular el gradiente, el método del gradiente conjugado resulta más eficiente (en nuestro caso solo necesita 17 evaluaciones de la función para hallar el mínimo). Naturalmente, cuanto más información le podamos aportar al método, más eficiente será el cálculo, tal como se muestra al utilizar el método de Newton, que provee la convergencia más rápida. En cualquier caso es necesario también evaluar la complejidad de las funciones para el cálculo del gradiente o del hessiano, debido a que puede resultar costosa la evaluación numérica de ellas, haciendo más conveniente el uso del método BFGS.

En general, los métodos que hemos visto hasta aquí convergen a mínimos locales. En los problemas con muchos mínimos locales, es muy posible que estos métodos converjan a uno de estos mínimos locales (dependiendo, naturalmente, del valor inicial de la iteración), y que nos resulte difícil localizar el mínimo global. En estos casos es recomendable realizar una primera búsqueda utilizando un método de fuerza bruta, que consiste simplemente en discretizar el dominio en una grilla y evaluar la función en los nodos de esta discretización, y utilizar el resultado de esta búsqueda como punto inicial de un método iterativo. SimPy provee la función `brute` de `optimize` para realizar esta búsqueda inicial, a la que le pasamos como argumento la función `f`, y una tupla de objetos `slice`, con un elemento por cada dimensión del espacio de parámetros (valor inicial,

valor final y paso para cada coordenada), que especifica la grilla sobre la que se evaluará la función. Como tercer argumento especificamos `finish=None` para evitar que la función `brute` refine la búsqueda (dejaremos esa tarea a uno de nuestros algoritmos, posteriormente):

CELDA 17
<pre>from scipy.optimize import brute sol_ini = brute(f, (slice(-5, 5, 0.5), slice(-5, 5, 0.5)), finish=None) sol_ini</pre>
array([-3., -3.])
CELDA 18
<pre>f(sol_ini)</pre>
array(-78.)

Vemos que el punto de la grilla $(-3, 3)$ ofrece el menor valor de f (-78). Ahora usamos ese punto como valor inicial del método BFGS:

CELDA 19
<pre>sol_bfgs_g = minimize(f, sol_ini, method='BFGS') sol_bfgs_g</pre>
<pre>message: Optimization terminated successfully. success: True status: 0 fun: -78.33233140754265 x: [-2.904e+00 -2.904e+00] nit: 3 jac: [-1.907e-06 -1.907e-06] hess_inv: [[5.145e-01 -4.855e-01] [-4.855e-01 5.145e-01]] nfev: 15 njev: 5</pre>

que converge al valor del mínimo global que anticipamos al inicio. Dado que nuestro punto de partida es bastante cercano al óptimo, el número de evaluaciones requeridas por el algoritmo BFGS es la mitad de la requerida para el punto inicial $(1, 1)$, que converge a un mínimo local.

22.3. Optimización con restricciones

La incorporación de restricciones a los problemas de optimización incrementa el nivel de complejidad de los métodos para obtener las soluciones posibles. No existe una única forma de categorizar estos problemas, en este capítulo abordaremos tres formas de considerar restricciones: límites en las variables de decisión, programación lineal y programación no lineal.

22.3.1. Restricciones en las variables de decisión

Tal vez la manera más simple de incluir restricciones en un problema de optimización consiste en limitar el dominio de búsqueda de los valores óptimos de las variables de decisión. Por ejemplo, encontrar el mínimo de $f(x, y)$ sujeto a $a \leq x \leq b$ y $c \leq y \leq d$. La restricción $a \leq x \leq b$ es simple dado que solo restringe los valores de la variable x independientemente de las otras variables del problema.

SciPy puede resolver estos problemas de optimización con restricciones simples por medio del algoritmo L-BFGS-B, que es una variante del método BFGS que utilizamos en la sección 22.2. Veamos cómo aplicamos este método para obtener el mínimo de la [función de Rosembrock](#):

$$f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

donde usaremos $a = 1$ y $b = 1$. Esta función tiene un mínimo global en (a, a^2) en donde la función se anula. Intentaremos localizar los mínimos en el problema sin restricciones, y restringiendo el dominio de las variables de decisión al rectángulo $2 \leq x_1 \leq 3$ y $0 \leq x_2 \leq 2$:

CELDA 20

```
def f_ros(X):
    x1, x2 = X
    a, b = 1, 1
    return (a - x1)**2 + b * (x2 - x1**2)**2

x0 = [0, 0]
res_opt = minimize(f_ros, x0, method='BFGS')
bnd_x1, bnd_x2 = (2, 3), (0, 2)
res_opt_bnd = minimize(f_ros, x0, method='L-BFGS-B',
                       bounds=[bnd_x1, bnd_x2])
print(f"Sin restricción: f = {res_opt.fun} en x = {res_opt.x}")
print(f"Con restricción: f = {res_opt_bnd.fun} en x = {res_opt_bnd.x}")
```

Sin restricción: f = 2.9986375725184793e-15 en x = [0.99999995 0.99999991]
 Con restricción: f = 5.0 en x = [2. 2.]

Aquí podemos ver que cuando optimizamos sin restringir los valores de x_1 y x_2 localizamos el mínimo global de la función de Rosembrock, pero al establecer la optimización limitada a los intervalos especificados para cada variable, el valor óptimo se establece en $(2, 2)$ en donde $f = 5$. Dado que este ejemplo presenta una optimización de dos variables, podemos realizar un gráfico para visualizar cómo se ubican los valores óptimos en cada caso. En la celda siguiente construimos esta visualización, realizando previamente una definición de la función de Rosembrock para que acepte como parámetros de entrada dos valores en vez de una tupla, ya que de este modo necesitamos ingresar los valores de la grilla de visualización. Marcamos también en gris el rectángulo que comprenden los intervalos de la restricción. La estrella azul representa los valores que minimizan la función sin restricciones, mientras que la roja lo hace al establecer las restricciones.

CELDA 21

```
def f_ros_p(x, y):
    return f_ros((x, y))

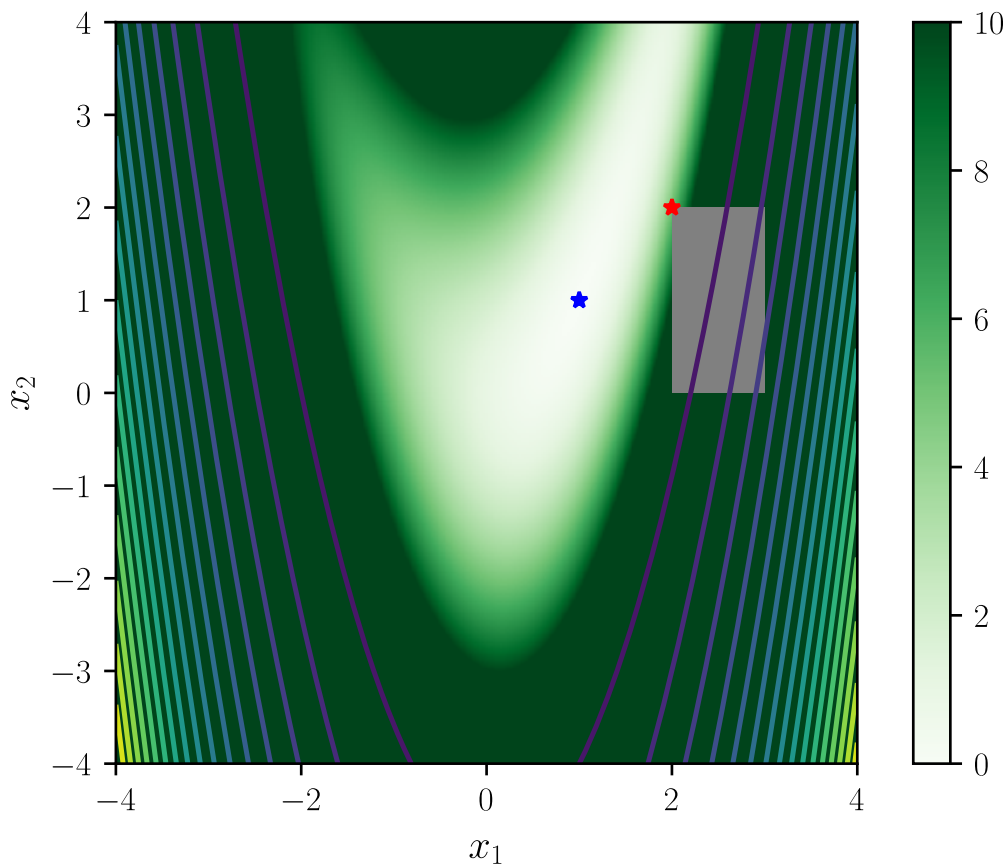
fig, ax = plt.subplots(figsize=(6, 4))
```



```

xLim = yLim = 4
x = y = np.linspace(-xLim, xLim, 200)
X, Y = np.meshgrid(x, y)
palette = plt.cm.Greens
im=ax.imshow(f_ros_p(X, Y), extent=[-xLim, xLim, -yLim, yLim],
             cmap=palette, norm=colors.Normalize(vmin=-0.0, vmax=10),
             origin='lower')
c = ax.contour(X, Y, f_ros_p(X, Y), 20)
ax.plot(res_opt.x[0], res_opt.x[1], 'b*', markersize=5)
ax.plot(res_opt_bnd.x[0], res_opt_bnd.x[1], 'r*', markersize=5)
bound_zone = plt.Rectangle((bnd_x1[0], bnd_x2[0]),
                           bnd_x1[1] - bnd_x1[0],
                           bnd_x2[1] - bnd_x2[0],
                           facecolor='gray')
ax.add_patch(bound_zone)
ax.set_xlabel(r"$x_1$", fontsize=12)
ax.set_ylabel(r"$x_2$", fontsize=12)
plt.colorbar(im, ax=ax)
plt.show()

```



22.3.2. Programación lineal

La programación lineal aborda el problema de optimizar una función objetivo lineal sujeta a restricciones lineales en forma de igualdades y desigualdades sobre las variables de decisión. Estos problemas se pueden expresar en una variedad de formas equivalentes. Veremos en un ejemplo simple cómo abordar un problema de programación lineal.

Supongamos que disponemos de tres tipos de alimentos, A_1 , A_2 y A_3 , que contienen tres tipos de nutrientes (carbohidratos, proteínas y vitaminas) en las siguientes cantidades:

Alimento	Carbohidratos	Proteínas	Vitaminas	Costo (\$/kg)
A_1	1	4	3	0.82
A_2	7	2	2	0.50
A_3	2	2	0	0.40

El requerimiento diario de carbohidratos, proteínas y vitaminas es 10, 15 y 6, respectivamente. El problema es determinar cuánto es necesario consumir de cada alimento para obtener las cantidades requeridas de cada nutriente al menor costo, por lo tanto, la elección natural de las variables de decisión a optimizar será x_1 , x_2 y x_3 como las cantidades a consumir por día de cada alimento.

El paso siguiente es determinar la función objetivo a maximizar o minimizar. En este ejemplo, queremos minimizar el costo diario total, que se obtiene mediante el cálculo $0,82 x_1 + 0,50 x_2 + 0,40 x_3$.

Por último necesitamos describir las restricciones que deben satisfacer x_1 , x_2 y x_3 . En primer lugar, estas variables deben ser positivas (no podemos comer cantidades negativas de alimentos). Estas restricciones se denominan “restricciones de no negatividad”, y se encuentran muy frecuentemente en los problemas de programación lineal. Además, no todos los valores de x_1 , x_2 y x_3 dan lugar a una dieta con las cantidades diarias necesarias de nutrientes. La cantidad de proteínas en x_1 unidades de A_1 , x_2 de A_2 y x_3 de A_3 es $4x_1 + 2x_2 + 2x_3$, y esa cantidad debe ser al menos de 15 por día. Esto significa que x_1 , x_2 y x_3 deben satisfacer $4x_1 + 2x_2 + 2x_3 \geq 15$.

Entonces, este problema de optimización dietaria se puede formular de la siguiente forma:

$$\begin{aligned}
 \text{minimizar: } & f(x_1, x_2, x_3) = 0,82 x_1 + 0,50 x_2 + 0,40 x_3 \\
 \text{sujeto a: } & x_1 + 7 x_2 + 2 x_3 \geq 10 \\
 & 4 x_1 + 2 x_2 + 2 x_3 \geq 15 \\
 & 3 x_1 + 2 x_2 \geq 6 \\
 & x_1 \geq 0; x_2 \geq 0; x_3 \geq 0
 \end{aligned} \tag{22.3}$$

Se dice que una solución $x = (x_1, x_2, x_3)$ es factible con respecto al programa lineal anterior si satisface todas las restricciones anteriores. El conjunto de soluciones factibles se denomina espacio factible o región factible. Una solución factible es óptima si el valor de su función objetivo es igual al valor más pequeño que f puede tomar en la región factible.

Para obtener la solución a este problema de programación lineal utilizaremos la función `linprog` de `scipy.optimize`, que puede resolver problemas con la forma:

$$\begin{aligned}
 & \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} \\
 & \text{tal que: } \mathbf{A}_{ub} \mathbf{x} \leq \mathbf{b}_{ub} \\
 & \mathbf{A}_{eq} \mathbf{x} \leq \mathbf{b}_{eq} \\
 & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}
 \end{aligned} \tag{22.4}$$

donde $\mathbf{x}$ representa el vector con las variables de decisión, $\mathbf{c}$ es el vector con los coeficientes de la función lineal f , $\mathbf{A}_{ub}$ es la matriz con los coeficientes de las restricciones de desigualdad, en la que cada fila especifica los coeficientes de una desigualdad lineal sobre $\mathbf{x}$, $\mathbf{b}_{ub}$ es el vector de las desigualdades lineales en el que cada elemento representa el límite superior del correspondiente valor de $\mathbf{A}_{ub} \mathbf{x}$. Del mismo modo, $\mathbf{A}_{eq}$ y $\mathbf{b}_{eq}$ representan restricciones mediante igualdades (que en este problema no tenemos). Finalmente, $\mathbf{l}$ y $\mathbf{u}$ son los límites inferiores y superiores del dominio de búsqueda de los valores óptimos de $\mathbf{x}$ (interpretando la desigualdad elemento por elemento).

Podemos ver que en nuestro problema representado en la forma (22.3) las restricciones de desigualdad tienen el sentido inverso al requerido por `linprog`. Esto no representa ninguna dificultad, ya que multiplicando cada ecuación miembro a miembro por (-1) las adaptamos para usarlas en la función.

La función `linprog` utiliza los métodos `highs-ds` y `highs-ipm` para resolver el problema de minimización, que consisten en [rutinas desarrolladas en C++ de alto desempeño](#) que implementan los métodos `simplex dual revisado` y `de punto interior`, cuyas descripciones exceden el alcance de este libro [45]. En la celda 22 definimos `c` como una lista con los coeficientes de f , `A` representa la matriz $\mathbf{A}_{ub}$, la lista `b` contiene los elementos del vector $\mathbf{b}_{ub}$, y finalmente establecemos los límites para cada variable de decisión con la tupla `(0, None)` que significa que el límite inferior en cada variable es cero, y el superior es infinito (todas las variables de decisión son no negativas):

CELDA 22

```

c = [0.82, 0.5, 0.4]
A = [[-1, -7, -2], [-4, -2, -2], [-3, -2, 0]]
b = [-10, -15, -6]
x0_bounds = (0, None)
x1_bounds = (0, None)
x2_bounds = (0, None)

```

Ahora podemos invocar la función `linprog`, guardando su resultado en un objeto `res` que contiene el valor mínimo de la función objetivo (`res.fun`) evaluada en los valores óptimos obtenidos (`res.x`):

CELDA 23

```

from scipy.optimize import linprog

res = linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds, x2_bounds])
print(f"Valor de f: {res.fun}, en x = {res.x}")
print(f"c . x = {c @ res.x}")

```

```

Valor de f: 3.052380952380952, en x = [1.9047619  0.14285714 3.54761905]
c . x = 3.0523809523809526

```

que verificamos mostrando el valor del producto `c @ res.x`. El objeto `res` contiene más información acerca de la ejecución de la rutina de optimización:

CELDA 24	
res	

message: Optimization terminated successfully. (HiGHS Status 7: Optimal)	
success: True	
status: 0	
fun: 3.052380952380952	
x: [1.905e+00 1.429e-01 3.548e+00]	
nit: 3	
lower: residual: [1.905e+00 1.429e-01 3.548e+00]	
marginals: [0.000e+00 0.000e+00 0.000e+00]	
upper: residual: [inf inf inf]	
marginals: [0.000e+00 0.000e+00 0.000e+00]	
eqlin: residual: []	
marginals: []	
ineqlin: residual: [0.000e+00 0.000e+00 0.000e+00]	
marginals: [-1.238e-02 -1.876e-01 -1.905e-02]	
mip_node_count: 0	
mip_dual_bound: 0.0	
mip_gap: 0.0	

Para la interpretación de la información contenida en `res`, invitamos a ver la [documentación](#).

22.3.3. Programación no lineal

Una vez más, cuando en el problema intervienen relaciones no lineales entre las variables de decisión, la complejidad aumenta y son necesarios métodos más sofisticados para encontrar el valor óptimo, y en muchas oportunidades no es posible obtener una minimización o maximización que respete todas las restricciones.

Una de las técnicas que se pueden utilizar para problemas de optimización con restricciones no lineal es la de los multiplicadores de Lagrange, que convierte un problema con restricciones en uno sin restricciones mediante la incorporación de variables adicionales. La idea es incorporar cada restricción a la función objetivo y resolver el problema como uno sin restricciones.

Por ejemplo, consideremos el problema de optimización mín $f(\mathbf{x})$ sujeto a restricciones de igualdad $g(\mathbf{x}) = 0$. Si no consideramos esta restricción, el gradiente de $f(\mathbf{x})$ se anula en los puntos extremos, $\nabla f(\mathbf{x}) = 0$. Se puede mostrar que al incorporar la restricción, el negativo del gradiente pertenece al espacio soportado por la restricción normal, es decir, $-\nabla f(\mathbf{x}) = \boldsymbol{\lambda} J_g^T(\mathbf{x})$, donde $J_g(\mathbf{x})$ es la matriz jacobiana de la función de restricción $g(\mathbf{x})$ y $\boldsymbol{\lambda}$ son las nuevas variables representadas por el vector de los multiplicadores de Lagrange. Esta condición tiene su origen en igualar a cero la función $L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T g(\mathbf{x})$ denominada “función lagrangiana”. Si ambas funciones f y g son continuas y diferenciables, un punto estacionario $(\mathbf{x}_0, \boldsymbol{\lambda}_0)$ de L corresponde a un $\mathbf{x}_0$ que es un óptimo del problema original con restricciones.

Veamos un ejemplo minimizando nuevamente la función de Rosembrock

$$f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

con la condición $g(x_1, x_2) = x_1^2 + x_2^2 = 1$. Al igual que en el ejemplo de la subsección 22.3.2. tomamos $a = 1$ y $b = 1$. Esta vez utilizaremos como argumento de la función `minimize` el método

SLSQP (*Sequential Least Squares Programming*), que implementa en forma iterativa la optimización de un modelo cuadrático utilizando aproximaciones de segundo orden del lagrangiano. La condición de restricción debe pasarse como diccionario en el argumento `constraints` de `optimize`, declarando en la definición que se trata de una restricción de igualdad a cero (`type='eq'`):

CELDA 25

```
def g(X):
    x1, x2 = X
    return x1**2 + x2**2 - 1

constrain = dict(type='eq', fun=g)
```

Ahora hacemos la minimización del mismo modo que en los ejemplos anteriores:

CELDA 26

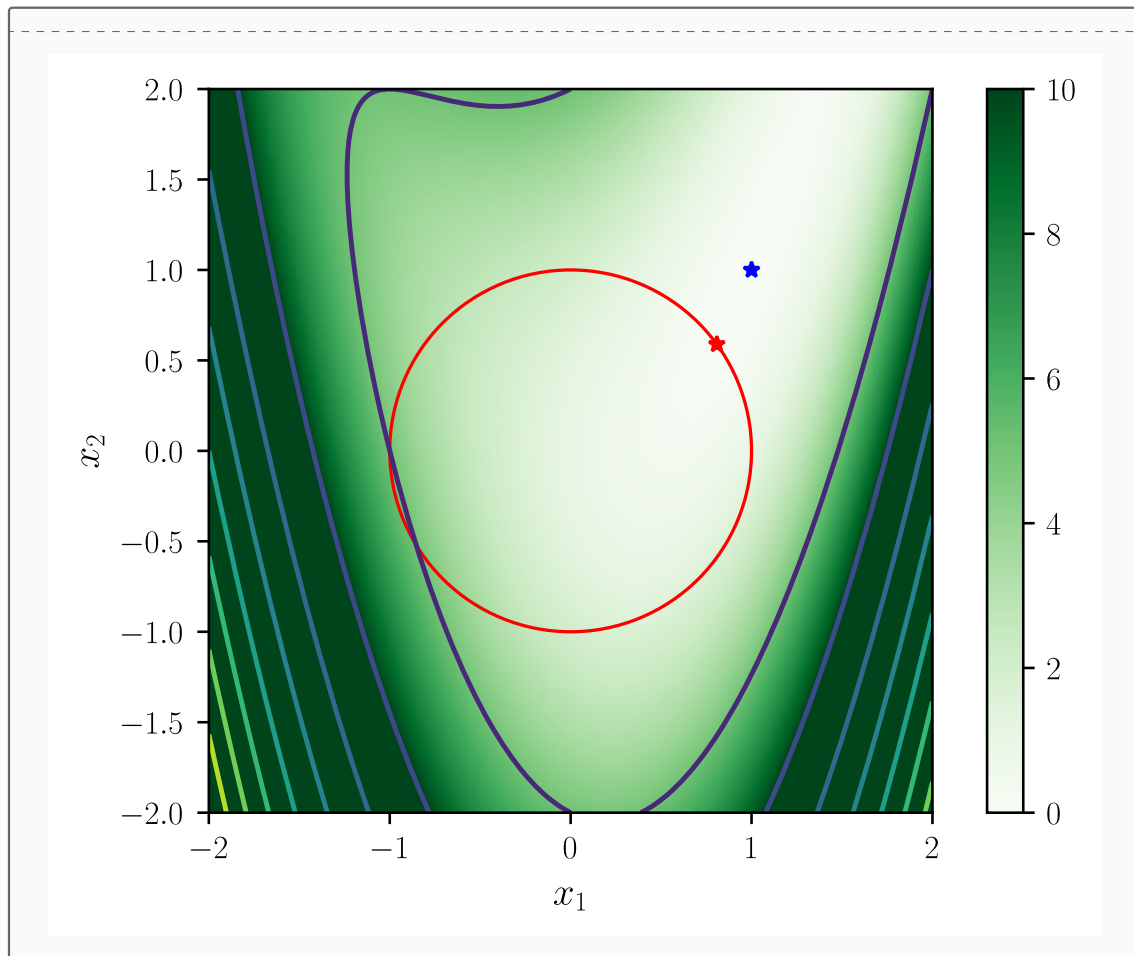
```
res = minimize(f_ros, x0, method='SLSQP', constraints=[constrain])
res
```

```
message: Optimization terminated successfully
success: True
status: 0
      fun: 0.04091903706230661
         x: [ 8.082e-01  5.889e-01]
        nit: 8
         jac: [-1.762e-01 -1.284e-01]
        nfev: 33
        njev: 8
```

Y visualizamos el proceso mostrando con una estrella azul el mínimo de la función de Rosbrock sin restricciones, y el nuevo óptimo (en rojo) cuya restricción exige que se encuentre sobre la circunferencia de radio 1:

CELDA 27

```
fig, ax = plt.subplots(figsize=(6, 4))
xLim = yLim = 2
x = y = np.linspace(-xLim, xLim, 200)
X, Y = np.meshgrid(x, y)
palette = plt.cm.Greens
im=ax.imshow(f_ros_p(X, Y), extent=[-xLim, xLim, -yLim, yLim], cmap=palette,
             norm=colors.Normalize(vmin=-0.0, vmax=10), origin='lower')
c = ax.contour(X, Y, f_ros_p(X, Y), 10)
circle = plt.Circle( (0, 0), 1, fill=False, color='red')
ax.add_artist(circle)
ax.plot(res_opt.x[0], res_opt.x[1], 'b*', markersize=5)
ax.plot(res.x[0], res.x[1], 'r*', markersize=5)
ax.set_xlabel(r"$x_1$", fontsize=12)
ax.set_ylabel(r"$x_2$", fontsize=12)
plt.colorbar(im, ax=ax)
plt.show()
```



22.4. Algoritmo genético

En esta sección tomaremos un abordaje cualitativamente diferente al que hemos usado hasta ahora para resolver problemas de optimización. Introduciremos en forma sintética el algoritmo genético (AG), como ejemplo de la clase de métodos heurísticos de optimización. En particular, los AG están inspirados en el proceso de selección natural en el que una “población” de posibles soluciones (denominados “individuos” o “cromosomas”) evolucionan hacia soluciones mejores.

