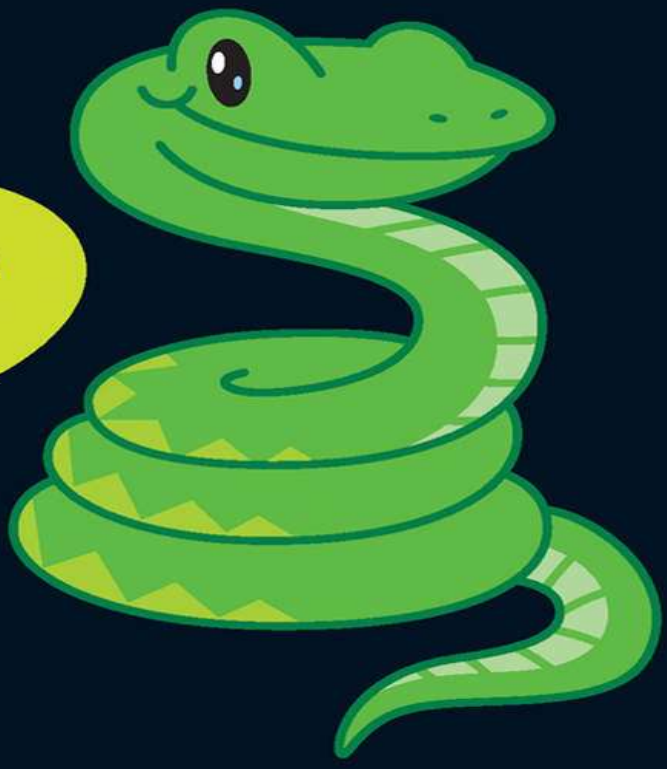
Python 3
al descubierto

2ª edición revisada

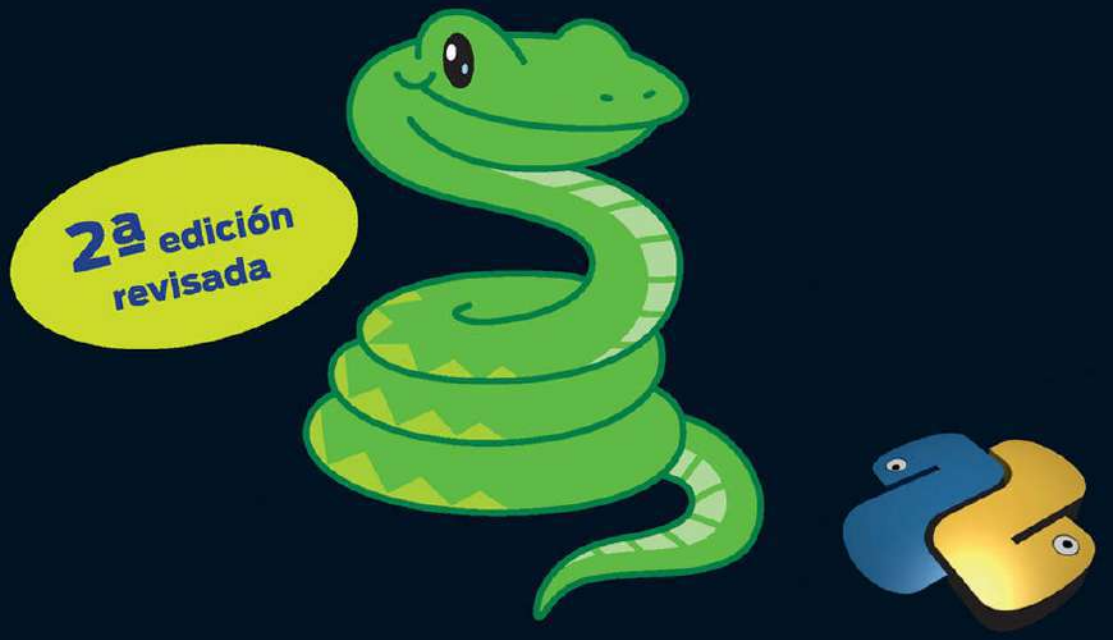


Arturo Fernández Montoro

Alfaomega

libros R

Python 3
al descubierto



Arturo Fernández Montoro

Alfaomega

libros R

Python 3
al descubierto

2ª edición revisada

Arturo Fernández Montoro

Alfaomega

libros RC

Python 3 al descubierto
Arturo Fernández Montoro
ISBN: 978-84-939450-4-6 edición original publicada por RC Libros, Madrid, España

Segunda edición: Alfaomega Grupo Editor, México, septiembre 2013

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: http://www.alfaomega.com.mx
E-mail: atencionalcliente@alfaomega.com.mx

ISBN: 978-607-707-718-3
eISBN: 978-607-62200-8-5

Datos Catalográficos
Fernández, Arturo
Python 3 al descubierto
Alfaomega Grupo Editor, S.A. de C.V.,
México

ISBN: 978-607-707-718-3
eISBN: 978-607-62200-8-5

Formato: 17 x 23 cm
Páginas: 276

La transformación a libro electrónico del presente título fue realizada por
Sextil Online, S.A. de C.V./ Editorial Ink ® 2016.
+52 (55) 52 54 38 52
contacto@editorial-ink.com
www.editorial-ink.com

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in México.

Empresas del grupo:
México: Alfaomega Grupo Editor, S.A. de C.V. - Pitágoras 1139, Col. Del Valle, México, D.F. - C.R 03100. Tel.: (52-55) 5575-5022 - Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 E-mail: atencionalcliente@alfaomega.com.mx
Colombia: Alfaomega Colombiana S.A. - Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 - E-mail: cliente@alfaomega.com.co
Chile: Alfaomega Grupo Editor, S.A. -Av. Providencia 1443. Oficina 24, Santiago, Chile Tel.: (56-2) 2235-4248 - Fax: (56-2) 2235-5786 - E-mail: agechile@alfaomega.cl
Argentina: Alfaomega Grupo Editor Argentino, S.A. - Paraguay 1307 PB. Of. 11, C.R 1057, Buenos Aires, Argentina, -Tel./Fax: (54-11) 4811-0887 y 4811 7183 - E-mail: ventas@alfaomegaeditor.com.ar

PRÓLOGO

En la actualidad Python es uno de los lenguajes de programación con mayor proyección. Su facilidad de uso, su librería estándar y la cantidad de librerías adicionales que existen contribuyen a que sean muchos los desarrolladores de software que optan por su utilización para llevar a cabo sus proyectos.

Python es un lenguaje de propósito general, de alto nivel, interpretado y que admite la aplicación de diferentes paradigmas de programación, como son, por ejemplo, la programación procedural, imperativa y la orientación a objetos.

La programación científica, la programación de sistemas o las aplicaciones web son ámbitos en los que habitualmente se emplea Python como lenguaje de programación principal. También puede ser empleado para desarrollar aplicaciones de escritorio con interfaz gráfica de usuario, integrar componentes escritos en diferentes lenguajes de programación o incluso desarrollar juegos.

Dadas sus principales características, Python es un lenguaje ideal para el prototipado. Para diversos tipos de aplicaciones pueden construirse rápidamente prototipos, facilitando el desarrollo del modelo final en otros lenguajes que ofrezcan mayor rendimiento como es el caso de C y C++. Todo ello sin perder de vista el hecho de que Python puede utilizarse como un lenguaje más de alto nivel.

El presente libro no pretende ser un manual de referencia al uso, sino ofrecer una completa visión del lenguaje desde un punto de vista práctico. Con ella se pretende que el lector consiga familiarizarse rápidamente con el lenguaje, aprendiendo sus fundamentos y descubriendo cómo utilizarlo para desarrollar diferentes tipos de aplicaciones.

Los primeros cinco capítulos están dedicados a los aspectos más importantes del lenguaje. En ellos aprenderemos sobre las estructuras y tipos de datos básicos, sentencias de control, cómo aplicar la programación orientada a objetos y detalles más avanzados sobre el lenguaje. Los siguientes capítulos, que pueden ser considerados como una segunda parte, están orientados a utilizar Python para

desarrollar aplicaciones que interactúen con bases de datos, manejen ficheros y utilicen diversos servicios de Internet. Seguidamente nos centraremos en la instalación y distribución de programas desarrollados en el lenguaje de programación que nos ocupa. Por último, descubriremos cómo diseñar y ejecutar pruebas unitarias, formando parte estas de una de las fases más importantes en el desarrollo de software.

Esperamos que el lector disfrute aprendiendo los fundamentos de este lenguaje de programación y pueda rápidamente aplicar los conceptos aprendidos a sus propios proyectos.

ÍNDICE

PRIMEROS PASOS

INTRODUCCIÓN

Este primer capítulo será nuestra primera toma de contacto con Python.

Comenzaremos con una sencilla descripción del lenguaje y una serie de datos que nos ayuden a tener una visión general del mismo. Posteriormente, haremos un breve recorrido a su historia, para pasar después a examinar sus principales características. Después, realizaremos la primera incursión práctica escribiendo nuestro primer código en este lenguaje. Los dos últimos apartados los dedicaremos a ver con qué herramientas de desarrollo contamos y cuáles son las principales novedades de Python 3.

¿QUÉ ES PYTHON?

Básicamente, Python es un lenguaje de programación de alto nivel, *interpretado* y *multipropósito.* En los últimos años su utilización ha ido constantemente creciendo y en la actualidad es uno de los lenguajes de programación más empleados para el desarrollo de software.

Python puede ser utilizado en diversas plataformas y sistemas operativos, entre los que podemos destacar lo más populares, como Windows, Mac OS X y Linux. Pero, además, Python también puede funcionar en *smartphones,* Nokia desarrolló un intérprete de este lenguaje para su sistema operativo *Symbian.*

¿Tiene Python un ámbito específico? Algunos lenguajes de programación sí que lo tienen. Por ejemplo, PHP fue ideado para desarrollar aplicaciones web. Sin embargo, este no es el caso de Python. Con este lenguaje podemos desarrollar software para aplicaciones científicas, para comunicaciones de red, para aplicaciones de escritorio con Interfaz gráfica de usuario (GUI), para crear juegos, para *smartphones* y por supuesto, para aplicaciones web.

Fig. 1-1 Logo de Python

Empresas y organizaciones del calibre de *Industrial Light & Magic, Walt Disney,* la *NASA,* Google, Yahoo!, *Red Hat* y Nokia hacen uso intensivo de este lenguaje para desarrollar sus productos y servicios. Esto demuestra que Python puede ser utilizado en diversos tipos de sectores, con independencia de su actividad empresarial.

Entre las principales razones para elegir Python, son muchos los que argumentan que sus principales características lo convierten en un lenguaje muy productivo. Se trata de un lenguaje potente, flexible y con una sintaxis clara y

concisa. Además, no requiere dedicar tiempo a su compilación debido a que es interpretado.

Python es *open source*, cualquiera puede contribuir a su desarrollo y divulgación. Además, no es necesario pagar ninguna licencia para distribuir software desarrollado con este lenguaje. Hasta su intérprete se distribuye de forma gratuita para diferentes plataformas.

La última versión de Python recibe varios nombres, entre ellos, *Python 3000* y *Py3K*, aunque, habitualmente, se le denomina simplemente *Python 3*.

Un poco de historia

El origen del lenguaje Python se remonta a principios de los noventa. Por este tiempo, un investigador holandés llamado Guido van Rossum, que trabajaba en el centro de investigación CWI *(Centrum Wiskunde & Informática)* de Ámsterdam, es asignado a un proyecto que consistía en el desarrollo de un sistema operativo distribuido llamado *Amoeba*. Por aquel tiempo, el CWI utilizaba un lenguaje de programación llamado *ABC*. En lugar de emplear este lenguaje para el proyecto *Amoeba*, Guido decide crear uno nuevo que pueda superar las limitaciones y problemas con los que se había encontrado al trabajar en otros proyectos con ABC. Así pues, es esta la principal motivación que dio lugar al nacimiento de Python.

La primera versión del lenguaje ve la luz en 1991, pero no es hasta tres años después cuando decide publicarse la versión 1.0. Inicialmente el CWI decidió liberar el intérprete del lenguaje bajo una licencia *open source* propia, pero en septiembre de 2000 y coincidiendo con la publicación de la versión 1.6, se toma la decisión de cambiar la licencia por una que sea compatible con la licencia GPL (*GNU General Public License).* Esta nueva licencia se denominará *Python Software Foundation License* y se diferencia de la GPL al ser una licencia no *copyleft*. Este hecho implica que es posible modificar el código fuente y desarrollar código derivado sin la necesidad de hacerlo *open source*.

Hasta el momento solo se ha liberado tres versiones principales, teniendo cada una de ellas diversas actualizaciones. En lo que respecta a la versión 2, la última en ser liberada fue la 2.7, en julio de 2010. En el momento de escribir estas líneas, la versión 3 cuenta con la actualización 3.2, liberada en febrero de

2011. Ambas versiones, la de 2 y 3, son mantenidas por separado. Esto implica que, tanto la 2.7 como la 3.2 se consideran estables pero, lógicamente, correspondientes a diferentes versiones. ¿Por qué mantener ambas versiones y no seguir una evolución lógica? La respuesta a esta pregunta es fácil de responder: Entre ambas versiones existen diferencias que las hacen incompatibles. Posteriormente, nos centraremos en este aspecto, comentando las principales diferencias entre ambas y viendo las novedades que supone la versión 3 con respecto a su predecesora.

Entre las características de las primeras versiones de Python cabe destacar el soporte de la orientación a objetos, el manejo de excepciones y el soporte de estructuras de datos de alto nivel, como, por ejemplo, las listas y los diccionarios. Además, desde su desarrollo inicial, se tuvo en cuenta que el código escrito en este lenguaje fuera fácil de leer y de aprender, sin que esto suponga renunciar a características y funcionalidades avanzadas.

Muchos se preguntan el origen del nombre de este lenguaje de programación. Guido van Rossum decidió darle este nombre en honor a la serie de televisión *Monty Python's Flying Circus*, de la cual era fan. Esta es una serie cómica protagonizada por el grupo de humoristas *Monty Python*, famoso por películas como *La vida de Brian o El sentido de la vida.* Desde el principio de su diseño, se pretendía que Python fuera un lenguaje que resultara divertido de utilizar, de ahí que en el nombre influyera la mencionada serie cómica. También resulta curioso que, tanto en tutoriales, como en ejemplos de código, se suelan utilizar referencias a los *Monty Python.* Por ejemplo, en lugar de emplear los tradicionales nombres de variables *foo* y *bar,* se suele utilizar *spam* y *egss*, en referencia a *sketchs* de este grupo de cómicos.

El desarrollo y promoción de Python se lleva a cabo a través de una organización, sin ánimo de lucro, llamada *Python Software Foundation*, que fue creada en marzo de 2001. Entre las actividades que realiza esta organización destacan el desarrollo y distribución oficial de Python, la gestión de la propiedad intelectual del código y documentos realizados, así como la organización de conferencias y eventos dedicados a poner en contacto a todas aquellas personas interesadas en este lenguaje de programación.

Python tiene un claro carácter *open source* y la *Python Software Foundation* invita, a cualquiera que quiera hacerlo, a contribuir al desarrollo y promoción de este lenguaje de programación. Aquellos lectores interesados en contribuir pueden echar un vistazo a la página oficial dedicada a la *comunidad* de Python

(ver referencias).

Principales características

No hay duda de que a la hora de elegir un lenguaje es muy importante conocer sus características. Ver qué nos puede ofrecer resulta determinante para tomar la decisión adecuada. Son muchas las empresas que se plantean esta cuestión a la hora de elegir un lenguaje de programación para un determinado proyecto. Esto también es extrapolable a proyectos *open source* o aquellos proyectos personales que requieren del uso de un lenguaje de programación. Ya sabemos que Python es un lenguaje de propósito general, dinámico e interpretado. Sin embargo, Python puede ofrecernos mucho más, tal y como descubriremos a continuación.

Dos de las principales características del lenguaje Python son, por un lado que es *interpretado* y, por otro lado, que es *multiplataforma*. Lo primero significa que no es necesario compilar el código para su ejecución, ya que existe un intérprete que se encarga de leer el fichero fuente y ejecutarlo. Gracias a este funcionamiento es posible ejecutar el mismo código en distintas plataformas y sistemas operativos sin necesidad de cambiar el código fuente, bastará con tener instalado el intérprete. Eso sí, la versión de este intérprete es *nativa* para cada plataforma. En este sentido, Python es similar a Perl o a Ruby y difiere de otros lenguajes como C++ y Objective-C.

Habitualmente, a los programas en Python se les denomina *scripts*. En realidad, *script* es el término que se suele emplear para los ficheros de código fuente escritos en Python, pudiendo un programa contar con uno o más de estos scripts.

Los programadores de Python suelen llamar indistintamente con este nombre tanto al lenguaje como al intérprete del mismo. Deberemos tener esto en cuenta, debido a que es habitual escuchar "voy a instalar Python" o "la versión que tengo instalada de Python es la 3.2". En estos casos se hace referencia directa al intérprete y no al lenguaje.

La interacción con el intérprete del lenguaje se puede hacer directamente a través de la *consola*. Tal y como veremos posteriormente, durante la instalación de Python, se instala un componente llamado *shell o consola* que permite

ejecutar directamente código Python a través de una *terminal o interfaz de comandos.*

En lo que respecta a la sintaxis del lenguaje cabe destacar su simplicidad; es decir, gracias a la misma, es sencillo escribir código que sea fácil de leer. Este factor es muy importante, ya que, además de facilitar el aprendizaje del lenguaje, también nos ayuda a que nuestro código sea más fácil de mantener.

Python carece de tipos propiamente dichos, es decir, es un lenguaje con *tipado dinámico.* Los programadores de C++ y Java están acostumbrados a declarar cada variable de un tipo específico. Este proceso no es necesario en Python, ya que el tipo de cada variable se fija en el momento de su asignación. Como consecuencia de este hecho, una variable puede cambiar su tipo durante su ciclo de vida sin necesidad explícita de ser declarado. Dado que puede ser interesante consultar el tipo de una variable en un momento dado, Python nos ofrece una serie de funciones que nos dan este tipo de información.

Además de soportar la orientación a objetos, Python también nos permite utilizar otros paradigmas de programación, como, por ejemplo, la programación funcional y la imperativa. En la actualidad, Python es considerado uno de los lenguajes que más facilidades ofrecen para enseñar programación orientada a objetos. A esto contribuyen su sintaxis, los mecanismos de introspección que incorpora y el soporte para la implementación de herencia sencilla y múltiple.

Con respecto a su sintaxis, una de las diferencias más destacables es el uso de la *indentación.* Diferentes niveles de indentación son utilizados para marcar las sentencias que corresponden al mismo bloque. Por ejemplo, todas las sentencias que deban ser ejecutadas dentro de un bloque *if* llevarán el mismo nivel de indentación, mientras que el resto utilizarán un nivel diferente, incluida la sentencia que contiene la condición o condiciones del mencionado *if.* Además, cada sentencia no necesita un punto y coma (;), como sí ocurre en lenguajes como C/C++, PHP y Java. En Python basta con que cada sentencia vaya en una línea diferente. Por otro lado, tampoco se hace uso de las llaves *({})* para indicar el principio y fin de bloque. Tampoco se emplean palabras clave como *begin* y *end.* Simplemente se utilizan los dos puntos (:) para marcar el comienzo de bloque y el cambio de indentación se encarga de indicar el final.

Para facilitar la programación, Python incluye una serie de estructuras de datos de alto nivel, como son, por ejemplo, las listas, los diccionarios, cadenas de texto *(strings),* tuplas y conjuntos. Por otro lado, su *librería estándar* incorpora multitud de funciones que pueden ser utilizadas en diversos ámbitos,

entre ellas podemos mencionar, desde aquellas básicas para manejar *strings*, hasta las que pueden ser usadas en programación criptográfica, pasando por otros de nivel intermedio, como son las que permiten manejar ficheros ZIP, trabajar con ficheros CSV o realizar comunicaciones de red a través de distintos protocolos estándar. Todo ello, sin necesidad de instalar librerías adicionales. Comúnmente, se emplea la frase *batteries included* para resaltar este hecho.

A diferencia de lenguajes compilados, como C++, en Python existe un *recolector de basura (garbage collector).* Esto significa que no es necesario pedir y liberar memoria, de forma explícita, para crear y destruir objetos. El intérprete lo hará automáticamente cuando sea necesario y el recolector se encargará de gestionar la memoria para evitar los temidos *memory leaks.*

Otro de los aspectos interesantes del lenguaje es su facilidad para interactuar con otros lenguajes de programación. Esto es posible gracias a los *módulos* y *extensiones.* ¿Cuándo puede ser útil esto? Supongamos que ya contamos con un programa en C++ que se encarga de realizar, por ejemplo, una serie de complejas operaciones matemáticas. Por otro lado, estamos realizando un desarrollo en Python y nos damos cuenta que sería interesante contar con la funcionalidad que nos ofrece el mencionado programa en C++. En lugar de reescribir este programa en Python, podemos comunicar ambos a través de la interfaz que Python incorpora para ello.

Existen diversas implementaciones del intérprete de Python, es decir, el código escrito en Python puede ejecutarse desde diferentes sistemas preparados para ello. La implementación más popular es la llamada *CPython*, escrita en el lenguaje de programación C, aunque existen otras como *Jython*, la cual está desarrollada en el lenguaje Java, e *IronPython*, que permite la ejecución en la plataforma .NET de Microsoft. El siguiente apartado lo dedicaremos a la instalación del intérprete de Python implementado en CPython y para la cual existen versiones para diferentes sistemas operativos.

INSTALACIÓN

A continuación, nos centraremos en la instalación del intérprete de Python y sus herramientas asociadas en las tres familias más populares de sistemas operativos. Dentro de las mismas y en concreto, explicaremos el proceso de instalación en Windows, Mac OS X y las principales distribuciones de GNU/Linux.

Windows

Para la instalación de Python en Windows recurriremos al programa de instalación ofrecido desde el sitio web oficial (ver referencias) de este lenguaje de programación. En concreto, accederemos a la página principal de descargas (ver referencias) y haremos clic sobre el enlace que referencia a la última versión liberada de Python 3. Dicho enlace nos llevará a una nueva página web donde se nos ofrecen una serie de ficheros, tanto binarios, como fuentes, para diferentes sistemas operativos y arquitecturas de procesador. Antes de continuar es conveniente averiguar si nuestro Windows 7 es de 32 o de 64 bits. La mayoría de fabricantes de PC instalan la versión de este sistema operativo en función del tipo de arquitectura que incorpora el procesador de la máquina en cuestión. Los actuales PC suelen contar con procesadores de 64b. Podemos comprobar qué tipo de sistema operativo tiene instalado nuestro PC accediendo a la opción de menú *Panel de Control > Sistema,* apartado *Tipo de sistema.* Una vez que conocemos este dato, podemos volver a la página web de descargas y buscar el enlace para el fichero de instalación de Python que corresponde a Windows y al tipo de arquitectura de nuestro PC. Por ejemplo, si contamos con un sistema de 64b, haremos clic sobre *Windows x86-64 MSI Installer (3.2.2).* Automáticamente comenzará la descarga del fichero binario apuntado por el enlace, que no es otro que un programa de instalación guiado a través de un asistente o *wizard.*

Figura 1-2. Selección de la instalación de Python para todos los usuarios o solo para el actual

Al finalizar la descarga del programa de instalación, haremos doble clic sobre el mismo para comenzar el proceso de instalación propiamente dicho. El primer cuadro de diálogo (figura 1-2) nos pregunta si deseamos realizar la instalación para todos los usuarios del sistema o solamente para el usuario que está ejecutando el asistente. Por defecto aparece seleccionada la primera opción.

Pulsando sobre el botón *Next* accederemos al siguiente paso, el cual nos pide seleccionar el directorio donde serán instalados los ficheros (figura 1-3).

Figura 1-3. Selección del directorio base para de la instalación de Python

Avanzamos un paso más y se nos ofrece la personalización de la instalación, siendo posible elegir qué componentes deseamos instalar (figura 1-4). Salvo que tengamos muy claro cómo hacer esta selección, es recomendable utilizar las opciones marcadas por defecto. Al pulsar sobre el botón *Next* se procederá a la copia de ficheros al disco duro y al finalizar este proceso veremos un mensaje informándonos de ello. Por último, el asistente nos pide reiniciar el PC para completar la instalación.