Los individuos de un AG codifican las variables de decisión del problema de optimización en cadenas de longitud finita de algún alfabeto. Estas cadenas que representan candidatos a la solución están compuestas de “genes” dados por el alfabeto utilizado. Para evolucionar hacia mejores soluciones e implementar un mecanismo de selección natural, es necesario una medida que distinga buenas soluciones de las que no son tan adecuadas para resolver el problema. Por lo general, esta medida es una función objetivo, que puede estar dada tanto por una expresión matemática como por el resultado de una simulación, y a la que generalmente se la denomina *fitness*.

Una vez que el problema ha sido codificado a través de un determinado alfabeto en forma de cromosomas, y que disponemos de la correspondiente función de *fitness*, el algoritmo de evolución se implementa de la siguiente manera:

1. Inicialización: se establece una población inicial de individuos, generalmente generados de forma aleatoria sobre el dominio de búsqueda. Si es posible incorporar conocimiento específico del problema en cuestión, se puede hacer en este paso.
2. Evaluación: una vez que la población ha sido inicializada, o cuando se obtiene una nueva generación de hijos, es necesario evaluar el *fitness* de cada individuo.
3. Selección: Un conjunto de individuos de la población, priorizando aquellos con mejor *fitness*, es elegido para formar el conjunto de padres de la siguiente generación de individuos.
4. Recombinación (o *crossover*): el cromosoma de dos padres elegidos en el paso anterior es recombinado para obtener nuevos individuos hijos.
5. Mutación: en forma aleatoria se modifica un individuo de la población.
6. Reemplazo: la generación de hijos generada por selección, recombinación y mutación reemplaza (total o parcialmente) la generación anterior de individuos.
7. Se repiten los pasos 2 – 6 hasta que se satisfaga una condición de finalización.

Cada uno de estos pasos requiere de la implementación de operadores (selección, *crossover*, mutación y reemplazo) que puede hacerse de diversas maneras, y de la elección de cada uno de ellos dependerá la eficacia con la que el AG evolucione hacia el óptimo. Mencionaremos algunas opciones a continuación.

Los métodos de selección se pueden clasificar en selección proporcional al *fitness* o en métodos ordinales. Un ejemplo de implementación del primer caso lo constituye el método de la ruleta, en el que a cada individuo se le asigna una “caja” cuyo tamaño es proporcional a su *fitness*, de modo que al lanzar la bola de la ruleta haya más probabilidades de que caiga en la caja de los individuos con mayor *fitness*. Un ejemplo de método ordinal es la selección por torneo, en el que se selecciona un número arbitrario de candidatos (mínimamente dos), con o sin reemplazo, y el que tiene el mayor *fitness* gana el torneo y es seleccionado como padre para la próxima generación. El torneo debe repetirse hasta seleccionar la cantidad de padres necesaria.

La operación de recombinación o *crossover* consiste en mezclar los genes de los cromosomas de dos padres. Típicamente, esto se logra intercambiando tramos de genes de los padres para formar a los hijos de la nueva generación, a través del “*crossover* de uno o dos puntos”, en los que el o los puntos de corte se eligen aleatoriamente a lo largo del cromosoma; o del “*crossover* uniforme”, en el que se genera una máscara binaria y se intercambian los genes a los que les corresponde el valor 1 en la posición de la máscara. Los siguientes esquemas ejemplifican estas variantes del operador de *crossover*.

$$\text{Crossover de un punto: } \left\{ \begin{array}{ccc} & \text{Padres} & \text{Hijos} \\ & a_1 a_2 a_3 a_4 | a_5 a_6 a_7 & a_1 a_2 a_3 a_4 b_5 b_6 b_7 \\ & b_1 b_2 b_3 b_4 | b_5 b_6 b_7 & b_1 b_2 b_3 b_4 a_5 a_6 a_7 \end{array} \right. \Rightarrow$$

$$\begin{aligned}
\text{Crossover de dos puntos: } & \left\{ \begin{array}{cc} \text{Padres} & \text{Hijos} \\ a_1 a_2 | a_3 a_4 | a_5 a_6 a_7 & a_1 a_2 b_3 b_4 a_5 a_6 a_7 \\ b_1 b_2 | b_3 b_4 | b_5 b_6 b_7 & b_1 b_2 a_3 a_4 b_5 b_6 b_7 \end{array} \right. \Rightarrow \\
\text{Crossover uniforme: } & \left\{ \begin{array}{cc} \text{Padres} & \text{Hijos} \\ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 & \leftarrow \text{Máscara} \\ a_1 a_2 a_3 a_4 a_5 a_6 a_7 & a_1 b_2 b_3 a_4 b_5 a_6 b_7 \\ b_1 b_2 b_3 b_4 b_5 b_6 b_7 & b_1 a_2 a_3 b_4 a_5 b_6 a_7 \end{array} \right. \Rightarrow
\end{aligned}$$

También existen diferentes maneras de implementar el operador de mutación. Tal vez la más simple consiste en reemplazar aleatoriamente un gen del cromosoma por un valor arbitrario. Preferentemente este valor debe estar dentro del dominio de búsqueda de la variable asociada al gen por medio del procedimiento de codificación. Si no es el caso, es posible obtener cromosomas que no representen una solución factible. Un método más sofisticado consiste en reemplazar un gen aleatorio por un valor obtenido mediante alguna distribución de probabilidad alrededor del valor del gen que estamos reemplazando. Una representación de este mecanismo se muestra en el siguiente esquema:

$$\text{Mutación: } \left\{ \begin{array}{cc} \text{Antes} & \text{Después} \\ a_1 a_2 a_3 a_4 a_5 a_6 a_7 & \Rightarrow a_1 a_2 a_3 b_4 a_5 a_6 a_7 \end{array} \right.$$

Por último, también existen variantes para el operador de reemplazo. El más simple consiste en reemplazar completamente la población actual con la nueva población de hijos. Esto es sencillo de realizar, pero puede suceder que en la nueva generación los individuos no tengan *fitness* tan buenos como el de algunos padres. Una forma de sortear esta posibilidad es implementar “elitismo”, que consiste en reemplazar solamente una porción r de los individuos originales, dejando sobrevivir en la nueva generación la porción $1 - r$ de los que tienen mejor *fitness*. Es usual que a través de la evolución de la población se mantenga constante el número de individuos que la componen.

Implementaremos ahora un AG básico para resolver un problema que, según su tamaño (es decir, el número de variables de decisión), puede ser muy difícil de abordar. El “problema de la mochila” se puede enunciar de la siguiente forma:

Dado un conjunto de ítems, cada uno con un peso y un valor, determinar qué ítems incluir en una colección de forma tal que el peso total sea menor o igual a la capacidad máxima de la mochila, maximizando el valor de los objetos incluidos en la colección.⁴

Formalmente, si tenemos n ítems cuyos valores son v_i y sus pesos p_i ($0 \leq i \leq n$), con un valor máximo P de carga, el problema consiste en:

⁴ Recomendamos la lectura de H. Kellerer, U. Pferschy y D. Pisinger. *Knapsack Problems*. Berlin: Springer-Verlag, 2004. DOI: [10.1007/978-3-540-24777-7](https://doi.org/10.1007/978-3-540-24777-7).

$$\begin{aligned}
 \text{maximizar: } f(\mathbf{x}) &= \sum_{i=1}^n v_i x_i \\
 \text{tal que: } \sum_{i=1}^n p_i x_i &\leq P \\
 \text{y } 0 \leq x_i &\leq q_i, \text{ para } 1 \leq i \leq n
 \end{aligned}
 \tag{22.5}$$

Si $q_i = 1$ para cada ítem, se trata del problema de la mochila 0-1. Si uno o más q_i es infinito se trata del problema de la mochila no acotado, mientras que en otro caso se llama problema de la mochila acotado. Abordaremos el caso en el que $x_i \in \{0, 1\}$.

El código a continuación contiene dos clases, que permiten representar objetos de tipo Individual y Population, siendo este último el objeto que contiene una lista de individuos y métodos que implementan los operadores del AG. Comenzamos describiendo la clase Individuo

```

1  #!/usr/bin/env python3
2
3  import random
4
5
6  class Individual():
7      genes = (0, 1)
8      max_weight = 102
9
10     def __init__(self, elements, chromosome=None):
11         # structures after elements to simplify later calculus
12         self.values, self.weights = zip(*elements)
13         self.size = len(elements)
14
15         # if the chromosome is not specified build a random one
16         if chromosome is None:
17             chromosome = [random.choice(self.genes) for _ in range(self.size)]
18         self.chromosome = chromosome
19
20         # these two values will be set on evaluation
21         self.fitness = None
22         self.weight = None
23
24     def evaluate(self):
25         """Calculate own weight and fitness value.
26
27         This is not done on __init__ because the chromosome may change after creation, it
28         will be called automatically before comparison.
29         """
30         self.fitness = sum(gene * value for gene, value in zip(self.chromosome, self.values))
31         self.weight = sum(gene * weight for gene, weight in zip(self.chromosome, self.weights))
32         if self.weight > self.max_weight:
33             # not fit at all

```

En la clase Individual comenzamos estableciendo una tupla que indica el conjunto de genes con el que codificaremos el cromosoma (nuestro alfabeto), que está limitado a 0 y 1. También indicamos cuál es la capacidad de carga de la mochila, determinado su valor en la variable max_weight.

Para la inicialización de los objetos necesitamos pasar como argumentos dos listas: `elements`, y `chromosome`. La primera contiene tuplas de pares (`valor`, `peso`) que son necesarios para la evaluación del *fitness*, mientras que la segunda lista permite inicializar un individuo con un cromosoma dado. En caso que no suministremos un cromosoma, creamos uno seleccionando al azar elementos de `self.genes`, la cantidad de veces necesaria para completar el tamaño del cromosoma, que queda determinado por la longitud de la lista de los valores.

El método `evaluate` calcula el *fitness* del individuo según (22.5), y también guarda el valor del peso total de los objetos que incluye el cromosoma. Nótese que si el peso del individuo es mayor que la capacidad de carga, asignamos un *fitness* de -1, indicando de este modo que el individuo no es apto como solución del problema ya que no será favorable un cromosoma que represente un valor negativo.

```

35
36     def mutate(self):
37         """Change a random gene (maybe)."""
38         self.fitness = self.weight = None
39         self.chromosome[random.randrange(self.size)] = random.choice(self.genes)
40
41     def __lt__(self, other):
42         # ensure fitness is useful
43         if self.fitness is None:
44             self.evaluate()
45         if other.fitness is None:
46             other.evaluate()
47
48         return other.fitness < self.fitness
49
50     def __str__(self):
51         return f"<Individual {id(self)} fitness={self.fitness}>"

```

El método `mutate` implementa el operador de mutación. Simplemente, determina un gen aleatorio dentro del cromosoma y le asigna una elección al azar de `self.genes`. En este ejemplo, tenemos igual probabilidad de cambiar el valor del gen o dejarlo como estaba. También hacemos `None` los valores de `self.fitness` y `self.weight`, ya que ante un eventual cambio en el cromosoma es necesario evaluar nuevamente estas cantidades. Finalmente, los dos métodos siguientes permiten comparar individuos según su *fitness* (`__lt__`), evaluando sus *fitness* de ser necesario, y realizar una representación prolija de un individuo (`__str__`).

Describiremos ahora la clase `Population`. En su inicialización pasamos como argumentos (con valores por defecto) los parámetros que controlan la ejecución del algoritmo y los correspondientes operadores genéticos, así como una lista con los pares (`valor`, `peso`) de cada ítem a considerar durante el proceso de optimización. Para generar la población inicial, construimos una lista de individuos cuyos genes se seleccionan aleatoriamente, hasta obtener el número de elementos establecidos en `size`

```

54
55 class Population():
56
57     def __init__(
58         self,

```

```

59     size=20,                # Cantidad de individuos
60     max_gen=50,             # Máximo número de generaciones
61     crossover_p=0.9,        # Probabilidad de crossover
62     mutation_p=0.05,       # Probabilidad de mutación
63     optimo=None,            # Valor óptimo de finalización
64     replacement_ratio=0.5,  # Tasa de reemplazo (elitismo)
65     elements=None,          # Pares (valor, peso) de los ítems
66 ):
67     if size % 2:
68         raise ValueError("Population size must be even.")
69     self.size = size
70
71     # set of individuals, always kept sorted (best at beginning)
72     self.individuals = sorted(Individual(elements) for _ in range(size))
73
74     self.elements = elements
75     self.crossover_p = crossover_p
76     self.mutation_p = mutation_p
77     self.max_gen = max_gen
78     self.optimo = optimo
79     self.replacement_ratio = replacement_ratio

```

Dado que ordenaremos la población desde el mejor individuo en orden decreciente del *fitness*, el método `best` nos permite obtener la mejor solución de la generación actual de la población:

```

81
82     def best(self):
83         """Return the best individual in the population."""

```

Para evolucionar la población utilizamos el método `run` que, mientras no se alcance una situación de finalización, invoca al método `step`. Las condiciones de finalización para este ejemplo son alcanzar el número máximo de iteraciones o el valor óptimo del *fitness*.

```

85
86     def run(self):
87         while not self.final():
88             self.step()
89
90     def final(self):

```

El método `step` mencionado arriba consiste en aplicar cada uno de los operadores genéticos selección, *crossover*, mutación, y reemplazo de la población en orden secuencial, e incrementar en uno la variable `self.generation`:

```

92
93     def step(self):
94         parents = self.selection()
95         children = self.do_crossover(parents)
96         self.do_mutation(children)
97         self.do_new_population(children)

```

```

98         self.generation += 1
99
100     def selection(self):
101         return [self.tournament() for _ in range(self.size)]
102
103     def tournament(self, size=3, goliat=0.9):
104         """Tournament selection."""
105         competidores = random.choices(self.individuals, k=size)
106         competidores.sort()
107         if random.random() < goliat:
108             return competidores[0]
109         else:

```

El operador de selección, implementado en el método `selection`, devuelve una lista de padres seleccionados mediante un torneo, conteniendo la misma cantidad de individuos que la población inicial. Es posible que los individuos con mejor *fitness* sean seleccionados múltiples veces en la lista de padres, pero el torneo permite, en forma aleatoria y con baja probabilidad, que cada tanto un individuo con un *fitness* no muy bueno integre la selección de padres, agregando de esta forma diversidad genética en la población. El torneo implementado en este ejemplo consiste en seleccionar aleatoriamente `size` individuos de la población (por defecto, `size` vale tres), y con una probabilidad definida en la variable `goliat` (con un valor por defecto del 90 %) gana el torneo el individuo con mayor *fitness*, y con una probabilidad de $(1 - \text{goliat})$ es seleccionado alguno de los restantes participantes del torneo.

El método `do_crossover` implementa el operador de recombinación de dos puntos. Comienza generando una lista vacía de hijos que se irá completando hasta alcanzar la cantidad de individuos originales en la población. El método selecciona al azar dos padres de la lista que constituye el argumento de `do_crossover` (generada por `selection`), y decide si los recombina con una probabilidad `self.crossover_p` (con un valor por defecto de 0.9), o si incluye en la lista de hijos los cromosomas de los padres sin recombinar. En el caso en que efectivamente se produzca la recombinación, se seleccionan al azar dos puntos en la cadena de cromosomas de los padres y se intercambia el segmento medio de genes entre estos puntos, dando origen a los nuevos cromosomas de los hijos (dos padres generan dos hijos).

```

111
112     def do_crossover(self, parents):
113         """Create a population of children from parents, maybe altering their chromosomes."""
114         children = []
115         crom_size = parents[0].size
116
117         for _ in range(self.size // 2):
118             parent1, parent2 = random.choices(parents, k=2)
119             if random.random() < self.crossover_p:
120                 new_chromol = parent1.chromosome.copy()
121                 new_chromo2 = parent2.chromosome.copy()
122
123                 # cross the "randomly central" part of one chromosome with the other
124                 x_i = random.randrange(0, crom_size - 1)
125                 x_d = random.randrange(x_i + 1, crom_size)
126                 new_chromol[x_i:x_d] = parent2.chromosome[x_i:x_d]
127                 new_chromo2[x_i:x_d] = parent1.chromosome[x_i:x_d]
128
129                 child1 = Individual(self.elements, chromosome=new_chromol)

```

```

130         child2 = Individual(self.elements, chromosome=new_chromo2)
131     else:
132         child1 = parent1
133         child2 = parent2
134     children.extend((child1, child2))

```

El operador de mutación se implementa en el método `do_mutation` de `Poblacion`. Simplemente, un bucle recorre la lista de hijos, y con una probabilidad dada por `self.mutation_p` decide mutar o no cada uno de ellos:

```

136
137     def do_mutation(self, children):
138         """Mutate some of the children."""
139         for child in children:
140             if random.random() < self.mutation_p:

```

Finalmente, el método `do_new_population` genera la nueva población implementando elitismo, esto es, determina qué fracción de la población debe ser reemplazada por los hijos en la generación siguiente. Mantiene entonces a los mejores padres y a los mejores hijos según sus *fitness*.

```

142
143     def do_new_population(self, children):
144         """Select a portion of the children into current population."""
145         children.sort()
146         cut_position = int(self.replacement_ratio * self.size)
147         self.individuals[self.size - cut_position:] = children[:cut_position]

```

Utilizamos ahora nuestro AG para resolver el ejemplo 2.1 del libro de Martello y Toth [47], que consiste en un conjunto de 8 elementos cuyos valores y pesos están definidos en la lista `elements` del código a continuación, y que tiene como solución óptima $x = (1, 1, 1, 1, 0, 1, 0, 0)$ con valor 280:

```

150
151 if __name__ == "__main__":
152     # Martello-Toth Example 2.1 pg 31: Optimo en {1, 1, 1, 1, 0, 1, 0, 0} valor=280 cap. 102
153     # conjunto de pares (valor, peso) para cada elemento a poner en la mochila
154     elements = [
155         (15, 2),
156         (100, 20),
157         (90, 20),
158         (60, 30),
159         (40, 40),
160         (15, 30),
161         (10, 60),
162         (1, 10),
163     ]
164     population = Population(elements=elements, size=20, max_gen=50)
165     population.run()
166     print("Mejor individuo:")

```

```
167 print(population.best().chromosome)
168 print("Valor:", population.best().fitness)
```

Al ejecutar este código obtenemos:

```
$ ./ag.py
Mejor individuo:
[1, 1, 1, 1, 0, 1, 0, 0]
Valor: 280.0
Peso: 102.0
```

Dado el carácter estocástico del AG, este método de optimización no garantiza que se obtenga el valor óptimo global del problema. Es posible que si ejecutamos nuevamente el programa, con otra semilla para el generador de números aleatorios, obtengamos otro resultado:

```
$ ./ag.py
Mejor individuo:
[1, 1, 1, 0, 1, 0, 0, 1]
Valor: 246.0
Peso: 92.0
```

Este efecto se vuelve más crítico cuanto mayor sea la dimensionalidad del problema, por lo que es recomendable realizar múltiples ejecuciones para observar la distribución de los resultados. Por otra parte, para cada problema constituye una cuestión casi artesanal la determinación de los parámetros de ejecución tales como el tamaño de la población, las probabilidades de recombinación y mutación, y la tasa de reemplazo. Cuando el problema es complejo, es conveniente también incorporar otras representaciones de los operadores genéticos. La experiencia en el uso de esta técnica resulta una buena guía para generar el algoritmo genético más eficiente para cada problema.

22.5. Lecturas recomendadas

- Stephen Boyd y Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. DOI: [10.1017/CB09780511804441](https://doi.org/10.1017/CB09780511804441). Se puede descargar desde <https://web.stanford.edu/~boyd/cvxbook/>.
- Wayne L Winston y Jeffrey B Goldberg. *Operations research: Applications and algorithms*. Brooks/Cole, 2004.
- Jorge Nocedal y Stephen Wright. *Numerical Optimization*. en. 2.^a ed. Springer Series in Operations Research and Financial Engineering. New York, NY: Springer, jul. de 2006.
- David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.
- Melanie Mitchell. *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996. DOI: <https://doi.org/10.7551/mitpress/3927.001.0001>.
- H. Kellerer, U. Pferschy y D. Pisinger. *Knapsack Problems*. Berlin: Springer-Verlag, 2004. DOI: [10.1007/978-3-540-24777-7](https://doi.org/10.1007/978-3-540-24777-7)



23 | Aprendizaje automático

En el contexto de la inteligencia artificial coexisten diversas metodologías tales como sistemas expertos, agentes racionales, procesamiento del lenguaje natural, visión de computadoras, robótica, y aprendizaje de máquinas o *machine learning* (ML). Particularmente esta última metodología ha mostrado un desarrollo impresionante de aplicaciones en los últimos años, debido principalmente al avance en los procesadores de cálculo especializados (GPUs¹ y TPUs²) y a la existencia sistematizada de grandes cantidades de información (*big data*).

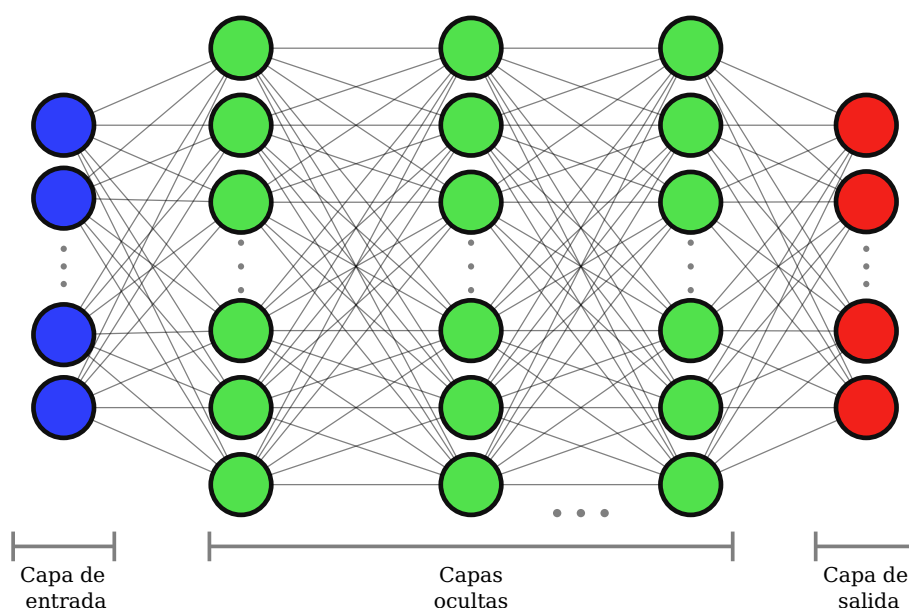
Entre las diferentes metodologías de ML se encuentran las redes neuronales (o NN por su sigla en inglés), cuyos orígenes se remontan al trabajo de McCulloch y Pitts de 1943 [53], quienes intentaron modelar las redes de neuronas en el cerebro mediante redes de cálculo computacional. Existen numerosas

Módulo	Versión
Keras	3.6.0
Matplotlib	3.9.2
NumPy	1.26.4
scikit-learn	1.5.2
tensorflow	2.18.0
tqdm	4.66.6

[Código disponible](#)

¹ *Graphics processing unit*, unidad de procesamiento gráfico.

² *Tensor processing unit*, unidad de procesamiento tensorial.

FIGURA 23.1: Representación de la estructura de una red *feedforward* multicapa.

topologías o “arquitecturas”³ de estas redes, que se especializan para realizar diferentes tareas de como ajuste de funciones, clasificación, identificación de patrones, detección de agregados (“*clustering*”), etc.

Analizaremos aquí un modelo simple de red del tipo *feedforward* (o de avance o propagación directa), en las cuales las conexiones entre distintas neuronas son solo en una dirección, formando de este modo un grafo acíclico entre las entradas y las salidas. Cada nodo de la red computa una función de sus entradas y pasa el resultado a las entradas de los nodos siguientes, sin generar ciclos. Por otra parte, existen otros tipos de redes como las recurrentes, en las cuales las entradas de los nodos se retroalimentan con sus propias salidas (este tipo de redes es apropiado para el ajuste de series temporales, por ejemplo).

23.1. Nodos, pesos y funciones de activación

Una red neuronal constituye un grafo en el que los nodos representan las “neuronas”, que son unidades de cálculo inspiradas en las neuronas biológicas. Estas últimas están conectadas unas a otras mediante axones y dendritas por medio de enlaces sinápticos que realizan la transmisión de impulsos nerviosos. Este proceso es el que se simula en las redes neuronales artificiales, en las que los nodos de cada capa están conectados a los nodos de la capa siguiente de la red a través de “pesos” que representan la intensidad de la conexión sináptica entre los nodos. Una red neuronal artificial calcula una función de los valores de entrada hacia los nodos de salida, utilizando estos pesos como parámetros de dicha función. Un esquema de esta estructura se ve en la Figura 23.1, en la que se muestra la capa de nodos de entrada (en azul), varias capas intermedias (u “ocultas”, en verde), y una capa de nodos de salida (en rojo). Las neuronas de cada capa están densamente conectadas con las capas vecinas.

³ Ver [The Neural Network Zoo](#) (en inglés).

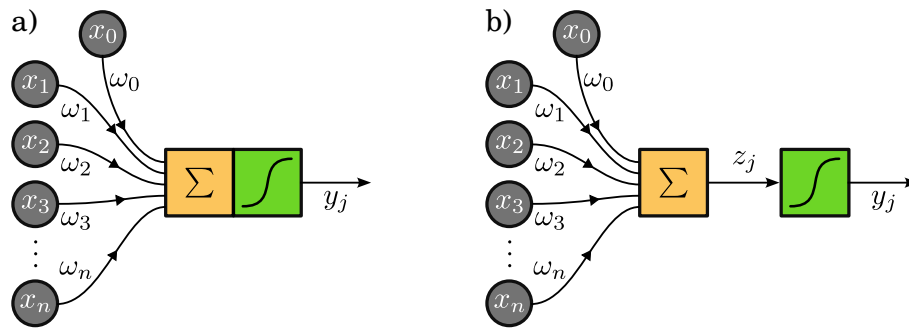


FIGURA 23.2: a) Esquema de un nodo o “neurona”. b) Esquema con la separación de operaciones indicando explícitamente el valor de preactivación z_j .

Cada nodo (o neurona) en la red computa una suma ponderada de las entradas de sus nodos predecesores, y a este resultado le aplica una función no lineal para producir la salida, tal como se representa en la Figura 23.2. Si denotamos por y_j la salida del nodo j , y $\omega_{i,j}$ el peso correspondiente a la conexión entre el nodo predecesor i y el j , la salida del nodo j da como resultado:

$$y_j = g_j \left(\sum_{i=0}^n \omega_{ij} x_i \right) = g_j(z_j)$$

donde $z_j = \sum_{i=0}^n \omega_{ij} x_i$ es la suma ponderada de los n nodos precedentes conectados con el nodo j , denominado “valor de activación”. Aquí, $g_j(\cdot)$ es alguna función no lineal denominada “función de activación”. Es usual asumir que $x_0 = 1$ por lo que ω_{0j} es un término de *bias* o sesgo, y es un parámetro que permite que los nodos de la red aprendan aún cuando todas las entradas sean nulas.

Si bien tanto la transformación lineal como la posterior evaluación de la función de activación son cálculos que se realizan en una neurona, es posible considerar un nodo con su correspondiente función de activación no lineal como dos nodos computacionales, uno que realiza la transformación lineal $z_j = \omega^T \cdot \mathbf{x}$, siendo ω el vector que representa los pesos que conectan las entradas $\mathbf{x} = [x_0, x_1, \dots, x_n]$ del nodo j (aquí denotamos las cantidades vectoriales con letras en negrita). En el ejemplo que desarrollaremos a continuación, usaremos este esquema desacoplado de cálculo, lo que simplificará la representación computacional de los nodos.

El hecho de que la función de activación sea no lineal es de central importancia, porque si fuese lineal, cualquier composición de nodos solo podría representar una función lineal. La no linealidad de g_j es lo que permite que redes suficientemente grandes puedan representar funciones arbitrarias⁴. Existe una variedad de funciones de activación que se usan con diferentes propósitos. Adicionalmente, tienen un impacto significativo en la velocidad de aprendizaje, factor que es uno de los principales criterios para su elección. Entre las más utilizadas están:

- La función logística o sigmoidea:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

⁴ Ver el teorema de aproximación universal [54, 55, 56].

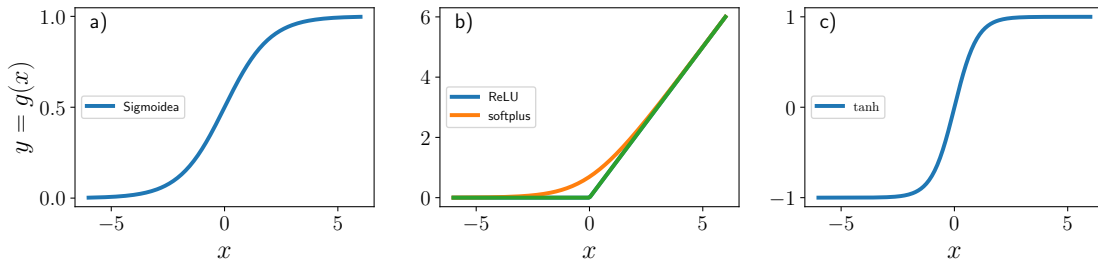


FIGURA 23.3: Funciones de activación: a) sigmoidea, b) ReLU y softplus y c) tangente hiperbólica.

- La función lineal rectificada (o ReLU, por su sigla en inglés *rectified linear unit*):

$$\text{ReLU}(x) = \max(0, x)$$

- La función *softplus*, que es una versión suavizada de la ReLU:

$$\text{softplus}(x) = \log(1 + e^x)$$

y cuya derivada es la función sigmoidea.

- La función tangente hiperbólica:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

La Figura 23.3 muestra las representaciones gráficas de estas funciones. Se puede apreciar que tanto la sigmoidea como la tangente hiperbólica acotan el recorrido a un intervalo finito: $(0, 1)$ para el caso de la sigmoidea y $(-1, 1)$ para la $\tanh$. Un comportamiento diferente tienen ReLU y softplus, ya que estas funciones no están acotadas. En particular, la función softplus tiene derivada continua en $x = 0$, a diferencia de ReLU. En todos los casos, las funciones son monótonamente crecientes, por lo que en todos sus dominios las derivadas correspondientes son positivas.

Una función de activación particular, dado que se utiliza casi exclusivamente en la capa de salida, es la *softmax* o función exponencial normalizada. Esta función es una generalización de la función logística que convierte un vector de k números reales en una distribución de probabilidad de k eventos discretos. Específicamente, si la entrada es un vector $\mathbf{z} = [z_1, z_2, \dots, z_k]$ que representa los valores de activación de los k nodos en una capa dada, la función de activación *softmax* para la salida i -ésima es:

$$\sigma(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}, \quad \forall i \in \{1, 2, \dots, k\}$$

Al utilizar un esquema desacoplado de cálculo, una capa que implementa solo la función *softmax* no requiere pesos que deban ajustarse, ya que solo transforma un vector de valores reales en probabilidades.

Una forma general de abordar el tratamiento matemático que realiza una red *feedforward* es la de considerar la red como grafo de cálculo o de “flujo de datos”, esto es, un grafo donde cada

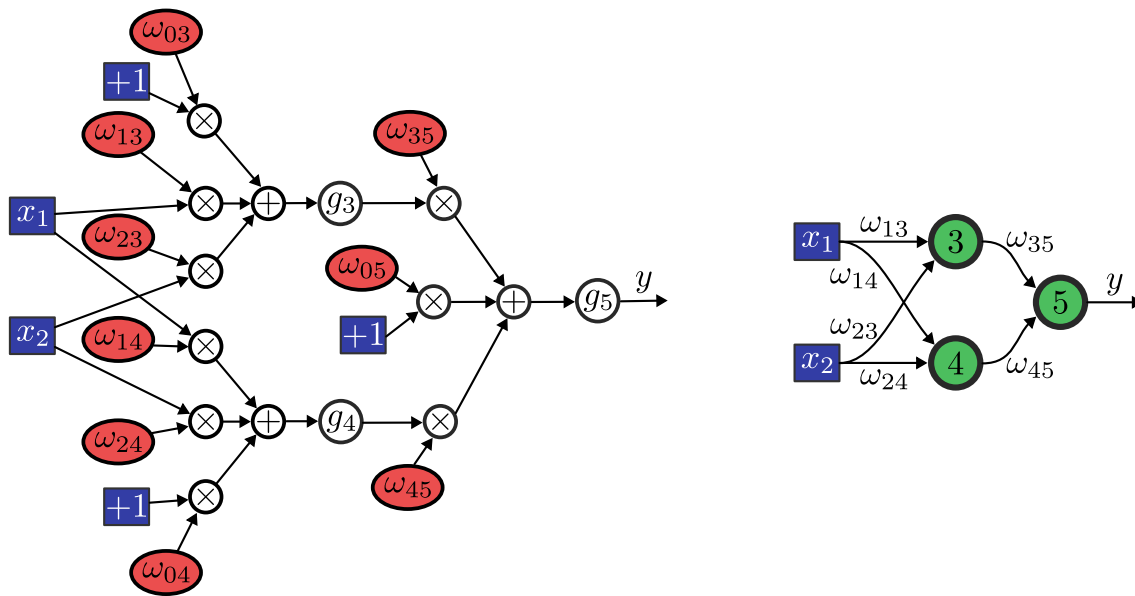


FIGURA 23.4: Flujo del cálculo de operaciones matemáticas en una red “*feedforward*”. A la izquierda, el detalle para una red con dos neuronas de entrada (x_1, x_2), dos neuronas en la capa oculta, con funciones de activación g_3 y g_4 , y una neurona en la capa de salida con función de activación g_5 que produce la predicción y . A la derecha, la forma simplificada usual de representación.

nodo representa un cálculo elemental y las uniones entre los nodos representan la relación de entrada/salida de los diferentes nodos. La Figura 23.4 muestra, a la izquierda, el flujo de cálculos que se realiza a partir de las entradas x_1 y x_2 (en azul) y que realizan los nodos 3, 4 y 5 utilizando los pesos ω_{ij} (en rojo), dando como resultado el valor y . A la derecha de la Figura 23.4 se representa la red en su forma simplificada habitual.

23.2. Aprendizaje

El procedimiento por el cual se modifican los valores de los pesos de una red neuronal, con el propósito de minimizar el error en sus predicciones, se denomina “aprendizaje”. Este procedimiento se inicia con un conjunto de datos de entrada a los que se expone la red, y a partir de las predicciones que obtiene de estas entradas se genera una retroalimentación que pueden clasificarse, en general, en tres tipos:

- **Aprendizaje supervisado.** Cada valor de entrada se acompaña de la correspondiente salida conocida (que usualmente se denomina “etiqueta”), y la red aprende una función que mapea la entrada con la salida. Este será el caso que abordaremos a continuación, en la que cada entrada es una imagen y la correspondiente salida es un dígito decimal.
- **Aprendizaje no supervisado.** En este caso, la red aprende a detectar patrones en las entradas sin la especificación de la retroalimentación. Uno de los ejemplos más usuales de este caso es el agrupamiento o *clustering*.
- **Aprendizaje por refuerzo.** Aquí se utiliza un sistema de refuerzo, por lo general un premio y castigo. Esta forma de aprendizaje es la típica de redes que aprenden a jugar, obteniendo

un premio al ganar el juego o un castigo al perderlo. De este modo, la red aprende de qué forma puede actuar para ganar más premios en el futuro.

Dado que este aprendizaje representa formalmente un proceso de optimización, podría usarse en principio cualquier algoritmo con ese propósito (por ejemplo, un algoritmo genético [57, 58, 59]). Sin embargo, en la práctica el método predominante para el entrenamiento de redes neuronales es el del descenso del gradiente (o alguna variante). En particular, examinaremos el descenso estocástico del gradiente (SGD por su sigla en inglés), que es un método iterativo que permite optimizar una función suficientemente suave (como las que mencionamos como funciones de activación y transformaciones lineales) por medio de una aproximación en la que se reemplaza el gradiente de la función calculado sobre el conjunto completo de datos por una estimación obtenida a partir de un subconjunto elegido al azar de dichos datos. Esta estrategia de optimización permite abordar problemas de alta dimensión (como el conjunto ω de parámetros de una red neuronal) disminuyendo el costo computacional, pagando el precio de una menor velocidad de convergencia. Una forma muy utilizada de implementar el método SGD es a través del algoritmo de propagación hacia atrás, o *backpropagation*, que describiremos a continuación.

Formalmente, el entrenamiento para el caso del aprendizaje supervisado se puede proponer del siguiente modo: dado el conjunto de entrenamiento con N muestras

$$X = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

donde cada par fue generado por una función desconocida $y = f(x)$, determinar una función $\hat{f}(X, \omega)$ que aproxime a f . Naturalmente, tanto x_i como y_i son usualmente cantidades vectoriales. Con este fin, se define una “función costo”⁵, C , que resulta de alguna medida de la diferencia entre los valores $\hat{y}_i = (y_1^L, y_2^L, \dots, y_{m_L}^L)$ que predice la red para una determinada entrada x_i y parámetros de pesos y sesgos ω . Aquí, L es el número de capas de la red (L es el índice de la capa de salida) y m_L es la cantidad de nodos que tiene la capa L . Claramente, la idea es que durante el aprendizaje de la red, los valores de esta función disminuyan progresivamente, dado que idealmente si las predicciones coinciden con los valores reales, la “distancia” entre ellas será nula. Según el problema que abordamos con la red (regresión, clasificación, salidas discretas o continuas), pueden elegirse distintas opciones para C , por ejemplo el valor absoluto de la diferencia entre el valor predicho y el valor verdadero $\|y - \hat{y}\|$ (conocida usualmente como la norma L^1), el error cuadrático $\|y - \hat{y}\|^2$, o la función de costo 0/1 que devuelve un valor 1 si la predicción es incorrecta y es útil para salidas con valores discretos 0 si $y = \hat{y}$, si no 1.



Alan Mathison Turing - Matemático, lógico, informático teórico, criptógrafo, filósofo y biólogo teórico británico (1912-1954), es considerado como uno de los padres de la ciencia de la computación y precursor de la informática moderna. Proporcionó una formalización influyente de los conceptos de algoritmo y computación: la máquina de Turing. En el campo de la inteligencia artificial, es conocido sobre todo por la concepción de la prueba de Turing (1950), un criterio según el cual puede juzgarse la inteligencia de una máquina si sus respuestas en la prueba son indistinguibles de las de un ser humano.

⁵ Es usual referirse a la función costo también como función de pérdida (*loss*) o función objetivo.

La función $\hat{f}$ que mencionamos anteriormente depende los valores de las entradas del conjunto de entrenamiento, x , y también de los valores de los pesos y sesgos ω . No obstante, durante el aprendizaje los valores tanto de las entradas como las etiquetas están fijos, por lo que solo tenemos la posibilidad de minimizar la función costo a través de ω . En lo siguiente, adoptaremos como C la función error cuadrático medio⁶:

$$C(X, \omega) = \frac{1}{2n} \sum_{i=1}^N \|\hat{y}_i - y_i\|^2 \quad (23.1)$$

La suma se realiza sobre cada caso individual del conjunto de entrenamiento y $\|\cdot\|$ es la norma L^2 o euclídea. Esta elección de la función costo según la expresión (23.1) cumple con una condición necesaria para poder utilizar el algoritmo de *backpropagation*, esto es, que C debe ser *aditiva*, ya que de este modo podemos escribir la función costo como un promedio de funciones costo para muestras de entrenamiento individuales $C(x_i)$, lo que nos permitirá calcular las derivadas parciales, y obtener luego el gradiente $\partial C / \partial \omega$ a partir de promedios sobre las muestras de entrenamiento.

Si bien esta función depende de los valores de y_i , recordemos que estos valores están fijos durante el entrenamiento, al igual que los valores de las entradas correspondientes x_i , por lo que estrictamente los valores de C quedan determinados por la elección de los pesos ω . De este modo, minimizar la función consiste en utilizar el descenso del gradiente para que iterativamente generemos una sucesión de valores de ω que disminuyan el valor de C :

$$\omega^{t+1} = \omega^t - \eta \frac{\partial C(X, \omega^t)}{\partial \omega} \quad (23.2)$$

donde ω^t denota los parámetros de la red neuronal en la iteración t del descenso del gradiente. η se denomina “tasa de aprendizaje” y determina el tamaño del paso que se da en la dirección del descenso del gradiente y por lo tanto condiciona la forma en que converge la minimización. El valor de η debe realizarse cuidadosamente para evitar inestabilidades en el proceso de aprendizaje, y usualmente está definido en el rango $0 < \eta < 1$.

Dado que la función costo se puede descomponer en una suma sobre términos de costo o error para cada par individual de entrada salida (x_i, y_i) , la derivada se puede calcular para cada par entrada/salida individualmente y luego combinar estos términos al final, ya que la derivada de una suma de funciones es la suma de las derivadas de cada función:

$$\frac{\partial C(X, \omega)}{\partial \omega_{ij}^k} = \frac{1}{N} \sum_{d=1}^N \frac{\partial}{\partial \omega_{ij}^k} \left(\frac{1}{2} (\hat{y}_d - y_d)^2 \right) = \frac{1}{N} \sum_{d=1}^N \frac{\partial C_d}{\partial \omega_{ij}^k}$$

donde ω_{ij}^k es el peso (o sesgo) para el nodo j de la capa l_k del nodo de entrada i en la capa l_{k-1} . De este modo, con el propósito de derivar el algoritmo de *backpropagation* vamos a considerar solo un par de entrada/salida. Luego de esta derivación, la forma general para todos los pares de entrada/salida del conjunto X puede ser generada combinando los gradientes individuales. Así, la función costo que utilizaremos es:

$$C = \frac{1}{2} \|\hat{y} - y\|^2$$

⁶ En el ejemplo de aplicación utilizaremos como función costo la función de entropía cruzada o *crossentropy*.

donde omitimos el subíndice d en C_d , $\hat{y}_d$ y y por simplicidad.

La derivación del algoritmo de *backpropagation* comienza aplicando la regla de la cadena a la derivada parcial de la función costo

$$\frac{\partial C}{\partial \omega_{ij}^k} = \frac{\partial C}{\partial z_j^k} \frac{\partial z_j^k}{\partial \omega_{ij}^k}$$

donde z_j^k es el valor de activación del nodo j de la capa k antes de pasar por la función de activación para generar la salida y_j^k . Esta descomposición de la derivada parcial establece que el cambio en la función costo debido a un peso es el producto del cambio de la función C debido a la activación z_j^k multiplicado por el cambio en la activación z_j^k debido al peso ω_{ij}^k . Generalmente se llama “error” al primer factor de la expresión anterior, y se denota por

$$\delta_j^k \equiv \frac{\partial C}{\partial z_j^k}$$

El segundo factor se puede calcular a partir de la definición del valor de activación:

$$\frac{\partial z_j^k}{\partial \omega_{ij}^k} = \frac{\partial}{\partial \omega_{ij}^k} \left(\sum_{l=0}^{m_{k-1}} \omega_{lj}^k y_l^{k-1} \right) = y_i^{k-1}$$

siendo m_{k-1} la cantidad de nodos en la capa $k - 1$. Entonces, la derivada parcial de la función costo C con respecto de un peso ω_{ij}^k es

$$\frac{\partial C}{\partial \omega_{ij}^k} = \delta_j^k y_i^{k-1}$$

Se puede ver entonces que la derivada parcial respecto a un peso es el producto del término de error δ_j^k del nodo j en la capa k , y de la salida y_i^{k-1} del nodo i de la capa $k - 1$. Esto resulta bastante intuitivo porque ω_{ij}^k conecta la salida del nodo i de la capa $k - 1$ con la entrada del nodo j de la capa k en el grafo computacional.

Comenzamos con el cálculo de errores de la última capa (L), que pueden ser evaluados directamente a partir de la “distancia” entre los valores predichos por la red y los correspondientes del conjunto de entrenamiento X . Para el nodo j en la capa de salida cuya función de activación es $g_o(z_j^L)$, tenemos:

$$C_j = \frac{1}{2} (\hat{y}_j - y_j)^2 = \frac{1}{2} (g_o(z_j^L) - y_j)^2$$

Aplicando la derivada parcial y usando la regla de la cadena resulta:

$$\delta_j^L = (g_o(z_j^L) - y_j) g'_o(z_j^L) = (\hat{y}_j - y_j) g'_o(z_j^L)$$

donde como es usual, g' es la derivada de g . La derivada parcial de C con respecto a un peso de la capa final es, entonces,

$$\frac{\partial C_j}{\partial \omega_{ij}^L} = \delta_j^L z_i^{L-1} = (\hat{y}_j - y_j) g'_o(z_j^L) y_i^{L-1}$$

Ahora podemos empezar a propagar hacia atrás los errores de la última capa, y para ello volvemos a utilizar la regla de la cadena para la derivada de funciones multivariadas. El término del error δ_k^k en la capa $1 \leq k \leq L$ es

$$\delta_j^k = \sum_{l=1}^{m_{k+1}} \frac{\partial C}{\partial z_l^{k+1}} \frac{\partial z_l^{k+1}}{\partial z_j^k}$$

donde l va desde 1 hasta m_{k+1} (el número de nodos de la capa siguiente). Notar que debido a que la entrada de sesgo y_0^k correspondiente a ω_{0j}^{k+1} es fijo, su valor no depende de las salidas de las capas previas, y por lo tanto l no toma el valor cero. Insertando el término de error δ_l^{k+1} obtenemos la siguiente ecuación:

$$\delta_j^k = \sum_{l=1}^{m_{k+1}} \delta_l^{k+1} \frac{\partial z_l^{k+1}}{\partial z_j^k}$$

A partir de la definición del valor de activación:

$$z_l^{k+1} = \sum_{j=1}^{m_k} \omega_{jl}^{k+1} g(z_j^k)$$

donde $g(\cdot)$ es la función de activación de las capas ocultas, tenemos

$$\frac{\partial z_l^{k+1}}{\partial z_j^k} = \omega_{jl}^{k+1} g'(z_j^k)$$

Reemplazando esta expresión en la ecuación previa obtenida para δ_j^k , resulta la derivada parcial de la función costo con respecto a un peso en las capas ocultas ω_{ij}^k para $1 \leq k < L$:

$$\frac{\partial C}{\partial \omega_{ij}^k} = \delta_j^k y_i^{k-1} = g'(z_j^k) y_i^{k-1} \sum_{l=1}^{m_{k+1}} \omega_{jl}^{k+1} \delta_l^{k+1}$$

Esta ecuación es el motivo del nombre del algoritmo de aprendizaje. El error δ_j^k en la capa k depende de los errores δ_l^{k+1} de la capa siguiente $k+1$. De este modo, los errores se propagan hacia atrás desde la última capa hacia la primera. El algoritmo comienza entonces calculando los errores de la última capa a partir de la salida predicha por la red $\hat{y}_j = g_o(z_j^L)$ y del valor de entrenamiento y_j . A partir de aquí, los términos de error de la capa previa son calculados realizando una suma de los errores recién calculados ponderada por los pesos ω_{jl}^{k+1} de los términos de error de la capa siguiente y escalados por $g'(z_j^k)$, y esto se repite hasta que se alcanza la primera capa.