Comprobar si la instalación de Python 3 se ha realizado correctamente en nuestro Windows es sencillo, basta con acceder al menú de inicio y teclear *python* en el cuadro de diálogo para buscar programas. Como resultado de la búsqueda nos deben aparecer varios programas, entre ellos, *IDLE (Python GUI)* y *Python (command line)*. El primero nos da acceso a una interfaz de comandos, en modo gráfico, donde podemos interactuar con el intérprete del lenguaje. El segundo nos permite abrir la misma interfaz pero en modo consola, como si lanzáramos un comando a través de la interfaz de comandos de Windows invocada a través del comando *cmd*.

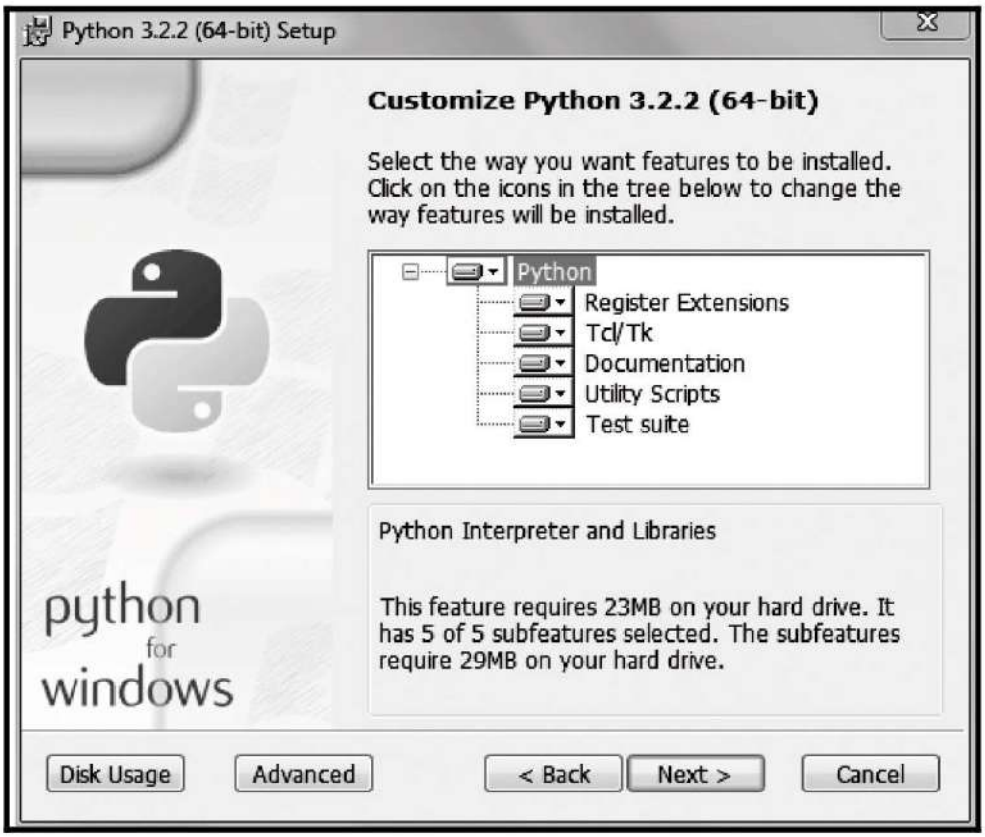


Figura 1-4. Personalización de la instalación de Python

La interfaz gráfica presenta algunas ventajas funcionales con respecto a la textual, por ejemplo, el resaltado de sintaxis del código, el autocompletado de palabras clave o la opción de utilizar un depurador. En ambas interfaces de comandos, observaremos cómo en la primera línea aparece el número de versión del intérprete de Python que tenemos instalado y que estamos usando.

En realidad, *IDLE* es algo más que una interfaz gráfica para interactuar con el intérprete de Python, ya que es un sencillo, pero funcional *entorno integrado de desarrollo.* De ahí, que cuenta con características ya comentadas, como la posibilidad de depurar código. También es posible editar ficheros y ejecutarlos directamente. Para más información sobre las características de este entorno de desarrollo, recomendamos echar un vistazo a la documentación oficial sobre el mismo (ver referencias).

Esta interfaz de comandos del intérprete de Python nos será muy útil para llevar a cabo nuestra primera práctica toma de contacto con el lenguaje. Además, podemos recurrir a ella siempre que lo necesitemos, para, por ejemplo, probar

ciertas líneas de código o sentencias de control.

Obviamente, además de la mencionada interfaz de comandos, el intérprete de Python ha sido instalado. Esto significa que podemos crear un fichero de texto con código Python, salvarlo con la extensión *.py* y ejecutarlo haciendo clic sobre el mismo.

Mac OS X

El sistema operativo de Apple incluye Python preinstalado de serie. En concreto, la versión *Lion* (10.7) incorpora la versión 2.7 de Python, mientras que su predecesora, llamada *Snow Leopard,* cuenta, por defecto, con la versión 2.6. Sin embargo, para utilizar Python 3 en nuestro Mac deberemos instalarlo. Para ello, basta con recurrir al binario de instalación ofrecido desde la página web de descargas del sitio oficial de Python. Desde esta página se ofrecen dos binarios diferentes: uno para Mac OS X 10.6 y 10.7, para ordenadores con procesador Intel, y otro específico para la arquitectura de procesador *PPC.* Haciendo clic sobre el correspondiente enlace, deberemos elegir en función del sistema que tenga instalado nuestro Mac, se procederá a la descarga de un fichero *DMG,* el cual podemos ejecutar una vez descargado. Para ello, bastará con hacer clic sobre el mismo. Será entonces cuando se abrirá una nueva ventana en *Finder* que nos mostrará una serie de archivos (figura 1-5).

Figura 1-5. Ficheros contenidos en la imagen DMG del instalador de Python

Haciendo doble clic sobre el fichero *Python.mpkg* se lanzará el asistente que nos guiará en el proceso de instalación. La primera ventana que aparece nos describe los programas que van a ser instalados, nos invita a leer el fichero *ReadMe.text* y nos propone continuar a través del botón *Continue.*

En el siguiente paso del asistente se nos solicita que indiquemos la unidad de

disco donde se va a realizar la instalación. Después de pulsar el botón para continuar el proceso, el software será instalado en la ubicación seleccionada. Finalmente, aparecerá un mensaje indicándonos que la instalación se ha realizado correctamente.

Al abrir una ventana del *Finder* y acceder a *Aplicaciones*, observaremos que tenemos una nueva carpeta llamada *Python 3.2*. Dentro de la misma aparecen varios archivos. Entre ellos *IDLE*, un fichero HTML de documentación y un *script* que nos permitirá fijar la versión 3.2 de Python como el intérprete por defecto, sustituyendo así a la versión 2 que Apple incluye por defecto en su sistema operativo.

Figura 1-6. Pantalla inicial del asistente para la instalación de Python

Aquellos programadores de Mac, acostumbrados a utilizar la interfaz de comandos, pueden lanzar *Terminal* y ejecutar el comando *python3.2*. Este comando invocará al intérprete del lenguaje y nos permitirá utilizar la terminal para interactuar con él.

Figura 1-7. Selección del disco para la instalación de Python

Linux

La mayoría de las distribuciones de GNU/Linux, como, por ejemplo, Ubuntu, Fedora y Debian, incluyen e instalan Python por defecto. Algunas de ellas utilizan la versión 2.6, mientras que otras se decantan por la 2.7. La instalación de Python 3 en Linux es sencilla, ya que las mencionadas distribuciones incluyen paquetes binarlos listos para su instalación. En función de la distribución que estemos utilizando, basta con emplear una de las herramientas de instalación de software con las que cuenta específicamente cada una de ellas. Por ejemplo, en Ubuntu 11.10 basta con acceder al *Centro de Software* y realizar una búsqueda por *python3*. Entre los resultados de la búsqueda, veremos que aparecerá un paquete llamado *python3*, haciendo doble clic sobre el mismo se procederá a la instalación. Si preferimos utilizar la interfaz de comandos, bastará con lanzar una consola y ejecutar el siguiente comando:

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

En distribuciones de GNU/Linux basadas en paquetes con formato *RPM*, como, por ejemplo, Fedora, lanzaremos el siguiente comando, como usuario *root*, desde una terminal:

```
# yum install python3
```

Una vez que finalice la instalación, con independencia de la distribución que estemos utilizando, bastará con acceder a la línea de comandos y lanzar el siguiente comando para comenzar a utilizar la consola interactiva del intérprete

de Python:

```
$ python3
```

Al contrario que en Mac y en Windows, para utilizar IDLE en Linux deberemos instalar el correspondiente binario ofrecido por nuestra distribución. El nombre del paquete binario en cuestión se llama idle3 en Ubuntu. En el caso de Fedora, será necesario instalar un paquete llamado python3-tools. Sin embargo, el nombre del ejecutable para ambas distribuciones es idle3, lo que significa que, lanzando directamente este comando desde la consola podemos disfrutar de este entorno integrado de desarrollo.

Debemos tener en cuenta que la invocación al comando *python* seguirá lanzando la versión 2 del intérprete. Si deseamos cambiar este comportamiento, podemos crear un *enlace simbólico* para que el comando *python* apunte directamente a la versión 3. Para ello basta ejecutar, como usuario *root*, los siguientes comandos:

```
# mv /usr/bin/python /usr/bin/python2
# ln -s /usr/bin/python3 /usr/bin/python
```

De esta forma, con el comando *python2* estaremos invocando a la versión 2 del intérprete, y *python* será el encargado de lanzar la versión 3. Como el lector habrá podido averiguar, es posible disponer de dos versiones diferentes del intérprete en la misma máquina.

HOLA MUNDO

La primera toma de contacto práctica con el lenguaje la realizaremos a través del famoso *Hola Mundo.* Comenzaremos lanzando la interfaz de comandos del intérprete de Python. Dependiendo del sistema operativo que estemos utilizando, accederemos a la mencionada interfaz de forma diferente. Por ejemplo, en Linux comenzaremos abriendo una *shell* y lanzando el comando *python.* En Mac OS X procederemos de la misma forma a través del programa *Terminal.* Los usuarios de Windows pueden acceder al menú *Inicio* y buscar el programa *IDLE.*

Nada más lanzar la interfaz de comandos del intérprete, también llamado *intérprete interactivo,* comprobaremos que aparece un mensaje inicial con el número de versión del intérprete y una serie de información adicional que hace referencia a la plataforma donde está siendo ejecutado. Justo en la línea siguiente aparece otro mensaje que nos indica de qué forma podemos acceder a la información sobre la licencia del intérprete. La última línea comienza por los caracteres >>> y nos muestra un cursor parpadeando. Este es el *prompt* del intérprete que nos permite interactuar directamente con él. Por ejemplo, si tecleamos *copyright y* pulsamos *enter,* veremos cómo se lanza información sobre el *copyright* de Python y después, vuelve a aparecer el *prompt,* invitándonos a lanzar otro comando o sentencia.

A lo largo de este libro, los ejemplos de código que comiencen por los mencionados caracteres >>> representarán sentencias que pueden ser lanzadas directamente en el intérprete. Si debajo de la misma apareciera otra más, sin los caracteres >>>, esta hará referencia al resultado obtenido como consecuencia de la ejecución en el intérprete de la línea de código correspondiente.

Como es tradicional, cuando se está aprendiendo un lenguaje de programación, nuestras primeras líneas de código imprimirán en pantalla el mensaje *Hola Mundo.* Para ello, desde el prompt del intérprete escribiremos el siguiente comando y pulsaremos enter:

```
>>> print ("Hola Mundo")
```

Veremos, entonces, cómo aparece el mencionado mensaje en la siguiente línea y después volverá a aparecer el prompt del intérprete. Obviamente, no hace

falta teclear los caracteres >>>, ya que estos aparecen por defecto y nos indican que el prompt se encuentra en espera y listo para que tecleemos y ejecutemos nuestro código.

A pesar de que la interfaz del intérprete es muy práctica y nos puede servir para realizar pruebas, habitualmente, nuestro código será ejecutado desde un fichero de texto. Siguiendo con nuestro ejemplo, crearemos un nuevo fichero con nuestro editor de textos favorito al que añadiremos la misma línea de código que hemos ejecutado desde el intérprete (sin añadir los caracteres >>>). Lo salvaremos con el nombre *hola.py.* Efectivamente, la extensión *.py* es la que se utiliza para los ficheros de código Python. Seguidamente, los usuarios de Mac OS X y Linux pueden invocar directamente al intérprete desde la shell o desde *Terminal:*

```
$ python hola.py
```

El resultado aparecerá directamente en la siguiente línea, cuando el comando finalice su ejecución. Los usuarios de Windows tendrán que hacer un poco de trabajo extra para ejecutar el mismo comando. Esto se debe a que, por defecto, el ejecutable del intérprete de Python no se añade a la variable de entorno *PATH,* como sí ocurre en los sistemas operativos basados en UNIX. Así pues, para modificar el valor de esta variable, en Windows, accederemos a *Panel de control > Sistema > Configuración avanzada del sistema* y pulsaremos el botón *Variables de entorno...* de la pestaña *Opciones Avanzadas.* Dentro de *Variables del sistema*, localizaremos la variable *Path* y haremos clic sobre el botón *Editar....* Aparecerá una nueva ventana que nos permite modificar el valor de la variable, al final de la línea añadiremos el directorio donde se encuentra el ejecutable del intérprete de Python. Por defecto, este directorio es *C:/Python32.* Una vez realizada esta configuración, bastará con lanzar el comando *cmd* para poder acceder a la shell del sistema e invocar directamente al comando, igual que en Mac OS X y en Linux. Asimismo, si deseamos utilizar directamente la interfaz de comandos en Windows, sin invocar a *IDLE,* podemos hacerlo desde la misma *cmd*, tecleando *python.*

Código fuente y *bytecode*

Hasta ahora solo hemos hablado de los ficheros de código Python, que utilizan la extensión *.py*. También sabemos que este lenguaje es interpretado y no compilado. Sin embargo, en realidad, internamente el intérprete Python se encarga de generar unos ficheros binarios que son los que serán ejecutados. Este proceso se realiza de forma transparente, a partir de los ficheros fuente. Al código generado automáticamente se le llama *bytecode* y utiliza la extensión *.pyc*. Así pues, al invocar al intérprete de Python, este se encarga de leer el fichero fuente, generar el bytecode correspondiente y ejecutarlo. ¿Por qué se realiza este proceso? Básicamente, por cuestiones de eficiencia. Una vez que el fichero .pyc esté generado, Python no vuelve a leer el fichero fuente, sino que lo ejecuta directamente, con el ahorro de tiempo que esto supone. La generación del bytecode es automática y el programador no debe preocuparse por este proceso. El intérprete es lo suficientemente inteligente para volver a generar el bytecode cuando es necesario, habitualmente, cuando el fichero de código correspondiente cambia.

Por otro lado, también es posible generar ficheros binarios listos para su ejecución, sin necesidad de contar con el intérprete. Recordemos que los ficheros Python requieren del intérprete para ser ejecutados. Sin embargo, en ocasiones necesitamos ejecutar nuestro código en máquinas que no disponen de este intérprete. Este caso suele darse en sistemas Windows, ya que, por defecto, tanto Mac OS X, como la mayoría de las distribuciones de GNU/Linux, incorporan dicho intérprete. Para salvar este obstáculo contamos con programas como *py2exe* (ver referencias), que se encarga de ejecutar un binario para Windows *(.exe)* a partir de un fichero fuente escrito en Python.

HERRAMIENTAS DE DESARROLLO

Uno de los factores importantes a tener en cuenta, a la hora de abordar el desarrollo de software, es el conjunto de herramientas con el que podemos contar para realizar el trabajo. Con independencia de la tecnología y el lenguaje, existen diferentes tipos de herramientas de desarrollo de software, desde un sencillo editor de texto, hasta complejos depuradores, pasando por entornos integrados de desarrollo que ofrecen bastantes funcionalidades en un solo programa. Python no es una excepción y cuenta con diferentes herramientas de desarrollo que nos ayudarán a ser más productivos.

Dado que entrar en profundidad, en cada una de las herramientas de desarrollo que podemos utilizar para trabajar con Python, escapa al ámbito de este libro, nos centraremos en mencionar y describir las más populares. El objetivo es que el lector tenga un punto de referencia sobre las mismas y no se encuentre perdido a la hora de elegir.

Por funcionalidad hemos realizado una agrupación en categorías. En concreto, se trata de editores, entornos integrados de desarrollo, depuradores, herramientas de *profiling* y entornos *virtuales.*

Editores

Podemos considerar a los editores de texto como las herramientas básicas para desarrollar software, ya que nos permiten escribir el código fuente y crear un fichero a partir del mismo.

Dentro de este grupo, existen multitud de programas, desde los básicos como *Bloc de Notas*, hasta aquellos más complejos como *Vim o TextMate.* Aunque cualquier editor de texto es válido para escribir código, es interesante que este cuente con ciertas funcionalidades que nos hagan el trabajo más fácil. Por ejemplo, el *resaltado* de sintaxis *(syntax highlighting),* la búsqueda utilizando expresiones regulares, la autoindentación, la personalización de *atajos* de teclado o la navegación de código, resultan muy prácticas a la vez que nos ayudan a mejorar la productividad.

En la actualidad existen multitud de editores de texto que incorporan otras muchas funcionalidades, además de las mencionadas anteriormente, que nos serán muy válidos para escribir código Python. Algunos son multiplataforma, mientras que otros solo existen para un sistema operativo concreto.

Vim y *Emacs* son los editores más populares en el mundo UNIX y de los cuales podemos encontrar versiones para Mac OS X, Linux y Windows. En realidad, muchos consideran a ambos mucho más que un editor de texto, ya que ambos se pueden personalizar ampliando sus funcionalidades hasta convertirlos en un moderno entorno integrado de desarrollo. En la red existen multitud de recursos (ver referencias) que podemos añadir a ambos editores para convertirlos en herramientas imprescindibles para desarrollar aplicaciones en Python. Aunque Vim y Emacs son muy potentes, deberemos tener en cuenta que ambos tienen una curva de aprendizaje elevada.

Muchos desarrolladores que trabajan en Mac OS X están habituados a *TextMate* (ver referencias). Se trata de un potente editor que también cuenta con útiles herramientas para Python. Este editor no es *open source* y deberemos adquirir una licencia para su uso.

Distribuciones de Linux, como Ubuntu y Fedora, instalan por defecto un sencillo y práctico editor que también podemos utilizar para Python. Su nombre es *gedit* y su funcionalidad puede ser ampliada a través de *plugins.*

Otro editor de código digno de mención es *Notepad++*. Se distribuye bajo la licencia GPL, aunque solo existe una versión para sistemas Windows.

Entornos integrados de desarrollo (IDE)

La evolución natural de los editores de código son los entornos integrados de desarrollo. Estos amplían la funcionalidad de los editores añadiendo facilidades para la depuración de código, la creación de proyectos, el auto completado, la búsqueda de referencias en la documentación o el marcado de sintaxis errónea. Dos de los más populares son *Eclipse* y *NetBeans.* Aunque se hicieron populares para el desarrollo Java, actualmente, ambos soportan Python como lenguaje y ofrecen funcionalidades específicas para él mismo. Entre las ventajas de estos dos IDE caben destacar su carácter *open source,* la gran comunidad de usuarios con la que cuentan y que existen versiones para distintas plataformas. Por otro

lado, la dependencia del *runtime* de Java y el consumo de recursos hardware son algunas de sus desventajas.

Aunque menos conocido, *Komodo* es otra de las opciones. Desarrollado por la empresa *ActiveState*, es multiplataforma, no consume demasiados recursos y ofrece bastantes prácticas funcionalidades. A diferencia de Eclipse y NetBeans, no es *open source* y requiere del pago de una licencia para su uso. No obstante, existe una versión más limitada en funcionalidades, llamada *Komodo Edit* y que sí es gratuita y *open source.*

En lo que respecta a algunos IDE específicos para Python, son tres los más populares. El primero de ellos es fríe, que está escrito en Python utilizando el *toolkit* gráfico *Qt.* La última versión de este IDE es la 5 y requiere de Python 3 para su ejecución. Por otro lado tenemos a *PyCharm,* desarrollado por la empresa *JetBrains* y caracterizado por tener un amplio soporte para el desarrollo para *Django,* el popular framework web de Python. *Wingware* es el tercero de este grupo y entre sus características cabe destacar el soporte para populares *toolkits* y frameworks para Python, como son, *Zope, PyQt, PyGTK,* Django y *wxPython.*

Intérprete interactivo mejorado

A pesar de que el intérprete interactivo estándar de Python es muy práctico para ejecutar código escrito en este lenguaje sin necesidad de crear fichero, tiene algunas carencias. Por ejemplo, no es posible usar el tabulador para autocompletar código, no numera las líneas de código que se van escribiendo, no contiene una ayuda interactiva y no permite la introspección dinámica de objetos. Con el objetivo de disponer de una herramienta, similar al intérprete interactivo estándar, pero que pudiera suplir las carencias de este, se desarrolló *IPython.* Esta herramienta puede ser utilizada como sustituta del mencionado intérprete, el cual está incluido en la instalación estándar de Python.

La instalación de *IPython* puede realizarse como si de un módulo de Python más se tratara, siendo, pues, posible su utilización en diferentes sistemas operativos. Recomendamos leer el capítulo 9 *(Instalación y distribución de módulos)* para realizar la instalación a través del gestor de paquetes *pip.*

IPython puede facilitarnos en gran medida el trabajo de desarrollo y es

recomendable su utilización como intérprete interactivo, sobre todo para aquellos programadores avanzados de Python.

Para más información sobre las características, método de instalación y documentación en general sobre *IPython*, podemos visitar la página web (ver referencias) que existe a tal efecto.

Depuradores

La acción de *depurar* código es de gran ayuda a la hora de resolver *bugs.* Dentro del proceso de desarrollo de software es una de las tareas más habituales llevadas a cabo por los programadores.

¿En qué consiste la depuración? Básicamente se trata de seguir paso a paso la ejecución de un programa o una parte del mismo. Contar con una herramienta automática que nos ayude a ello, resulta imprescindible. Al igual que para otros lenguajes, para Python contamos con la herramienta llamada *pdb* que soporta la fijación de *breakpoints,* el avance paso a paso, la evaluación de expresiones y variables y el listado del código actual en ejecución. Esta utilidad puede ser invocada directamente desde la interfaz del intérprete de Python o a través del ejecutable *python.*

El funcionamiento básico de pdb es sencillo. Podemos comenzar por fijar un *breakpoint* en un punto determinado de nuestro código fuente. Esto se realiza a través de dos sencillas líneas de código:

```
import pdb
pdb.set_trace()
```

Al lanzar pdb y llegar al punto donde hemos puesto el breakpoint, entrará en marcha el depurador, parando la ejecución del programa y esperando, a través del prompt, para que introduzcamos un comando que nos permita, por ejemplo, evaluar una variable o continuar la ejecución del programa paso a paso. El lanzamiento de pdb para nuestro script de ejemplo se haría de la siguiente forma:

```
$ python -m pdb hola.py
```

Para una referencia completa sobre los comandos que pueden lanzarse desde el prompt ofrecido por pdb, recomendamos visitar la página web oficial (ver

referencias) de este depurador.

Profiling

En ingeniería de software, un *profiler* es un programa que mide el rendimiento de la ejecución de otro programa, ofreciendo una serie de estadísticas sobre dicho rendimiento. Este tipo de herramientas es muy útil para mejorar un determinado programa, debido a que la información que nos proporciona es difícil obtenerla de otra manera.

Además, en ocasiones se da la circunstancia de que durante el desarrollo es muy complicado predecir que partes de una aplicación contribuirán a bajar su rendimiento. Para averiguar cuáles son las secciones o componentes de código, tendremos que esperar al tiempo de ejecución y es aquí donde los *profilers* realizan su trabajo.