Esta propagación hacia atrás de los errores es muy similar a la propagación hacia adelante que calcula la predicción de la red. Por esto, el cálculo de la salida de la red se suele llamar la “fase de propagación hacia adelante”, mientras que el cálculo de los errores y derivadas se denomina usualmente “fase de propagación hacia atrás”, o *backpropagation*. Durante la fase de avance, las entradas se combinan repetidamente desde la primer capa hasta la última por medio de las sumas ponderadas por los pesos ω_{ij}^k y las funciones de activación $g(x)$ y $g_o(x)$. En la fase de retroceso, las “entradas” son los términos de error de la capa final, que son repetidamente re combinados desde

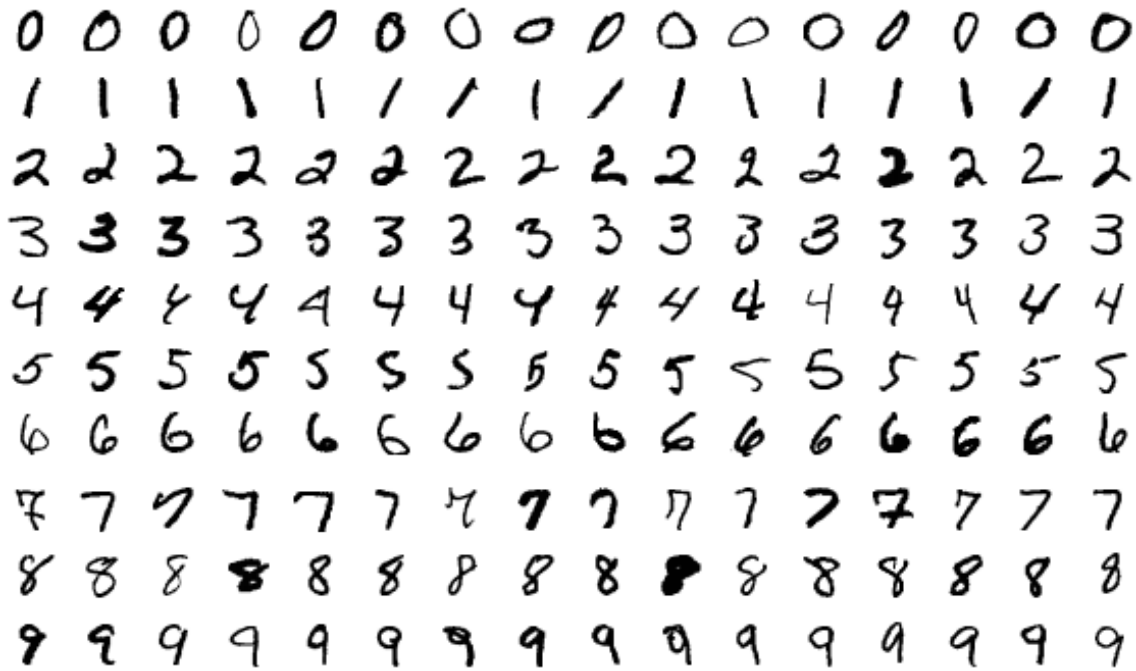


FIGURA 23.5: Ejemplo de imágenes del conjunto MNIST. Fuente: [Wikimedia Commons](#).

la última capa hacia la primera mediante sumas ponderadas por los pesos ω_{ij}^{k+1} y transformados por los factores de escala no lineales $g'_o(z_j^L)$ y $g'(z_j^k)$.

Con las expresiones obtenidas para el cálculo del gradiente de C con respecto de ω , podemos realizar la secuencia de actualizaciones de los pesos dada por la ecuación (23.2). Esta actualización no se realiza calculando el gradiente completo sobre todo el conjunto de entrenamiento X simultáneamente, sino que se realiza una estimación del gradiente utilizando un subconjunto de X elegido aleatoriamente, en lo que se denomina un *minibatch*. El uso de *minibatches* otorga eficiencia al método, dado que permiten un entrenamiento más rápido al calcular el gradiente y actualizar los pesos solo con una parte de los datos. Además, al promediar los gradientes de múltiples ejemplos, reduce el ruido y mejora la estabilidad del proceso de aprendizaje. No obstante, este método puede afectar a la precisión final de la red, ya que la actualización de los pesos se basa en una aproximación del gradiente total. El tamaño de la muestra que compone el minibatch es un parámetro importante que se debe ajustar para obtener un buen rendimiento.

Una vez que el conjunto X se particiona en subconjuntos aleatorios de muestras en *minibatch*, se llama una “era” a la secuencia de actualizaciones de los pesos en cada *minibatch*, hasta que se recorre el conjunto completo de aprendizaje. Luego, se iteran sucesivas eras hasta que la función costo alcanza un valor suficientemente bajo. En el ejemplo siguiente analizaremos una implementación práctica del algoritmo de *backpropagation* para el entrenamiento de una red *feedforward*.

23.3. Ejemplo de red usando NumPy

Vamos a poner en código los conceptos desarrollados para la red de propagación hacia adelante, entrenando una red para que pueda reconocer dígitos escritos a mano. Utilizaremos para ello el conjunto de datos MNIST⁷ que consiste en una colección de 60 000 imágenes de entrenamiento y 10 000 imágenes de prueba. Cada imagen tiene una resolución de 28×28 píxeles en valores que representan una escala de grises en el rango $[0, 255]$. Una muestra de las imágenes que componen el conjunto se puede ver en la Figura 23.5.

Como es usual, en las primeras líneas importamos los módulos necesarios para crear clases abstractas (abc); para interpretar los argumentos de la línea de comandos (argparse); para serializar y guardar objetos (pickle); para realizar conteos de frecuencia (Counter); los usuales para graficar (matplotlib) y para operaciones numéricas (numpy); y finalmente para visualizar una barra de progres (trange). A continuación establecemos el valor de la semilla del generador de números aleatorios de modo de poder reproducir los resultados.⁸

```

3 import abc
4 import argparse
5 import pickle
6 from collections import Counter
7
8 import matplotlib.pyplot as plt
9 import numpy as np
10 from tqdm import trange
11
12 # predictibilidad durante el desarrollo
13 np.random.seed(216091)

```

Comenzamos definiendo la función `load_dataset`, que como su nombre indica será la responsable de leer los datos desde un archivo csv (ver subsección 15.2.1.1). Este archivo contiene en cada fila una imagen, la primer columna de la fila es el dígito que representa dicha imagen (valor entre 0 y 9), y las restantes 784 columnas contiene el valor en escala de gris de cada píxel. Esta función devuelve dos arrays de $(N, 1)$ y $(N, 784)$ en `Y_train` y `X_train`, respectivamente, siendo N la cantidad de datos que contiene el *dataset*. El conjunto de entrada (`X_train`) se normaliza antes de devolver ambos arrays.

```

16 def load_dataset(datasource_path):
17     """Carga los datos de trabajo."""
18     # Lectura de datos
19     train_data = np.genfromtxt(datasource_path, delimiter=',', skip_header=1, dtype=float)
20     Y_train = train_data[:, 0].astype(int)
21     X_train = train_data[:, 1:]
22
23     # Normalizamos X
24     X_train = X_train / 255
25
26     return X_train, Y_train

```

⁷ Ver entrada en [Wikipedia](#). El conjunto MNIST se puede descargar desde [aquí](#).

⁸ Usamos para esto el trigésimo primer número perfecto.

La estructura de nuestra red consiste en una secuencia de capas que contienen a los nodos separados en dos unidades de cálculo: la que calcula los valores de activación z_i y la que representa las correspondientes funciones de activación. En este contexto, definimos primero una clase abstracta `Layer` que define los métodos que luego cada clase derivada debe implementar: `forward`, que realiza un paso de propagación hacia adelante, y `backward` que utilizaremos durante la fase de entrenamiento para implementar el algoritmo de *backpropagation*:

```

29 class Layer(abc.ABC):
30     """Capa abstracta.
31
32     Cada capa puede realizar dos tareas:
33
34     - Procesar la entrada para generar la salida: output = layer.forward(inputdata)
35
36     - Implementar backpropagation: grad_input = layer.backward(inputdata, grad_output)
37
38     Algunas capas tienen parámetros que se actualizan durante backpropagation (w, b).
39     """
40
41     @abc.abstractmethod
42     def forward(self, inputdata):
43         """Realiza un paso de propagación para adelante.
44
45         Recibe una entrada de shape [batch, input_nodes], y devuelve la salida con
46         shape [batch, output_nodes].
47         """
48
49     @abc.abstractmethod
50     def backward(self, inputdata, grad_output):
51         """Realiza un paso de backpropagation a través de la capa, respecto de la entrada.
52
53         Si la capa tiene parámetros (capa densa), se actualiza en este paso.
54         """

```

Para todas las capas ocultas utilizaremos ReLU como función de activación. Los nodos de esta capa no contienen información sobre los pesos de las conexiones sinápticas, ya que solamente toman como entrada el valor de activación calculado en la capa previa y transforma este valor según la función ReLU. Esto es lo que implementa el método `forward`. Para el cálculo de la propagación hacia atrás del error, esto simplemente es el error de la capa siguiente multiplicado por la derivada de la función ReLU (que es 1 si $x > 0$, y 0 para $x \leq 0$). Este es el cálculo que realiza el método `backward`.

```

57 class ReLU(Layer):
58     """Capa ReLU que aplica una unidad lineal rectificadora elemento por elemento."""
59
60     def forward(self, inputdata):
61         """Aplica ReLU elemento por elemento a la matriz [batch, input_nodes]."""
62         relu_forward = np.maximum(0, inputdata)
63         return relu_forward
64
65     def backward(self, inputdata, grad_output):
66         """Calcula el gradiente de la función de costo respecto de la entrada ReLU."""

```

```

67     relu_grad = inputdata > 0
68     return grad_output * relu_grad

```

La clase siguiente permite instanciar capas cuyos nodos están todos conectados con todos los nodos de la capa anterior, y por eso se denomina Dense. A diferencia de la capa ReLU, esta capa contiene los valores ω_{ij}^l (`self.W`) que representan los pesos de las conexiones entre los nodos i de la capa $l - 1$ con los nodos j de la capa l . Por lo tanto esta capa contiene un método adicional (`__init__`) que permite inicializar estos valores (utilizamos para ello el método de inicialización de Xavier[60], que asigna valores en un rango razonable de valores obtenidos de una distribución normal con media cero y varianza que depende de la cantidad de conexiones de entrada y salida). Además, en esta capa separamos explícitamente el valor de los sesgos o *bias* (`self.bias`). El método `forward` realiza el cálculo del valor de activación por medio del producto matricial del array con los valores de entrada y los pesos, sumando los valores de los sesgos. Por su parte, el método `backward` no solo calcula el gradiente de la función costo respecto de los pesos, sino que también implementa la actualización de los mismos con el método del descenso del gradiente incorporando el correspondiente parámetro η (`self.learning_rate`).

```

71 class Dense(Layer):
72     """Capa densa; la salida es  $W * x + b = z$ ."""
73
74     def __init__(self, input_nodes, output_nodes, learning_rate=0.1):
75         self.learning_rate = learning_rate
76         self.W = np.random.normal(
77             loc=0.0,
78             scale=np.sqrt(2 / (input_nodes + output_nodes)),
79             size=(input_nodes, output_nodes))
80         self.bias = np.zeros(output_nodes)
81
82     def forward(self, inputdata):
83         """Calcula el producto matricial entre la entrada y los pesos, más los bias."""
84         return np.dot(inputdata, self.W) + self.bias
85
86     def backward(self, inputdata, grad_output):
87         """Calcula  $df / dx = df / d[capa] d[capa] / dx$  donde  $d[capa] / dx = W.T$ ."""
88         grad_input = np.dot(grad_output, self.W.T)
89         # cálculo del gradiente respecto de W y b
90         grad_W = np.dot(inputdata.T, grad_output)
91         grad_b = grad_output.mean(axis=0) * inputdata.shape[0]
92         assert grad_W.shape == self.W.shape and grad_b.shape == self.bias.shape
93         # Actualización de W y b por medio del descenso del gradiente.
94         self.W = self.W - self.learning_rate * grad_W
95         self.bias = self.bias - self.learning_rate * grad_b
96         return grad_input

```

En nuestro ejemplo, la última capa será una instancia de Dense, por lo que su salida consiste en los valores de activación que genera esta clase sin pasar por una función de activación. Llamaremos *logits* a estos valores (o vector logit como se lo utiliza habitualmente), que serán utilizados para predecir el valor numérico de la imagen de entrada como una medida de probabilidad. En este contexto utilizaremos una función de activación softmax para la última capa junto con una función costo del tipo entropía cruzada, o *crossentropy*, que se usa frecuentemente para estimar

la diferencia entre dos distribuciones de probabilidad y se basa en el concepto de entropía de la información propuesta por Shannon [61].

Llamando l_i a la componente i -ésima del vector logit, y p_i al valor correspondiente luego de pasar por la función softmax, la función costo es:

$$C(\mathbf{y}, \mathbf{l}) = - \sum_i y_i \left[l_i + \log \left(\sum_j e^{l_j} \right) \right]$$

Esta expresión se simplifica si codificamos las etiquetas (dígitos entre cero y nueve) en forma de un vector binario que tiene un 1 en la posición que corresponde a cada dígito (conocido también como *one-hot encoding*, o OHE). Por ejemplo, si el valor de la etiqueta es 4, la codificamos en forma OHE como el vector (0, 0, 0, 0, 1, 0, 0, 0, 0, 0), es decir, $y_l = 1$ para $l = 4$ y cero en otro caso. De este modo, combinamos el cálculo de la función softmax sobre el vector logit de la última capa con la función de pérdida de *crossentropy*, y obtenemos para cada nodo j de la última capa:

$$C_j = -l_j + \log \left(\sum_i e^{l_i} \right)$$

La función `softmax_crossentropy_with_logits` implementa este cálculo, devolviendo un array (`xentropy`) que contiene los valores de C_j para cada nodo j de la última capa. El correspondiente cálculo del gradiente de esta función costo se realiza en la función

`grad_softmax_crossentropy_with_logits`:

```

99 def softmax_crossentropy_with_logits(logits, reference_labels):
100     """Cálculo de crossentropy.
101
102     Se realiza a partir de logits[batch, n_clases] y las etiquetas de los datos.
103     """
104     logits_for_labels = logits[np.arange(len(logits)), reference_labels]
105     xentropy = - logits_for_labels + np.log(np.sum(np.exp(logits), axis=-1))
106     return xentropy
107
108
109 def grad_softmax_crossentropy_with_logits(logits, reference_labels):
110     """Cálculo del gradiente de crossentropy.
111
112     Se realiza a partir de logits[batch, n_clases] y las etiquetas de los datos.
113     """
114     ones_for_labels = np.zeros_like(logits)
115     ones_for_labels[np.arange(len(logits)), reference_labels] = 1
116     softmax = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)
117     return (- ones_for_labels + softmax) / logits.shape[0]
```



Claude Elwood Shannon - Matemático, ingeniero eléctrico y criptógrafo estadounidense (1916-2001). Reconocido por haber fundado el campo de la teoría de la información y por haber sentado las bases de la teoría del diseño de circuitos digitales. Demostró que las aplicaciones electrónicas de álgebra booleana podrían construir cualquier relación lógico-numérica.

Con todo lo anterior ya tenemos lo necesario para crear una red. Esto lo hacemos a partir de la clase `Network`, que básicamente consiste en una secuencia (en forma de lista) sobre la que actuarán los correspondientes métodos `_forward`, para propagar los cálculos hacia adelante a partir de los datos de entrada X , `predict` para obtener la predicción de la red ante una entrada determinada, el método `train` para entrenar la red a partir de pares (X, Y) , implementando *backpropagation*, y dos métodos más: `dump` para guardar los parámetros de una red entrenada y `load` para recuperar dichos parámetros desde un archivo en disco y crear una red operativa para realizar predicciones.

Declaramos la clase `Network` junto con el método `__init__`, que recibe una lista (`layers`) que contiene objetos instanciados a partir de las clases `ReLU` y `Dense`. A continuación, el método `_forward` calcula los valores de activación de cada capa, secuencialmente a partir de la capa de entrada, y devuelve estos valores en el array `activations`:

```

120 class Network:
121     """Opera sobre una secuencia de capas."""
122
123     def __init__(self, layers):
124         self.layers = layers
125
126     def _forward(self, inputdata):
127         """Calcula las activaciones de todas las capas, secuencialmente.
128
129         La salida de una capa es la entrada de la próxima. Devuelve una lista con
130         las activaciones de cada capa.
131         """
132         activations = []
133         for layer in self.layers:
134             inputdata = layer.forward(inputdata)
135             activations.append(inputdata)
136
137         return activations

```

El método `predict` recibe una entrada X y propaga los cálculos hacia adelante en la red hasta la última capa, donde se calcula el vector logit. La predicción de la red resulta de la posición de este vector que tiene el máximo valor (recordar que usamos OHE para codificar los dígitos, por lo que si el valor máximo del vector logit ocurre en la sexta posición, la predicción es el dígito 5). No hace falta aplicar una función de activación sobre los logits, como softmax, dado que por ser una función estrictamente no negativa, solo realiza un cambio de escala pero no altera la posición del máximo:

```

139 def predict(self, X):
140     """Calcula la predicción de la red.
141
142     Devuelve el índice del mayor valor de probabilidad en logits.
143     """
144     layer_activations = self._forward(X)
145     logits = layer_activations[-1]
146     return logits.argmax(axis=-1)

```

El método `train` realiza secuencialmente los pasos que detallamos del algoritmo *backpropagation*. Recibe como argumentos una entrada X y su correspondiente etiqueta (el valor del dígito

al que corresponde la imagen) y . En primer lugar realiza el avance hacia adelante en la red de los cálculos que llevan desde la entrada a las activaciones de la última capa (vector logits). Luego de esto le agregamos al principio el propio array X para tenerlo como entrada de la primera capa. A continuación obtenemos el array `loss`, que corresponde al valor del costo de cada nodo de la capa de salida⁹, así como su gradiente, `loss_grad`, de la última capa, y luego mediante un bucle `for` recorreremos en forma inversa la lista que contiene cada capa de la red, invocando el método `backward` de cada capa que propaga hacia atrás los gradientes y actualiza los pesos y sesgos en las capas del tipo `Dense` (las capas `ReLU` no contienen parámetros para optimizar, ya que simplemente implementan la función no lineal de activación). Este método devuelve el valor medio de las funciones de costo de todos los nodos de la última capa.

```

148 def train(self, X, y):
149     """Entrena a la red con X e y."""
150     # Etapa de avance de activaciones (forward)
151     layer_activations = self._forward(X)
152
153     # Agregamos al principio la entrada original para tener la entrada de cada capa
154     # y separamos las últimas activaciones como logits
155     *layer_inputs, logits = [X] + layer_activations
156
157     # Cálculo de la función de pérdida (indicativo de lo bien entrenado)
158     # para devolver al final
159     loss = softmax_crossentropy_with_logits(logits, y)
160
161     # Comenzamos la propagación de gradientes para atrás en la red (backpropagation)
162     # por la última capa especial de entropía cruzada
163     loss_grad = grad_softmax_crossentropy_with_logits(logits, y)
164
165     # Seguimos la propagación para atrás por cada capa
166     for layer, originalinput in reversed(list(zip(self.layers, layer_inputs))):
167         loss_grad = layer.backward(originalinput, loss_grad)
168
169     return np.mean(loss)

```

Por último, agregamos los métodos `dump` y `load`. El primero recibe como argumento una cadena que será el nombre del archivo donde se guarda la instancia de la clase `Network` que contiene todos los parámetros luego del entrenamiento, de forma de poder recuperarlos después con el método `load` y poder utilizar la red sin tener la necesidad de pasar nuevamente por el costoso proceso de entrenamiento:

```

171 def dump(self, filepath):
172     """Guarda la red actual en un archivo."""
173     with open(filepath, "wb") as fh:
174         pickle.dump(self, fh)
175
176     @classmethod
177     def load(cls, filepath):
178         """Recupera de disco una instancia de la red."""
179         with open(filepath, "rb") as fh:
180             return pickle.load(fh)

```

⁹Recordemos que a esta función también es habitual llamarla función objetivo o función de pérdida (*loss*).

Usamos el decorador `@classmethod` en load de forma de poder instanciar objetos de la clase Network directamente a partir de la estructura de la red y los valores de los parámetros almacenados en filepath (ver la sección 5.5).

El método definido a continuación, `get_minibatches`, implementa la división del conjunto completo de entrenamiento en pequeñas tandas que permiten acelerar el cálculo, a partir de estimar el gradiente de la función de costo con muestras más pequeñas del conjunto completo (X, Y). Recibe como argumentos las entradas X (inputs), las etiquetas Y (targets), el tamaño de la tanda de datos a utilizar en cada etapa de entrenamiento, y una variable booleana `shuffle` que si tiene el valor `True`, realiza una permutación aleatoria de los índices de ambos conjuntos de modo de acceder aleatoriamente a estos datos. Esta función está implementada a modo de generador, que va produciendo estos los índices a medida que se van requiriendo:

```

183 def get_minibatches(inputs, targets, batchsize, shuffle=False):
184     assert len(inputs) == len(targets)
185     if shuffle:
186         indices = np.random.permutation(len(inputs))
187         for start_idx in trange(0, len(inputs) - batchsize + 1, batchsize):
188             if shuffle:
189                 excerpt = indices[start_idx:start_idx + batchsize]
190             else:
191                 excerpt = slice(start_idx, start_idx + batchsize)
192             yield inputs[excerpt], targets[excerpt]

```

La función `train` es la que realiza el entrenamiento de la red y muestra el grado de precisión que obtiene a medida que progresa. Comienza cargando los datos de las entradas y las etiquetas desde un archivo, y separa estos datos en un conjunto que se utilizará para el entrenamiento (X_{train} y Y_{train}) de otros que se utilizarán como validación (X_{val} , y_{val}), que contienen solo un sexto del total de los datos. Esta validación será útil para cuantificar el desempeño de la red prediciendo a partir de datos que no fueron utilizados durante la etapa de entrenamiento:

```

195 def train(datasource_path):
196     # Cargamos los datos y reservamos los últimos ejemplos de entrenamiento para validación
197     X_train, Y_train = load_dataset(datasource_path)
198     limit = len(X_train) // 6
199     X_train, X_val = X_train[:-limit], X_train[-limit:]
200     Y_train, y_val = Y_train[:-limit], Y_train[-limit:]

```

Luego instanciamos un objeto de la clase Network, como una secuencia de una capa Dense cuya entrada tiene la misma dimensión que los datos de entrada X (en este caso, un array de $28 \times 28 = 784$ elementos) conectados a 100 nodos, la segunda capa es la capa ReLU que recibe la salida de las activaciones de la capa anterior y calcula las correspondientes funciones de activación, luego otra capa Dense, ahora con 200 nodos, su correspondiente capa ReLU, y la capa final Dense con las 10 unidades que generan el vector logit:

```

202 network = Network([
203     Dense(X_train.shape[1], 100),
204     ReLU(),
205     Dense(100, 200),

```

```

206         ReLU(),
207         Dense(200, 10),
208     ])

```

A continuación creamos dos listas vacías en las que iremos guardando los resultados de las precisiones obtenidas en cada época de entrenamiento (recordemos que una época consiste en recorrer todos los datos de entrenamiento mediante minibatches), y luego generamos el entrenamiento propiamente dicho en un número fijo de 25 épocas, informando el progreso de la función junto con las precisiones obtenidas en el conjunto de entrenamiento y el de validación, entendiendo estas precisiones como el número medio de aciertos de la red en ambos casos. Al finalizar el recorrido por las épocas, guardamos en un archivo la red entrenada y generamos un gráfico que muestra la evolución de las precisiones obtenidas en los conjuntos de entrenamiento y validación.

```

210     train_log = []
211     val_log = []
212
213     for epoch in range(25):
214         total_loss = 0
215         for x_batch, y_batch in get_minibatches(X_train, Y_train, batchsize=32, shuffle=True):
216             loss = network.train(x_batch, y_batch)
217             total_loss += loss
218
219         train_log.append(np.mean(network.predict(X_train) == Y_train))
220         val_log.append(np.mean(network.predict(X_val) == y_val))
221
222         print(f"Época {epoch}")
223         print(f"Precisión de entrenamiento: {train_log[-1]:8.4f}")
224         print(f"    Precisión de validación: {val_log[-1]:8.4f}")
225         print(f"                Total pérdida: {total_loss:8.4f}")
226
227     # guardar la red con su estado actual entrenado
228     network.dump("trained.pkl")
229
230     plt.plot(train_log, label='Precisión de entrenamiento')
231     plt.plot(val_log, label='Precisión de validación')
232     plt.xlabel("Época")
233     plt.legend(loc='best')
234     plt.grid()
235     plt.show()

```

La función `show_images` tiene la utilidad de generar imágenes a partir de un conjunto de entrada, con el solo propósito de visualizar dichos datos. Es útil para comprender la tarea de clasificación que le encomendaremos a la red neuronal:

```

238 def show_images(datasource_path):
239     """Muestra un ejemplo de las imágenes usadas como datos fuente."""
240     X_train, Y_train = load_dataset(datasource_path)
241
242     # reshape the sequence of bits so each 784 chunk is 28x28 (square image)
243     X_train = X_train.reshape(-1, 28, 28)
244

```

```

245 plt.figure(figsize=[6, 6])
246 for i in range(4):
247     plt.subplot(2, 2, i + 1)
248     plt.title(f"Label: {Y_train[i]}")
249     plt.imshow(X_train[i].reshape([28, 28]), cmap='gray')
250 plt.show()

```

Para tener una estimación cuantitativa del grado de precisión de la red, la función `evaluate` recibe como argumento una cadena (`datasource_path`) que apunta a un archivo que contiene pares de entrada y etiquetas: `X_test` y `Y_test`, respectivamente. Luego instancia una red a partir de un archivo donde se ha guardado previamente una red entrenada y predice los valores correspondientes a las entradas. Finalmente muestra un histograma con los porcentajes de aciertos que ha obtenido, a partir de los valores reales de las etiquetas.

```

253 def evaluate(datasource_path):
254     """Evalúa una red entrenada sobre un conjunto de datos de prueba."""
255     print("Cargando red")
256     network = Network.load("trained.pkl")
257     print("Cargando dataset de prueba")
258     X_test, Y_test = load_dataset(datasource_path)
259     print("Prediciendo")
260     predicted = network.predict(X_test)
261     print("Listo")
262
263     cnt = Counter(zip(Y_test, predicted == Y_test))
264
265     digits = []
266     pred_perc = []
267     for digit in range(10):
268         digits.append(digit)
269         quant_ok = cnt[(digit, True)]
270         quant_bad = cnt[(digit, False)]
271         pred_perc.append(100 * quant_ok / (quant_ok + quant_bad))
272
273     fig, ax = plt.subplots()
274     ax.bar(digits, pred_perc, 0.7)
275     ax.set_title("Precisión de las predicciones")
276     ax.set_xticks(digits)
277     ax.set_ylim([90, 100])
278     ax.set_yticks(ticks=range(90, 101), labels=[f"{n}%" for n in range(90, 101)])
279     plt.show()

```

Por último, generamos un diccionario (`_actions`) que define diversas acciones que se pueden realizar con este código, en forma de pares función/datos. Las claves de este diccionario son las opciones de argumentos en línea de comando con los que se ejecutará el código. Se pueden realizar las siguientes opciones:

- `devtrain`: realiza un entrenamiento de la red con un conjunto reducido de datos (`mnist_train_dev.csv`), con el propósito de usarlo para pruebas de depuración del código.
- `train`: realiza el entrenamiento sobre el conjunto completo de datos de MNIST (`mnist_train.csv`).

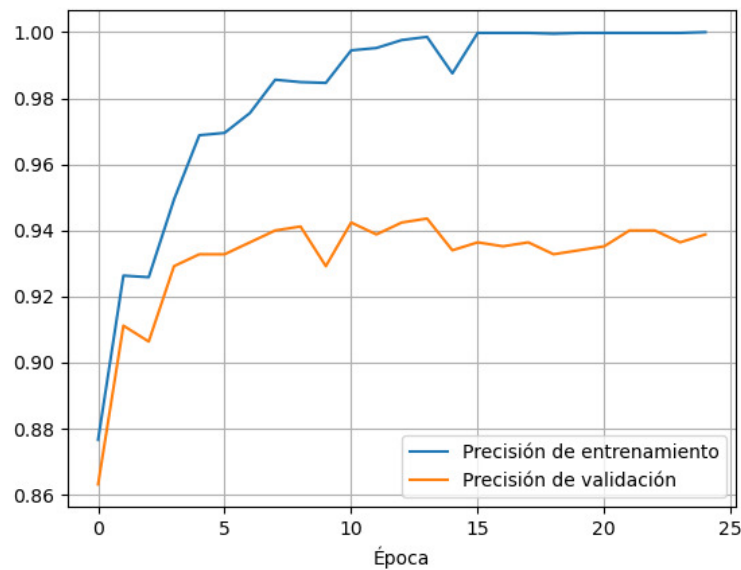


FIGURA 23.6: Evolución de la precisión en el entrenamiento utilizando el conjunto reducido de datos

- `show`: invoca a la función `show_images` sobre un conjunto reducido de datos para visualizar las imágenes que contiene.
- `evaluate`: ejecuta la función `evaluate` sobre el conjunto de prueba `mnist_test.csv` para cuantificar la precisión del entrenamiento realizado sobre la red.

Estas opciones se interpretan con el módulo `argparse`, y según el argumento pasado en la línea de órdenes ejecuta cada opción:

```

282 _actions = {
283     "devtrain": [train, "mnist_train_dev.csv"],
284     "train": [train, "mnist_train.csv"],
285     "show": [show_images, "mnist_train_dev.csv"],
286     "eval": [evaluate, "mnist_test.csv"],
287 }
288 parser = argparse.ArgumentParser()
289 parser.add_argument("action", choices=_actions.keys(), help="Qué acción realizar")
290 args = parser.parse_args()
291 func, *params = _actions[args.action]
292 func(*params)

```

Entonces, para el caso de la opción `devtrain`, que contiene solo 5000 datos, obtenemos una precisión final de 1,0000 para los datos de entrenamiento y 0,9388 para los de validación. La evolución del entrenamiento se puede ver en la Figura 23.6.

Si ahora realizamos el entrenamiento sobre el conjunto completo de datos de MNIST, usando

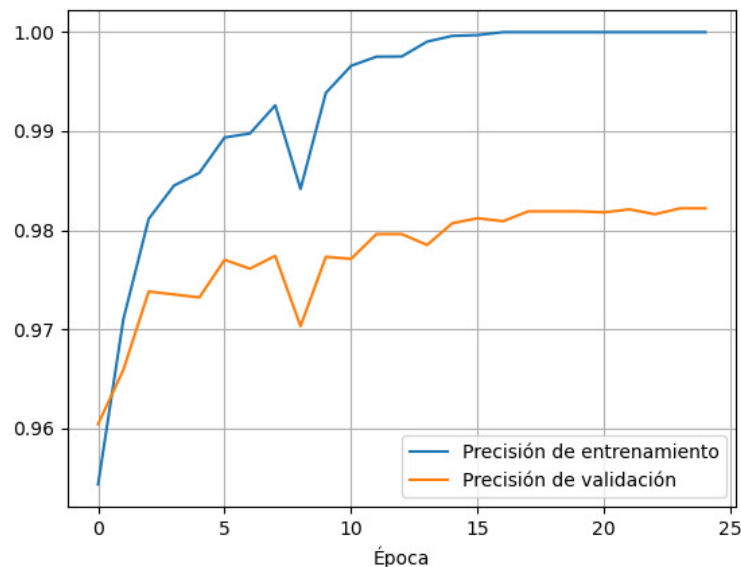


FIGURA 23.7: Evolución de la precisión en el entrenamiento utilizando el conjunto completo de datos MNIST

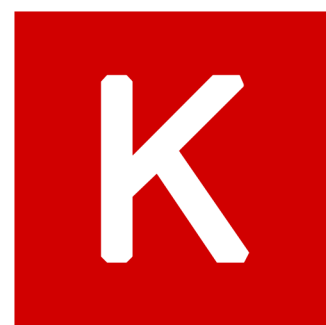
```
^^I./nn-mnist.py train
```

vemos que la precisión sobre el conjunto de validación mejora hasta el 0,9822, tal como muestra la Figura 23.7.

Por último, evaluaremos la precisión alcanzada por cada dígito del conjunto reducido de datos, con la opción `eval` en la línea de comandos. Tal como muestra la Figura 23.8, vemos que el peor desempeño sucede con la identificación de los dígitos 5 y 4, pero en todos los casos la tasa de aciertos es superior al 97 %.

23.4. Ejemplo usando Keras

En la sección anterior implementamos una red *feedforward* con herramientas básicas de Python (NumPy, Matplotlib) con la idea de desmenuzar las operaciones básicas involucradas en el entrenamiento y uso de la red. Afortunadamente existen bibliotecas que permiten abstraer los detalles de cálculo y facilitan la construcción de estructuras complejas con gran variedad de herramientas (distintos nodos de cálculo, funciones de activación, métodos de optimización, etc.), y también la utilización de *hardware* específico para estas aplicaciones (GPUs, TPUs). Estas bibliotecas mantienen un desarrollo muy activo en los últimos años, acompañado al explosivo crecimiento de las aplicaciones



Keras - Biblioteca de alto nivel para construir y entrenar redes neuronales de manera sencilla e intuitiva.

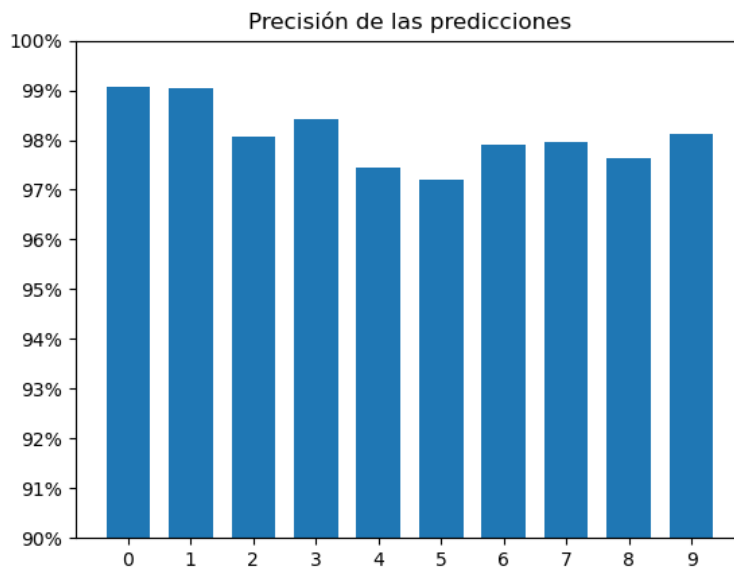


FIGURA 23.8: Porcentaje de aciertos de la red sobre el conjunto reducido de datos.

de las técnicas de las redes neuronales de la actualidad.

A continuación abordaremos la identificación de las imágenes del conjunto MNIST por medio de la biblioteca [Keras](#), que a su vez se ejecuta sobre la biblioteca [TensorFlow](#), en este caso usando un *notebook* de Jupyter.

Comenzamos importando numpy y fijando una semilla para los números al azar, con el propósito de tener repetibilidad en los cálculos:

CELDA 1 DE NN-MNIST-KERAS.IPYNB

```
import numpy as np
np.random.seed(216091)
```

En la segunda celda vemos lo simple que resulta acceder a los datos del conjunto MNIST, y además contamos cuántas valores tienen los conjuntos de entrenamiento y de prueba para cada etiqueta:

CELDA 02

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Contamos el número de etiquetas de entrenamiento
unique, counts = np.unique(y_train, return_counts=True)
print("\nEtiquetas de entrenamiento: ", dict(zip(unique, counts)))
# Contamos el número de etiquetas de prueba
unique, counts = np.unique(y_test, return_counts=True)
print("\nEtiquetas de prueba: ", dict(zip(unique, counts)))
```

Etiquetas de entrenamiento: {0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7:
↳ 6265, 8: 5851, 9: 5949}

Etiquetas de prueba: {0: 980, 1: 1135, 2: 1032, 3: 1010, 4: 982, 5: 892, 6: 958, 7: 1028, 8: 974,
↳ 9: 1009}

Para ver cómo son las imágenes que la red tratará de identificar, elegimos 25 muestras aleatoriamente del conjunto de entrenamiento y las visualizamos:

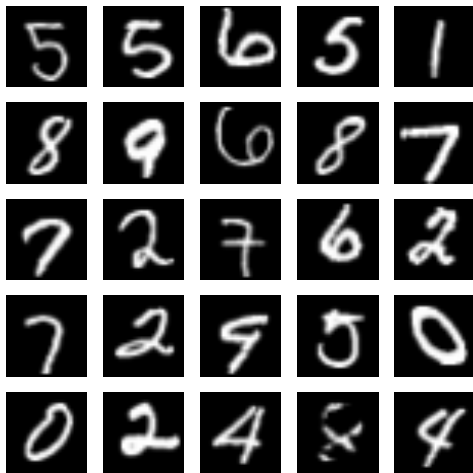
CELDA 03

```
import matplotlib.pyplot as plt

# Visualizamos 25 muestras del conjunto de entrenamiento
indexes = np.random.randint(0, x_train.shape[0], size=25)
images = x_train[indexes]
labels = y_train[indexes]

plt.figure(figsize=(2,2))
for i in range(len(indexes)):
    plt.subplot(5, 5, i + 1)
    image = images[i]
    plt.imshow(image, cmap='gray')
    plt.axis('off')

plt.show()
plt.close('all')
```



Al igual que hicimos en la sección anterior, convertimos las etiquetas que tenemos en formato de dígito (0, 1, ..., 9) en un array codificado en forma *one-hot encoding* (OHE):

CELDA 04

```
from keras.utils import to_categorical

# Convertimos a un array OHE
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

En las dos celdas siguientes obtenemos la dimensión del array que representa la imagen de entrada, y adaptamos los arrays de entrada de una imagen cuadrada de bits en el rango [0, 255] a un array unidimensional cuyos valores están normalizados:

CELDA 05

```
# Calculamos la dimensión del array que representa la imagen de entrada
# (asumimos que es cuadrada)
image_size = x_train.shape[1]
input_size = image_size * image_size
input_size
```

784

CELDA 06

```
# Cambiamos la forma de los arrays de entrada y normalizamos
x_train = np.reshape(x_train, [-1, input_size])
x_train = x_train.astype('float32') / 255
x_test = np.reshape(x_test, [-1, input_size])
x_test = x_test.astype('float32') / 255
```

A continuación construimos una red con la misma estructura que usamos en la sección anterior, es decir, una capa de entrada, dos capas densas con activación ReLU con 100 y 200 nodos, respectivamente, y una capa de salida de 10 nodos con activación softmax. Para ello instanciamos un objeto `model` de la clase `Sequential`:

CELDA 07

```
from keras.models import Sequential
from keras.layers import Input, Dense, Activation

# La red (modelo) consiste una capa de entrada, dos capas con activación ReLU
# y la capa de salida que contiene solo los valores de activación
model = Sequential()
model.add(Input(shape=(input_size,)))
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dense(200))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

Podemos invocar el método `summary()` para ver los detalles de la red, particularmente la cantidad de parámetros que se ajustarán en la fase de entrenamiento:

CELDA 08		
<code>model.summary()</code>		
Model: "sequential"		
Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	78,500
activation (Activation)	(None, 100)	0
dense_1 (Dense)	(None, 200)	20,200
activation_1 (Activation)	(None, 200)	0
dense_2 (Dense)	(None, 10)	2,010
activation_2 (Activation)	(None, 10)	0
Total params: 100,710 (393.40 KB)		
Trainable params: 100,710 (393.40 KB)		
Non-trainable params: 0 (0.00 B)		

Esta celda muestra que tenemos un total de 100710 parámetros que entrenar, que ocupan un total de 393.40 KB.

En la celda siguiente invocamos el método `compile` sobre el objeto `model`. Este método verifica que la arquitectura del modelo (red) sea válida y que todas las capas estén correctamente conectadas, y prepara el modelo para la fase de entrenamiento en una CPU, GPU o TPU. `compile` recibe tres argumentos: `loss`, que especifica cuál será la función de costo (o pérdida) a utilizar, en este caso entropía cruzada con valores categóricos; `optimizer`, que corresponde al método de optimización, y que para este ejemplo elegimos `adam` (*Adaptive Moments*) que es una variación del método del descenso estocástico del gradiente; y finalmente una métrica, `accuracy`, que es la fracción o porcentaje de predicciones correctas:



TensorFlow

[tensorflow](#) - Biblioteca de código abierto para el aprendizaje automático y el aprendizaje profundo, desarrollada por Google Brain.

CELDA 09
<code>model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])</code>

Ya tenemos entonces todo listo para iniciar el entrenamiento:

CELDA 10
<code>model.fit(x_train, y_train, epochs=25, batch_size=128, verbose=0)</code>

```

Epoch 22/25
469/469 ————— 0s 1ms/step - accuracy: 1.0000 - loss: 4.9750e-07
Epoch 23/25
469/469 ————— 0s 1ms/step - accuracy: 1.0000 - loss: 3.9767e-07
Epoch 24/25
469/469 ————— 0s 1ms/step - accuracy: 1.0000 - loss: 3.2803e-07
Epoch 25/25
469/469 ————— 1s 1ms/step - accuracy: 1.0000 - loss: 2.6730e-07

```

FIGURA 23.9: Últimas líneas de la salida de ejecución del método fit.

```
<keras.src.callbacks.history.History at 0x7eaeaf88440>
```

El método `fit` recibe los arrays de entrada y etiquetas (`x_train` y `y_train`), la cantidad de épocas y el tamaño del batch. Adicionalmente pasamos el argumento `verbose=0` para evitar la salida extensa de esta celda, pero si pasamos el valor por defecto de este parámetro (omitiéndolo por ejemplo), el método mostrará el progreso del entrenamiento informando el tiempo que le toma cada época, la precisión obtenida y el valor de la función de costo, tal como se muestra en la Figura 23.9.

Una vez entrenada la red, podemos probar cuán eficaz es para predecir las imágenes del conjunto de prueba. Para ello ejecutamos la celda siguiente:

CELDA 11

```

loss, acc = model.evaluate(x_test, y_test, batch_size=128)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))

```

```

79/79 ————— 0s 2ms/step - accuracy: 0.9729 - loss: 0.1324

Test accuracy: 97.8%

```

Vemos que la tasa de aciertos alcanza el 98 %, muy similar a la precisión que alcanzamos con el modelo de la sección anterior. Una descripción más completa que muestra el desempeño de la red la da la matriz de confusión. Esta matriz representa la cantidad de aciertos para cada clase o etiqueta, así como los “desaciertos” al predecir etiquetas erróneas. En las celdas siguientes definimos una función que realiza el gráfico de la matriz de confusión (tomada de la web de `scikit-learn`, versión 0.18), y obtenemos los valores predichos y verdaderos transformándolos a dígitos desde su representación como arrays OHE:

CELDA 12

```

import itertools

# Matriz de confusión
# Nota: este código está tomado directamente de la web de scikit-learn 0.18
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)

```



```

plt.title(title)
plt.colorbar()
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation=30)
plt.yticks(tick_marks, classes)

if normalize:
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j],
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('Etiqueta verdadera')
plt.xlabel('Etiqueta predicha')

```

CELDA 13

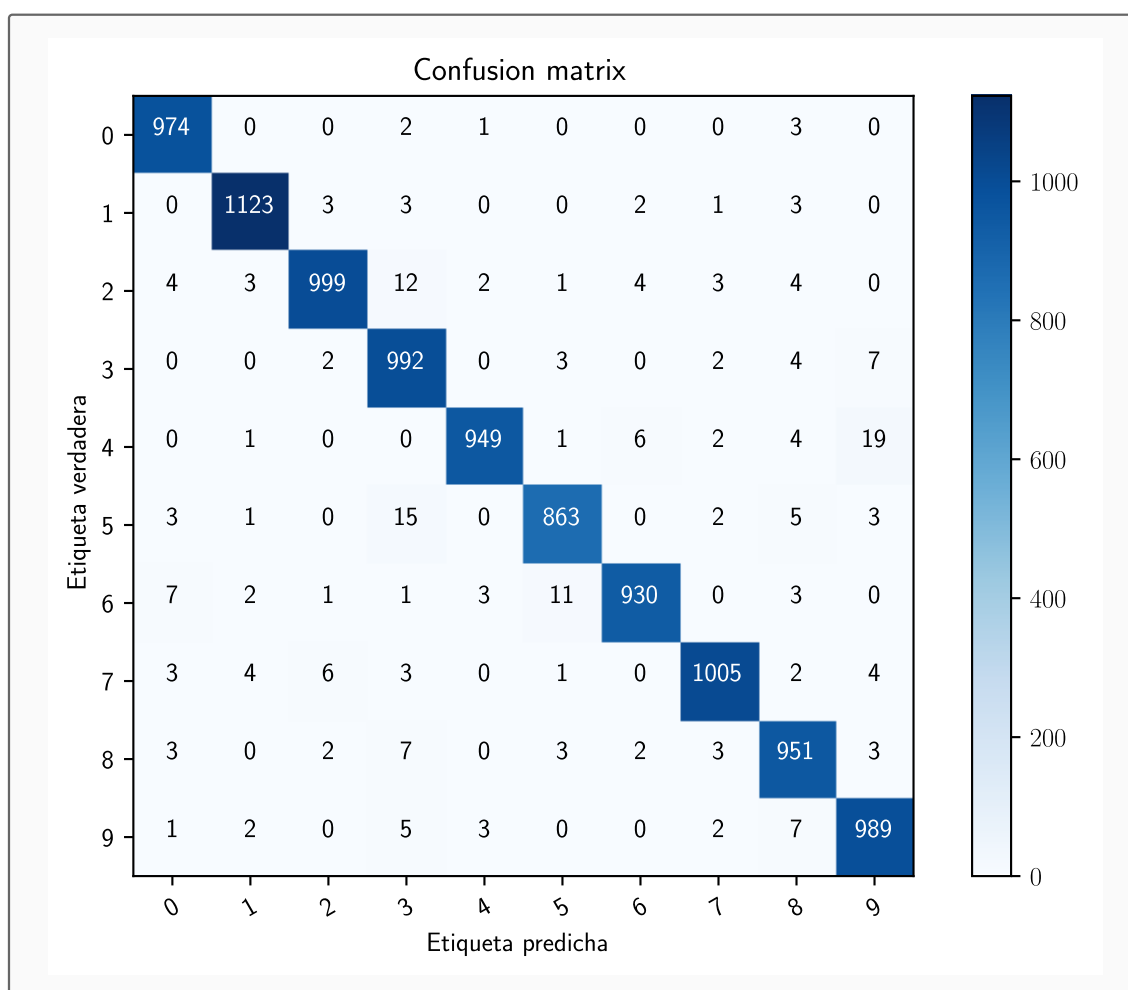
```