Dentro de la librería estándar de Python contamos con tres *profilers* diferentes: *cProfile*, *profile* y *hotshot.* El primero de ellos fue introducido en la versión 2.5 y es el más recomendado, tanto por su facilidad de uso, como por la información que nos ofrece. Por otro lado, *profile* está escrito en Python, es más lento que *cProfile* y además su funcionalidad está limitada a este. El uso de *hotshot* no es aconsejable para principiantes, dado que es experimental, además hemos de tener en cuenta que será eliminado en futuras versiones del intérprete.

El uso básico de *cProfile* es bastante sencillo, bastará con invocar al intérprete de Python pasando un parámetro específico, seguido del programa que deseamos comprobar. Por ejemplo, hagámoslo con nuestro primer programa:

```
$ python -m cProfile hola.py
```

Como salida de la ejecución del comando anterior, obtendremos lo siguiente:

```
Hola Mundo
8 function calls in 0.000 seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:
                                           lineno(function)
2     0.000  0.000  0.000  0.000  cp850.py:18(encode)
1     0.000  0.000  0.000  0.000  hola.py:1(<module>)
2     0.000  0.000  0.000  0.000  {built-in
                                           method charmap_encode}
```

```
1     0.000  0.000  0.000  0.000  {built-in
                                   method exec}
1     0.000  0.000  0.000  0.000  {built-in
                                   method print}
1     0.000  0.000  0.000  0.000  {method 'disable' of
                                   '_lsprof.Profiler' objects}
```

Dado que nuestro programa ejemplo es muy sencillo, no obtendremos valiosa información, pero sí que nos servirá para descubrir cómo funcionan este tipo de herramientas.

Otra herramienta que podemos utilizar para hacer *profiling* es el módulo *timeit*, el cual nos permite medir el tiempo que tarde en ejecutarse una serie de líneas de código. Esta herramienta forma parte de la librería estándar de Python, lo que significa que no tenemos que realizar ninguna instalación adicional.

NOVEDADES EN PYTHON 3

La última versión de Python trae consigo una serie de claras novedades y diferencias con respecto a la serie 2.x. Aquellos programadores de Python 2, que deseen migrar sus aplicaciones para que funcionen en la versión 3, deberán tener en cuenta estas diferencias. A continuación, resumiremos las más significativas, los lectores no familiarizados con Python pueden pasar por alto este apartado y saltar hacia el siguiente capítulo.

En lo que respecta a los *strings*, el cambio más significativo es que en la versión 3 todos son *Unicode*. Como consecuencia de ello, se ha suprimido la función *unicode()*. Además, el operador *%*, utilizado para la concatenación de *strings*, ha sido reemplazado por la nueva función *format()*. Así pues, por ejemplo, la siguiente sentencia en Python 2:

```
>>> cad = "%s %s" % (cadl, cad2)
```

Pasa a ser de esta forma en Python 3:

```
>>> cad = "{o} {1}".format(cadl, cad2)
```

Otra nueva función introducida en Python 3 es *print()*, siendo ahora necesario utilizar paréntesis cuando la invocamos. Igualmente ocurre con la función *exec()*, utilizada para ejecutar código a través de un objeto. Relacionada con esta funcionalidad, en Python 3 ha sido eliminada *execfile()*. Para simular su funcionamiento, deberemos leer un fichero línea a línea y ejecutar *exec()* para cada una de ellas.

En Python 2.x la operación aritmética para realizar la división exacta debe hacerse entre dos números reales, utilizando para ello el operador /. Sin embargo, en la nueva versión de Python esta operación puede hacerse directamente con números enteros. Para la división entera utilizaremos el operador //. Veamos unos ejemplos al respecto. La siguiente sentencia nos devolverá el número real 3.5 en Python 2.x:

```
>>> 7.0 / 2.0
```

La misma operación puede realizarse en Python 3:

```
>>> 7 / 2
```

Por otro lado, para la división entera, en Python 3, ejecutaríamos el siguiente comando, siendo el resultado 3:

```
>>> 7 // 2
```

La representación de números en octal (base 8) ha sido también cambiada en la nueva versión de Python. Ahora se debe poner la letra o justo detrás del 0 y antes del número que va a ser representado. Es decir, la siguiente expresión ha dejado de ser válida en Python 3:

```
>>> x = 077
```

En su lugar debe emplear esta otra:

```
>>> x = 0o77
```

Si implementamos clases *iterator,* deberemos escribir un método *__next __(),* esto implica que no podremos utilizar el método *next()* de nuestra clase. Así pues, en Python 3, invocaremos directamente al mencionado método pasando como argumento la clase *iterator.*

Con respecto a los diccionarios, la forma de iterar entre sus claves y valores ha cambiado. Ahora las funciones *iterkeys(), iteritems()* y *itervalues()* no son necesarias, en su lugar emplearemos las funciones *keys(), ítems()* y *values(),* respectivamente. Para comprobar si una clave se encuentra en un diccionario, en lugar de invocar a la función *has_key(),* bastará con preguntar directamente a través del operador *if:*

```
>>> if mykey in mydict: print("Clave en diccionario")
```

Si trabajamos con comprensión de listas y dentro de ellas usamos tuplas, estas deberán, en Python 3, ir entre paréntesis. Además, la función *sorted()* devuelve directamente una lista, sin necesidad de convertir su argumento a este tipo de dato. Para emplear esta función de ordenación deberemos tener en cuenta que la lista o tupla debe contener elementos del mismo tipo. En Python 3 la función *sorted()* y el método *sort()* devolverán una excepción si los elementos que van a ser ordenados son de diferentes tipos.

La librería estándar ha reemplazado los nombres de algunos módulos, lo que significa que debemos tenerlo en cuenta a la hora de utilizar la sentencia *import.*

Por ejemplo, el módulo *Cookie* ha sido renombrado a *http.Cookies*. Otro ejemplo es *httplib* que ahora se encuentra dentro de *http* y se llama *client (import http.Client)*.

Las excepciones que capturan un objeto, en Python 3, requieren de la palabra clave *as*. De esta forma, escribiremos:

```
try:
    myfun()
except ValueError as myerror:
    print(err)
```

En relación también con las excepciones, para invocar a *raise* con argumentos necesitaremos paréntesis en la llamada. Además, los *strings* no pueden ser usados como excepciones. Si necesitamos de esta funcionalidad, podemos escribir:

```
raise Exception("Ha ocurrido un error")
```

La nueva versión del lenguaje no solo nos permite *desempaquetar* diccionarios, también podemos hacerlo con conjuntos. Por ejemplo, la siguiente sentencia nos devuelve el valor *1* para la variable *a* y una lista con los valores 2 y 3 para la variable *b:*

```
a, *b = (1, 2, 3)
```

Para migrar nuestro código de una versión a otra, existe una herramienta llamada *2to3* (ver referencias). Gracias a ella, automáticamente podemos obtener una versión de nuestro código compatible con Python 3. Si bien es cierto, que esta herramienta no es perfecta, es recomendable repasar el código generado automáticamente para asegurarnos que el proceso se ha realizado correctamente. *2to3.py* es un script escrito en Python y que se distribuye junto al intérprete del lenguaje. Por ejemplo, en Windows podemos localizarlo en el subdirectorio *Tools\Scripts*, que se encuentra dentro del directorio donde, por defecto, fue instalado el intérprete.

Hasta aquí las novedades y diferencias más interesantes entre versiones de este lenguaje. Si estamos interesados en obtener una completa referencia de todas las novedades de Python 3, podemos echar un vistazo a la página oficial dedicada a este efecto (ver referencias).

ESTRUCTURAS Y TIPOS DE DATOS BÁSICOS

INTRODUCCIÓN

Realizada una primera toma de contacto con Python en el capítulo anterior, dedicaremos el presente a descubrir cuáles son las estructuras de datos y tipos básicos con los que cuenta este lenguaje.

Comenzaremos describiendo y explicando una serie de conceptos básicos propios de este lenguaje y que serán empleados a lo largo del libro. Posteriormente, pasaremos a centrarnos en los tipos básicos, como son, los números y las cadenas de texto. Después, llegará el turno de las estructuras de datos como las tuplas, las listas, los conjuntos y los diccionarios.

CONCEPTOS BÁSICOS

Uno de los conceptos básicos y principales en Python es el *objeto.* Básicamente, podemos definirlo como un componente que se aloja en memoria y que tiene asociados una serie de valores y operaciones que pueden ser realizadas con él. En realidad, los datos que manejamos en el lenguaje cobran vida gracias a estos *objetos.* De momento, estamos hablando desde un punto de vista bastante general; es decir, no debemos asociar este *objeto* al concepto del mismo nombre que se emplea en programación orientada a objetos (OOP). De hecho, un *objeto* en Python puede ser una cadena de texto, un número real, un diccionario o un objeto propiamente dicho, según el paradigma OOP, creado a partir de una clase determinada. En otros lenguajes de programación se emplea el término *estructura de datos* para referirse al *objeto.* Podemos considerar ambos términos como equivalentes.

Habitualmente, un programa en Python puede contener varios componentes. El lenguaje nos ofrece cinco tipos de estos componentes claramente diferenciados. El primero de ellos es el *objeto,* tal y como lo hemos definido previamente. Por otro lado tenemos las *expresiones,* entendidas como una combinación de valores, constantes, variables, operadores y funciones que son aplicadas siguiendo una serle de reglas. Estas expresiones se suelen agrupar formando *sentencias,* consideradas estas como las unidades mínimas ejecutables de un programa. Por último, tenemos los *módulos* que nos ayudan a formar grupos de diferentes sentencias.

Para facilitarnos la programación, Python cuenta con una serie de *objetos* integrados *(built-in).* Entre las ventajas que nos ofrecen, caben destacar, el ahorro de tiempo, al no ser necesario construir estas estructuras de datos de forma manual, la facilidad para crear complejas estructuras basadas en ellos y el alto rendimiento y mínimo consumo de memoria en tiempo de ejecución. En concreto, contamos con números, cadenas de texto, *booleanos,* listas, diccionarios, tuplas, conjuntos y ficheros. Además, contamos con un tipo de objeto especial llamado *None* que se emplea para asignar un valor nulo. A lo largo de este capítulo describiremos cada uno de ellos, a excepción de los ficheros, de los que se ocupa el capítulo 7.

Antes de comenzar a descubrir los objetos *built-in* de Python, es conveniente explicar qué es y cómo funciona el *tipado dinámico,* del que nos ocuparemos en el siguiente apartado.

Tipado dinámico

Los programadores de lenguajes como Java y C++ están acostumbrados a definir cada variable de un *tipo* determinado. Igualmente ocurre con los objetos, que deben estar asociados a una clase determinada cuando son creados. Sin embargo, Python no trabaja de la misma forma, ya que, al declarar una variable, no se puede indicar su tipo. En tiempo de ejecución, el tipo será asignado a la variable, empleando una técnica conocida como *tipado dinámico.*

¿Cómo es esto posible, cómo diferencia el intérprete entre diferentes tipos y estructuras de datos? La respuesta a estas preguntas hay que buscarla en el funcionamiento interno que el intérprete realiza de la memoria. Cuando se asigna a una variable un valor, el intérprete, en tiempo de ejecución, realiza un proceso que consiste en varios pasos. En primer lugar se crea un objeto en memoria que representará el valor asignado. Seguidamente se comprueba si existe la variable, si no es así se crea una referencia que enlaza la nueva variable con el objeto. Si por el contrario ya existe la variable, entonces, se cambia la referencia hacia el objeto creado. Tanto las variables, como los objetos, se almacenan en diferentes zonas de memoria.

A bajo nivel, las variables se guardan en una tabla de sistema donde se indica a qué objeto referencia cada una de ellas. Los objetos son trozos de memoria con el suficiente espacio para albergar el valor que representan. Por último, las referencias son punteros que enlazan objetos con variables. De esta forma, una variable referencia a un objeto en un determinado momento de tiempo. ¿Cuál es la consecuencia directa de este hecho? Es sencillo, en Python los tipos están asociados a objetos y no a variables. Lógicamente, los objetos conocen de qué tipo son, pero no las variables. Esta es la forma con la que Python consigue implementar el tipado dinámico.

Internamente, el intérprete de Python utiliza un contador de las referencias que se van asignando entre objetos y variables. En función de un algoritmo determinado, cuando estás van cambiando y ya no son necesarias, el recolector

de basura se encargará de marcar como disponible el espacio de memoria ocupado por un objeto que ha dejado de ser referenciado.

Gracias al tipado dinámico podemos, en el mismo bloque de código, asignar diferentes tipos de datos a la misma variable, siendo el intérprete en tiempo de ejecución el que se encargará de crear los objetos y referencias que sean necesarios.

Para clarificar el proceso previamente explicado, nos ayudaremos de un ejemplo práctico. En primer lugar asignaremos el valor numérico *8* a la variable *x* y seguidamente asignaremos a una nueva variable *y* el valor de *x:*

```
>>> x = 8
>>> y = x
```

Después de la ejecución de las sentencias anteriores, en memoria tendríamos una situación como la que muestra la figura 2-1.

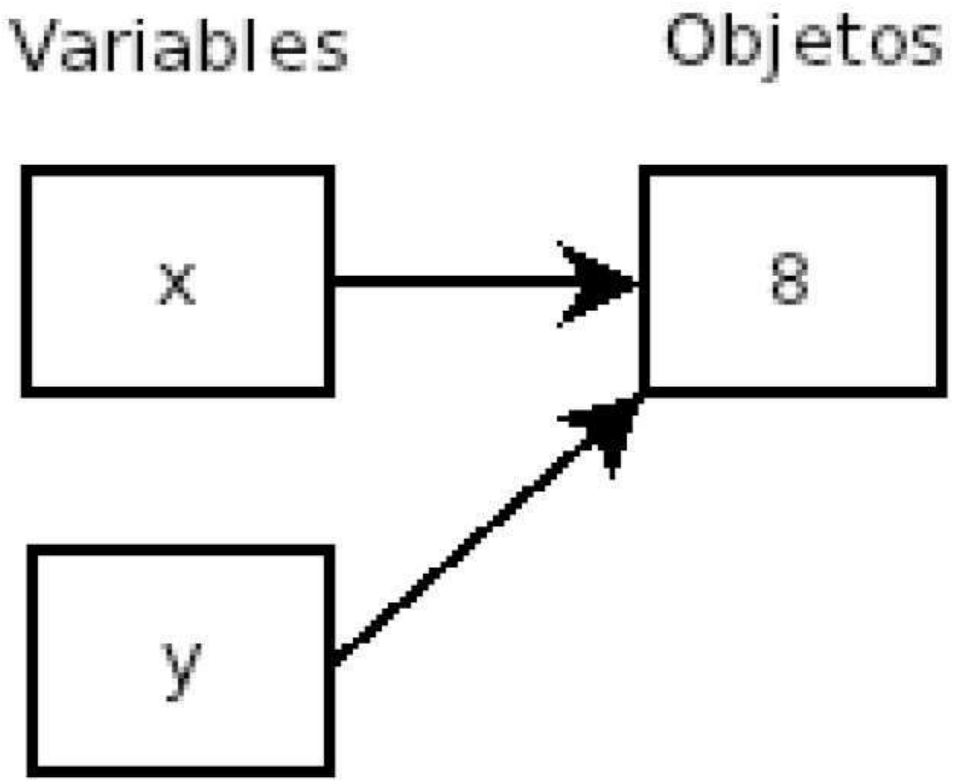


Fig. 2-1 Variables y valor asignado

Posteriormente ejecutamos una nueva sentencia como la siguiente:

```
>>> x = "test"
```

Los cambios efectuados en memoria pueden apreciarse en la figura 2-2, donde comprobaremos cómo ahora las variables tienen distinto valor puesto que apuntan a diferentes objetos.

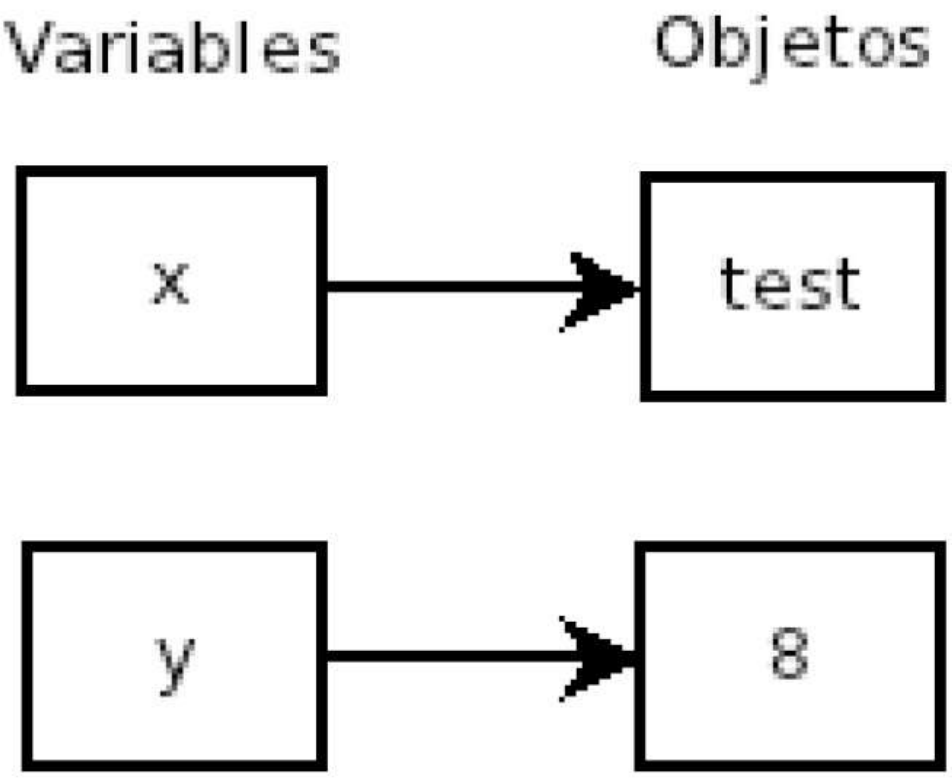


Fig. 2-1 Cambio de valores

Sin embargo, para algunos tipos de objetos, Python realiza la asignación entre objetos y variables de forma diferente a la que hemos explicado previamente. Un ejemplo de este caso es cuando se cambia el valor de un elemento dentro de una lista. Supongamos que definimos una *lista* (un simple *array o* vector) con una serie de valores predeterminados y después, asignamos esta nueva variable a otra diferente llamada *lista_2:*

```
>>> lista_1 = [9, 8, 7]
>>> lista_2 = lista_1
```

Ahora modificamos el segundo elemento de la primera lista, ejecutando la siguiente sentencia:

```
>>> lista_1[2] = 5
```

Como resultado, ambas listas habrán sido modificadas y su valor será el mismo. ¿Por qué se da esta situación? Simplemente, porque no hemos cambiado el objeto, sino un componente del mismo. De esta forma, Python realiza el cambio sobre la marcha, sin necesidad de crear un nuevo objeto y asignar las correspondientes referencias entre variables. Como consecuencia de ello, se ahorra tiempo de procesamiento y memoria cuando el programa es ejecutado por el intérprete.

Una función que nos puede resultar muy útil para ver de qué tipo es una variable es *type()*. Como argumento recibe el nombre de la variable en cuestión y devuelve el tipo precedido de la palabra clave *class*. Gracias a esta función y debido a que una variable puede tomar distintos tipos durante su ejecución, podremos saber a qué tipo pertenece en cada momento de la ejecución del código. Veamos un sencillo ejemplo, a través de las siguientes sentencias y el resultado devuelto por el intérprete:

```
>>> z = 35
>>> type(z)
<class 'int'>
>>> z = "ahora es una cadena de texto"
<class 'str'>
```

NÚMEROS

Como en cualquier lenguaje de programación, en Python, la representación y el manejo de números se hacen prácticamente imprescindibles. Para trabajar con ellos, Python cuenta con una serie de tipos y operaciones integradas, de ambos nos ocuparemos en el presente apartado.

Enteros, reales y complejos

Respecto a los tipos de números soportados por Python, contamos con que es posible trabajar con números enteros, reales y complejos. Además, estos pueden ser representados en decimal, binario, octal y hexadecimal, tal y como veremos más adelante.

De forma práctica, la asignación de un número entero a una variable se puede hacer a través de una sentencia como esta:

```
>>> num_entero = 8
```

Por supuesto, en el contexto de los números enteros, la siguiente expresión también es válida:

```
>>> num_negativo = -78
```

Por otro lado, un número real se asignaría de la siguiente forma:

```
>>> num_real = 4.5
```

En lo que respecta a los números complejos, aquellos formados por una parte *real* y otra *imaginaria*, la asignación sería la siguiente:

```
>>> num_complejo = 3.2 + 7j
```

Siendo también válida la siguiente expresión:

```
>>> num_complex = 5J + 3
```

Como el lector habrá deducido, en los números complejos, la parte imaginaria aparece representada por la letra *j,* siendo también posible emplear la misma letra en mayúscula.

Python 2.x distingue entre dos tipos de enteros en función del tamaño del valor que representan. Concretamente, tenemos los tipos *int* y *long.* En Python 3 esta situación ha cambiado y ambos han sido integrados en un único tipo *int.*

Los valores para números reales que podemos utilizar en Python 3 tienen un amplio rango, gracias a que el lenguaje emplea para su representación un bit para el signo (positivo o negativo), 11 para el exponente y 52 para la mantisa. Esto también implica que se utiliza la precisión doble. Recordemos que en algunos lenguajes de programación se emplean dos tipos de datos para los reales, que varían en función de la representación de su precisión. Es el caso de C, que cuenta con el tipo *float* y *double.* En Python no existe esta distinción y podemos considerar que los números reales representados equivalen al tipo *double* de C.

Además de expresar un número real tal y como hemos visto previamente, también es posible hacerlo utilizando notación científica. Simplemente necesitamos añadir la letra e, que representa el exponente, seguida del valor para él mismo. Teniendo esto en cuenta, la siguiente expresión sería válida para representar al número $0.5*10^{-7}$:

```
>>> num_real = 0.5e-7
```

Desde un punto de vista escrito los *booleanos* no son propiamente números; sin embargo, estos solo pueden tomar dos valores diferentes: *True* (verdadero) o *False* (falso). Dada esta circunstancia, parece lógico utilizar un tipo entero que necesite menos espacio en memoria que el original, ya que, solo necesitamos dos números: 0 y 1. En realidad, aunque Python cuenta con el tipo integrado *bool,* este no es más que una versión personalizada del tipo *int.*

Si necesitamos trabajar con números reales que tengan una precisión determinada, por ejemplo, dos cifras decimales, podemos utilizar la clase *Decimal.* Esta viene integrada en la librería básica que ofrece el intérprete e incluye una serie de funciones, para, por ejemplo, crear un número real con precisión a través de un variable de tipo *float*

Sistemas de representación

Tal y como hemos adelantado previamente, Python puede representar los números enteros en los sistemas decimal, octal, binarlo y hexadecimal. La representación decimal es la empleada comúnmente y no es necesario indicar nada más que el número en cuestión. Sin embargo, para el resto de sistemas es necesario que el número sea precedido de uno o dos caracteres concretos. Por ejemplo, para representar un número en binario nos basta con anteponer los caracteres *Ob*. De esta forma, el número 7 se representaría en binario de la siguiente forma:

```
>>> num_binario = 0b111
```

Por otro lado, para el sistema octal, necesitamos los caracteres Oo, seguidos del número en cuestión. Así pues, el número entero 8 quedaría representado utilizando la siguiente sentencia:

```
>>> num_octal = 0o10
```

En lo que respecta al sistema hexadeclmal, el carácter necesario, tras el número 0, es la letra x. Un ejemplo sería la representación del número 255 a través de la siguiente sentencia:

```
>>> num_hex = 0xff
```

Operadores

Obviamente, para trabajar con números, no solo necesitamos representarlos a través de diferentes tipos, sino también es importante realizar operaciones con ellos. Python cuenta con diversos operadores para aplicar diferentes operaciones numéricas. Dentro del grupo de las aritméticas, contamos con las básicas suma, división entera y real, multiplicación y resta. En cuanto a las operaciones de bajo nivel y entre bits, existen tanto las operaciones NOT y NOR, como XOR y AND. También contamos con operadores para comprobar la igualdad y desigualdad y para realizar operaciones lógicas como AND y OR.