from sklearn.metrics import confusion_matrix

# Predict the values from the validation dataset
Y_pred = model.predict(x_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test, axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))

```

313/313 ————— 1s 2ms/step

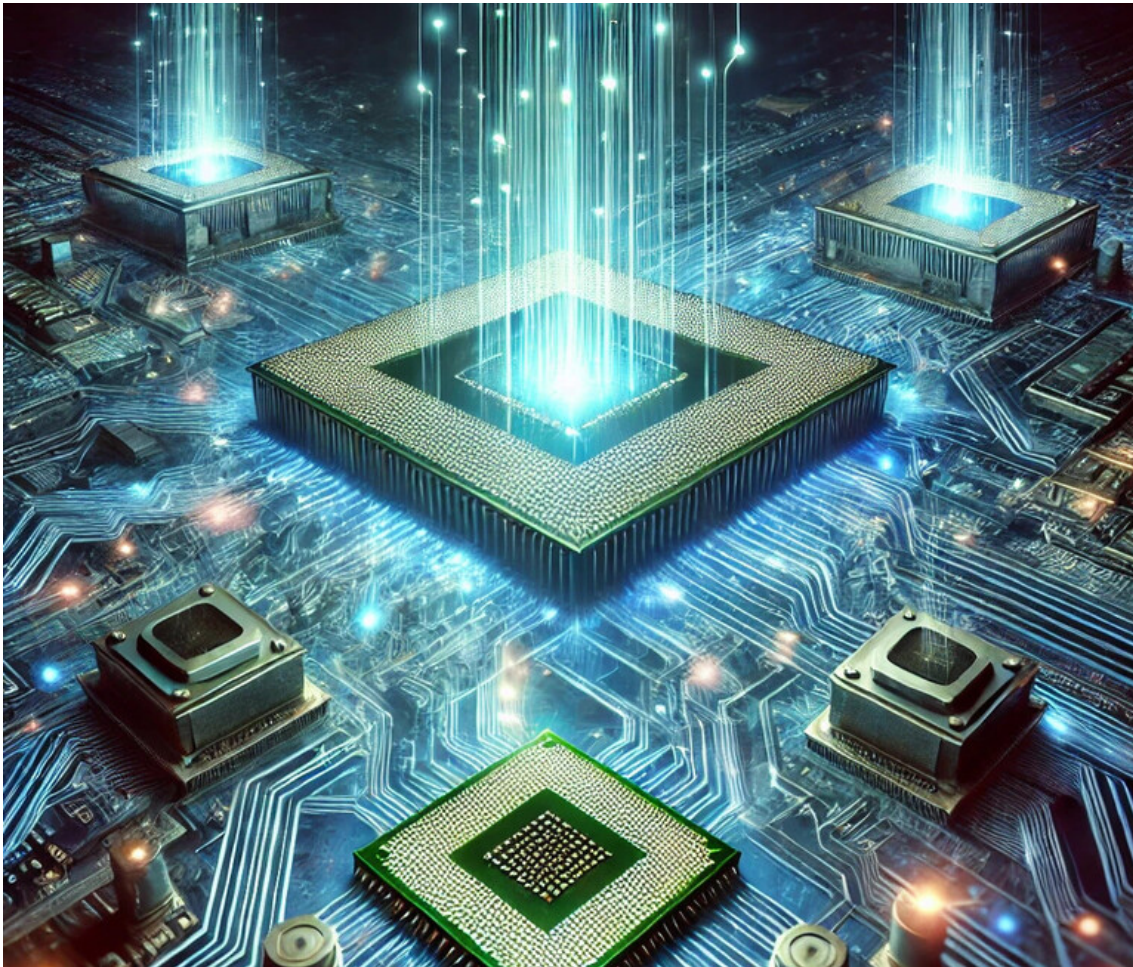


Los valores sobre la diagonal son los aciertos de la red, mientras que los que están fuera de la diagonal muestran cómo se distribuyen los fallos de la red sobre el resto de las etiquetas. Tal como ocurrió en la sección anterior, las imágenes que contienen al número 5 son las que más le cuesta identificar a la red. Si pasamos como argumento `normalize=True` a la función que construye la red, se mostrarán las fracciones de acierto en vez de los números absolutos.

En este ejemplo hemos visto solo las capacidades básicas de Keras, pero la biblioteca ofrece una amplia variedad de herramientas tales como capas Dropout, que es una técnica de regularización para reducir el sobreajuste de los datos, funciones de penalización, etc. Además, ofrece la posibilidad de construir otros tipos de redes como convolucionales, o recurrentes.

23.5. Lecturas recomendadas

- Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- Stuart J. Russel y Peter Norvig. *Artificial Intelligence. A Modern Approach*. 4.^a ed. Harlow, United Kingdom: Pearson Education Limited, 2022.



24 | Procesamiento en GPU

En el ámbito del cálculo científico y tecnológico, la demanda de potencia de cálculo es siempre creciente, dada la necesidad de resolver problemas cada vez más complejos, más grandes, más rápido.

Afortunadamente, la industria ha respondido a tales requerimientos con adelantos tecnológicos formidables, tal como lo muestra la ley empírica de Moore¹. Hasta el fin del siglo pasado, las computadoras incrementaban su potencia de cálculo principalmente por medio del aumento de la frecuencia del reloj (también mejorando el tamaño de las memorias caché y el conjunto de instrucciones de los procesadores), lo que permitía realizar más operaciones aritméticas o lógicas por segundo. Sin embargo, esto requería disipar cantidades mayores de calor, lo que impone un límite físico a la velocidad de los relojes. Por este motivo, desde mediados de los 2000, se comenzaron a implementar estrategias de paralelismo incorporando más núcleos de cálculo en las computadoras. Esto se evidencia claramente en la

Módulo	Versión
NumPy	1.26.4
PyOpenCL	2024.2.7
PyCUDA	2024.1.2

[Código disponible](#)

¹ Ver su entrada en [Wikipedia](#).

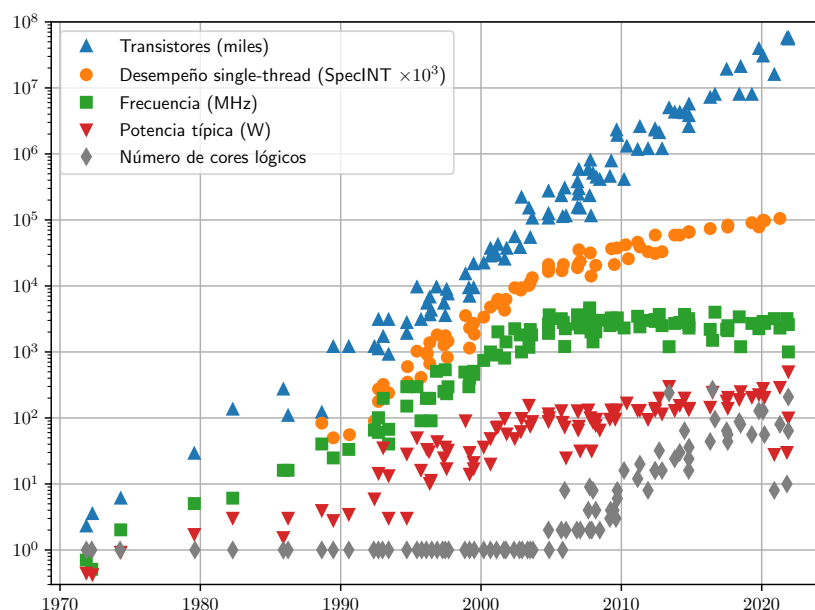


FIGURA 24.1: Evolución de 50 años de microprocesadores. Fuente: [Karl Rupp](#).

Figura 24.1.

Por otra parte, las placas gráficas fueron desarrolladas inicialmente como *hardware* especializado en acelerar el procesamiento gráfico de aplicaciones científicas o de ingeniería, aunque ya en la última década del siglo pasado se produjeron adelantos tecnológicos significativos traccionados principalmente por la industria de los juegos de video y la demanda creciente de gráficos 3D. En 1999, la empresa [NVIDIA](#) presentó la GeForce 256 como la primera GPU (*Graphics Processing Unit*) más que una aceleradora de gráficos, y desde entonces los avances en la tecnología de estas placas se han sostenido a un ritmo vertiginoso, liderados por compañías como NVIDIA, [AMD](#) e [Intel](#).

Debido a la naturaleza inherentemente paralela del cálculo de grandes cantidades de píxeles, investigadores y desarrolladores extendieron el uso de las GPUs más allá de las aplicaciones gráficas al ámbito del procesamiento general, en los primeros años del siglo XXI. En particular, algunos problemas asociados a operaciones sobre vectores y matrices fueron adaptados fácilmente a su ejecución en GPU [64, 65, 66], logrando realizar dichas operaciones en menos tiempo que sobre una CPU. Al desarrollo en el *hardware* lo acompañó la mejora en los lenguajes de programación, lo que facilitó significativamente la representación de estructuras y algoritmos sobre las placas gráficas por medio de abstracciones orientadas hacia el cálculo paralelo masivo, más que hacia los detalles de cálculo gráfico (los primeros desarrollos se hicieron expresando los cálculos como una forma de “pintar” un píxel). El lenguaje de programación sobre GPUs más difundido actualmente es CUDA [67], introducido por NVIDIA en 2006, como una extensión del lenguaje C/C++. Una alternativa que constituye un estándar abierto y multiplataforma es [OpenCL](#) del Kronos Group, desarrollado inicialmente por Apple Inc. en 2009.

Una vez consolidado el desarrollo del *hardware* y los lenguajes para utilizarlos, la comunidad científica y tecnológica adoptó rápidamente la enorme capacidad de cálculo de bajo costo dispo-

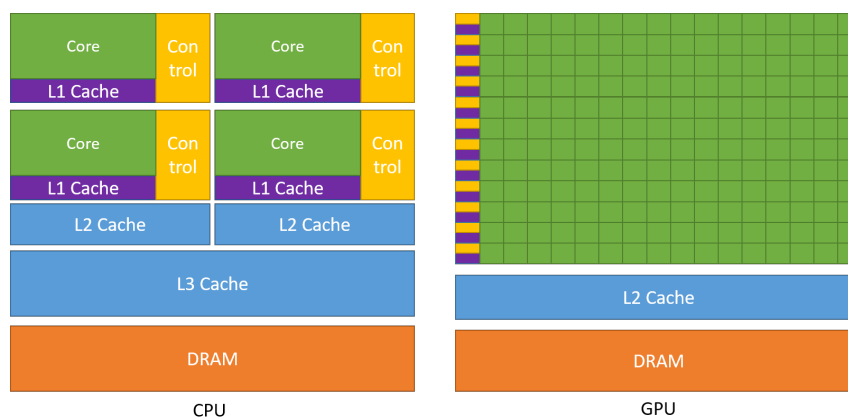


FIGURA 24.2: Comparación de arquitecturas de CPU y GPU: en verde se representan las unidades de cómputo o núcleos (*cores*), en amarillo las unidades de control y el resto (violeta, celeste y anaranjado) diferentes jerarquías de memoria. Fuente: [CUDA C++ Programming Guide](#).

nible, y el número de aplicaciones ejecutadas sobre GPUs comenzó a crecer sostenidamente. Un ejemplo notable de los primeros esfuerzos por aprovechar el paralelismo masivo provisto por las placas gráficas fue el trabajo de Raina, Madhavan y Ng en 2009 sobre la posibilidad de entrenar redes neuronales profundas con GPUs[68].

Existen varias herramientas en el ecosistema de Python que permiten acceder al procesamiento en una placa gráfica, abordando de diversas maneras las formas de delegar en una GPU parte del procesamiento que realizamos en un código. Por ejemplo, podemos mencionar [Numba](#), [CuPy](#), [PyCUDA](#) o [PyOpenCL](#). En este capítulo haremos una breve introducción a las dos últimas, con algo más de detalle sobre PyCUDA.

24.1. Arquitectura de una GPU

En una CPU actual podemos encontrar varios (decenas) núcleos de cómputo (o *cores*), que son procesadores capaces de realizar tareas secuenciales complejas en tiempos muy cortos gracias a su conjunto de instrucciones. Por ejemplo, con el propósito de optimizar el tiempo de ejecución, una CPU puede ejecutar instrucciones en un orden diferente al establecido por el programa (*out-of-order executions*), o puede predecir las instrucciones más probables a ejecutar en el futuro próximo cuando se encuentra una bifurcación (*multiple branch prediction*). De este modo puede anticiparse preparando los operandos correspondientes y ejecutando con anticipación tales instrucciones (*speculative execution*).

Por su parte, las GPUs proveen una plataforma especializada en cálculo paralelo de operaciones con números representados en punto flotante. Los núcleos de las GPUs tienen un conjunto de instrucciones más simple, por lo que cada uno puede realizar tareas menos complejas y en forma más lenta que una CPU. Sin embargo, una GPU actual dispone de miles de núcleos² trabajando simultáneamente con muy poco *overhead*, lo que le permite procesar datos hasta cientos de veces más rápido que una CPU con varios núcleos. Aprovechando el paralelismo masivo que proveen

²El modelo GeForce RTX 4090 de NVIDIA dispone de 16 384 CUDA *cores*.

las GPUs, el modelo de programación que utilizan es el SIMT (*Single Instruction, Multiple Threads*) que significa que todos los núcleos ejecutan exactamente la misma operación sobre datos diferentes (SIMD, *Single Instruction, Multiple Data*), combinado con el uso de múltiples hilos de ejecución.

En el caso de las GPUs de NVIDIA, la arquitectura de una GPU dada depende de la familia de dispositivos a la que pertenece. Cada GPU se caracteriza por su capacidad de cálculo o *compute capability*, que se representa por números de revisión mayor y menor (X, Y). Los dispositivos que comparten el mismo número de revisión mayor X pertenecen a la misma arquitectura (Kepler: 3, Maxwell: 5, Pascal: 6, Volta y Turing: 7, Ampere: 8 y Hopper: 9), mientras que el número de revisión menor Y corresponde a mejoras incrementales dentro de la misma arquitectura. Las especificaciones técnicas de cada capacidad de cálculo se pueden ver en las tablas que ofrece NVIDIA ([aquí](#) y [aquí](#)).

Todas estas arquitecturas se basan en un conjunto de *streaming multiprocessors* (SM), cada uno de los cuales controla la ejecución de un número de hilos o *threads*. Para ello, los SM disponen (según su capacidad de cálculo) de un número de CUDA *cores* para operaciones aritméticas con números enteros o de diferentes grados de precisión de punto flotante, unidades de procesamiento para el cálculo de funciones trascendentales con precisión simple, unidades de cálculo tensorial y procesadores para la administración de los hilos, además de una cantidad limitada memoria *on-chip* que es accesible por los *cores* que administra, y un gran número de registros.

Estos SM son capaces de crear, administrar y ejecutar grupos de 32 hilos que se ejecutan simultáneamente denominados *warps*. Cada *warp* ejecuta una instrucción común a los hilos que administra en forma simultánea. Cuando un SM recibe un bloque de hilos para ejecutar, los divide en *warps* que contienen hilos con identificadores (ID) consecutivos.

24.2. Kernels y threads en CUDA

La plataforma de programación provista por NVIDIA denominada CUDA (*Compute Unified Device Architecture*) permite escribir programas para ejecutar en sus GPUs, y consiste en una extensión del lenguaje C/C++ que ofrece un capa de *software* que permite el acceso directo al conjunto de instrucciones y núcleos de cálculo paralelo para la ejecución de funciones denominadas “*kernels*”. Estos kernels se ejecutan en la GPU, en forma paralela, utilizando un número N establecido de hilos o *threads*, sobre arreglos (*arrays*) de números que reciben como argumento. Cada thread que ejecuta el kernel tiene un identificador único que es accesible por el kernel a través de variables propias del lenguaje: `threadIdx`. Este identificador puede ser utilizado para calcular y acceder a elementos de los arrays.

El modelo de programación en CUDA para ejecutar código en la GPU consiste, básicamente, en proceder en la ejecución de las siguientes tareas:

- Definir el código que compone el kernel, proceder a su compilación y generar una función que será invocada desde nuestro programa en la CPU (o *host*).
- Copiar los datos que constituyen los argumentos del kernel desde la memoria del *host* a la memoria de la GPU (o *device*), o transferencia *host* a *device*.
- Lanzar el kernel en la GPU, con una determinada configuración de la división de tareas.

- Copiar los datos resultantes desde la memoria del *device* a la del *host*, o transferencia *device* to *host*.

Veamos un ejemplo simple que consiste en la suma de dos arrays, utilizando PyCUDA [69]. Primero importamos los módulos necesarios para acceder a la GPU y los inicializamos:

CELDA 1 DE SUMA-ARRAYS.IPYNB

```
import pycuda.driver as driver
import pycuda.autoinit
```

Los primeros dos módulos permiten establecer la comunicación con la GPU y realizar los pasos necesarios para enviarle kernels para su ejecución. El submódulo `driver` contiene funciones para la administración de la memoria, propiedades del dispositivo, etc., mientras que el submódulo `autoinit` se utiliza para la inicialización de la GPU, creación de contexto y limpieza de memoria. Este submódulo no es obligatorio, ya que estas funciones se pueden hacer en forma explícita.

En la celda siguiente vamos a generar el código que constituye el *kernel*. Para esto debemos importar módulo `pycuda.compiler`, que es el que crea un módulo ejecutable en la GPU a partir del código fuente que le suministramos en forma de cadena, haciendo uso del compilador de NVIDIA “nvcc”. A continuación definimos el *kernel* definiendo una variable `mod` que contiene el módulo con la definición del kernel. Éste se pasa como argumento al constructor de la clase compuesta por una cadena, que define la función a ejecutar sobre los arrays escrita con sintaxis de C/C++:

CELDA 02

```
from pycuda.compiler import SourceModule
mod = SourceModule("""
    __global__ void producto_arrays(float *a, float *b, float *c)
    {
        const int i = threadIdx.x;
        c[i] = a[i] * b[i];
    }
""")
producto_arrays = mod.get_function("producto_arrays")
```

La definición del *kernel* comienza con la palabra `__global__`, que se utiliza para declarar que la función que sigue a continuación es un kernel que se ejecuta en la GPU, invocable desde el *host* (para arquitecturas con capacidad de cómputo 5.0 o superior, la función también se puede invocar desde el *device*). Una función `__global__` debe devolver un tipo `void`, que es la palabra que precede al nombre de la función (`producto_arrays`). Luego, los tres arrays que constituyen los argumentos se pasan como punteros a datos de tipo `float`.

En el cuerpo de la función, delimitado por llaves, la variable entera `i` contiene el valor de `threadIdx.x`, que es el identificador del hilo que ejecuta el kernel. De este modo, cada hilo accede al elemento *i*-ésimo de cada array `a` y `b`, y almacena la suma de ambos en el elemento *i*-ésimo de `c`, paralelizando de este modo la suma sobre los arrays.

La variable `producto_arrays` recibe un *handler* a la función que contiene el módulo `mod`. En la celda siguiente definimos los tres arrays que utilizaremos: `a` y `b`, de tipo `np.float32`, y que contienen valores aleatorios mientras que `c`, del mismo tipo de `a`, se inicializa con ceros.

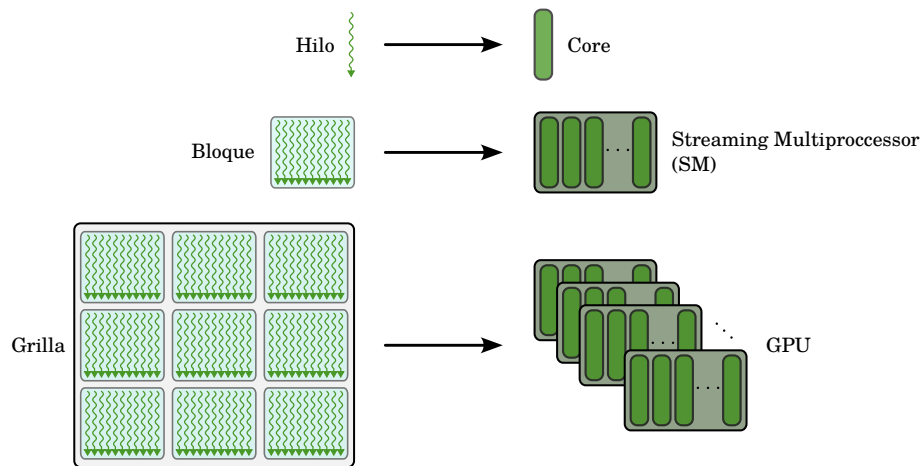


FIGURA 24.3: Jerarquía de hilos en CUDA.

CELDA 03

```
import numpy as np

n_elementos = 400
a = np.random.randn(n_elementos).astype(np.float32)
b = np.random.randn(n_elementos).astype(np.float32)
c = np.zeros_like(a)
```

Ahora lanzamos la ejecución de hilos en la GPU utilizando el *handler* `producto_arrays`, al que le pasamos como argumentos los dos arrays que vamos a sumar, y el array en el que almacenamos el resultado. La transferencia de estos arrays desde y hacia el device lo hacemos con los métodos `In`, que transfiere el array al device *antes* de ejecutar el kernel, y `Out`, que indica que el array debe transferirse desde el device *después* de que el kernel finaliza su tarea. Otro parámetro que debemos pasar es `block`, que indica el modo en que se distribuirán las tareas en la GPU y que describiremos en la sección siguiente. Finalmente, comparamos los valores almacenados en `c` con el cálculo que realiza NumPy para verificar que sean iguales.

CELDA 04

```
producto_arrays(driver.In(a), driver.In(b), driver.Out(c), block=(400,1,1))
np.allclose(c, a * b)
```

True

24.3. Organización de *threads*

La unidad de ejecución de kernels es el hilo o *thread*, y lo usual es utilizar un gran número de *threads* que debemos gestionar para aprovechar la cantidad de *cores* que nos provee la GPU. Un grupo de hilos se denomina “bloque”, y a su vez un conjunto de bloques define una “grilla”, tal como se esquematiza en la Figura 24.3. Cada bloque se ejecuta en un *streaming multiprocessor* (salvo excepciones, un bloque no se puede migrar a otro SM) en forma concurrente, según los

requerimientos de recursos que necesita. De esta forma, dado que los hilos de un mismo bloque comparten los recursos que provee el SM, dichos hilos pueden comunicarse entre sí a través de la memoria compartida del SM, por medio de barreras de sincronización u operaciones atómicas.

El número de hilos que pueden agruparse en un bloque está limitado, y por esto es posible agrupar bloques en una grilla para hacer factible la ejecución de una gran cantidad de hilos. La división de tareas en un procesamiento en paralelo debe planificarse teniendo en cuenta la cantidad de hilos que puede administrar un bloque, lo que en última instancia depende de la GPU (o múltiples GPUs) que tengamos disponible. En el código siguiente vemos cómo podemos interrogar a la GPU para obtener información sobre los recursos que ofrece:

CELDA 1 DE GPU_INFO.IPYNB

```
import pycuda.driver as cuda
import pycuda.autoinit

print(f'{cuda.Device.count()} dispositivo(s) encontrado.')
device = cuda.Device(0)
print(f'Modelo de GPU: {device.name()}')
print('  Compute Capability: %d.%d' % device.compute_capability())
print(f'  Memoria total: {device.total_memory()//((1024)**2)} MB')
atts = [(str(att), value) for att, value in device.get_attributes().items()]
atts.sort()

for att, value in atts:
    print(' %s: %s' % (att, value))
```

La salida de esta celda es extensa, por lo que resumimos aquí algunas líneas:

```
1 dispositivo(s) encontrado.
Modelo de GPU: NVIDIA GeForce GTX 1060 3GB
Compute Capability: 6.1
Memoria total: 3072 MB
...
MAX_BLOCKS_PER_MULTIPROCESSOR: 32
MAX_BLOCK_DIM_X: 1024
MAX_BLOCK_DIM_Y: 1024
MAX_BLOCK_DIM_Z: 64
MAX_GRID_DIM_X: 2147483647
MAX_GRID_DIM_Y: 65535
MAX_GRID_DIM_Z: 65535
MAX_THREADS_PER_BLOCK: 1024
MAX_THREADS_PER_MULTIPROCESSOR: 2048
...
MULTIPROCESSOR_COUNT: 9
WARP_SIZE: 32
```

Aquí podemos ver que en la computadora donde se ejecutó este notebook existe solo una placa cuyo modelo es «NVIDIA GeForce GTX 1060 3GB», que soporta una capacidad de cálculo 6.1 y dispone de una memoria total de 3.072 MB. Hacia el final de la salida de la celda, confirmamos que el tamaño de warp es 32 (cantidad que puede cambiar en futuras versiones del *hardware*), y que la GPU dispone de 9 SM. Además, el número máximo de bloques que puede ejecutar cada SM es 32. Si corremos el mismo código en Google Colab, obtenemos la siguiente salida:

```

1 dispositivo(s) encontrado.
Modelo de GPU: Tesla T4
Compute Capability: 7.5
Memoria total: 15102 MB
...
MAX_BLOCKS_PER_MULTIPROCESSOR: 16
MAX_BLOCK_DIM_X: 1024
MAX_BLOCK_DIM_Y: 1024
MAX_BLOCK_DIM_Z: 64
MAX_GRID_DIM_X: 2147483647
MAX_GRID_DIM_Y: 65535
MAX_GRID_DIM_Z: 65535
MAX_THREADS_PER_BLOCK: 1024
MAX_THREADS_PER_MULTIPROCESSOR: 1024
...
MULTIPROCESSOR_COUNT: 40
WARP_SIZE: 32

```

es decir que la plataforma nos asigna una GPU con mayor capacidad de cálculo (7.5), la mitad de bloques por SM (16) pero más de 4 veces la cantidad de streaming multiprocessors (40 SM), y cinco veces la memoria de la placa anterior.

Por otra parte, podemos notar que los máximos valores para las dimensiones de bloques y grilla se dividen en `_X`, `_Y` y `_Z`, lo que sugiere que estas categorías pueden adquirir representaciones tridimensionales. De hecho, la Figura 24.3 muestra un arreglo bidimensional de bloques en una grilla, y un arreglo unidimensional de hilos en un bloque.

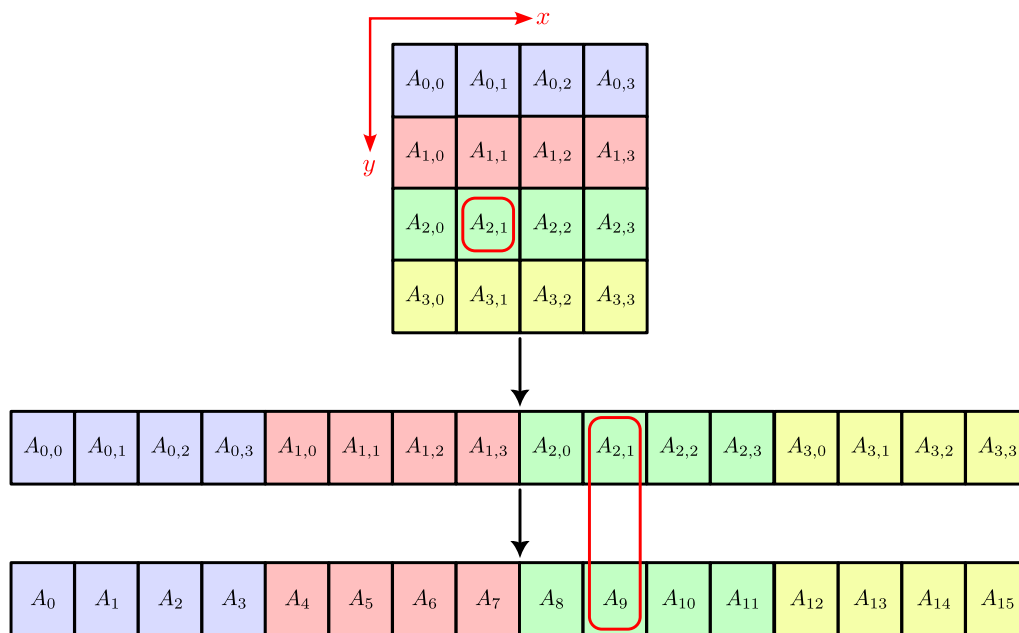
Efectivamente, la variable `threadIdx` es un vector de tres componentes, de modo que los hilos se pueden identificar utilizando un índice unidimensional, bidimensional o tridimensional, lo que genera un bloque de threads de una, dos o tres dimensiones respectivamente, permitiendo de este modo una representación natural de dominios de cómputo como vectores, matrices o volumen. Podemos acceder a cada componente del vector con `threadIdx.x`, `threadIdx.y` y `threadIdx.z`. La definición de un bloque, entonces, queda determinada por una tupla de tres enteros que indican la cantidad de hilos en cada dimensión. Por ejemplo, un bloque bidimensional de 5 hilos en `x` y 4 hilos en `y` se indica con `block=(5, 4, 1)`.

En consecuencia, la lógica para determinar el índice de un hilo a partir de su `threadIdx` en un bloque, se puede realizar como se esquematiza en la Figura 24.4. En ese caso, pasamos de una representación bidimensional a una unidimensional «concatenando» consecutivamente cada fila (orden por fila o *raw-major order*). Específicamente, para un bloque de dimensiones `blockDim.x` en la dirección `x` y `blockDim.y` en la dirección `y`, el índice de cada hilo resulta `threadIdx.x + blockDim.x × threadIdx.y` ($1 + 4 \times 2 = 9$ en el elemento resaltado de la figura).

Esta forma de identificar hilos en un bloque puede generalizarse a una y tres dimensiones:

- 1D: `threadID = threadIdx.x`
- 2D: `threadID = threadIdx.x + blockDim.x × threadIdx.y`
- 3D: `threadID = threadIdx.x + blockDim.x × threadIdx.y + blockDim.x × blockDim.y × threadIdx.z`

Tal como comentamos anteriormente, existe un límite en la cantidad de hilos que se pueden agrupar en un bloque, dadas las limitaciones de memoria y cantidad de núcleos de cálculo que

FIGURA 24.4: Identificación del índice de un hilo a partir de su `threadIdx` según el orden por filas.

dispone un SM. Actualmente ese límite es de 1.024 hilos por bloque. Sin embargo, es posible ejecutar un kernel sobre muchos bloques que tengan la misma forma (o dimensiones en cada «eje»), de modo que el número total de hilos en los que se puede desplegar un cálculo es el número de hilos por bloque multiplicado por el número de bloques.

Al igual que los hilos en un bloque, los bloques pueden organizarse en un arreglo unidimensional, bidimensional o tridimensional, formando de este modo una grilla. La cantidad de bloques en la grilla se suele determinar por la cantidad de datos a procesar, que por lo general excede la cantidad de procesadores en la GPU. De la misma forma que para los bloques, la definición de la grilla requiere de una tupla de tres enteros. Durante la ejecución de un kernel, cada bloque de la grilla se puede identificar mediante las variables `blockIdx.x`, `blockIdx.y` y `blockIdx.z`, mientras que las dimensiones de la grilla se pueden acceder mediante `blockDim.x`, `blockDim.y` y `blockDim.z`. Finalmente, se puede asignar un índice a cada hilo a partir de su ubicación dentro de un bloque y la localización del bloque en la grilla. Veamos un ejemplo en el siguiente código:

CELDA 1 DE HILOS_ID.IPYNB

```
import pycuda.driver as driver
import pycuda.autoinit
```

CELDA 02

```
from pycuda.compiler import SourceModule

mod = SourceModule(r"""
#include <stdio.h>

__global__ void t_ident()
{
```

```

int idx = threadIdx.x + blockDim.x * threadIdx.y + blockDim.x * blockDim.y * threadIdx.z;
int block_id = blockIdx.x + gridDim.x * (blockIdx.y + blockIdx.z * gridDim.y);
int thread_id = idx + block_id * (blockDim.x * blockDim.y);
printf("%9d %11d %11d %11d %10d %10d %10d \n",
      thread_id, threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y, blockIdx.z);
}
"""
func = mod.get_function("t_ident")

```

En la celda 1 importamos los módulos necesarios e inicializamos la comunicación con la GPU. En la segunda celda definimos el kernel que se ejecutará en el *device*. Esta función no recibe argumentos, pues solo realizará operaciones sobre variables internas provistas por la GPU. La primera línea de la función determina un índice que identifica cada hilo dentro de un bloque (*idx*), y la línea siguiente identifica el bloque que se está ejecutando (*block_id*), por último, combinando los índices anteriores, podemos asignar un identificador único al hilo en ejecución (*thread_id*), y la tarea de la función será mostrar los valores de todas estas variables. En la celda siguiente, luego de mostrar en la pantalla un encabezamiento, invocamos al *handler* de nuestro módulo pasándole como argumentos dos parámetros, *block* y *grid* que proveen el contexto en el que se ejecutará el kernel en el *device*. Podemos ahora lanzar un total de 16 hilos organizados en una grilla con un solo bloque, compuesto por un arreglo bidimensional de 4 hilos por dimensión, tal como se muestra en la celda 3:

CELDA 03							
<pre> print('thread_id threadIdx.x threadIdx.y threadIdx.z blockIdx.x blockIdx.y blockIdx.z') func(block=(4, 4, 1), grid=(1, 1, 1)) </pre>							
thread_id	threadIdx.x	threadIdx.y	threadIdx.z	blockIdx.x	blockIdx.y	blockIdx.z	
0	0	0	0	0	0	0	
1	1	0	0	0	0	0	
2	2	0	0	0	0	0	
3	3	0	0	0	0	0	
4	0	1	0	0	0	0	
5	1	1	0	0	0	0	
6	2	1	0	0	0	0	
7	3	1	0	0	0	0	
8	0	2	0	0	0	0	
9	1	2	0	0	0	0	
10	2	2	0	0	0	0	
11	3	2	0	0	0	0	
12	0	3	0	0	0	0	
13	1	3	0	0	0	0	
14	2	3	0	0	0	0	
15	3	3	0	0	0	0	

Claro que, según resulte conveniente para el problema que queremos descomponer en múltiples procesos, podemos organizar la misma cantidad de hilos en forma diferente. Por ejemplo, en la siguiente celda vemos cómo lanzamos el mismo kernel pero ahora en una grilla bidimensional de dos bloques por dimensión, conteniendo un arreglo unidimensional de cuatro hilos cada bloque:

CELDA 04

```
print('thread_id threadIdx.x threadIdx.y threadIdx.z blockIdx.x blockIdx.y blockIdx.z')
func(block=(4, 1, 1), grid=(2, 2, 1))
```

thread_id	threadIdx.x	threadIdx.y	threadIdx.z	blockIdx.x	blockIdx.y	blockIdx.z
0	0	0	0	0	0	0
1	1	0	0	0	0	0
2	2	0	0	0	0	0
3	3	0	0	0	0	0
4	0	1	0	0	0	0
5	1	1	0	0	0	0
6	2	1	0	0	0	0
7	3	1	0	0	0	0
8	0	2	0	0	0	0
9	1	2	0	0	0	0
10	2	2	0	0	0	0
11	3	2	0	0	0	0
12	0	3	0	0	0	0
13	1	3	0	0	0	0
14	2	3	0	0	0	0
15	3	3	0	0	0	0
0	0	0	0	0	0	0
1	1	0	0	0	0	0
2	2	0	0	0	0	0
3	3	0	0	0	0	0
8	0	0	0	0	1	0
9	1	0	0	0	1	0
10	2	0	0	0	1	0
11	3	0	0	0	1	0
4	0	0	0	1	0	0
5	1	0	0	1	0	0
6	2	0	0	1	0	0
7	3	0	0	1	0	0
12	0	0	0	1	1	0
13	1	0	0	1	1	0
14	2	0	0	1	1	0
15	3	0	0	1	1	0

Observamos en la última salida, a diferencia de la celda anterior, que los hilos no aparecen en orden, lo que es muy común que ocurra al lanzar procesos en forma concurrente, que es lo que sucede a nivel de bloque. No sabemos de antemano en qué orden se ejecutarán los cuatro bloques que lanzamos.

24.4. Producto de matrices en GPU con PyCUDA

El producto de matrices, operación ubicua en métodos numéricos, constituye un cálculo muy conveniente para realizar en forma paralela, por lo que implementaremos este cálculo para dar un ejemplo práctico de uso de la GPU.

Nuestro objetivo es calcular el producto:

$$C = A \cdot B, \quad A \in \mathbb{R}^{m \times l}, \quad B \in \mathbb{R}^{l \times n} \text{ y } C \in \mathbb{R}^{m \times n}$$

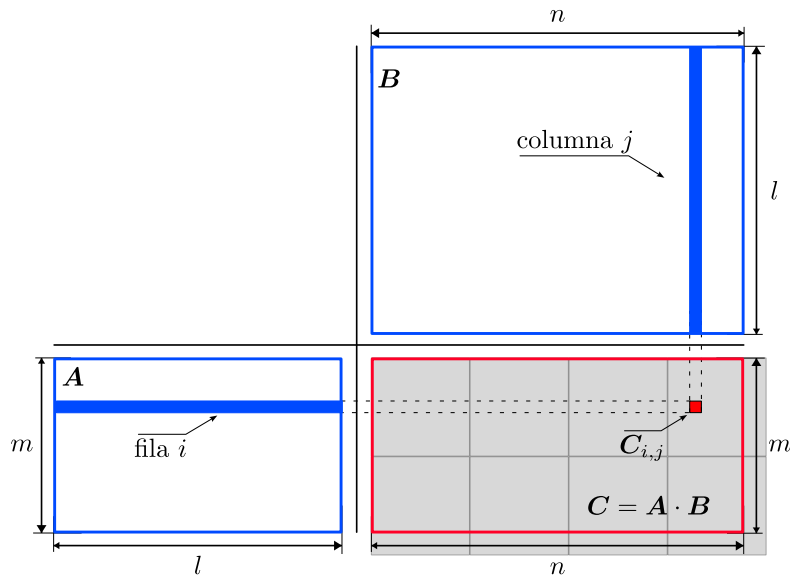


FIGURA 24.5: Esquema de procesamiento de hilos en el producto de matrices. En gris, sobre la matriz C , se representa una grilla de bloques que cubren (en exceso) las dimensiones de C .

Cada elemento c_{ij} de C resulta de realizar la operación:

$$c_{ij} = \sum_{k=1}^l a_{ik} b_{kj}$$

Vemos de aquí que el cálculo de c_{ij} es independiente del resto de los valores de los elementos de C , por lo que es posible paralelizar de una manera muy simple el cálculo asignando a cada hilo el cómputo de c_{ij} . Además, considerando la estructura «bidimensional» de las matrices involucradas, la representación natural para los hilos es agruparlos en forma bidimensional en los bloques.

Como mencionamos anteriormente, cada bloque puede contener un número limitado de hilos, que según hemos visto en los ejemplos previos es 1.024. Una posibilidad consiste en organizar nuestros bloques de forma que contengan un arreglo cuadrado de 32 hilos por fila y 32 hilos por columna, lo que nos lleva a considerar la necesidad de utilizar varios bloques de hilos si nuestras matrices superan estas dimensiones. La Figura 24.5 esquematiza esta situación, en la que la matriz C es cubierta con un arreglo de bloques cuadrados.

En nuestro ejemplo vamos a multiplicar matrices con $m = 4000$, $l = 6000$ y $n = 2000$, lo que da un total de 8 millones de hilos para realizar los 48 mil millones de productos y sumas que requiere calcular C . Vamos a necesitar varios bloques.

Iniciamos nuestro código importando los módulos necesarios para acceder e inicializar el *device*, y definimos el kernel. Como argumentos, recibirá los arrays que representan las matrices involucradas, y también las dimensiones de las mismas:

CELDA 1 DE MATPROD.IPYNB

```
import pycuda.driver as driver
import pycuda.autoinit
```

CELDA 02

```
kernel = """
__global__ void matrix_prod(float *a, float *b, float *c, int n, int l, int m)
{
    // Identificación del hilo 2D en un bloque
    int i = blockIdx.x * blockDim.x + threadIdx.y; // Fila i
    int j = blockIdx.y * blockDim.y + threadIdx.x; // Columna j

    // Verificamos estar dentro de los límites (n x m)
    if(i < n && j < m) {
        float valor = 0;
        for (int k = 0; k < l; k++) {
            valor += a[i * l + k] * b[k * m + j];
        }
        // Almacenamos el resultado
        c[i * m + j] = valor;
    }
}
"""
```

Asociamos la fila i con la variable `threadIdx.y`, y la columna j con la variable `threadIdx.x`, dentro del bloque correspondiente. Luego, realizamos el cálculo del elemento c_{ij} verificando previamente que los índices se encuentran dentro de los límites de filas y columnas de las matrices. Esto es necesario debido a que se crearán más hilos que los que ocuparemos para el cálculo de c_{ij} , dado que la cantidad de bloques en cada dimensión debe exceder a las filas y columnas que tienen A y B , como veremos en la celda siguiente.

En la celda 3, con Numpy, definimos las dimensiones de las matrices A y B , y a continuación generamos los elementos de estas matrices con números aleatorios. Dado que las matrices son creadas en el *host*, nombramos los arrays con `a_cpu` y `b_cpu` respectivamente. Es necesario mencionar aquí que dichos arrays bidimensionales son representados en memoria como arrays unidimensionales por fila, y hacemos uso de esto en el kernel al recorrer los arrays con los índices i y j . Con el propósito de validar los resultados que obtendremos con la GPU, realizamos el cálculo en la CPU utilizando el método `dot` de NumPy. Registramos el tiempo que tarda en ejecutarse esta última operación (para ello importamos también el módulo `time` al comienzo de la celda), para comparar la eficiencia del *host* y del *device*.

CELDA 03

```
import numpy as np
import time

# Definimos el tamaño de las matrices A(n x l) y B (l x m)
m = np.int32(4000)
l = np.int32(6000)
n = np.int32(2000)

# creamos dos matrices aleatorias
np.random.seed(13)
```

```

a_cpu = np.random.randn(m, l).astype(np.float32)
b_cpu = np.random.randn(l, n).astype(np.float32)

# calculamos el resultado en la CPU para comparar con el de la GPU

start_time_CPU = time.time()
c_cpu = np.dot(a_cpu, b_cpu)
end_time_CPU = time.time()
elapsed_time_CPU = end_time_CPU - start_time_CPU

print(f"El tiempo de ejecución fue: {elapsed_time_CPU} segundos")

```

El tiempo de ejecución fue: 0.31313204765319824 segundos

Vemos que el cálculo del producto matricial llevó un poco más de tres décimas de segundo.

Organizaremos nuestros hilos en bloques cuadrados, conteniendo un número de hilos por dimensión que resulta de la raíz cuadrada del máximo número de hilos por bloque que nos informa la GPU. La cantidad de bloques que definen nuestra grilla resulta, en cada dimensión, del entero superior más próximo al cociente entre la cantidad de elementos de matriz y el número de hilos. Al realizar esa cuenta, vemos que necesitamos 125 bloques en la dirección x y 63 en la y . Recordemos que el número total de elementos de C es de 8 millones. Si dividimos este número por la cantidad de hilos por bloque (1.024) nos indica que necesitamos 7812,5 bloques, pero dado que usamos bloques cuadrados sobre matrices que no lo son, el cálculo determina que necesitaremos un total de 7875 bloques para cubrir las filas y columnas de C , generando de este modo más hilos que los necesarios, y por este motivo debemos verificar que los índices i y j en el cálculo que realizamos en el kernel no superen los límites de filas y columnas de A y B , respectivamente. Esto se muestra en la celda 4.

CELDA 04

```

from math import sqrt, ceil

device = driver.Device(0)
max_threads_per_block = device.get_attribute(driver.device_attribute.MAX_THREADS_PER_BLOCK)
t_xy = int(ceil(sqrt(max_threads_per_block)))
NBlocks_x = ceil(m / t_xy)
NBlocks_y = ceil(n / t_xy)
print(f"Bloques en x: {NBlocks_x}, bloques en y: {NBlocks_y}, con t_xy: {t_xy}")

```

Bloques en x: 125, bloques en y: 63, con t_xy: 32

Lo que nos queda por hacer es compilar el kernel, obtener el *handler* a la función, transferir los arrays a la memoria del *device* (mediante el método `gpuarray.to_gpu()`), crear en el *device* el espacio necesario para almacenar los valores de la matriz C (`c_gpu`), y ejecutar el kernel con los datos de las matrices y la organización de hilos en bloques, tal como mostramos a partir de la celda 5.

CELDA 05

```

from pycuda import compiler

# Compilamos el kernel y obtenemos el handler a la función

```



```
mod = compiler.SourceModule(kernel)
func = mod.get_function("matrix_prod")
```

CELDA 06

```
from pycuda import gpuarray, tools

# transferimos de la memoria host (CPU) a la memoria del device (GPU)
start_time_CPU2GPU = time.time()
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
end_time_CPU2GPU = time.time()
elapsed_time_CPU2GPU = end_time_CPU2GPU - start_time_CPU2GPU
# creamos un array vacío en la GPU para el resultado (C = A . B)
c_gpu = gpuarray.empty((m, n), np.float32)

print(f"El tiempo de transferencia de datos fue: {elapsed_time_CPU2GPU} segundos")

# Ejecutamos el kernel en la GPU
start_time_GPU = time.time()
func(a_gpu, b_gpu, c_gpu, m, l, n, block=(t_xy, t_xy, 1), grid=(NBlocks_x, NBlocks_y, 1))
end_time_GPU = time.time()
elapsed_time_GPU = end_time_GPU - start_time_GPU

print(f"El tiempo de ejecución fue: {elapsed_time_GPU} segundos")
```

```
El tiempo de transferencia de datos fue: 0.03423953056335449 segundos
El tiempo de ejecución fue: 0.0003933906555175781 segundos
```

En estas celdas registramos el tiempo que lleva realizar dos operaciones: la transferencia de los arrays al *device* y la ejecución del kernel en la GPU. Podemos ver que la transferencia es dos órdenes de magnitud más lenta que la ejecución del kernel (al menos en el *hardware* particular en el que fue ejecutado el código), y que el cálculo del producto matricial, realizado en la GPU, es unas 700 veces más rápido que el cálculo en la CPU con NumPy. Observando lo “costoso” que resulta la transferencia de datos hacia y desde el *host*, es de suma importancia un buen diseño del programa de modo de minimizar estas transferencias.

Finalmente, comparamos el resultado obtenido en el *host* con el obtenido con la GPU, realizando previamente la transferencia *device* a *host* con el método `get()`:

CELDA 07