Como en otros lenguajes de programación, en Python también existe la precedencia de operadores, lo que deberemos tener en cuenta a la hora de escribir expresiones que utilicen varios de ellos. Sin olvidar que los paréntesis pueden ser usados para marcar la preferencia entre unas operaciones y otras

dentro de la misma expresión.

La tabla 2-1 resume los principales operadores y operaciones numéricas a las que hacen referencia, siendo o y *b* dos variables numéricas.

Expresión con operador	Operación
a + b	Suma
a - b	Resta
a * b	Multiplicación
a % b	Resto
a / b	División real
a // b	División entera
a ** b	Potencia
a | b	OR (bit)
a ^ b	XOR (bit)
a & b	AND (bit)
a == b	Igualdad
a != b	Desigualdad
a or b	OR (lógica)
a and b	AND (lógica)
not a	Negación (lógica)

Tabla 2-1. Principales operaciones y operadores numéricos

Funciones matemáticas

A parte de las operaciones numéricas básicas, anteriormente mencionadas, Python nos permite aplicar otras muchas funciones matemáticas. Entre ellas, tenemos algunas como el valor absoluto, la raíz cuadrada, el cálculo del valor máximo y mínimo de una lista o el redondo para números reales. Incluso es posible trabajar con operaciones trigonométricas como el seno, coseno y tangente. La mayoría de estas operaciones se encuentran disponibles a través de un módulo (ver definición en capítulo 3) llamado *math*. Por ejemplo, el valor absoluto del número -47,67 puede ser calculado de la siguiente forma:

```
>>> abs(-47,67)
```

Para algunas operaciones necesitaremos importar el mencionado módulo *math*, sirva como ejemplo el siguiente código para calcular la raíz cuadrada del número 169:

```
>>> import math
>>> math.sqrt(169)
```

Otras interesantes operaciones que podemos hacer con números es el cambio de base. Por ejemplo, para pasar de decimal a binario o de octal a hexadecimal. Para ello, Python cuenta con las funciones *int(), hex(), oct()* y *bin()*. La siguiente sentencia muestra cómo obtener en hexadecimal el valor del entero 16:

```
>>> hex(16)
'0x10'
```

Si lo que necesitamos es el valor octal, por ejemplo, del número 8, bastará con lanzar la siguiente sentencia:

```
>>> oct(8)
'0o10'
```

Debemos tener en cuenta que las funciones de cambio de base admiten como argumentos cualquier representación numérica admitida por Python. Esto quiere decir, que la siguiente expresión también sería válida:

```
>>> bin(0xfe)
'0b11111110'
```

Conjuntos

Definir y operar con conjuntos matemáticos también es posible en Python. La función para crear un conjunto se llama *set()* y acepta como argumentos una serie de valores pasados entre comas, como si se tratara de una cadena de texto. Por ejemplo, la siguiente línea de código define un conjunto tres números diferentes:

```
>>> conjunto = set ('846')
```

Un conjunto también puede ser definido empleando llaves *({})* y separando los elementos por comas. Así pues, la siguiente definición es análoga a la sentencia anterior:

```
>>> conjunto = {8, 4, 6}
```

Operaciones como unión, intersección, creación de subconjuntos y diferencia están disponibles para conjuntos en Python. Algunas operaciones se pueden hacer directamente a través de operadores o bien, llamando al método en cuestión de cada instancia creada. Un ejemplo de ello es la operación intersección. Creemos un nuevo conjunto, utilicemos el operador *&* y observemos el resultado:

```
>>> conjunto_2 = set('785')
>>> conjunto & conjunto_2
('8')
```

Si en su lugar ejecutamos la siguiente sentencia, veremos que el resultado es el mismo:

```
>>> conjunto.intersection(conjunto_2)
```

A través de los métodos *add()* y *remove()* podemos añadir y borrar elementos de un conjunto.

Si creamos un conjunto con valores repetidos, estos serán automáticamente eliminados, es decir, no formarán parte del conjunto:

```
>>> duplicados = {2, 3, 6, 7, 6, 8, 2, 1}
>>> duplicados

{1, 3, 2, 7, 6, 8}
```

CADENAS DE TEXTO

No cabe duda de que, a parte de los números, las cadenas de texto (*strings*) son otro de los tipos de datos más utilizados en programación. El intérprete de Python integra este tipo de datos, además de una extensa serie de funciones para interactuar con diferentes cadenas de texto.

En algunos lenguajes de programación, como en C, las cadenas de texto no son un tipo integrado como tal en el lenguaje. Esto implica un poco de trabajo extra a la hora de realizar operaciones como la concatenación. Sin embargo, esto no ocurre en Python, lo que hace mucho más sencillo definir y operar con este tipo de dato.

Básicamente, una cadena de texto o *string* es un conjunto inmutable y ordenado de caracteres. Para su representación y definición se pueden utilizar tanto comillas dobles ("), como simples ('). Por ejemplo, en Python, la siguiente sentencia crearía una nueva variable de tipo *string:*

```
>>> cadena = "esto es una cadena de texto"
```

Si necesitamos declarar un string que contenga más de una línea, podemos hacerlo utilizando comillas triples en lugar de dobles o simples:

```
>>> cad_multiple = """Esta cadena de texto
... tiene más de una línea. En concreto, cuenta

... con tres líneas diferentes"""
```

Tipos

Por defecto, en Python 3, todas las cadenas de texto son *Unicode.* Si hemos trabajado con versiones anteriores del lenguaje deberemos tener en mente este hecho, ya que, por defecto, antes se empleaba ASCII. Así pues, cualquier *string* declarado en Python será automáticamente de tipo *Unicode*.

Otra de las novedades de Python 3 con referencia a las cadenas de texto es el tipo de estas que soporta. En concreto son tres las incluidas en esta versión:

Unicode, *byte* y *bytearray*. El tipo *byte* solo admite caracteres en codificación ASCII y, al igual que los de tipo Unicode, son inmutables. Por otro lado, el tipo *bytearray* es una versión mutable del tipo *byte*.

Para declarar un *string* de tipo *byte*, basta con anteponer la letra *b* antes de las comillas:

```
>>> cad = b"cadena de tipo byte"
>>> type(cad)
<class 'bytes'>
```

La declaración de un tipo *bytearray* debe hacerse utilizando la función integrada que nos ofrece el intérprete. Además, es imprescindible indicar el tipo de codificación que deseamos emplear. El siguiente ejemplo utiliza la codificación de caracteres *latin1* para crear un string de este tipo:

```
>>> lat = bytearray("España", 'latin1')
```

Observemos el siguiente ejemplo y veamos la diferencia al emplear diferentes tipos de codificaciones para el mismo string:

```
>>> print(lat)
bytearray(b'Esp\xfla')
>>> bytearray("España", "utf16")
bytearray(b'\xff\xfeE\xOOs\xOOp\xOOa\xOO\xf1\xOOa\xOO')
```

Para las cadenas de texto declaradas por defecto, internamente, Python emplea el tipo denominado *str*. Podemos comprobarlo sencillamente declarando una cadena de texto y preguntando a la función *type():*

```
>>> cadena = "comprobando el tipo str"
>>> type(cadena)
<class 'str'>
```

Realizar conversión entre los distintos tipos de strings es posible gracias a dos tipos de funciones llamadas *encode()* y *decode()*. La primera de ellas se utiliza para transformar un tipo *str* en un tipo *byte*. Veamos cómo hacerlo en el siguiente ejemplo:

```
>>> cad = "es de tipo str"
>>> cad. encode()
b'es de tipo str'
```

La función *decode()* realiza el paso inverso, es decir, convierte un string *byte*

a otro de tipo *str*. Como ejemplo, ejecutaremos las siguientes sentencias:

```
>>> cad = b"es de tipo byte"
>>> cad.decode()
'es de tipo byte'
```

Como el lector habrá podido deducir, cada una de estas funciones solo se encuentra definida para cada tipo. Esto significa que *decode()* no existe para el tipo *str* y que *encode()* no funciona para el tipo *byte*.

Alternativamente, la función *encode()* admite como parámetro un tipo de codificación específico. Si este tipo es indicado, el intérprete utilizará el número de bytes necesarios para su representación en memoria, en función de cada codificación. Recordemos que, para su representación interna, cada tipo de codificación de caracteres requiere de un determinado número de bytes.

Principales funciones y métodos

Para trabajar con strings Python pone a nuestra disposición una serie de funciones y métodos. Las primeras pueden ser invocadas directamente y reciben como argumento una cadena. Por otro lado, una vez que tenemos declarado el string, podemos invocar a diferentes métodos con los que cuenta este tipo de dato.

Una de las funciones más comunes que podemos utilizar sobre strings es el cálculo del número de caracteres que contiene. El siguiente ejemplo nos muestra cómo hacerlo:

```
>>> cad = "Cadena de texto de ejemplo"
>>> len(cad)
26
```

Otro ejemplo de función que puede ser invocada, sin necesidad de declarar una variable de tipo string, es *print()*. En el capítulo anterior mostramos cómo emplearla para imprimir una cadena de texto por la salida estándar.

Respecto a los métodos con los que cuentan los objetos de tipo string, Python incorpora varios de ellos para poder llevar a cabo funcionalidades básicas relacionadas con cadenas de texto. Entre ellas, contamos con métodos para buscar una subcadena dentro de otra, para reemplazar subcadenas, para borrar

espacios en blanco, para pasar de mayúsculas a minúsculas, y viceversa.

La función *find()* devuelve el índice correspondiente al primer carácter de la cadena original que coincide con el buscado:

```
>>> cad ="xyza"
>>> cad.find("y")
1
```

Si el carácter buscado no existe en la cadena, *find()* devolverá -1.

Para reemplazar una serie de caracteres por otros, contamos con el método *replace()*. En el siguiente ejemplo, sustituiremos la subcadena "Hola" por "Adiós":

```
>>> cad = "Hola Mundo"
>>> cad.replace("Hola", "Adiós")
'Adiós Mundo'
```

Obsérvese que *replace()* no altera el valor de la variable sobre el que se ejecuta. Así pues, en nuestro ejemplo, el valor de la variable *cad* seguirá siendo *"Hola Mundo"*.

Los métodos *strip()*, *lstrip()* y *rstrip()* nos ayudarán a eliminar todos los espacios en blanco, solo los que aparecen a la izquierda y solo los que se encuentran a la derecha, respectivamente:

```
>>> cad = " cadena con espacios en blanco "
>>> cad.strip()
"cadena con espacios en blanco"
>>> cad.lstrip()
"cadena con espacios en blanco "
>>> cad.rstrip()
" cadena con espacios en blanco"
```

El método *upper()* convierte todos los caracteres de una cadena de texto a mayúsculas, mientras que *lower()* lo hace a minúsculas. Veamos un sencillo ejemplo:

```
>>> cad2 = cad.upper()
>>> print(cad2)
"CADENA CON ESPACIOS EN BLANCO"
>>> print(cad3.lower())
" cadena con espacios en blanco "
```

Relacionados con *upper()* y *lower()* encontramos otro método llamado

capitalize(), el cual solo convierte el primer carácter de un string a mayúsculas:

```
>>> cad = "un ejemplo"
>>> cad.capitalize()
'Un ejemplo'
```

En ocasiones puede ser muy útil dividir una cadena de texto basándonos en un carácter que aparece repetidamente en ella. Esta funcionalidad es la que nos ofrece *split()*. Supongamos que tenemos un cadena con varios valores separadas por *;* y que necesitamos una lista donde cada valor se corresponda con los que aparecen delimitados por el mencionado carácter. El siguiente ejemplo nos muestra cómo hacerlo:

```
>>> cad = "primer valor;segundo;tercer valor"
>>> cad.split(";")
['primer valor', 'segundo', 'tercer valor']
```

join() es un método que devuelve una cadena de texto donde los valores de la cadena original que llama al método aparecen separados por un carácter pasado como argumento:

```
>>> "abc".join(',')
'a,b,c'
```

Operaciones

El operador + nos permite concatenar dos strings, el resultado puede ser almacenado en una nueva variable:

```
>>> cad_concat = "Hola" + " Mundo!"
>>> print(cad_concat)
Hola Mundo!
```

También es posible concatenar una variable de tipo string con una cadena o concatenar directamente dos variables. Sin embargo, también podemos prescindir del operador + para concatenar strings. A veces, la expresión es más fácil de leer si empleamos el método *format()*. Este método admite emplear, dentro de la cadena de texto, los caracteres *{}*, entre los que irá, número o el nombre de una variable. Como argumentos del método pueden pasarse variables

que serán sustituidas, en tiempo de ejecución, por los marcadores *{}*, indicados en la cadena de texto. Por ejemplo, veamos cómo las siguientes expresiones son equivalentes y devuelven el mismo resultado:

```
>>> "Hola " + cad2 + ". Otra " + cad3
>>> "Hola {0}. Otra {1}".format(cad2, cad3)
>>> "Hola {cad2}. Otra {cad3}".format(cad2=cad2,cad3=cad3)
```

La concatenación entre strings y números también es posible, siendo para ello necesario el uso de funciones como *int()* y *str()*. El siguiente ejemplo es un caso sencillo de cómo utilizar la función *str():*

```
>>> num = 3
>>> "Número: " + str(num)
```

Interesante resulta el uso del operador * aplicado a cadenas de texto, ya que nos permite repetir un string *n* veces. Supongamos que deseamos repetir la cadena "Hola Mundo" cuatro veces. Para ello, bastará con lanzar la siguiente sentencia:

```
>>> print("Hola Mundo" * 4)
```

Gracias al operador *in* podemos averiguar si un determinado carácter se encuentra o no en una cadena de texto. Al aplicar el operador, como resultado, obtendremos *True o False*, en función de si el valor se encuentra o no en la cadena. Comprobémoslo en el siguiente ejemplo:

```
>>> cad = "Nueva cadena de texto"
>>> "x" in cad
False
```

Un string es inmutable en Python, pero podemos acceder, a través de índices, a cada carácter que forma parte de la cadena:

```
>>> cad = "Cadenas"
>>> print(cad[2])
d
```

Los comentados índices también nos pueden ayudar a obtener subcadenas de texto basadas en la original. Utilizando la variable *cad* del ejemplo anterior podemos imprimir solo los tres primeros caracteres:

```
>>> print(cad[:3])
```

```
Cad
```

En el ejemplo anterior el operador *:* nos ha ayudado a nuestro propósito. Dado que delante del operador no hemos puesto ningún número, estamos indicando que vamos a utilizar el primer carácter de la cadena. Detrás del operador añadimos el número del índice de la cadena de texto que será utilizado como último valor. Así pues, obtendremos los tres primeros caracteres. Los índices negativos también funcionan, simplemente indican que se empieza contar desde el último carácter. La siguiente sentencia devolverá el valor *a*:

```
>>> cad[-2]
```

A continuación, veamos un ejemplo donde utilizamos un número después del mencionado operador:

```
>>> cad [3:]
'enas'
```

TUPLAS

En Python una *tupla* es una estructura de datos que representa una colección de objetos, pudiendo estos ser de distintos tipos. Internamente, para representar una tupla, Python utiliza un array de objetos que almacena referencias hacia otros objetos.

Para declarar una tupla se utilizan paréntesis, entre los cuales deben separarse por comas los elementos que van a formar parte de ella. En el siguiente ejemplo, crearemos una tupla con tres valores, cada uno de un tipo diferente:

```
>>> t = (1, 'a', 3.5)
```

Los elementos de una tupla son accesibles a través del índice que ocupan en la misma, exactamente igual que en un *array:*

```
>>> t [1]
'a'
```

Debemos tener en cuenta que las tuplas son un tipo de dato inmutable, esto significa que no es posible asignar directamente un valor a través del índice.

A diferencia de otros lenguajes de programación, en Python es posible declarar una tupla añadiendo una coma al final del último elemento:

```
>>> t = (1, 3, 'c', )
```

Dado que una tupla puede almacenar distintos tipos de objetos, es posible *anidar* diferentes tuplas; veamos un sencillo ejemplo de ello:

```
>>> t = (1, ('a', 3), 5.6)
```

Una de las peculiaridades de las tuplas es que es un objeto *iterable*; es decir, con un sencillo bucle *for* podemos recorrer fácilmente todos sus elementos:

```
>>> for ele in t:
...     print(ele)
...
1
('a', 3)
5.6
```

Concatenar dos tuplas es sencillo, se puede hacer directamente a través del operador +. Otros de los operadores que se pueden utilizar es *, que sirve para crear una nueva tupla donde los elementos de la original se repiten *n* veces. Observemos el siguiente ejemplo y el resultado obtenido:

```
>>> (’r', 2) * 3
>>> ('r', 2, 'r', 2, 'r', 2)
```

Los principales métodos que incluyen las tupas son *index()* y *count()* El primero de ellos recibe como parámetro un valor y devuelve el índice de la posición que ocupa en la tupla. Veamos el siguiente ejemplo:

```
>>> t = (1, 3, 7)
>>> t.index(3)
1
```

El método *count()* sirve para obtener el número de ocurrencias de un elemento en una tupla:

```
>>> t = (1, 3, 1, 5, 1,)
>>> t.count(1)
3
```

Sobre las tuplas también podemos usar la función integrada *len()*, que nos devolverá el número de elementos de la misma. Obviamente, deberemos pasar la variable tupla como argumento de la mencionada función.

LISTAS

Básicamente, una *lista* es una colección ordenada de objetos, similar al *array dinámico* empleado en otros lenguajes de programación. Puede contener distintos tipos de objetos, es mutable y Python nos ofrece una serie de funciones y métodos integrados para realizar diferentes tipos de operaciones.

Para definir una lista se utilizan corchetes *([])* entre los cuales pueden aparecer diferentes valores separados por comas. Esto significa que ambas declaraciones son válidas:

```
>>> lista = []
>>> li = [2, 'a' , 4]
```

Al igual que las tuplas, las listas son también *iterables*, así pues, podemos recorrer sus elementos empleando un bucle:

```
>>> for ele in li:
...   print(ele)
...
2
'a'
4
```

A diferencia de las tuplas, los elementos de las listas pueden ser reemplazados accediendo directamente a través del índice que ocupan en la lista. De este modo, para cambiar el segundo elemento de nuestra lista *li*, bastaría como ejecutar la siguiente sentencia:

```
>>> 1i [ 1] = ' b'
```

Obviamente, los valores de las listas pueden ser accedidos utilizando el valor del índice que ocupan en la misma:

```
>>> li [2]
4
```

Podemos comprobar si un determinado valor existe en una lista a través del operado *in*, que devuelve *True* en caso afirmativo y *False* en caso contrario:

```
>>> 'a' in li
True
```

Existen dos funciones integradas que relacionan las listas con las tuplas: *list()* y *tuple()*. La primera toma como argumento una tupla y devuelve una lista. En cambio, *tuple()* devuelve una tupla al recibir como argumento una lista. Por ejemplo, la siguiente sentencia nos devolverá una tupla:

```
>>> tuple(li)
(2, 'a', 4)
```

Operaciones como la suma (+) y la multiplicación (*) también pueden ser aplicadas sobre listas. Su funcionamiento es exactamente igual que en las tuplas.

Inserciones y borrados

Para añadir un nuevo elemento a una lista contamos con el método *append()*. Como parámetro hemos de pasar el valor que deseamos añadir y este será insertado automáticamente al final de la lista. Volviendo a nuestra lista ejemplo de tres elementos, uno nuevo quedaría insertado a través de la siguiente sentencia:

```
>>> li.append('nuevo')
```

Nótese que, para añadir un nuevo elemento, no es posible utilizar un índice superior al número de elementos que contenga la lista. La siguiente sentencia lanza un error:

```
>>> li [4] = 23
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Sin embargo, el método *insert()* sirve para añadir un nuevo elemento especificando el índice. Si pasamos como índice un valor superior al número de elementos de la lista, el valor en cuestión será insertado al final de la misma, sin tener en cuenta el índice pasado como argumento. De este modo, las siguientes sentencias producirán el mismo resultado, siendo *'c'* el nuevo elemento que será insertado en la lista *li:*

```
>>> li.insert(3, 'c')
>>> li.insert(12, 'c')
```

Por el contrario, podemos insertar un elemento en una posición determinada cuyo índice sea menor al número de valores de la lista. Por ejemplo, para insertar un nuevo elemento en la primera posición de nuestra lista *li*, bastaría con ejecutar la siguiente sentencia:

```
>>> li.insert(0, 'd')
>>> li
>>> ['d’, 2, 'a', 4]
```

Si lo que necesitamos es borrar un elemento de una lista, podemos hacerlo gracias a la función *del()*, que recibe como argumento la lista junto al índice que referencia al elemento que deseamos eliminar. La siguiente sentencia ejemplo borra el valor *2* de nuestra lista *li:*

```
>>> del(li[1])
```

Como consecuencia de la sentencia anterior, la lista queda reducida en un elemento. Para comprobarlo contamos con la función *len()*, que nos devuelve el número de elementos de la lista:

```
>>> len(li)
2
```

Obsérvese que la anterior función también puede recibir como argumento una tupla o un string. En general, *len()* funciona sobre tipos de objetos iterables.

También es posible borrar un elemento de una lista a través de su valor. Para ello contamos con el método *remove():*

```
>>> li.remove('d')
```

Si un elemento aparece repetido en la lista, el método *remove()* solo borrará la primera ocurrencia que encuentre en la misma.

Otro método para eliminar elementos es *pop()*. A diferencia de *remove()*, *pop()* devuelve el elemento borrado y recibe como argumento el índice del elemento que será eliminado. Si no se pasa ningún valor como índice, será el último elemento de la lista el eliminado. Este método puede ser útil cuando necesitamos ambas operaciones (borrar y obtener el valor) en una única sentencia.

Ordenación

Los elementos de una lista pueden ser ordenados a través del método *sort() o* utilizando la función *sorted()*. Como argumento se puede utilizar *reverse* con el valor *True* o *False*. Por defecto, se utiliza el segundo valor, el cual indica que la lista será ordenada de mayor a menor. Si por el contrario el valor es *True*, la lista será ordenada inversamente. Veamos un ejemplo para ordenar una lista de enteros:

```
>>> lista = [3, 1, 9, 8, 7]
>>> sorted(lista)
[1, 3, 7, 8, 9]
>>> sorted(lista, reverse=True)
[9, 8, 7, 3, 1]
>>> lista
[3, 1, 9, 8, 7]
```

Como el lector habrá podido observar, la lista original ha quedado inalterada. Sin embargo, si en lugar de utilizar la función *sorted()*, empleamos el método *sort()*, la lista quedará automáticamente modificada. Ejecutemos las siguientes setencias para comprobarlo:

```
>>> lista.sort()
>>> lista
[1, 3, 7, 8, 9]
```

Tanto para aplicar *sort()* como *sorted()* debemos tener en cuenta que la lista que va a ser ordenada contiene elementos que son del mismo tipo. En caso contrario, el intérprete de Python lanzará un error. No obstante, es posible realizar ordenaciones de listas con elementos de distinto tipo si es el programador el encargado de establecer el criterio de ordenación. Para ello, contamos con el parámetro *key* que puede ser pasado como argumento. El valor del mismo puede ser una función que fijará cómo ordenar los elementos. Además, el mencionado parámetro también puede ser utilizado para cambiar la forma de ordenar que emplee el intérprete por defecto, aunque los elementos sean del mismo tipo. Supongamos que definimos la siguiente lista:

```
>>> lis = ['aA', 'Ab', 'Cc', ’ca']
```

Ahora ordenaremos con la función *sorted()* sin ningún parámetro adicional y

observaremos que el criterio de ordenación que utiliza el intérprete, por defecto, es ordenar primero las letras mayúsculas:

```
>>> sorted(lis)
['Ab', 'Cc' , 'aA', 'ca']
```

Sin embargo, al pasar como argumento un determinado criterio de ordenación, el resultado varía:

```
>>> sorted(lis, key=str.lower)
['aA', 'Ab', 'ca', 'Cc']
```

Otro método que contienen las listas relacionado con la ordenación de valores es *reverse()*, que automáticamente ordena una lista en orden inverso al que se encuentran sus elementos originales. Tomando el valor de la última lista de nuestro ejemplo, llamaremos al método para ver qué ocurre:

```
>>> lista.reverse()

>>> lista

[9, 8, 7, 3, 1]
```

Los métodos y funciones de ordenación no solo funcionan con números, sino también con caracteres y con cadenas de texto:

```
>>> lis = ['be', 'ab', 'cc', 'aa', 'cb']
>>> lis.sort()
>>> lis

['aa', 'ab','be', 'cb', 'cc']
```

Comprensión

La *comprensión* de listas es una construcción sintáctica de Python que nos permite declarar una lista a través de la creación de otra. Esta construcción está basada en el principio matemático de la teoría de comprensión de conjuntos. Básicamente, esta teoría afirma que un conjunto se define por comprensión cuando sus elementos son nombrados a través de sus características. Por ejemplo, definimos el conjunto *S* como aquel que está formado por todos los

meses del año: *S = {meses del año}*

Veamos un ejemplo práctico para utilizar la mencionada construcción sintáctica en Python:

```
>>> lista = [ele for ele in (1, 2, 3)]
```

Como resultado de la anterior sentencia, obtendremos una lista con tres elementos diferentes:

```
>>> print(lista)
[1, 2, 3]
```

Gracias a la comprensión de listas podemos definir y crear listas ahorrando líneas de código y escribiendo el mismo de forma más elegante. Sin la comprensión de listas, deberíamos ejecutar las siguientes sentencias para lograr el mismo resultado:

```
>>> lista = []
>>> for ele in (1, 2, 3):
...   lista.append(ele)
...
```

Matrices

Anidando listas podemos construir *matrices* de elementos. Estas estructuras de datos son muy útiles para operaciones matemáticas. Debemos tener en cuenta que complejos problemas matemáticos son resueltos empleando matrices. Además, también son prácticas para almacenar ciertos datos, aunque no se traten estrictamente de representar matrices en el sentido matemático.

Por ejemplo, una matriz matemática de dos dimensiones puede definirse de la siguiente forma:

```
>>> matriz = [[1, 2, 3],[4, 5, 6]]
```

Para acceder al segundo elemento de la primera matriz, bastaría con ejecutar la siguiente sentencia:

```
>>> matriz = [0][1]
```

Asimismo, podemos cambiar un elemento directamente:

```
>>> matriz[0][1] = 33
>>> m

[[1, 33, 3],[4, 5, 6]]
```

DICCIONARIOS

Un *diccionario* es una estructura de datos que almacena una serie de valores utilizando otros como referencia para su acceso y almacenamiento. Cada elemento de un diccionario es un par *clave-valor* donde el primero debe ser único y será usado para acceder al valor que contiene. A diferencia de las tuplas y las listas, los diccionarios no cuentan con un orden específico, siendo el intérprete de Python el encargado de decidir el orden de almacenamiento. Sin embargo, un diccionario es iterable, mutable y representa una colección de objetos que pueden ser de diferentes tipos.

Gracias a su flexibilidad y rapidez de acceso, los diccionarios son una de las estructuras de datos más utilizadas en Python. Internamente son representadas como una tabla *hash*, lo que garantiza la rapidez de acceso a cada elemento, además de permitir aumentar dinámicamente el número de ellos. Otros muchos lenguajes de programación hacen uso de esta estructura de datos, con la diferencia de que es necesario implementar la misma, así como las operaciones de acceso, modificación, borrado y manejo de memoria. Python ofrece la gran ventaja de incluir los diccionarios como estructuras de datos integradas, lo que facilita en gran medida su utilización.

Para declarar un diccionario en Python se utilizan las llaves *({})* entre las que se encuentran los pares clave-valor separados por comas. La clave de cada elemento aparece separada del correspondiente valor por el carácter *:*. El siguiente ejemplo muestra la declaración de un diccionario con tres valores:

```
>>> diccionario = {'a': 1, 'b': 2, 'c': 3}
```

Alternativamente, podemos hacer uso de la función *dict()* que también nos permite crear un diccionario. De esta forma, la siguiente sentencia es equivalente a la anterior:

```
>>> diccionario = dict(a=1, b=2, c=3)
```

Acceso, inserciones y borrados

Como hemos visto previamente, para acceder a los elementos de las listas y las tuplas, hemos utilizado el índice en función de la posición que ocupa cada elemento. Sin embargo, en los diccionarios necesitamos utilizar la clave para acceder al valor de cada elemento. Volviendo a nuestro ejemplo, para obtener el valor indexado por la clave *'c'* bastará con ejecutar la siguiente sentencia:

```
>>> diccionario 'c']
3
```

Para modificar el valor de un diccionario, basta con acceder a través de su clave:

```
>>> diccionario['b'] = 28
```

Añadir un nuevo elemento es tan sencillo como modificar uno ya existente, ya que si la clave no existe, automáticamente Python la añadirá con su correspondiente valor. Así pues, la siguiente sentencia insertará un nuevo valor en nuestro diccionario ejemplo:

```
>>> diccionario['d'] = 4
```

Tres son los métodos principales que nos permiten iterar sobre un diccionario: *items(), values()* y *keys()*. El primero nos da acceso tanto a claves como a valores, el segundo se encarga de devolvernos los valores, y el tercero y último es el que nos devuelve las claves del diccionario. Veamos estos métodos en acción sobre el diccionario original que declaramos previamente:

```
>>> for k, v in diccionario.items():
...   print("clave={0}, valor={1}".format(k, v))
...
clave=a, valor=1
clave=b, valor=2
clave=c, valor=3

>>> for k in diccionario.keys():
...   print("clave={0}".format(k))
...
clave=a
clave=b
clave=c

>>> for v in diccionario.values():
...   print("valor={0}".format(v))
...
```

```
valor=1
valor=2
valor=3
```

Por defecto, si iteramos sobre un diccionario con un bucle *for,* obtendremos las claves del mismo sin necesidad de llamar explícitamente al método *keys()*:

```
>>> for k in diccionario:
...   print(k)
a
b
c
```

A través del método *keys()* y de la función integrada *list()* podemos obtener una lista con todas las claves de un diccionario:

```
>>> list(diccionario.keys())
```

Análogamente es posible usar *values()* junto con la función *list()* para obtener un lista con los valores del diccionario. Por otro lado, la siguiente sentencia nos devolverá una lista de tuplas, donde cada una de ellas contiene dos elementos, la clave y el valor de cada elemento del diccionario:

```
>>> list(diccionario.items())
[ ('a', 1), (’b', 2), ('c', 3)]
```

La función integrada *del()* es la que nos ayudará a eliminar un valor de un diccionario. Para ello, necesitaremos pasar la clave que contiene el valor que deseamos eliminar. Por ejemplo, para eliminar el valor que contiene la clave *'c'* de nuestro diccionario, basta con ejecutar:

```
>>> del(diccionario['b'])
```

El método *pop()* también puede ser utilizado para borrar eliminar elementos de un diccionario. Su funcionamiento es análogo al explicado en el caso de las listas.

Otra función integrada, en este caso *len(),* también funciona sobre los diccionarios, devolviéndonos el número total de elementos contenidos.

El operador *in* en un diccionario sirve para comprobar si una clave existe. En caso afirmativo devolverá el valor *True* y *False* en otro caso:

```
>>> 'x' in diccionario
False
```

Comprensión

De forma similar a las listas, los diccionarios pueden también ser creados por comprensión. El siguiente ejemplo muestra cómo crear un diccionario utilizando la iteración sobre una lista:

```
>>> {k: k+1 for k in (1, 2, 3)}
{1: 2, 3: 4, 4: 5}
```

La comprensión de diccionarios puede ser muy útil para inicializar un diccionario a un determinado valor, tomando como claves los diferentes elementos de una lista. Veamos cómo hacerlo a través del siguiente ejemplo que crea un diccionario inicializándolo con el valor *1* para cada clave:

```
>>> {clave: 1 for clave in ['x', 'y', 'z']}
{'x': 1, ’y': 1, 'z': 1}
```

Ordenación

A diferencia de las listas, los diccionarios no tienen el método *sort()*, pero sí que es posible utilizar la función integrada *sorted()* para obtener una lista ordenada de las claves contenidas. Volviendo a nuestro diccionario ejemplo inicial, ejecutaremos la siguiente sentencia:

```
>>> sorted(diccionario)
['a', 'b', 'c']
```

También podemos utilizar el parámetro *reverse* con el mismo resultado que en las listas:

```
>>> sorted(diccionario, reverse=True)
['c', ' b' , 'a']
```

SENTENCIAS DE CONTROL, MÓDULOS Y FUNCIONES

INTRODUCCIÓN

Las sentencias de control es uno de los primeros aspectos que deben ser abordados durante el aprendizaje de un lenguaje de programación. Entre las sentencias de las que dispone Python, las básicas son las que nos permiten crear condiciones y realizar iteraciones. Es por ello que dedicaremos el primer apartado de este capítulo a las mismas.

Continuaremos entrando de lleno en uno de los principales conceptos de la programación procedural: las funciones. Aprenderemos cómo se definen, cómo son tratadas por el intérprete y cómo pasar parámetros. Además, presentaremos a un tipo especial llamado *lambda,* muy utilizado en programación funcional.

Python permite agrupar nuestro código en *módulos* y *paquetes,* gracias a los cuales podemos organizar adecuadamente nuestros programas. De ellos hablaremos a continuación de las funciones.

Por último, veremos qué son las *excepciones* y cómo podemos trabajar con ellas en Python. El mecanismo de *tratamiento de excepciones* nos ahorra muchos problemas en tiempos de ejecución y se ha convertido en una de las más importantes funcionalidades que incorporan los modernos lenguajes de programación.

PRINCIPALES SENTENCIAS DE CONTROL

Al igual que otros lenguajes de programación, Python incorpora una serie de sentencias de control. Entre ellas, encontramos algunas tan básicas y comunes a otros lenguajes como *if/else, while* y *for,* y otras específicas como *pass* y *with.* A continuación, echaremos un vistazo a cada una de estas sentencias.

if, else y elif

La sentencia *if/else* funciona evaluando la condición indicada, si el resultado es *True* se ejecutará la siguiente sentencia o sentencias, en caso negativo se ejecutarán las sentencias que aparecen a continuación del *else.* Recordemos que Python utiliza la indentación para establecer sentencias que pertenecen al mismo bloque. Además, en el carácter dos puntos (*:*) indica el comienzo de bloque. A continuación, vemos un ejemplo:

```
x = 4
y = 0
if x == 4:
    y = 5
else:
    y = 2
```

Obviamente, también es posible utilizar solo la sentencia *if* para comprobar si se cumple una determinada condición y actuar en consecuencia. Además, podemos anidar diferentes niveles de comprobación a través de *elif*:

```
if X = = 4 :
    y = 1
elif x = = 5
    y = 2
elif x = = 6
    y = 3
else:
    y = 5
```

Como el lector habrá podido observar y a diferencia de otros lenguajes de

programación, los paréntesis para indicar las condiciones han sido omitidos. Para Python son opcionales y habitualmente no suelen ser utilizados. Por otro lado, a pesar de que Python emplea la indentación, también es posible escribir una única sentencia a continuación del final de la condición. Así pues, la siguiente línea de código es válida:

```
if a > b: print("a es mayor que b")
```

for y while

Para iterar contamos con dos sentencias que nos ayudarán a crear bucles, nos referimos a *for* y a *while*. La primera de ellas aplica una serle de sentencias sobre cada uno de los elementos que contiene el objeto sobre el que aplicamos la sentencia *for*. Python incorpora una función llamada *range()* que podemos utilizar para iterar sobre una serie de valores. Por ejemplo, echemos un vistazo al siguiente ejemplo:

```
>>> for x in range(1, 3):
...   print(x)
...
1
2
3
```

Asimismo, tal y como hemos visto en el capítulo anterior, es muy común iterar a través de *for* sobre los elementos de una tupla o de una lista:

```
>>> lista = ["uno", "dos", "tres"]
>>> cad = ""
>>> for ele in lista:
...   cad += ele
...
>>> cad
"unodostres"
```

Opcionalmente, *for* admite la sentencia *else*. Si esta aparece, todas las sentencias posteriores serán ejecutadas si no se encuentra otra sentencia que provoque la salida del bucle. Por ejemplo, en la ejecución de un *bucle for* que no contiene ningún *break*, siempre serán ejecutadas las sentencias que pertenecen al *else* al finalizar el bucle. A continuación, veamos un ejemplo para ilustrar este

caso:

```
>>> for item in (1, 2, 3):
...      print(item)
...   else:
...      print ("fin")
...
1
2
3
fin
```

Otra sentencia utilizada para iterar es *while*, la cual ejecuta una serie de sentencias siempre y cuando se cumpla una determinada condición o condiciones.

Para salir del bucle podemos utilizar diferentes técnicas. La más sencilla es cambiar la condición o condiciones iniciales para así dejar que se cumplan y detener la iteración. Otra técnica es llamar directamente a *break* que provocará la salida inmediata del bucle. Esta última sentencia también funciona con *for.* A continuación, veamos un ejemplo de cómo utilizar *while:*

```
>>> x = 0
>>> y = 3
>>> while x < y:
...      print(x)
...      x += 1
0
1
2
```

Al igual que *for*, *while* también admite opcionalmente *else.* Observemos el siguiente código y el resultado de su ejecución:

```
>>> x = 0
>>> y = 3
>>> while x < y:
...      print(x)
...      x+ = 1
...      if x == 2:
...         break
...      else:
...         print("x es igual a 2")
0
1
```

Si en el ejemplo anterior eliminamos la sentencia *break*, comprobaremos cómo la última sentencia *print* es ejecutada.

Además de *break*, otra sentencia asociada a *for* y *while* es *continue*, la cual se emplea para provocar un salto inmediato a la siguiente iteración del bucle. Esto puede ser útil, por ejemplo, cuando no deseamos ejecutar una determinada sentencia para una iteración concreta. Supongamos que estamos iterando sobre una secuencia y solo queremos imprimir los números pares:

```
>>> for i in range(l, 10):
...       if i % 2 != 0:
...          continue
...       print(i)
2
4
6
8
```

pass y with

Python incorpora una sentencia especial para indicar que no se debe realizar ninguna acción. Se trata de *pass* y especialmente útil cuando deseamos indicar que no se haga nada en una sentencia que requiere otra. Por ejemplo, en un sencillo *while:*

```
>>> while True:
...   pass
```

Muchos desarrolladores emplean *pass* cuando escriben *esqueletos* de código que posteriormente rellenarán. Dado que inicialmente no sabemos qué código contendrá una determinada sentencia, es útil emplear *pass* para mantener el resto del programa funcional. Posteriormente, el *esqueleto* de código será rellenado con código funcional y la sentencia *pass* será reemplazada por otras que realicen una función específica.

La sentencia *with* se utiliza con objetos que soportan el protocolo de *manejador de contexto* y garantiza que una o varias sentencias serán ejecutadas automáticamente. Esto nos ahorra varias líneas de código, a la vez que nos garantiza que ciertas operaciones serán realizadas sin que lo indiquemos explícitamente. Uno de los ejemplos más claros es la lectura de las líneas de un

fichero de texto. Al terminar esta operación siempre es recomendable cerrar el fichero. Gracias a *with* esto ocurrirá automáticamente, sin necesidad de llamar al método *close()*. Las siguientes líneas de código ilustran el proceso:

```
>>> with open(r'info.txt') as myfile:
...      for line in myfile:
...         print(line)
```

FUNCIONES

En programación estructurada, las funciones son uno de los elementos básicos. Una *función* es un conjunto de sentencias que pueden ser invocadas varias veces durante la ejecución de un programa. Las ventajas de su uso son claras, entre ellas, la minimización de código, el aumento de la legibilidad y la fomentación de la reutilización de código.

Una de las principales diferencias de las funciones en Python con respecto a lenguajes compilados, como C, es que estas no existen hasta que son invocadas y el intérprete pasa a su ejecución. Esto implica que la palabra reservada *def*, empleada para definir una función, es una sentencia más. Como consecuencia de ello, otras sentencias pueden contener una función. Así pues, podemos utilizar una sentencia *if*, definiendo una función cuando una determinada condición se cumple.

Internamente, al definir una función, Python crea un nuevo objeto y le asigna el nombre dado para la función. De hecho, una función puede ser asignada a una variable o almacenada en una lista.

Como hemos comentado previamente, la palabra reservada *def* nos servirá para definir una función. Seguidamente deberemos emplear un nombre y, opcionalmente, una serie de argumentos. Esta será nuestra primera función:

```
def test():
    print("test ejecutada")
```

Para invocar a nuestra nueva función, basta con utilizar su nombre seguido de paréntesis. En lugar de utilizar el intérprete para comprobar su funcionamiento, lo haremos a través de un fichero. Basta con abrir nuestro editor de textos favorito y crear un fichero llamado *test.py* con el siguiente código:

```
def test () :
    print("test ejecutada")
test()
nueva = test
nueva()
```

Una vez que salvemos el fichero con el código, ejecutaremos el programa

desde la línea de comandos a través del siguiente comando:

```
python test.py
```

Como resultado veremos cómo aparece dos veces la cadena de texto *test ejecutada.*

Paso de parámetros

En los ejemplos previos hemos utilizado una función sin argumentos y, por lo tanto, para invocar a la misma no hemos utilizado ningún parámetro. Sin embargo, las funciones pueden trabajar con parámetros y devolver resultados. En Python, el paso de parámetros implica la asignación de un nombre a un objeto. Esto significa que, en la práctica, el paso de parámetros ocurre por *valor* o por *referencia* en función de si los tipos de los argumentos son mutables o inmutables. En otros lenguajes de programación, como por ejemplo C, es el programador el responsable de elegir cómo serán pasados los parámetros. Para ilustrar este funcionamiento, observemos el siguiente ejemplo:

```
>>> def test2 (a, b)
...       : a = 2
...       b = 3
...
>>> C = 5
>>> d = 6
>>> test2 (c, d)
>>> print("c={0}, d={l}". format(x, y))
c=5, d=6
```

Como podemos observar, el valor de las variables *c* y *d*, que son inmutables, no ha sido alterado por la función.

Sin embargo, ¿qué ocurre si utilizamos un argumento inmutable como parámetro? Veámoslo a través del siguiente código de ejemplo:

```
>>> def variable(lista):
...       lista [0] = 3
>>> lista = [1, 2, 3]
>>> print(variable(lista))
[3, 2, 3]
```

Efectivamente, al pasar como argumento una lista, que es de mutable, y modificar uno de sus valores, se modificará la variable original pasada como argumento.

Internamente, Python siempre realiza el paso de parámetros a través de referencias, es decir, se puede considerar que el paso se realiza siempre por *variable* en el sentido de que no se hace una copia de las variables. Sin embargo, tal y como hemos mencionado, el comportamiento de esta asignación de referencias depende de si el parámetro en cuestión es mutable o inmutable. Este comportamiento en funciones es similar al que ocurre cuando hacemos asignaciones de variables. Si trabajamos con tipos inmutables y ejecutamos las siguientes sentencias, observaremos cómo el valor de la variable *a* se mantiene:

```
>>> a = 3
>>> b = a
>>> b = 2
>>> print("a={0}, b={1}".format(a, b))
a=3, b=2
```

Por otro lado, si aplicamos el mismo comportamiento a un tipo mutable, veremos cómo varía el resultado:

```
>>> a = [0, 1]
>>> b = a
>>> b[0] = 1
>>> print("a={0}; b={1}".format(a, b))
a= [1, 1] ; b= [1, 1]
```

Si necesitamos modificar una o varias variables inmutables a través de una función, podemos hacerlo utilizando una técnica, consistente en devolver una tupla y asignar el resultado de la función a las variables. Partiendo del ejemplo anterior, reescribiremos la función de la siguiente forma:

```
def test(a, b):
    a = 2
    b = 3
    return(a, b)
```

Posteriormente, realizaremos la llamada a la función y la asignación directamente con una sola sentencia:

```
>>> c, d = test (c, d)
```

A pesar de ser el comportamiento por defecto, es posible no modificar el parámetro mutable pasado como argumento. Para ello, basta con realizar, cuando se llama a la función, una copia explícita de la variable:

```
>>> variable(lista[:])
```

Dado el manejo que realiza Python sobre el paso de parámetros a funciones, en lugar de utilizar los términos *por valor o por referencia*, sería más exacto decir que Python realiza el paso de parámetros *por asignación.*

Valores por defecto y nombres de parámetros

En Python es posible asignar un valor a un parámetro de una función. Esto significa que, si en la correspondiente llamada a la función no pasamos ningún parámetro, se utilizará el valor indicado por defecto. El siguiente código nos muestra un ejemplo:

```
>>> def fun(a, b=1):
...       print(b)
>>> fun(4)
1
```

Hasta ahora hemos visto cómo pasar diferentes argumentos a una función según la posición que ocupan. Es decir, la correspondencia entre parámetros se realiza según el orden. Sin embargo, Python también nos permite pasar argumentos a funciones utilizando nombres y obviando la posición que ocupan. Comprobémoslo a través del siguiente código:

```
>>> def fun(a, b, c):
...       print("a={0}, b={1}, c={2}".format(a, b, c))
>>> fun(c=5, b=3, a=1)
a=1, b=3, c=5
```

Obviamente, podemos combinar valores por defecto y nombres de argumentos para invocar a una función. Supongamos que definimos la siguiente función:

```
>>> def fun(a, b, c=4):
...       print("a={0}, b={1}, c={2}".format(a, b, c))
```

Dada la anterior función, las siguientes sentencias son válidas:

```
>>> fun(1, 2, 4)
>>> fun(a=1, b=2, c=4)
>>> fun(a=1, b=2)

>>> fun(1, 2)
```

Número indefinido de argumentos

Python nos permite crear funciones que acepten un número indefinido de parámetros sin necesidad de que todos ellos aparezcan en la cabecera de la función. Los operadores * y ** son los que se utilizan para esta funcionalidad. En el primer caso, empleando el operador *, Python recoge los parámetros pasados a la función y los convierte en una tupla. De esta forma, con independencia del número de parámetros que pasamos a la función, esta solo necesita el mencionado operador y un nombre. A continuación, un ejemplo que nos muestra esta funcionalidad:

```
>>> def fun(*items):
...       for ítem in ítems:
...          print(ítem)
...
>> fun(1, 2, 3)
1
2
3
>>> fun(5, 6)
5
6
>>> t = ('a', 'b', 'c')
>>> fun(t)
'a'
'b'
'c'
```

Como podemos comprobar, en el código anterior, el comportamiento de la función siempre es el mismo, con independencia del número de argumentos que pasamos.

Por otro lado, gracias al operador ** podemos pasar argumentos indicando un nombre para cada uno de ellos. Internamente, Python construye un

diccionario y los parámetros pasados a la función son tratados como tal. El siguiente ejemplo nos muestra cómo emplear este operador en la cabecera de una función:

```
>>> def fun(**params):
...      print(params)
...
>>> fun(x=5, y=8)
{'x': 5, 'y': 8}
>>> fun(x=5, y=8, z=4)
{'x': 5, 'y': 8, 'z': 4}
```

También es posible que la cabecera de una función utilice uno o varios argumentos posicionales, seguidos del operador * o **. Esto nos proporciona bastante flexibilidad a la hora de invocar a una función. Observemos la siguiente función:

```
def print_record(nombre, apellido, **rec):
    print("Nombre: ", nombre)
    print("Apellidos:", apellidos)
    for k in rec:
        print("{0}: {1}".format(k, rec[k]))
```

Las siguientes invocaciones a la función recién definida serían válidas:

```
>>> print_record("Juan", "Coll", edad=43, localidad="Madrid")
>>> print_record("Manuel", "Tip", edad=34)
```

Desempaquetado de argumentos

En el apartado anterior hemos aprendido a utilizar los operadores * y ** en la cabecera de una función. Sin embargo, dichos operadores también pueden ser empleados en la llamada a la función. El comportamiento es similar y la técnica empleada se conoce como *desempaquetado de argumentos.* Por ejemplo, supongamos que una función tiene en su cabecera tres parámetros diferentes. En la llamada a la misma, en lugar de utilizar tres valores, podemos emplear el operador *, tal y como muestra el siguiente código:

```
>>> def fun(x, y, z):
...      print(x, y, z)
...
```

```
>>> t = (1, 2, 3)
>>> fun(*t)
1 2 3
```

En lugar de una tupla, el operador ** se basa en el uso de un diccionario. Tomando como ejemplo la función definida previamente, veamos el comportamiento de este operador:

```
>>> d = {'y': 1, 'z': 2, 'x': 0}
>>> fun(**d)
0 1 2
```

También es posible combinar el paso de parámetros con el operador ** utilizando valores por defecto en la cabecera de la función:

```
>>> def fun(a=1, b=2, c=3):
...      print(a, b, c)
...
>>> d = {'a': 3, 'b': 4}
>>> fun(**d)
3 4 1
```

Funciones con el mismo nombre

Python permite definir diferentes funciones con el mismo nombre y diferente número de argumentos. Sin embargo, su comportamiento es distinto al que hacen otros lenguajes de programación, como es el caso de Java. Si definimos más de una función como el mismo nombre y con el mismo o diferente número de argumentos, Python empleará siempre la última que ha sido definida. Este comportamiento se debe a que Python trata las funciones como un tipo determinado de objeto. De esta forma, al volver a definir una función, estamos creando un nueva variable que será asignada a un nuevo valor. Ilustremos este hecho con un ejemplo, donde vamos a definir dos funciones con el mismo nombre y diferente número de argumentos:

```
>>> def fun(x, y):
...      print(x, y)
...
>>> def fun(x):
...      print(x)
...
```

Seguidamente invocaremos a la función pasando dos parámetros:

```
>>> fun(1, 3)
Traceback (most recent cali last):
    File "<stdin>", line 1, in <module>
TypeError: fun() takes exactly 1 positional argument (2 given)
```

Como el lector habrá podido observar, el intérprete de Python muestra un error indicándonos que debemos pasar exactamente un único argumento. Sin embargo, la siguiente llamada a la función es válida:

```
>>> fun(4)
```

Para comprobar que Python trata a las funciones como un tipo de dato específico, basta con ejecutar la siguiente sentencia y echar un vistazo a su resultado:

```
>>> type(fun)
<class 'function'>
```

Funciones lambda

Al igual que otros lenguajes de programación, Python permite el uso de funciones *lambda.* Este tipo especial de función se caracteriza por devolver una función *anónima* cuando es asignada a una variable. Aquellos lectores que hayan trabajado con Lisp, u otros lenguajes funcionales, seguro que están familiarizados con su uso. En definitiva, las funciones lambda ejecutan una determinada expresión, aceptando o no parámetros y devuelven un resultado. A su vez, la llamada a este tipo de funciones puede ser utilizada como parámetros para otras.

En Python, las funciones lambda no pueden contener bucles y no pueden utilizar la palabra clave *return* para devolver un valor. La sintaxis para este tipo de funciones es del siguiente tipo:

```
lambda <parámetros>:<expresión>
```

Técnicamente, las funciones lambda no son una sentencia, sino una expresión. Esto las hace diferentes de las funciones definidas con *def*, ya que estas siempre hacen que el intérprete las asocie a un nombre determinado, en

lugar de simplemente devolver un resultado, tal y como ocurre con las lambda.

En la práctica, la utilidad de las funciones lambda es que nos permite definir una función directamente en el código que va a hacer uso de ella. Es decir, nos permite definir funciones *inline.* Esto puede ser útil, por ejemplo, para definir una lista con diferentes acciones que serán ejecutadas bajo demanda. Supongamos que necesitamos ejecutar dos funciones diferentes pasando el mismo parámetro, estando ambas funciones definidas en una determinada lista. En lugar de definir tres funciones diferentes utilizando *def*, vamos a emplear funciones lambda:

```
>>> li = [lambda x: x + 2, lambda x: x + 3]
>>> param = 4
>>> for accion in li:
...       print(accion(param))
...
4
5
```

A continuación, veremos un ejemplo de asignación de función lambda a una variable y su posterior invocación:

```
>>> lam = lambda x: x*5
>>> print(lam(3))
15
```

Equivalente en funcionalidad al anterior ejemplo, sería el siguiente código:

```
>>> def lam(x):
...       return x*5
...
>>> print(lam(3))
15
```

Para ilustrar el paso como parámetro de una función lambda a otra función convencional presentaremos primero a la función integrada de Python llamada *map().* Esta función recibe dos parámetros, el primero es una función que debe ser ejecutada para cada uno de los elementos que contiene el segundo parámetro. Como ejemplo tomaremos una lista con una serie de valores y aplicaremos sobre cada uno de ellos una simple función: sumar el número dos. Después imprimiremos por la salida estándar el resultado. El código en cuestión sería el que aparece a continuación:

```
>>> li = [1, 2, 3]
>>> new_li = map(lambda x: x+2, li)
>>> for item in new_li: print(item)
...
3
4

5
```

Tipos mutables como argumentos por defecto

Cuando fijamos el valor de un argumento en la cabecera de una función es conveniente tener en cuenta que este debería ser de tipo inmutable. En caso contrario podríamos obtener resultados inesperados. Es decir, no se debe emplear objetos como listas, diccionarios o instancias de clase como argumentos por defecto de una función. Observemos el siguiente ejemplo, donde fijamos una lista vacía como argumento por defecto:

```
>>> fun(x=1, 1i = []) :
        li.append(x)
...
>>> fun()
[1]
>>> fun()
[1, 1]
>>> fun(2)
[1, 1, 2]
```

¿Inesperado resultado? En lugar de devolver una lista con un único valor, dado que la lista es inicializada en el argumento por defecto, Python devuelve una lista con un nuevo valor cada vez que la función es invocada. Esto se debe a que el intérprete crea los valores por defecto cuando la sentencia que define la función es ejecutada y no cuando la función es invocada. Sin embargo, ¿por qué el comportamiento de nuestro parámetro *x* es diferente? Simplemente porque *x* es inmutable, mientras que *li* es mutable. Así pues, la lista puede cambiar su valor, que es precisamente lo que hace nuestra función de ejemplo, añadiendo un nuevo elemento cada vez que la misma es invocada.

¿Y si deseamos precisamente inicializar una lista cada vez que nuestra función ejemplo es invocada? Bastaría con una sencilla técnica consistente en utilizar el valor *None* por defecto para nuestro parámetro y en crear una lista

vacía cuando esto ocurre. El siguiente código muestra cómo hacerlo:

```
>>> def fun(x=1, li=None):
...      if lis is None:
...         lis = []
...      lis.append(x)
...      print(lis)
...
>>> fun()
[1]
>>> fun()
[1]
>>> fun(2)
[2]
>>> li = [3, 4]
>>> fun(6, li)
[3, 4, 6]
```

MÓDULOS Y PAQUETES

La organización del código es imprescindible para su eficiente mantenimiento y posible reutilización. Cuanto mayor es el número de ficheros que forman parte de un programa, más importante es mantener su organización. Para ello, Python nos ofrece dos unidades básicas: los módulos y los paquetes. Comenzaremos aprendiendo sobre los primeros.

Módulos

Básicamente, un *módulo* de Python es un fichero codificado en este lenguaje y cumple dos roles principales: permitir la reusabilidad de código y mantener un *espacio de nombres* de variables único. Para explicar estos roles, debemos pensar en cómo Python estructura el código. Un script de Python tiene un punto de entrada para su ejecución, consta de varias sentencias y puede tener definidas funciones que serán invocadas durante su ejecución. Sin embargo, si tenemos definidos dos scripts diferentes, en uno de ellos podemos invocar a una o varias funciones que existan en el otro. Esto sería un ejemplo básico de reutilización de código. ¿Qué ocurre si tenemos definidas funciones con el mismo nombre en ambos scripts? Aquí es donde entra en juego el espacio de nombres único, ya que cada fichero es por sí mismo un módulo y como tal mantiene la unicidad de su espacio. Pongamos esta teoría en práctica a través del siguiente ejemplo. En primer lugar crearemos un fichero llamado *first.py* con el siguiente código:

```
def say_hello():
    print("Hola Mundo!")
print("Soy el primero")
```

A continuación, crearemos nuestro segundo fichero, al que llamaremos *second.py* y añadiremos este código:

```
import first
first.say_hello()
```

Ahora utilizaremos el intérprete para ejecutar este último script creado. Desde la línea de comandos ejecutamos la siguiente sentencia:

```
$ python second.py
```

Siendo el resultado mostrado el siguiente:

```
Soy el primero
Hola Mundo!
```

Como habrá observado el lector, hemos introducido una nueva sentencia llamada *import.* Esta nos permite indicar qué módulo va a ser utilizado. A partir de esta sentencia, como hemos invocado a un módulo con su propio espacio de nombres, podremos invocar a cualquier objeto que esté definido en el primero. Para ello, hacemos uso del carácter punto que actúa como separador entre módulo y objeto. Dado que una función es un objeto, esta puede ser utilizada en nuestro segundo script, igual como ocurriría, por ejemplo, con una variable.

Además de la sentencia *import*, también podemos utilizar *from* con un comportamiento similar. Ambas difieren en que la segunda nos permitirá utilizar objetos sin necesidad de indicar el módulo al que pertenecen. De esta forma, el script *second.py* puede reescribirse de la siguiente forma:

```
from first import say_hello
say_hello()
```

Cuando utilicemos *from* hemos de tener cuidado de que no exista un objeto definido con el mismo nombre, en cuyo caso se utilizaría el último en ser definido. Básicamente, en la práctica, el uso de *import y from* dependerán de este hecho.

El operador * puede ser utilizado para importar todos y cada uno de los objetos declarados en un módulo. Así pues, con una única sentencia tendríamos acceso a todos ellos. Por otro lado, si solo necesitamos importar unos cuantos, podemos separarlos utilizando la coma. Supongamos que nuestro primer script de ejemplo tuviera definidas cinco funciones, pero nosotros solo necesitamos utilizar dos de ellas. Bastaría con utilizar la siguiente sentencia:

```
from first import say_hello, say_bye
```

Adicionalmente, un módulo puede ser importado para ser referenciado con un nombre diferente. Para ello, se utiliza la palabra clave *as* seguida del nombre

que deseamos emplear. Por ejemplo, supongamos que deseamos importar nuestro módulo *first* con el nombre de *primero,* bastaría con la siguiente sentencia:

```
import first as primero
```

A partir de la redefinición anterior, todas las referencias al módulo *first* se realizarán a través de *primero,* tal y como muestra el siguiente ejemplo:

```
primero.say_hello()
```

Volviendo a nuestro ejemplo inicial, tal y como habremos observado, la sentencia print("Soy el primero") ha sido ejecutada a pesar de no formar parte de la función *say_hello().* Para explicar este comportamiento, debemos entender cómo funciona la *importación* de módulos en Python.

FUNCIONAMIENTO DE LA IMPORTACIÓN

Algunos lenguajes permiten trabajar de forma estructurada de forma análoga a como lo hace Python. Se pueden crear componentes y desde unos ficheros invocar a funciones y/o variables declaradas en otros. Sin embargo, a diferencia, por ejemplo del lenguaje C, en Python la importación de un módulo no es simplemente la inserción de código de un fichero en otro, ya que esta es una operación que ocurre en tiempo de ejecución. En concreto, se llevan a cabo tres pasos durante la importación. El primero de ellos localiza el fichero físico de código que corresponde al módulo en cuestión. Para llevar a cabo este paso, se utiliza un *path de búsqueda* determinado del que nos ocuparemos en el apartado siguiente de este capítulo. Una vez localizado el fichero se procede a la generación del *bytecode* asociado al mismo. Este proceso ocurre en función de las fechas del fichero de código *(.py)* y del *bytecode* asociado *(.pyc)*. Si la del primero es posterior a la del segundo, entonces se vuelve a generar todo automáticamente. El último paso durante la importación es la ejecución del código del módulo importado. Esta es la razón por la cual se ejecuta la sentencia *print* de nuestro ejemplo.

Dado que los pasos llevados a cabo durante la importación pueden ser costosos en tiempo, el intérprete de Python solo importa cada módulo una vez por proceso. Esto implica que sucesivas importaciones se saltarán los tres pasos

y el Intérprete pasará directamente a utilizar el módulo contenido en memoria. No obstante, siendo este el comportamiento por defecto, Python nos permite emplear la función *reload* del módulo *imp*, que se encuentra en la librería estándar. Gracias a la cual, es posible indicar que se vuelva a realizar la importación de un determinado módulo.

PATH DE BÚSQUEDA

Anteriormente hemos mencionado que el primer paso del proceso de importación utiliza un *path* para la búsqueda. Es decir, el intérprete necesita saber desde qué localización o *path* del sistema de ficheros debe comenzar a buscar. Por defecto, el primer lugar donde busca es el directorio donde reside el fichero que está siendo ejecutado. Así pues, dado que nuestros scripts de ejemplo han sido creados en el mismo directorio, la importación se realiza correctamente. El segundo lugar que comprueba el intérprete es el directorio referenciado por el valor de la variable de entorno *PYTHONPATH*. Esta variable puede estar creada o no, dependiendo ello del sistema operativo y de si estamos utilizando la línea de comandos. Finalmente, se examinan los directorios de la librería estándar. A la misma, dedicaremos el siguiente apartado.

Gracias al *path* de búsqueda que emplea el intérprete, es posible importar cualquier módulo con independencia de su localización en el sistema de ficheros. Obviamente, para ello es necesaria la definición de la mencionada variable *PYTHONPATH*. Además, esta variable nos aporta bastante flexibilidad, ya que la misma puede ser modificada en tiempo de ejecución.

Comprobar cuáles son los directorios actuales en el path de búsqueda es fácil gracias a la lista *path* que pertenece al módulo *sys* de la librería estándar de Python. Por ejemplo, ejecutemos las siguientes sentencias en Windows y observemos el resultado:

```
>>> import sys
>>> sys.path
['','C:\\Windows\\system32\\python32.zip',
'C:\\Python32\\DLLs', 'C:\\Python32\\lib',
'C:\\Python32', 'C:\\Python32\\lib\\site-packages']
```

Dado que *sys.path* nos devuelve una lista, esta puede ser modificada en tiempo de ejecución, logrando así cambiar los directorios por defecto asociados

al path de búsqueda.

LIBRERÍA ESTÁNDAR

Python incorpora una extensa colección de módulos puesta automáticamente a disposición del programador. A esta colección se le conoce con el nombre de *librería estándar* y, entre los módulos que incorpora, encontramos utilidades para interactuar con el sistema operativo, trabajar con expresiones regulares, manejar protocolos de red como HTTP, FTP y SSL, leer y escribir ficheros, comprimir y descomprimir datos, manejar funciones criptográficas y procesar ficheros XML.

Toda la información referente a la librería estándar de Python puede encontrarse en la página web oficial de la misma (ver referencias).

Paquetes

Hasta ahora hemos visto la relación directa entre ficheros y módulos. Sin embargo, los directorios del sistema de ficheros también pueden utilizarse en la importación. Esto nos lleva al concepto de *paquete*, que no es sino un directorio que contiene varios ficheros de código Python que guardan entre sí una relación conceptual basada en su funcionalidad. La peculiaridad del intérprete es que, automáticamente, genera un espacio de nombres de variables a partir de un directorio. Ello también afecta directamente a cada subdirectorio que exista a partir del directorio en cuestión. De esta forma, podemos organizar más cómodamente nuestro código.

Para crear nuestro primer paquete, comenzaremos creando un nuevo directorio al que llamaremos *mypackage*. Copiaremos dentro del mismo nuestros scripts *first.py* y *second.py.* Por último, crearemos un fichero vacío llamado *__init__.py*. Ya tenemos listo nuestro paquete, ahora probaremos su funcionamiento a través de un nuevo fichero *(main.py)*, que debe estar fuera del directorio creado, con el siguiente contenido:

```
from mymodule import first
first.say_hello()
```

La ejecución del último script creado nos mostrará cómo es realizado el correspondiente *import* a través de nuestro paquete:

```
$ python main.py
Soy el primero
Hola Mundo!
```

Seguro que al lector no se le ha pasado por alto el nuevo fichero *__init.py__*.

Todos los directorios que vayan a ser empleados como paquetes deben contenerlo. Se trata de un fichero en el que podemos añadir código y que se puede utilizar sin ninguno. En la práctica, este fichero se utiliza para incluir aquellas sentencias que son necesarias para realizar acciones de inicialización. Por otro lado, el fichero *__init__.py* también se utiliza para que Python considere los directorios del sistema de ficheros como contenedores de módulos. De esta forma, cualquier directorio del sistema de ficheros puede ser un paquete de Python, permitiéndose utilizar diferentes niveles de subdirectorios. Las sentencias *import* y *from* son las que son empleadas para importar los diferentes módulos que forman parte de un paquete.

COMENTARIOS

Al igual que en otros lenguajes de programación, en Python podemos añadir *comentarios* a nuestro código. En realidad, estos comentarios son un tipo de sentencias especiales que el intérprete entiende que es código que no debe ejecutar. Los comentarios son muy útiles para describir qué acción lleva a cabo determinado código. En general, es una buena práctica de programación documentar nuestro código añadiendo comentarios. Estos son muy prácticos para entender el funcionamiento del código, tanto para otros programadores, como para nosotros mismos. Es habitual, cuando transcurre cierto tiempo, olvidar qué hace un determinado trozo de código, aunque lo haya escrito el mismo programador. De forma que los comentarios nos serán muy útiles en casos como este.

Python utiliza dos tipos de comentarios, los que se utilizan para comentar una única línea y los que se emplean para más de una línea. El siguiente ejemplo muestra cómo utilizar los del primer tipo:

```
# Esto es un comentario
print("Hola Mundo!")
```

Por otro lado, para los comentarios multilínea, emplearemos tres comillas dobles (") seguidas, tal y como muestra el siguiente ejemplo:

```
"""Comienzo de primera línea de comentario
Segunda línea de comentario
Tercera línea de comentario"""
```

EXCEPCIONES

El control de *excepciones* es un mecanismo que nos ofrece el lenguaje para detectar ciertos eventos y modificar el flujo original del programa en ejecución. Habitualmente, estos eventos son errores que ocurren en tipo de ejecución. Este control de errores nos permite detectarlos, realizar una serie de acciones en consecuencia y modificar el flujo de nuestro programa.

Cuando un error ocurre en tiempo de ejecución, el intérprete de Python aborta la ejecución del programa. Python dispara automáticamente un evento cuando uno de estos errores ocurre. El control de excepciones del lenguaje nos permitirá comprobar si uno de estos eventos ha sido lanzado. Además de controlar los errores, las excepciones nos permiten notificar el evento que se genera como consecuencia del error producido.

Capturando excepciones

Para explicar cómo funciona la captura de excepciones, partiremos de una lista que contiene dos elementos. Si intentamos acceder a la lista utilizando el valor de índice 2, ocurrirá un error:

```
>>> li = [0, 1]
>>> li [2]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Suponiendo que las líneas anteriores de código forman parte de un programa, lo que ocurriría al ejecutar la última de ellas, es que el intérprete detendría

automáticamente la ejecución del programa. Para evitar esto, podemos capturar la excepción *IndexError*, la cual ha sido lanzada al detectar el error:

```
>>> try:
...       print(li[2])
...    except IndexError:
...       print("Error: índice no válido")
...
Error índice no válido
```

Empleando el código anterior la ejecución no será detenida y el intérprete continuará ejecutando el programa. Efectivamente, el bloque *try/except* es el utilizado para capturar excepciones. Justo después de la palabra clave *except* debemos indicar el tipo de excepción que deseamos detectar. Por defecto, si no indicamos ninguna, cualquier excepción será capturada. A partir de la sentencia *try*, se tendrá en cuenta cualquier línea de código y si, como consecuencia de la ejecución de una de ellas, se produce una excepción serán ejecutadas las sentencias de código que aparecen dentro de la sentencia *except*. En concreto, la sintaxis de la cláusula *try/except* es como sigue:

```
try:
    <sentencias_susceptibles_de_lanzar_error>
except [<Nombre_excepcion>]:
    <sentencias_ejecutadas_cuando_error>
finally:
    <sentencias_ejecutadas_siempre>
else:
    <sentencias_ejecutadas_si_no_error>
```

Como ejemplo para ilustrar la sintaxis de *try/except* basta con modificar ligeramente el último ejemplo de código:

```
try:
    <li [2]
except:
    <print("Error: índice no válido")
else:
    <print("Sin error")
finally:
    <print("Bloque ejecutado")
```

Si ejecutamos el código anterior, comprobaremos cómo el intérprete lanzará la primera y tercera sentencia *print*. Sin embargo, si sustituimos la sentencia *li[2]* por *li[0]* observaremos cómo son las dos últimas sentencias *print* las

ejecutadas.

Lanzando excepciones

En determinadas ocasiones puede ser interesante que lancemos una excepción concreta desde nuestro código. Podemos lanzar cualquiera definida en el lenguaje o una propia que nosotros creemos. De este último tipo nos ocuparemos en el siguiente apartado.

La sentencia *raise* es la propuesta por Python para lanzar excepciones. El ejemplo más sencillo posible, sería invocarla directamente seguida del nombre de la excepción que deseamos lanzar. A continuación, he aquí el código en cuestión para nuestro ejemplo:

```
>>> try:
...       raise TypeError
...    except:
...       print("Error de tipo")
...
Error de tipo
```

Es importante tener en cuenta que si lanzamos una excepción y esta no es capturada, será propagada hacia el nivel superior de código hasta encontrar un bloque que la maneje. Si ningún bloque la captura, el intérprete mostrará el error y detendrá la ejecución del código.

Excepciones definidas por el usuario

Antes de continuar, el lector deberá entender los fundamentos de la programación orientada a objetos. En concreto, es interesante saber cómo implementar una clase y cómo funciona la herencia. El siguiente capítulo está dedicado a cómo emplear este paradigma de programación en Python.

El programador puede crear sus propios tipos de excepciones y lanzarlas cuando sea necesario. El nombre de excepción definido por el usuario debe corresponder a una clase que herede de la clase *Exception*, la cual está definida en la librería estándar de Python. La clase en cuestión contendrá las acciones que deben ser ejecutadas cuando la excepción es lanzada.

Para utilizar las excepciones que definamos necesitaremos emplear *raise* para

que puedan ser lanzadas en un momento determinado. Tengamos en cuenta que el intérprete no podrá lanzarlas automáticamente, es por ello que *raise* es necesario.

Trabajemos con un sencillo ejemplo donde crearemos y lanzaremos una excepción a la que llamaremos *ValorIncorrecto*. Se trata de comprobar si un número está en un rango determinado de valores. En caso afirmativo lanzaremos nuestra excepción. El primer paso es definir nuestra clase:

```
class ValorIncorrecto(Exception):
    def __init__(self, val):
        print(("{0} no permitido").format(val))
```

En este caso concreto, la excepción definida simplemente imprimirá un mensaje informando de que el valor no está permitido. Obviamente, podemos utilizar diferentes sentencias para llevar a cabo distintas acciones. Por otro lado, el constructor de la clase utiliza un parámetro, que deberá ser el valor que estamos comprobando. Observemos cómo lanzar la excepción definida y lo que nos devuelve el intérprete:

```
>>> val = 8
>>> if val > 5 and val < 9:
...      raise ValorIncorrecto(val)
...
8 no permitido
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
__main__.ValorIncorrecto
```

Información sobre la excepción

Python nos permite trabajar con la información generada cuando se produce una excepción. Esto puede ser útil para indicarle al usuario detalles sobre lo ocurrido. Si además utilizamos un fichero de *log* para volcar esta información, tendremos una útil herramienta para detectar lo sucedido en tiempo de ejecución.

El módulo *sys* de la librería estándar dispone de una función llamada *exc_info()* que nos devuelve una tupla con tres valores principales: el tipo de la excepción, la instancia de clase correspondiente a la excepción y un objeto *traceback* que contiene toda la información que existe en el *stack* en el punto en el que se produjo la excepción. Si volvemos al primer ejemplo con el que

comenzamos nuestro apartado sobre excepciones y justo después de la sentencia *except* añadimos el siguiente código, al ejecutarlo, veremos cómo el intérprete muestra la información requerida:

```
>>> try:
...       print(li[2])
...    except IndexError:
...       import sys
...       sys.exc.info()
...
(<class 'IndexError'>, IndexError('list assignment
index out of range',), <trace
back object at 0x00000000022AB848>)
```

Si quisiéramos solo mostrar el error ocurrido o bien volcarlo a un fichero de log, podríamos obtenerlo utilizando la siguiente sentencia:

```
sys.exc.info()[2]
```

El resultado de sustituir la anterior línea de código por la de *sys.exc.info()* nos lanzaría el siguiente resultado:

```
IndexError('list assignment index out of range')
```

Aquí finaliza este capítulo que sienta las bases para comenzar a programar con Python. Los siguientes capítulos del libro nos mostrarán cómo profundizar en el lenguaje, comenzando por el paradigma de la orientación a objetos.

ORIENTACION A OBJETOS

INTRODUCCIÓN

Entre los paradigmas de programación soportados por Python destacan el procedural, el funcional y la orientación a objetos. De este último y de su aplicación en el lenguaje nos ocuparemos en este capítulo.

En líneas generales, la orientación a objetos se basa en la definición e interacción de unas unidades mínimas de código, llamadas objetos, para el diseño y desarrollo de aplicaciones. Este paradigma comenzó a popularizarse a mediados de los años 90 y, en la actualidad, es uno de los más utilizados. Algunos lenguajes, como Java, están basados completamente en el uso del mismo. Otros como Ruby, C++ y PHP5 también ofrecen un amplio y completo soporte de la orientación a objetos. A este paradigma se le conoce también por sus siglas en inglés: OOP (Object Oriented Programming).

La orientación a objetos es ampliamente utilizada por los programadores de Python, especialmente en aquellos proyectos que, por su tamaño, requieren de una buena organización que ayude a su mantenimiento. Además, modernos componentes, como son los frameworks web, hacen un uso intensivo de este paradigma. Entre las claras ventajas de la orientación a objetos podemos destacar la reusabilidad de código, la modularidad y la facilidad para modelar componentes del mundo real.

A los programadores de C++ y Java les resultará familiar la terminología y conceptos explicados en este capítulo. Sin embargo, deberán prestar atención a las diferencias que presenta Python con respecto a estos lenguajes.

Para aquellos lectores que nunca han trabajado con orientación a objetos, recomendamos la lectura de la página de Wikipedia correspondiente (ver referencias).

En este capítulo veremos cómo definir y utilizar clases y objetos. Explicaremos qué tipos de variables y métodos pueden ser declarados en la definición de una clase. Descubriremos cuáles son las diferencias que presenta Python con respecto a otros lenguajes a la hora de trabajar con el paradigma de la OOP. Los dos últimos apartados están dedicados a dos conceptos

fundamentales en la orientación a objetos: la herencia y el polimorfismo. Comencemos aprendiendo cómo definir clases y cómo instanciar objetos de las mismas.

CLASES Y OBJETOS

Hasta ahora hemos utilizado indistintamente el concepto de objeto como tipo o estructura de datos. Sin embargo, dentro del contexto de la OOP, un objeto es un componente que tiene un rol específico y que puede interactuar con otros. En realidad, se trata de establecer una equivalencia entre un objeto del mundo real con un componente software. De esta forma, el modelado para la representación y resolución de problemas del mundo real a través de la programación, es más sencillo e intuitivo. Si echamos un vistazo a nuestro alrededor, todos los objetos presentan dos componentes principales: un conjunto de características y propiedades y un comportamiento determinado. Por ejemplo, pensemos en una moto. Esta tiene características como color, marca y modelo. Asimismo, su comportamiento puede ser descrito en función de las operaciones que puede realizar, por ejemplo, frenar, acelerar o girar. De la misma forma, en programación, un objeto puede tener atributos y se pueden definir operaciones que puede realizar. Los atributos equivaldrían a las características de los objetos del mundo real y las operaciones se relacionan con su comportamiento.

Es importante distinguir entre clase y objeto. Para ello, introduciremos un nuevo concepto: la instancia. La definición de los atributos y operaciones de un objeto se lleva a cabo a través de una clase. La instanciación es un mecanismo que nos permite crear un objeto que pertenece a una determinada clase. De esta forma, podemos tener diferentes objetos que pertenezcan a la misma clase.

Crear una clase en Python es tan sencillo como emplear la sentencia *class* seguida de un nombre. Por convención, el nombre de la clase empieza en mayúscula. También suele estar contenida en un fichero del mismo nombre de la clase, pero todo en minúscula. La clase más sencilla que podemos crear en Python es la siguiente:

```
class First:
    pass
```

Una vez que tenemos la clase definida, crearemos un objeto utilizando la siguiente línea de código:

```
a = First ()
```

Al igual que otros lenguajes de programación, al declarar una clase podemos hacer uso de un método constructor que se encargue de realizar tareas de inicialización. Este método siempre es ejecutado cuando se crea un objeto. Python difiere con otros lenguajes al disponer de dos métodos involucrados en la creación de un objeto. El primero de ellos se llama *__init__* y el segundo *__new__*. Este último es el primero en ser invocado al crear el objeto. Después, se invoca a *__init__* que es el que habitualmente se emplea para ejecutar todas las operaciones iniciales que necesitemos. En la práctica utilizaremos *__init__* como nuestro método constructor. Sobre el método*__new__* nos ocuparemos en el apartado "Métodos especiales" del presente capítulo. Así pues, nuestra clase con su constructor quedaría de la siguiente forma:

```
class First:
    def __init__ (self):
        print("Constructor ejecutado")
```

Al crear una nueva instancia de la clase, es decir, un nuevo objeto, veríamos cómo se imprime el mensaje:

```
>>> f = First()
Constructor ejecutado
```

Seguro que el lector se ha percatado del parámetro formal que contiene nuestro método constructor, nos referimos a *self*. Este parámetro contiene una referencia al objeto que ejecuta el método y, por lo tanto, puede ser utilizada para acceder al espacio de nombres del objeto. Debemos tener en cuenta que cada objeto contiene su propio espacio de nombres. Los métodos de instancia deben contener este parámetro y debe ser el primero en el orden. Otros lenguajes utilizan un mecanismo similar a través de la palabra clave *this*, por ejemplo PHP; sin embargo, en Python *self* no es una palabra clave y podemos emplear cualquier nombre para esta referencia. Sin embargo, esto no es aconsejable, ya que, por convención, está ampliamente aceptado y arraigado el nombre de *self* para este propósito. Por otro lado, cuando se invoca a un método, no es necesario incluir la mencionada referencia. Como ejemplo, modifiquemos nuestra clase añadiendo el siguiente método:

```
def nuevo(self):
    print("Soy nuevo")
```

Posteriormente, creemos un nuevo objeto de nuestra clase e invoquemos al método recién creado:

```
>>> f = First()
Constructor ejecutado
>>> f.nuevo()
Soy nuevo
```

Variables de instancia

Anteriormente hemos mencionado a los atributos y los hemos señalado como la correspondencia a las características de los objetos del mundo real. En la terminología de Python, a estos atributos se les denomina variables de instancia y siempre deben ir precedidos de la referencia *self.* Volviendo a nuestro ejemplo de la moto, modelemos una clase que contenga unas cuantas variables de instancia:

```
class Moto:
    def __init__ (self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
```

Los atributos de nuestra clase son creados al inicializar el objeto, utilizando tres variables diferentes que son pasadas como parámetros del constructor. De esta forma, la siguiente creación e inicialización de objetos sería válida:

```
>>> bmw_r1000 = Moto("BMW", "r1OO", "blanca")
>>> suzuki_gsx = Moto("Suzuki", "GSX", "negra")
```

Dado que *self* es empleado en todos los métodos de instancia, podemos acceder a los atributos en cualquiera de estos métodos. De hecho esta es una de las ventajas de usar *self,* ya que, cuando lo hacemos seguido de un atributo, tenemos la seguridad de que nos referimos al mismo y no a una variable definida en el espacio de nombres del método en cuestión. Para ilustrar este comportamiento, añadamos el siguiente nuevo método:

```
def get_marca(self):
    marca = "Nueva marca"
    print(self.marca)
```

Ahora creemos volvamos a crear un objeto e invoquemos a este método:

```
>>> honda_cbr = Moto("Honda", "CBR", "roja")
>>> honda_cbr.get_marca()
Honda
```

Métodos de instancia

Este tipo de métodos son aquellos que, en la práctica, definen las operaciones que pueden realizar los objetos. De hecho, nuestro ejemplo anterior *(get_marca)* es un ejemplo de ello. Habitualmente a este tipo de métodos se les llama simplemente métodos, aunque, tal y como veremos en sucesivos apartados, existen otros tipos de métodos que pueden definirse en una clase.

Además de *self*, un método de instancia puede recibir un número *n* de parámetros. Una vez más, volvamos a nuestro ejemplo de la clase Moto y añadamos un nuevo método que nos permita acelerar:

```
def acelerar(self, km):
    print("acelerando {0} km".format(km))
```

La llamada al nuevo método quedaría de la siguiente forma:

```
>>> honda_cbr.acelerar(20)
acelarando 20 km
```

Formalmente, un método de instancia es una función que se define dentro de una clase y cuyo primer argumento es siempre una referencia a la instancia que lo invoca.

Variables de clase

Relacionados con los atributos, también existen en Python las variables de clase. Estas se caracterizan porque no forman parte de una instancia concreta, sino de la clase en general. Esto implica que pueden ser invocados sin necesidad de hacerlo a través de una instancia. Para declararlas bastan con crear una variable justo después de la definición de la clase y fuera de cualquier método. A

continuación, veamos un ejemplo que declara la variable *n_ruedas:*

```
class Moto:
    n_ruedas = 2
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color
```

Así pues, podemos invocar a la variable de clase directamente:

```
>>> Moto.n_ruedas
2
```

Además, una instancia también puede hacer uso de la variable de clase:

```
>>> m = Moto()
>>> m.n_ruedas
2
```

En realidad, estas variables de clase de Python son parecidas a las declaradas en Java o C++ a través de la palabra clave *static.* Sin embargo, y a diferencia de estos lenguajes, las variables de clase pueden ser modificadas. Esto se debe a cómo Python trata la visibilidad de componentes, aspecto del que nos ocuparemos próximamente.

Propiedades

Las variables de instancia, también llamados atributos, definen una serie de características que poseen los objetos. Como hemos visto anteriormente, se declaran haciendo uso de la referencia a la instancia a través de *self.* Sin embargo, Python nos ofrece la posibilidad de utilizar un método alternativo que resulta especialmente útil cuando estos atributos requieren de un procesamiento inicial en el momento de ser accedidos. Para implementar este mecanismo, Python emplea un decorador llamado *property.* Este decorador o *decorator* es, básicamente, un modificador que permite ejecutar una versión modificada o decorada de una función o método concreto. En el siguiente capítulo entraremos en detalle en la explicación de su funcionamiento. De momento y para entender cómo funciona el decorador *property,* nos bastará con la descripción recién

realizada.

Supongamos que tenemos una clase que representa un círculo y deseamos tener un atributo que se refiera al área del mismo. Dado que esta área puede ser calculada en función del radio del círculo, parece lógico utilizar *property* para llevar a cabo el procesamiento requerido. Teniendo esto en cuenta, nuestra clase quedaría de la siguiente forma:

```
from math import pi
class Circle:
    def __init__(self, radio):
        self.radio = radio
    @property
    def area(self):
        return pi * (self.radio ** 2)
```

Dada la anterior definición, podemos acceder directamente a *área*, tal y como lo haríamos con cualquier variable de instancia. El siguiente ejemplo muestra cómo hacerlo:

```
>>> c = Circle(25)
>>> print(c.area)
1661.9025137490005
```

Alternativamente, en lugar de emplear el decorator, también es posible definir un método en la clase y luego emplear la función *property()* para convertirlo en un atributo. Si optamos por este mecanismo, el código equivalente al método decorado sería el siguiente:

```
def area (self) :
    return pi * (self.radio ** 2)
area = property(area)
```

Ya que la declaración y uso del decorator es más sencilla y fácil de leer, recomendamos su utilización en detrimento de la mencionada función *property()*.

Seguro que los programadores de Java están habituados a utilizar métodos *setters* y *getters* para acceder a atributos de clase. En Java y otros lenguajes similares, se emplea una técnica diferente a la que se usa en Python. Normalmente, sobre todo en Java, los atributos de instancia se declaran con el modificador de visibilidad *private* y se emplea un método para acceder al atributo y otro para modificarlo. A los del primer tipo se les llama *getter* y a los

del segundo *setter*. Habitualmente, se emplea la nomenclatura *get<Nombre_atributo>* y *set<Nombre_atributo>*. Por ejemplo, tendríamos un método *getRadio()* y otro llamado *setRadio()*.

Nuestro decorador en Python, de momento, solo nos sirve para acceder al atributo en cuestión, pero no para modificarlo. De hecho, al ejecutar la siguiente sentencia, obtendremos un error:

```
>>> c.area = 23
Traceback (most recent cali last) :
    File "Dropbox\libro_python\code\circle.py", line 17, in <module>
     print(c.area)
    File "Dropbox\libro_python\code\circle.py", line 10, in area
    return self.__area
AttributeError: 'Circle' object has no attribute '_Circle area'
```

Para poder llevar a cabo esta acción, Python permite utilizar otros métodos disponibles a través del decorado *property*. En nuestro ejemplo estamos calculando el área y devolviendo su valor directamente a través del decorador, pero, para ilustrar el uso de *setters* y *getters* en Python, nos centraremos en el atributo *radio* de nuestra clase *Circulo*. Haremos esto, simplemente porque tiene más sentido hacerlo con el radio que no es un campo calculable. Volviendo a nuestro objetivo, para modificar el atributo radio haremos una serie de cambios en nuestra clase original. En primer lugar eliminaremos el constructor de la misma. Después, añadiremos el siguiente método, donde, hemos creado un atributo privado empleando el doble guión bajo (__:

```
@property
def radius(self) :
    return self.__radio
```

Seguidamente, escribiremos el método que se encargará de la modificación de nuestro nuevo atributo de clase. El código es este:

```
@radio.setter
def radio(self, radio):
    self. radio = radio
```

Ahora veremos cómo emplear nuestra clase con estos cambios, para ello simplemente instanciaremos un nuevo objeto y modificaremos su valor:

```
>>> circulo = Circulo()
>>> circulo.radio = 23
```

```
>>> print(circulo.radio)
23
```

El lector se habrá dado cuenta de que hemos utilizado el concepto de atributo privado, lo que implica que estamos trabajando con un modificador de visibilidad. Para entender cómo realmente funciona la misma en Python, pasaremos al siguiente apartado.

Visibilidad

Uno de los aspectos principales en los que difiere la orientación a objetos de Python con respecto a otros lenguajes, como Java y PHP5, es en el concepto de visibilidad de componentes de una clase. En Python no contamos con modificadores como *public* o *private* y simplemente, todos los métodos y atributos pueden considerarse como *public*. Es decir, realmente no se puede impedir el acceso a un objeto o atributo desde la instancia de una clase concreta. Pero ¿es posible de alguna forma indicar que un atributo o método solo debe ser invocado desde la misma clase? Esto correspondería a utilizar el modificador *private* en lenguajes como PHP5 y Java. La respuesta a la pregunta es sí, pero debemos tener en cuenta que hablamos de debe y no de puede. De esta forma, los programadores de Python utilizan una sencilla convención: Si un atributo debe ser privado, basta con anteponer un único guión bajo (underscore). Sin embargo, para Python, esto no significa nada, simplemente es una convención entre programadores.

No perdamos de vista que lo comentado sobre la visibilidad es aplicable, tanto a variables como a métodos. Así pues, si utilizamos el underscore para declarar un método como privado, veremos cómo eso no impide su acceso. Trabajemos con un sencillo ejemplo, declarando una clase y su correspondiente método:

```
class Test:
    def _privado(self) :
        print("Método privado")
```

Ahora pasemos a crear una instancia y observaremos cómo la llamada al método en cuestión es válida:

```
>>> t = Test()
>>> t._privado
Método privado
```

Por otro lado, en el apartado anterior hemos visto cómo el doble guión bajo (__) ha sido empleado para declarar un atributo privado de instancia. Realmente, si el atributo ha sido declarado de esta forma, Python previene el acceso desde la instancia. Observemos la siguiente clase que declara un atributo de esta forma y comprobemos cómo Python no nos deja acceder al mismo:

```
class Privado:
    def __ini__(self):
        self.__atributo = 1
>>> p = Privado
>>> p.__atributo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Privado' object has no attribute '__atributo'
```

Sin embargo, para acceder a este atributo y dado que Python en realidad no entiende el concepto de visibilidad como tal, el lenguaje emplea un mecanismo técnicamente denominado *name mangling*. Este consiste en convertir cualquier atributo declarado con doble guión en simple guión seguido del nombre de la clase y luego doble guión bajo para acceso directo al atributo. Así pues y continuando con nuestra clase *Privado*, la forma de acceder a nuestro atributo sería la siguiente:

```
>>> p_Privado__atributo
12
```

Algunos prefieren no emplear la denominación privado para referirse a este tipo de atributos. En su lugar, prefieren llamarlos *pseudoprivados.*

La técnica del *name mangling* se diseñó en Python para asegurar que una clase *hija* no sobrescribe accidentalmente los métodos y/o atributos de su clase *padre.* Esta es la verdadera razón de su existencia, no la de poder utilizar un modificador de visibilidad. Acabamos de introducir el concepto de *hija* y *padre* para referirnos a dos tipos de clases diferentes. Estos conceptos se engloban dentro del ámbito de la *herencia*, de la que nos ocuparemos detenidamente en apartados posteriores. El código que viene a continuación ilustra cómo evitar la mencionada sobrescritura del atributo *test:*

```
class Padre:
    def __init__(self):
        self.__test = "Padre"
class Hijo:
    def __init__(self):
        self.__test = "Hijo"
```

Sin aplicar la técnica del *name mangling,* en el ejemplo anterior, el atributo *test* de la clase padre sería sobrescrito por el de la clase hija y otra clase que heredara de la padre se vería afectada.

Métodos de clase

Ya hemos aprendido sobre atributos y métodos de instancia. Pero Python también permite crear *métodos de clase.* Estos se caracterizan porque pueden ser invocados directamente sobre la clase, sin necesidad de crear ninguna instancia.

Dado que no vamos a utilizar un objeto, no es necesario emplear el argumento *self* como referencia en los métodos de clase. En su lugar, sí que debemos contar con otro parámetro similar que referenciará a la clase en cuestión. Por convención, al igual que se emplea *self* para los métodos de instancia, cls es el nombre que suele ser empleado como referencia en los métodos de clase. Además, esto implica que el programador no tiene que pasar como parámetro la mencionada referencia, ya que Python lo hace automáticamente. Eso sí, es responsabilidad del programador utilizar el parámetro formal, para dicha referencia, en la definición del método. Esto es válido tanto para *self* como para cls.

Por otro lado y al igual que en el caso de los métodos de instancia, se pueden utilizar parámetros adicionales para el método de clase. Lo que sí que es importante que nunca olvidemos la referencia cls en su definición. El resto de parámetros son opcionales.

Los métodos de clase requieren el uso de un decorador definido por Python, su nombre es *classmethod* y funciona de forma similar a como lo hace el comentado *property.* Gracias al *decorator,* solo debemos definir el método con el argumento referencia cls y escribir su funcionalidad. Sirva como ejemplo el siguiente código:

```
class Test:
```

```
    def __init__(self):
        self.x = 8
    @classmethod
    def metodo_clase(cls, param1):
        print("Parámetro: {0}".format(param1))
```

Para poner en práctica el código anterior, primero invocaremos directamente al método de clase:

```
>>> Test.metodo_clase(6)
Parámetro: 6
```

También es fácil comprobar que el método de instancia puede ser invocado, creando previamente un objeto de la clase en cuestión:

```
>>> t = Test ()
>>> print(t.x)
8
```

Es interesante saber que los métodos de clase también pueden ser invocados a través de una instancia de la misma. Es tan sencillo como invocarlo como si de un método de instancia se tratase. Siguiendo con el ejemplo anterior, la siguiente sentencia es válida y produce el resultado que podemos apreciar:

```
>>> t.metodo_clase(5)
Párametro: 5
```

Lógicamente, si un método de clase puede ser invocado desde una instancia, también podemos crear un objeto e invocar directamente al método en cuestión en la misma línea de código:

```
>>> Test().metodo_clase(3)
Párametro: 3
```

El comportamiento anteriormente explicado tiene sentido, ya que, en realidad, Python no tiene modificadores de visibilidad como tales. Es por ello, que un método de clase es también un método de instancia. La diferencia es que un método de instancia no puede ser creado si esta no existe.

Lenguajes como Java y C++ emplean la palabra clave *static* para declarar métodos de clase. Además, en estos lenguajes, no es posible utilizar la referencia a la instancia, llamada en ambos casos *this*. Sin embargo, estos no pueden ser invocados desde una instancia, a diferencia de Python.

Otros lenguajes como Ruby y Smalltalk no tienen métodos *estáticos*, pero sí de clase. Esto implica que estos métodos sí que tienen acceso a los datos de la instancia. Python cuenta tanto con métodos de clase como con métodos estáticos. De estos últimos nos ocuparemos a continuación.

Métodos estáticos

Otro de los tipos de métodos que puede contener una clase en Python son los llamados *estáticos.* La principal diferencia con respecto a los métodos de clase es que los estáticos no necesitan ningún argumento como referencia, ni a la instancia, ni a la clase. Lógicamente, al no tener esta referencia, un método estático no puede acceder a ningún atributo de la clase.

De la misma forma que los métodos de clase en Python usan el decorador *classmethod*, para los estáticos contamos con otro decorador llamado *staticmethod.* La definición de un método estático requiere del uso de este decorador, que debe aparecer antes de la definición del mismo.

Para comprobar cómo funcionan los métodos estáticos, añadamos el siguiente método a nuestra clase *Test:*

```
@staticmethod
def metodo_estatico(valor):
    print("Valor: {0}".format(valor))
```

Seguidamente, invoquémoslo directamente desde una instancia determinada de la clase:

```
>>> t = Test()
>>> t.metodo_ estático (33)
Valor: 33
```

Echando un vistazo al código anterior, podemos apreciar que la diferencia de un método estático y otro de instancia es, además del uso del decorado en cuestión, que como primer parámetro no estamos usando *self* para referenciar a la instancia. Pero este hecho tiene otra implicación, tal y como hemos comentado previamente. Se trata de que no es posible acceder desde el mismo a un atributo de clase. Así pues si cambiamos el código del método anterior por el siguiente, se produciría una excepción en tiempo de ejecución:

```
@staticmethod
def metodo_estatico(valor):
    print(self.x)
```

El error en cuestión, producido por la ejecución del método estático desde una instancia, sería el siguiente:

```
NameError: global name self is not defined
```

Es lógico que se produzca el error, no olvidemos que *self* es solo una convención para referenciar a la instancia de clase. Dado que el método es estático y no existe tal referencia en el mismo, no es posible utilizar ningún atributo de instancia en su interior. *NameError* es una excepción predefinida por Python y que es lanzada como consecuencia de intentar acceder al atributo.

Algunos programadores prefieren crear una función en lugar de un método estático. Se puede llamar a la función y utilizar su resultado interactuando posteriormente con la instancia de la clase. Sin embargo, otros prefieren emplear los métodos estáticos, argumentando que, si la funcionalidad está estrechamente relacionada con la clase, se mantiene y cumple el *principio de abstracción* (ver referencias), comúnmente utilizado en la programación orientada a objetos.

Métodos especiales

Python pone a nuestra disposición una serie de métodos *especiales* que, por defecto, contendrán todas las clases que creemos. En realidad, cualquier clase que creemos será hija de una clase especial integrada llamada *object.* De esta forma, dado que esta clase cuenta por defecto con estos métodos especiales, ambos estarán disponibles en nuestros nuevos objetos. Es más, podemos sobrescribir estos métodos reemplazando su funcionalidad.