```
np.allclose(c_cpu, c_gpu.get(), rtol=1e-05, atol=1e-03)
```

```
True
```

Podemos observar que los cálculos en CPU y GPU son aproximados con una tolerancia absoluta de 1×10^{-3} . Esto se debe a que utilizamos una representación de punto flotante con precisión simple (32 bits), y que en las sumas acumuladas en bucles se suman también los errores de redondeo.

Este ejemplo sencillo muestra la forma en que podemos utilizar la GPU para realizar cálculos numéricos. Sin embargo, es un código que no explota toda la capacidad del *hardware* para hacer más eficiente la tarea. Para ello es necesario considerar aspectos tales como el ancho de banda y latencia de las memorias global, compartida y registros de la GPU, que condicionan el parti-

cionado del problema y la organización de los hilos, y otros detalles que exceden el alcance de este libro. Sin embargo, queremos enfatizar la necesidad de evaluar la conveniencia de utilizar la GPU para nuestros cálculos en función del *hardware* disponible y el “tamaño” del problema a resolver. En este ejemplo concreto, vemos que resulta muy conveniente realizar el producto en la GPU, pero el tiempo que requiere la transferencia de datos casi pone en igualdad de condiciones al cálculo en CPU, lo que indica que una buena planificación de las tareas requiere minimizar la transferencia de los datos y aprovechar la potencia de cálculo paralelo masivo provista por la GPU.

24.5. Producto de matrices con PyOpenCL

OpenCL (*Open Computing Language*) es un estándar abierto que permite la programación sobre casi cualquier dispositivo disponible en nuestros días: CPUs, GPUs, FPGAs (*Field Programmable Gate Arrays*), aprovechando las ventajas que cada plataforma pueda ofrecer. A diferencia de CUDA, que solo corre programas en los dispositivos de NVIDIA, OpenCL puede correr en cualquier arquitectura y CPU de múltiples *cores* siempre que el fabricante provea un *driver* OpenCL.

Existen muchas similitudes entre OpenCL y CUDA, debido a las semejanzas del abordaje en paralelo y las jerarquías de memoria. OpenCL asume que la ejecución de un programa ocurre en una plataforma conformada por un *host* (típicamente un CPU) y un número de *devices* conectados. Cada *device* está compuesto por un número de unidades de cómputo (CU, *compute units*) que a su vez contienen unidades de procesamiento (PEs, *processing elements*). En una analogía con el modelo CUDA, los PEs corresponden a los CUDA *cores*, y los CUs a los *streaming processors*.

Un programa OpenCL contiene típicamente dos componentes:

- Un programa que controla la ejecución del código en el *device* y provee funcionalidades de entrada y salida de datos.
- Uno o más programas que corren en el *device* o *kernels*.

Cuando un *host* lanza un kernel, éste se ejecuta en el *device* en forma de “elemento de trabajo” (equivalente a un *thread* de CUDA). Cada elemento de trabajo corre en un PE, y puede haber múltiples elementos de trabajo que se agrupan en «grupos de trabajo» (equivalentes a los bloques de CUDA). Cada grupo de trabajo es asignado a un CU.

PyOpenCL ofrece un abordaje «pitónico» a OpenCL, del mismo modo que PyCUDA lo hace con CUDA. Mostraremos a continuación el ejemplo del producto de matrices pero ahora con OpenCL. Para comenzar, importamos el módulo `pyopencl` y establecemos el contacto con la GPU. Primero detectamos la plataforma de ejecución (`platform`), y su correspondiente *device* (que asignamos a la variable `device`). Generamos el contexto de ejecución (`context`) y finalmente una cola de comandos (`queue`) que es por donde se envían los comandos al *device* (ejecución del kernel, sincronización y operaciones de transferencia de memoria).

CELDA 1 DE MATPROD_PYOPENCL.IPYNB

```
import pyopencl as cl

# Configuramos el contexto y cola de comandos
platform = cl.get_platforms()[0]
device = platform.get_devices()[0]
```

```
context = cl.Context([device])
queue = cl.CommandQueue(context)
```

En la celda 2 interrogamos al *device* para poder dimensionar el tamaño del grupo de trabajo:

CELDA 02

```
import numpy as np

# Obtenemos el tamaño máximo de grupo de trabajo y el tamaño máximo de los
# elementos de grupo de trabajo en cada dimensión
max_work_group_size = device.get_info(cl.device_info.MAX_WORK_GROUP_SIZE)
max_work_item_sizes = device.get_info(cl.device_info.MAX_WORK_ITEM_SIZES)
print(
    f"Máximo tamaño de grupo: {max_work_group_size}, "
    f"Máximo tamaño de elementos por dimensión: {max_work_item_sizes}")

# Definimos el tamaño de los bloques
block_size = int(np.sqrt(max_work_group_size))
block_size_x = min(block_size, max_work_item_sizes[0])
block_size_y = min(block_size, max_work_item_sizes[1])
print(f"Elementos por bloque: {block_size_x} x {block_size_y}")
```

```
Máximo tamaño de grupo: 1024, máximo tamaño de elementos por dimensión: [1024, 1024, 64]
Elementos por bloque: 32 x 32
```

Vemos que cada bloque puede administrar 1.024 elementos de trabajo que se pueden organizar en un arreglo tridimensional con un máximo de elementos por cada dimensión de 1.024, 1.024 y 64, respectivamente. Decidimos entonces trabajar con grupos bidimensionales cuadrados.

Tal como hicimos el en ejemplo con PyCUDA, dimensionamos y definimos las matrices, y las transferimos al *device*:

CELDA 03

```
# Definimos el tamaño de las matrices A(n x l) y B (l x m)
m = 4000
l = 6000
n = 2000
np.random.seed(13)
a_cpu = np.random.rand(m, l).astype(np.float32)
b_cpu = np.random.rand(l, n).astype(np.float32)
c_cpu = np.empty((m, n), dtype=np.float32)

# Creamos buffers en el dispositivo
mf = cl.mem_flags
a_buf = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_cpu)
b_buf = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_cpu)
c_buf = cl.Buffer(context, mf.WRITE_ONLY, c_cpu.nbytes)
```

En la celda 4 definimos el kernel. A diferencia del ejemplo con pyCUDA, el índice de fila y columna se obtienen a partir de las variables intrínsecas `get_global_id(0)` y `get_global_id(1)`, que ya contienen los identificadores globales de cada hilo, sin tener que hacer el cálculo a partir de su ubicación interna en cada grupo de trabajo. Finalmente, asignamos a la variable `program` al programa compilado listo para su ejecución.

CELDA 04

```
# Definimos el kernel OpenCL
program_source = """
__kernel void matmul(const int M, const int N, const int K,
                    __global const float *A, __global const float *B,
                    __global float *C) {
    int row = get_global_id(0);
    int col = get_global_id(1);

    float sum = 0.0f;
    for (int k = 0; k < K; k++) {
        sum += A[row * K + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}
"""
program = cl.Program(context, program_source).build()
```

En la celda 5 determinamos el tamaño en cada dimensión de la grilla que agrupa los grupos de trabajo, a partir de la división entera entre la cantidad de hilos necesarios en cada dimensión (que representan las filas y columnas de las matrices) y de la cantidad de hilos que asignamos a cada grupo de trabajo (que recordemos, usaremos un arreglo cuadrado de hilos). Estas dimensiones de la grilla las asignamos a la variable `global_size`, mientras que en `local_size` definimos el tamaño de cada grupo de trabajo. Una vez organizada la descomposición del problema en la agrupación de tareas, lanzamos el kernel con los argumentos correspondientes (dimensiones de las matrices y los arrays que las contienen), y finalmente transferimos el resultado desde la memoria del *device* (`c_buf`) a la memoria del *host* (`c_cpu`).

CELDA 05

```
# Ajustamos el tamaño global (grilla)
global_size_x = (m + block_size_x - 1) // block_size_x * block_size_x
global_size_y = (n + block_size_y - 1) // block_size_y * block_size_y
global_size = (global_size_x, global_size_y)
local_size = (block_size_x, block_size_y)

# Ejecutamos el kernel
matmul_kernel = program.matmul
matmul_kernel.set_args(np.int32(m), np.int32(n), np.int32(l), a_buf, b_buf, c_buf)
cl.enqueue_nd_range_kernel(queue, matmul_kernel, global_size, local_size)

# Transferimos el resultado al host
cl.enqueue_copy(queue, c_cpu, c_buf)

# Esperamos a que terminen las operaciones
queue.finish()
```

Para finalizar, realizamos el producto de matrices usando el método `dot()` de NumPy, y comparamos los resultados obtenidos. Podemos ver que con una buena tolerancia absoluta, ambos resultados coinciden.

CELDA 06

```
# Calculamos el resultado en el host para comparar con el del device
c_npy = np.dot(a_cpu, b_cpu)

# Comparamos los resultados obtenidos en la CPU y la GPU
np.allclose(c_npy, c_cpu, rtol=1e-01, atol=1e-8)

-----
True
```

Al igual que el caso de PyCUDA, realizamos una paralelización muy simple del producto de matrices, sin aprovechar detalles propios del *hardware* que permiten una optimización de los recursos disponibles, aumentando de este modo en forma significativa la eficiencia del cómputo.

24.6. Lecturas recomendadas

- Wen-Mei W. Hwu, David B. Kirk e Izzat El Hajj. *Programming massively parallel processors*. 4.^a ed. Cambridge, United States: Morgan Kaufmann, sep. de 2023.
- NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation. Jun. de 2024. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- Bhaumik Vaidya. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA*. Birmingham, England: Packt Publishing, nov. de 2018.

Bibliografía

- [1] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.
- [2] Charles R. Harris et al. «Array programming with NumPy». En: *Nature* 585.7825 (sep. de 2020), págs. 357-362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2). URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [3] Pauli Virtanen et al. «SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python». En: *Nature Methods* 17 (2020), págs. 261-272. DOI: [10.1038/s41592-019-0686-2](https://doi.org/10.1038/s41592-019-0686-2).
- [4] J. D. Hunter. «Matplotlib: A 2D graphics environment». En: *Computing in Science & Engineering* 9.3 (2007), págs. 90-95. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- [5] Aaron Meurer et al. «SymPy: symbolic computing in Python». En: *PeerJ Computer Science* 3 (ene. de 2017), e103. DOI: [10.7717/peerj-cs.103](https://doi.org/10.7717/peerj-cs.103). URL: <https://doi.org/10.7717/peerj-cs.103>.
- [6] Makoto Matsumoto y Takuji Nishimura. «Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator». En: *ACM Trans. Model. Comput. Simul.* 8.1 (ene. de 1998), págs. 3-30. DOI: [10.1145/272991.272995](https://doi.org/10.1145/272991.272995). URL: <https://doi.org/10.1145/272991.272995>.
- [7] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Inf. téc. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, sep. de 2014.
- [8] Thomas Haslwanter. *An Introduction to Statistics with Python*. 1.^a ed. Statistics and Computing. Cham, Switzerland: Springer International Publishing, 2016. DOI: [10.1007/978-3-319-28316-6](https://doi.org/10.1007/978-3-319-28316-6).
- [9] John J. Schiller, R. Alu Srinivasan y Murray R. Spiegel. *Schaum's outline of probability and statistics*. 4.^a ed. Schaum's outline series. New York, NY: McGraw-Hill Professional, dic. de 2012.
- [10] G. N. Wilkinson y C. E. Rogers. «Symbolic Description of Factorial Models for Analysis of Variance». En: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 22.3 (1973), págs. 392-399. DOI: <https://doi.org/10.2307/2346786>. eprint: <https://rss.onlinelibrary.wiley.com/doi/pdf/10.2307/2346786>. URL: <https://rss.onlinelibrary.wiley.com/doi/abs/10.2307/2346786>.
- [11] Skipper Seabold y Josef Perktold. «statsmodels: Econometric and statistical modeling with python». En: *9th Python in Science Conference*. 2010.
- [12] A. Colin Cameron y Pravin K. Trivedi. «Econometric models based on count data. Comparisons and applications of some estimators and tests». En: *Journal of Applied Econometrics* 1.1 (1986), págs. 29-53. DOI: <https://doi.org/10.1002/jae.3950010104>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/jae.3950010104>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/jae.3950010104>.

- [13] Charles D. Keeling, Stephen C. Piper, Robert B. Bacastow, Martin Wahlen, Timothy P. Whorf, Martin Heimann y Harro A. Meijer. «Atmospheric CO₂ and ¹³CO₂ Exchange with the Terrestrial Biosphere and Oceans from 1978 to 2000: Observations and Carbon Cycle Implications». En: *A History of Atmospheric CO₂ and Its Effects on Plants, Animals, and Ecosystems*. Ed. por I.T. Baldwin et al. New York, NY: Springer New York, 2005, págs. 83-113. DOI: [10.1007/0-387-27048-5_5](https://doi.org/10.1007/0-387-27048-5_5). URL: https://doi.org/10.1007/0-387-27048-5_5.
- [14] Peter McCullagh. «What is a statistical model?» En: *The Annals of Statistics* 30.5 (2002), págs. 1225-1310. DOI: [10.1214/aos/1035844977](https://doi.org/10.1214/aos/1035844977). URL: <https://doi.org/10.1214/aos/1035844977>.
- [15] Galit Shmueli. «To Explain or to Predict?» En: *Statistical Science* 25.3 (2010), págs. 289-310. DOI: [10.1214/10-STS330](https://doi.org/10.1214/10-STS330). URL: <https://doi.org/10.1214/10-STS330>.
- [16] Susan Holmes y Wolfgang Huber. *Modern statistics for modern biology*. Cambridge, England: Cambridge University Press, feb. de 2019. URL: <https://web.stanford.edu/class/bios221/book/introduction.html>.
- [17] George E.P. Box, J. Stuart Hunter y William G. Hunter. *Statistics for experimenters*. 2.^a ed. Wiley Series in Probability and Statistics. Chichester, England: Wiley-Blackwell, mayo de 2005.
- [18] A. Colin Cameron y Pravin K. Trivedi. *Regression Analysis of Count Data*. 2.^a ed. Econometric Society Monographs. Cambridge University Press, 2013. DOI: [10.1017/CB09781139013567](https://doi.org/10.1017/CB09781139013567).
- [19] Terence Mills. *Applied time series analysis*. San Diego, CA: Academic Press, 2019.
- [20] George E P Box, Gwilym M Jenkins, Gregory C Reinsel y Greta M Ljung. *Time Series Analysis*. en. 5.^a ed. Wiley Series in Probability and Statistics. Nashville, TN: John Wiley & Sons, jun. de 2015.
- [21] Gilbert Strang. *Linear Algebra and Its Applications, 4th Edition*. 4th. Brooks Cole, 2006.
- [22] Stephen H. Friedberg, Arnold J. Insel y Lawrence E. Spence. *Linear Algebra*. 4th ed. Pearson, 2014.
- [23] R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.
- [24] A. Quarteroni, R. Sacco y F. Saleri. *Numerical Mathematics*. New York, United States: Springer-Verlag, 2000.
- [25] Gilbert Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007.
- [26] Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.
- [27] R. Bulirsch J. Stoer. *Introduction to Numerical Analysis*. 3rd ed. Texts in Applied Mathematics 12. Springer New York, 2002.
- [28] G Peter Lepage. «A new algorithm for adaptive multidimensional integration». En: *Journal of Computational Physics* 27.2 (1978), págs. 192-203. DOI: [https://doi.org/10.1016/0021-9991\(78\)90004-9](https://doi.org/10.1016/0021-9991(78)90004-9). URL: <https://www.sciencedirect.com/science/article/pii/0021999178900049>.
- [29] G. Peter Lepage. «Adaptive multidimensional integration: vegas enhanced». En: *Journal of Computational Physics* 439 (2021), pág. 110386. DOI: <https://doi.org/10.1016/j.jcp.2021.110386>. URL: <https://www.sciencedirect.com/science/article/pii/S0021999121002813>.

- [30] Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, ago. de 2020. DOI: [10.1017/9781108772310](https://doi.org/10.1017/9781108772310). URL: <https://doi.org/10.1017/9781108772310>.
- [31] S. Weinzierl. «Introduction to Monte Carlo methods». En: *arXiv: High Energy Physics - Phenomenology* (2000). URL: <https://arxiv.org/abs/hep-ph/0006269>.
- [32] Malvin H. Kalos y Paula A. Whitlock. *Monte Carlo Methods*. Weinheim: John Wiley & Sons, 2009.
- [33] Dirk Kroese, Thomas Taimre y Zdravko I. Botev. *Handbook of Monte Carlo Methods*. Hoboken, New Jersey: John Wiley & Sons, 2011.
- [34] Richard P. Feynman. *The Feynman Lectures on Physics, boxed set: The New Millennium Edition*. Basic Books, ene. de 2011. URL: <https://feynmanlectures.caltech.edu/info/>.
- [35] L D. Humphreys y R. Shammass. «Finding Unpredictable Behavior in a Simple Ordinary Differential Equation». En: *The College Mathematics Journal* 31.5 (2000), págs. 338-346. DOI: [10.1080/07468342.2000.11974171](https://doi.org/10.1080/07468342.2000.11974171). eprint: <https://doi.org/10.1080/07468342.2000.11974171>. URL: <https://doi.org/10.1080/07468342.2000.11974171>.
- [36] John Charles Butcher. *Numerical methods for ordinary differential equations*. Chichester, England Hoboken, NJ: Wiley, 2008.
- [37] William H Press, Saul A Teukolsky, William T Vetterling y Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge, UK New York: Cambridge University Press, 2007.
- [38] A.N. Tijonov y A.A. Samarsky. *Ecuaciones de la Física Matemática*. Editorial Mir, Moscú, 1983.
- [39] Hans Petter Langtangen y Anders Logg. *Solving PDEs in Python*. Springer, 2017. DOI: [10.1007/978-3-319-52462-7](https://doi.org/10.1007/978-3-319-52462-7).
- [40] Douglas N. Arnold y Anders Logg. «Periodic table of the finite elements». En: *SIAM News* 47.9 (2014), pág. 212.
- [41] Hans Petter Langtangen y Svein Linge. *Finite Difference Computing with PDEs*. Springer International Publishing, 2017. DOI: [10.1007/978-3-319-55456-3](https://doi.org/10.1007/978-3-319-55456-3). URL: <https://doi.org/10.1007/978-3-319-55456-3>.
- [42] M.G. Larson y F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2013.
- [43] Anders Logg, Kent-Andre Mardal, Garth N. Wells et al. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. por Anders Logg, Kent-Andre Mardal y Garth N. Wells. Springer, 2012. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8).
- [44] M.A. Styblinski y T.-S. Tang. «Experiments in nonconvex optimization: Stochastic approximation with function smoothing and simulated annealing». En: *Neural Networks* 3.4 (1990), págs. 467-483. DOI: [https://doi.org/10.1016/0893-6080\(90\)90029-K](https://doi.org/10.1016/0893-6080(90)90029-K). URL: <https://www.sciencedirect.com/science/article/pii/089360809090029K>.
- [45] Q. Huangfu y J. A. J. Hall. «Parallelizing the dual revised simplex method». En: *Mathematical Programming Computation* 10.1 (mar. de 2018), págs. 119-142. DOI: [10.1007/s12532-017-0130-5](https://doi.org/10.1007/s12532-017-0130-5). URL: <https://doi.org/10.1007/s12532-017-0130-5>.

- [46] H. Kellerer, U. Pferschy y D. Pisinger. *Knapsack Problems*. Berlin: Springer-Verlag, 2004. DOI: [10.1007/978-3-540-24777-7](https://doi.org/10.1007/978-3-540-24777-7).
- [47] S. Martello y P. Toth. *Knapsack Problems*. West Sussex, England: John Wiley & Sons Ltd., 1990.
- [48] Stephen Boyd y Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. DOI: [10.1017/CB09780511804441](https://doi.org/10.1017/CB09780511804441).
- [49] Wayne L Winston y Jeffrey B Goldberg. *Operations research: Applications and algorithms*. Brooks/Cole, 2004.
- [50] Jorge Nocedal y Stephen Wright. *Numerical Optimization*. en. 2.^a ed. Springer Series in Operations Research and Financial Engineering. New York, NY: Springer, jul. de 2006.
- [51] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.
- [52] Melanie Mitchell. *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996. DOI: <https://doi.org/10.7551/mitpress/3927.001.0001>.
- [53] Warren S. McCulloch y Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». En: *The bulletin of mathematical biophysics* 5.4 (dic. de 1943), págs. 115-133. DOI: [10.1007/BF02478259](https://doi.org/10.1007/BF02478259). URL: <https://doi.org/10.1007/BF02478259>.
- [54] George Cybenko. «Approximation by superpositions of a sigmoidal function». En: *Mathematics of control, signals and systems* 2.4 (1989), págs. 303-314.
- [55] Kurt Hornik. «Approximation capabilities of multilayer feedforward networks». En: *Neural Networks* 4.2 (1991), págs. 251-257. DOI: [https://doi.org/10.1016/0893-6080\(91\)90009-T](https://doi.org/10.1016/0893-6080(91)90009-T). URL: <http://www.sciencedirect.com/science/article/pii/089360809190009T>.
- [56] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus y Shimon Schocken. «Multilayer feedforward networks with a nonpolynomial activation function can approximate any function». En: *Neural Networks* 6.6 (1993), págs. 861-867. DOI: [https://doi.org/10.1016/S0893-6080\(05\)80131-5](https://doi.org/10.1016/S0893-6080(05)80131-5). URL: <http://www.sciencedirect.com/science/article/pii/S0893608005801315>.
- [57] Jatinder N.D Gupta y Randall S Sexton. «Comparing backpropagation with a genetic algorithm for neural network training». En: *Omega* 27.6 (1999), págs. 679-684. DOI: [https://doi.org/10.1016/S0305-0483\(99\)00027-4](https://doi.org/10.1016/S0305-0483(99)00027-4). URL: <https://www.sciencedirect.com/science/article/pii/S0305048399000274>.
- [58] Seyedali Mirjalili. *Evolutionary Algorithms and Neural Networks*. 1.^a ed. Studies in computational intelligence. Cham, Switzerland: Springer International Publishing, jul. de 2019.
- [59] Micheal Lanham. *Evolutionary Deep Learning*. New York, NY: Manning Publications, jul. de 2023.
- [60] Xavier Glorot y Yoshua Bengio. «Understanding the difficulty of training deep feedforward neural networks». En: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. por Yee Whye Teh y Mike Titterton. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, mayo de 2010, págs. 249-256. URL: <https://proceedings.mlr.press/v9/glorot10a.html>.
- [61] C. E. Shannon. «A mathematical theory of communication». En: *The Bell System Technical Journal* 27.3 (1948), págs. 379-423. DOI: [10.1002/j.1538-7305.1948.tb01338.x](https://doi.org/10.1002/j.1538-7305.1948.tb01338.x).

- [62] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. [http : / / www . deeplearningbook.org](http://www.deeplearningbook.org). MIT Press, 2016.
- [63] Stuart J. Russel y Peter Norvig. *Artificial Intelligence. A Modern Approach*. 4.^a ed. Harlow, United Kingdom: Pearson Education Limited, 2022.
- [64] C.J. Thompson, Sahngyun Hahn y M. Oskin. «Using modern graphics architectures for general-purpose computing: a framework and analysis». En: *35th Annual IEEE/ACM International Symposium on Microarchitecture , 2002. (MICRO-35). Proceedings*. 2002, págs. 306-317. DOI: [10.1109/MICRO.2002.1176259](https://doi.org/10.1109/MICRO.2002.1176259).
- [65] Jeff Bolz, Ian Farmer, Eitan Grinspun y Peter Schröder. «Sparse matrix solvers on the GPU: conjugate gradients and multigrid». En: *ACM Trans. Graph.* 22.3 (jul. de 2003), págs. 917-924. DOI: [10.1145/882262.882364](https://doi.org/10.1145/882262.882364). URL: <https://doi.org/10.1145/882262.882364>.
- [66] Jens Krüger y Rüdiger Westermann. «Linear algebra operators for GPU implementation of numerical algorithms». En: *ACM Trans. Graph.* 22.3 (jul. de 2003), págs. 908-916. DOI: [10.1145/882262.882363](https://doi.org/10.1145/882262.882363). URL: <https://doi.org/10.1145/882262.882363>.
- [67] NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation. Jun. de 2024. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [68] Rajat Raina, Anand Madhavan y Andrew Y Ng. «Large-scale deep unsupervised learning using graphics processors». En: *Proceedings of the 26th annual international conference on machine learning*. 2009, págs. 873-880.
- [69] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov y Ahmed Fasih. «PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation». En: *Parallel Computing* 38.3 (2012), págs. 157-174. DOI: [10.1016/j.parco.2011.09.001](https://doi.org/10.1016/j.parco.2011.09.001).
- [70] Wen-Mei W. Hwu, David B. Kirk e Izzat El Hajj. *Programming massively parallel processors*. 4.^a ed. Cambridge, United States: Morgan Kaufmann, sep. de 2023.
- [71] Bhaumik Vaidya. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA*. Birmingham, England: Packt Publishing, nov. de 2018.