Cuando creamos una nueva clase esta hereda directamente de *object,* por lo que no es necesario indicar esto explícitamente al instanciar un objeto de una clase concreta.

Los métodos especiales se caracterizan principalmente porque utilizan dos caracteres *underscore* al principio y al final del nombre del método.

CREACIÓN E INICIALIZACIÓN

El principal de estos métodos especiales lo hemos presentado en apartados anteriores, se trata de *__init__()*. Tal y como hemos comentado previamente, este todo será el encargado de realizar las tareas de inicialización cuando la instancia de una clase determinada es creada. Necesita utilizar como parámetro formal una referencia *(self)* a la instancia y, adicionalmente, pueden contener otros parámetros.

Por otro lado, y relacionado con *__init__()*, encontramos a *__new__()* . En muchos lenguajes la operación de creación e inicialización de instancia es atómica; sin embargo, en Python esto es diferente. Primero se invoca a *__new__()* para crear la instancia propiamente dicha y posteriormente se llama a *__init__()* para llevar a cabo las operaciones que sean requeridas para inicializar la misma. En la práctica, suele utilizarse *__init__()* como constructor de instancia, de la misma forma que en Java, por ejemplo, se emplea un método que tiene el mismo nombre que la clase que lo contiene. Sin embargo, gracias a este mecanismo de Python, podemos tener más control sobre las operaciones de creación e inicialización de objetos. Dado que cualquier clase los contendrá al heredar de *object*, estos métodos podrán ser sobrescritos con la funcionalidad que deseemos.

A diferencia de *__init__()*, *__new__()* no requiere de *self*, en su lugar recibe una referencia a la clase que lo está invocando. Realmente, es este un método estático que siempre devuelve un objeto.

El método *__new__()* fue diseñado para permitir la personalización en la creación de instancias de clases que heredan de aquellas que son inmutables. Recordemos que para Python algunos tipos integrados son inmutables, como, por ejemplo, los *strings* y las *tuplas.*

Es importante tener en cuenta que el método *__new__()* solo llamará automáticamente a *__init__()* si el primero devuelve una instancia. Es práctica habitual realizar ciertas funciones de personalización dentro de *__new__* y luego llamar al método del mismo nombre de la clase padre. De esta forma, nos aseguraremos que nuestro *__init__()* será ejecutado.

Con el objetivo de comprender cómo realmente funcionan los métodos de creación de instancias e inicialización de las mismas, vamos a crear una clase que contenga ambos métodos:

```
class Ini :
    def __new__(cls):
```

```
        print("new")
        return super(Test, cls).__new__(cls)
    def __init__(self):
        print("init")
```

Ahora crearemos una instancia y observaremos el resultado producido que nos indicará el orden en el que ambos métodos han sido ejecutados:

```
>>> obj = Ini()
new
init
```

¿Qué pasaría si el método *__new__()* de la clase anterior no devolviera una instancia? La respuesta es sencilla: nuestro método *__init__()* nunca sería ejecutado. Realizar esta prueba es bastante sencillo, basta con borrar la última línea, la que contiene la sentencia *return* del método *__new__*. Al volver a crear la instancia, comprobaremos cómo solo obtenemos como salida la cadena de texto "init".

No olvidemos que cuando trabajamos con *__new__()* y este recibe parámetros adicionales, estos mismos deben constar como parámetros formales en el método *__init__()*. Sirva como ejemplo el siguiente código:

```
def __new__(cls, x):
    return super(Test, cls).__new__(cls)
def __init__(self, x):
    self.x = x
```

La invocación a la creación de un objeto de la clase que contendrá los módulos anteriores podría ser de la siguiente forma:

```
>>> t = Test("param")
```

DESTRUCTOR

Al igual que otros lenguajes de programación, Python cuenta con un método especial que puede ejecutar acciones cuando el objeto va a ser destruido. A diferencia de lenguajes como C++, Python incorpora un recolector de basura *(garbage collector)* que se encarga de llamar a este destructor y liberar memoria cuando el objeto no es necesario. Este proceso es transparente y no es necesario invocar al destructor para liberar memoria. Sin embargo, este método destructor

puede realizar acciones adicionales si lo sobrescribimos en nuestras clases. Su nombre es *__del__()* y, habitualmente, nunca se le llama explícitamente.

Debemos tener en cuenta que la función integrada *del()* no llama directamente al destructor de un objeto, sino que esta sirve para decrementar el contador de referencias al objeto. Por otro lado, el método especial *__del__()* es invocado cuando el contador de referencias llega a cero.

Volviendo a nuestra clase ejemplo anterior *(Ini)* podemos añadirle el siguiente método, para posteriormente crear una nueva instancia y observar cómo el destructor es llamado automáticamente por el intérprete. A continuación, el código del método en cuestión:

```
def __del__(self):
    print("del")
```

En lugar de utilizar directamente el intérprete a través de la consola de comandos, vamos a crear un fichero que contenga nuestra clase completa, incluyendo el destructor. Seguidamente invocaremos al intérprete de Python para que ejecute nuestro *script* y comprobaremos cómo la salida nos muestra la cadena *del* como consecuencia de la ejecución del destructor por parte del recolector de basura. Al terminar la ejecución del *script* y no ser necesario el objeto, el intérprete invoca al recolector de basura, que, a su vez, llama directamente al constructor. Sin embargo, esta situación varía si empleamos la consola de comandos y creamos la instancia desde la misma, ya que el destructor no será ejecutado inmediatamente. La explicación es trivial: el intérprete no invocará al recolector hasta que no sea necesario, si lo hiciera antes de tiempo, perderíamos la referencia a nuestro objeto. Cuando ejecutamos el *script*, el intérprete detectada que se ha producido la finalización del mismo, que ya no es necesaria la memoria, puesto que el objeto no va a ser utilizado y que puede proceder a la llamada al recolector de basura.

REPRESENTACIÓN Y FORMATOS

En algunas ocasiones puede ser útil obtener una representación de un objeto, que nos sirva, por ejemplo, para volver a crear un objeto con los mismos valores que el original. La representación de un objeto puede obtenerse a través de la función integrada *repr()*. Por ejemplo, si declaramos un *string* con un valor

determinado e invocamos a la mencionada función, conseguiremos el valor de la cadena. De hecho, en el caso de un *string* este es el único valor que necesitamos para recrear el objeto. Sin embargo, la situación cambia cuando creamos nuestras propias clases. En este caso, deberemos emplear el método especial *__repr__()* , el cual será invocado directamente cuando se llama a la función *repr().* Pensemos en una clase que representa a un coche, donde sus atributos principales son la marca y el modelo. Dado que estos son los únicos valores que necesitamos para recrear el objeto, serán los que utilizaremos en nuestro método *__repr__()*, tal y como muestra el siguiente ejemplo:

```
class Coche:
    def __init__(marca="Porsche", modelo="911")
        self.marca = marca
        self.modelo = modelo
    def __repr__(self):
        return("{0}-{1}".format(self.marca, self.modelo))
```

Al crear un objeto e invocar a la mencionada función de representación, obtendremos el siguiente resultado:

```
>>> c = Coche()
>>> print(repr(c))
```

Con la información devuelta por la función *repr* podríamos crear un nuevo objeto con los mismos valores que el original:

```
>>> arr = repr(c).split("-")
>>> d = Coche(arr[0], arr[1])
```

El método especial *__repr__()* siempre debe devolver un *string*, en caso contrario se lanzará una excepción de tipo *TypeError.* Habitualmente, esta representación de un objeto se emplea para poder depurar código y encontrar posibles errores.

Relacionado con *__repr__()* encontramos otro método especial denominado *__str__()*, el cual es invocado cuando se llama a las funciones integradas *str()* y *print(),* pasando como argumento un objeto. El método *__str__()* puede ser considerado también una representación del objeto en cuestión, la diferencia con *__repr__()* radica en que el primero debe devolver una cadena de texto que nos sirva para identificar y representar al objeto de una forma sencilla y concisa. De esta forma, la información que devuelve suele ser una línea de texto descriptiva

o una cadena similar. Al igual que *__repr__()*, el método *__str__* debe devolver siempre un *string*. Para nuestra clase de ejemplo *Coche*, bastará con añadir el siguiente código:

```
def __str__(self):
    return("{0} -> {1}".format(self.marca, self.modelo)
```

Si llamamos a la función *print()*, apreciaremos el resultado:

```
>>> print(d)
"Porsche->911"
```

En nuestros ejemplos para la clase *Coche* no existe mucha diferencia entre las cadenas que devuelven ambos métodos de representación; sin embargo, si la clase es muy compleja, sí que es más fácil establecer diferencias entre los resultados devueltos. No obstante, esto siempre queda a criterio del programador.

Similar a *__str__* también existe el método especial *__bytes()__* que devuelve también una representación del objeto utilizando el tipo predefinido *bytes*. Este método será ejecutado cuando llamamos a la función integrada *bytes()*, pasando como argumento un objeto.

Por último, *__format__()* es otro método especial utilizado para obtener una representación en un formato determinado de un objeto. Es decir, podemos indicar, valores como, por ejemplo, la alineación de la cadena de texto producida. Esto nos ayudará a crear una cadena de texto convenientemente formateada. La función integrada *format()* es la responsable de llamar a este método especial, y al igual que los otros métodos de representación, requiere pasar la instancia de la clase en cuestión.

COMPARACIONES

Comparar tipos sencillos es fácil. Por ejemplo, pensemos en dos números. Establecer si uno es mayor que otro es trivial. Igual ocurre para otras operaciones similares como son la igualdad, la comprobación de si es igual o mayor que y la desigualdad. Sin embargo, este hecho cambia cuando debemos aplicar estas operaciones a objetos de nuestras propias clases. Ello se debe a que el intérprete, a priori, no sabe cómo realizar estas operaciones. Para solventar

este problema, Python cuenta con una serie de métodos especiales que nos ayudarán a escribir nuestra propia lógica aplicable a cada operación de comparación concreta. Bastará con implementar el método que necesitemos sobrescribiendo el original ofrecido por el lenguaje para esta situación. Concretamente, contamos con los siguientes métodos especiales de comparación:

- *__It__()*: Menor que.
- *__le__()*: Menor o igual que.
- *__gt__()*: Mayor que.
- *__ge__()*: Mayor o igual que.
- *__eq__()*: Igual a.
- *__ne__():* Distinto de.

Cada uno de estos métodos será invocado en función del operador equivalente que empleemos en la comparación. Supongamos que deseamos saber qué modelo de coche es más moderno. Por simplicidad, nos basaremos en el atributo que representa la marca y tendremos en cuenta que este solo puede ser un número. Cuanto mayor es el número, más moderno será el coche en cuestión. En base a esta simple afirmación, escribiremos el código del método que se encargará de realizar la comprobación:

```
def __gt__(self, objeto):
    if int(self.modelo) > int(objeto.modelo):
        return True
    return False
```

Si lo añadimos a nuestra clase *Coche* y creamos dos instancias, podremos emplear el operador > para llevar a cabo nuestra prueba:

```
>>> modelo_911 = Coche("Porsche", 911)
>>> modelo_924 = Coche("Porsche", 924)
>>> if modelo_924 > modelo_911:
...   print("El {0} es un modelo
superior".format(modelo_924.modelo))
...
El modelo 924 es un modelo superior
```

De forma análoga podemos emplear el resto de métodos especiales de

comparación. Es importante tener en mente que estos métodos deben devolver *True o False.* Sin embargo, esto es solo una convención, ya que, en realidad, pueden devolver cualquier valor.

HASH Y BOOL

Los dos últimos métodos especiales que nos quedan por describir son *__hash__()* y_ *__bool__()*. El primero de ellos se ejecuta al invocar a la función integrada *hash()* y debe devolver un número entero que sirve para identificar de forma unívoca a cada instancia de la misma clase. De esta forma, es fácil comparar si dos objetos son el mismo, ya que deben tener el mismo valor devuelto por *hash().* A través del método especial *__hash__(),* podemos realizar operaciones que nos sean necesarias y devolver después el correspondiente valor. Por defecto todos los objetos de cualquier clase cuentan con los métodos *__eq__()* y *__hash__()* y siempre dos instancias serán diferentes a no ser que se comparen consigo mismas. A continuación, veamos un ejemplo de invocación al método *__hash__()* desde dos instancias diferentes:

```
>>> class TestHash:
...   pass
...
>>> t = TestHash()
>>> t.__hash__()
39175112
>>> hash(t)
39175112
>>> x = TestHash()
>>> x__hash__()
2448448
>>> hash(x)
2448448
>>> x == x
True
>>> t == x
False
```

Por otro lado, la función *bool()* será la encargada de llamar al método especial *__bool__* cuando esta recibe como argumento la instancia de una clase determinada. Por defecto, si no implementamos el mencionado método en nuestra clase, la llamada a *bool()* siempre devolverá *True.* Si implementamos

__bool__() en nuestra clase, este debe devolver un valor de tipo *booleano*, es decir, en la práctica, solo podremos devolver *True o False*. En el siguiente ejemplo, sobrescribiremos el método especial para que siempre devuelva *False*, cambiando así el resultado que se obtiene por defecto cuando este método no está implementado:

```
>>> class TestBool:
...   def __bool__():
...      return False
...
>>> t = TestBool()
>>> t.__bool__()
False
>>> bool(t)
False
>>> if t: print("Es falso")
...
Es falso
```

HERENCIA

El concepto de *herencia* es uno de los más importantes en la programación orientada a objetos. En apartados anteriores de este capítulo hemos adelantado la base en la que se fundamente este concepto. En términos generales, se trata de establecer una relación entre dos tipos de clases donde las instancias de una de ellas tengan directamente acceso a los atributos y métodos declarados en la otra. Para ello, debemos contar con una principal que contendrá las declaraciones e implementaciones. A esta la llamaremos *padre, superclase o principal*. La otra, será la clase *hija o secundaria*.

La herencia presenta la clara ventaja de la reutilización de código, además nos permite establecer relaciones y escribir menos líneas de código, ya que no es necesario, por ejemplo, volver a declarar e implementar métodos.

Python implementa la herencia basándose en los espacios de nombres, de tal forma que, cuando una instancia de una clase hija hace uso de un método o atributo, el intérprete busca primero en la definición de la misma y si no encuentra correspondencias accede al espacio de nombres de la clase padre. Técnicamente, Python construye un árbol en memoria que le permite localizar correspondencias entre los diferentes espacios de nombres de clases que hacen uso de la herencia.

Habitualmente, los modificadores de visibilidad como *public*, *prívate o protected*, empleados por lenguajes como Java y C++, están directamente relacionados con la herencia. Por ejemplo, un método privado es heredable, pero solo invocable a través de una instancia de la clase que lo implementa. En Python, tal y como hemos comentado anteriormente, los modificadores de visibilidad, como tal, no existen, ya que cualquier atributo o método es accesible.

Lenguajes como C++, Java y PHP5 soportan la herencia y establecen diferentes sintaxis para declararla. Python no es una excepción y también permite el uso de esta técnica. Además, a diferencia, por ejemplo de Java, Python soporta dos tipos de herencia: la simple y la múltiple. De ambos nos ocuparemos en los siguientes apartados.

Simple

La herencia simple consiste en que una clase hereda únicamente de otra. Como hemos comentado previamente, la relación de herencia hace posible utilizar, desde la instancia, los atributos de la clase padre. En Python, al definir una clase, indicaremos entre paréntesis de la clase que hereda. Comenzaremos definiendo nuestra clase padre:

```
class Padre:
    def __init__(self):
        self.x = 8
        print("Constructor clase padre")
    def metodo(self):
        print("Ejecutando método de clase padre")
```

Crear una clase que herede de la que acabamos de definir es bien sencillo:

```
class Hija:
    def met_hija(self):
        print("Método clase hija")
```

Ahora procederemos a crear una instancia de la clase hija y a comprobar cómo es posible invocar al método definido en su padre:

```
>>> h = Hija()
Constructor clase padre
>>> h.método()
Ejecutando método clase padre
```

Seguro que el lector se ha dado cuenta de que el método *__init__()* de clase padre ha sido invocado directamente al invocar la instancia de la clase hija. Esto se debe a que el método constructor es el primero en ser invocado al crear la instancia y cómo este existe en la clase padre, entonces se ejecuta directamente. Pero ¿qué ocurre si creamos un método constructor en la clase hija? Sencillamente, este será invocado en lugar de llamar al de padre. Es decir, habremos sobrescrito el constructor original. De hecho, esto puede ser muy útil cuando necesitamos nuestro propio constructor en lugar de llamar al de padre. Si modificamos nuestra clase *Hija*, añadimos el siguiente método y volvemos a crear una instancia, podremos apreciar el resultado:

```
def __init__(self):
    print("Constructor hija")
>> z = Hija()
Constructor hija
```

Pero no solo los métodos son heredables, también lo son los atributos. Así pues, la siguiente sentencia es válida:

```
>>> h.x
7
```

Obviamente, varias clases pueden heredar de otra en común, es decir, una clase *padre* puede tener varias *hijas.* En la práctica, podríamos definir un clase *Vehículo,* que será la padre y otras dos hijas, llamadas *Coche* y *Moto.* De hecho, la relación que vamos a establecer entre ellas será de *especialización,* ya que un coche y una moto son un tipo de vehículo determinado. A continuación, mostramos el código necesario para ello:

```
class Vehiculo:
    n_ruedas = 2
    def __init__(self, marca, modelo):
        self.marca = marca
    self.modelo = modelo
    def acelerar(self):
        pass
    def frenar(self):
        pass
class Moto(Vehiculo):
    pass
class Coche(Vehiculo):
    pass
```

En este punto podemos crear dos instancias de las dos clases hija y modificar el atributo de clase:

```
>>> c = Coche("Porsche", "944")
>>> c.n_ruedas = 4
>>> m = Moto("Honda", "Goldwin")
>>> m.n_ruedas
2
```

Como las motos y los coches tienen en común las operaciones de aceleración y frenado, parece lógico que definamos métodos, en la clase padre, para representar dichas operaciones. Por otro lado, cualquier clase que herede de *Vehiculo,* también contendrá estas operaciones, con independencia del número de ruedas que tenga.

Múltiple

Como hemos comentado previamente, Python soporta la herencia múltiple, del mismo modo que C++. Otros lenguajes como Java y Ruby no la soportan, pero sí que implementan técnicas para conseguir la misma funcionalidad. En el caso de Java, contamos con las *clases abstractas* y las *interfaces*, y en Ruby tenemos los *mixins.*

La herencia múltiple es similar en comportamiento a la sencilla, con la diferencia que una clase hija tiene uno o más clases padre. En Python, basta con separar con comas los nombres de las clases en la definición de la misma. Pensemos en un ejemplo de la vida real para implementar la herencia múltiple. Por ejemplo, una clase genérica sería *Persona*, otra *Personal* y la hija sería *Mantemiento.* De esta forma, una persona que trabajara en una empresa determinada como personal de mantenimiento, podría representarse a través de una clase de la siguiente forma:

```
class Mantenimiento(Persona, Personal):
    pass
```

De esta forma, desde la clase *Mantenimiento*, tendríamos acceso a todos los atributos y métodos declarados, tanto en *Persona*, como en *Personal.*

La herencia múltiple presenta el conocido como *problema del diamante*. Este problema surge cuando dos clases heredan de otra tercera y, además una cuarta clase tiene como padre a las dos últimas. La primera clase padre es llamada *A* y las clases *B* y C heredan de ella, a su vez la clase *D* tiene como padres a *B* y *C*. En esta situación, si una instancia de la clase *D* llama a un método definido en *A*, ¿lo heredará desde la clase *B* o desde la clase *C*?. Cada lenguaje de programación utiliza un algoritmo para tomar esta decisión. En el caso de Python, se toma como referencia que todas las clases descienden de la clase padre *object.* Además, se crea una lista de clases que se buscan de derecha a Izquierda y de abajo arriba, posteriormente se eliminan todas las apariciones de una clase repetida menos la última. De esta forma queda establecido un orden.

La figura geométrica que representa la relación entre clases definida anteriormente tiene forma de diamante, de ahí su nombre. Podemos apreciar esta relación en la figura.

Teniendo en cuenta esta figura, Python crearía la lista de clases *D, B, A, C, A.* Después eliminaría la primera *A*, quedan el orden establecido en *D, B, C, A.*

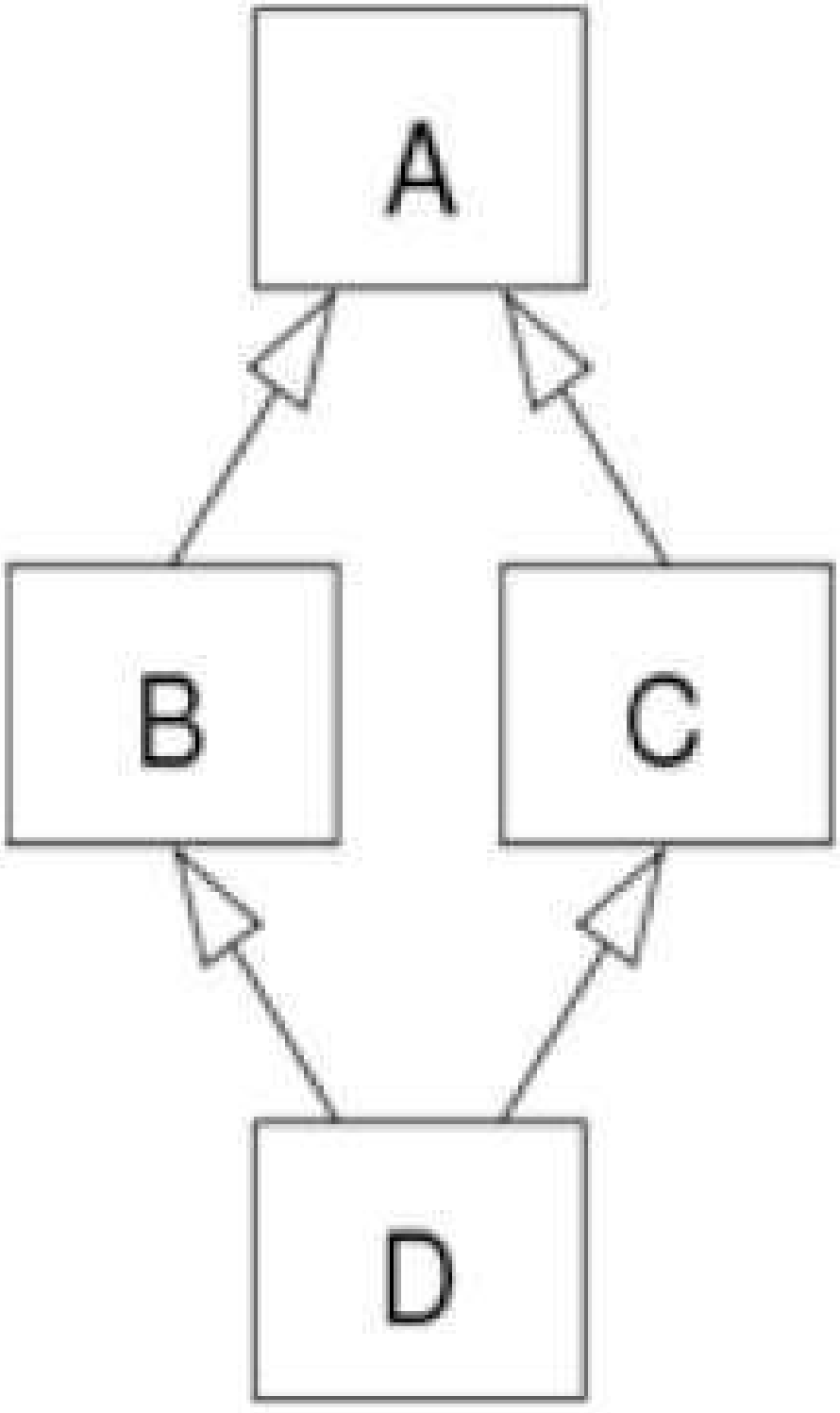


Fig. 4-1 El problema del diamante

Dadas las ambigüedades que pueden surgir de la utilización de la herencia múltiple, son muchos los programadores que deciden emplearla lo mínimo posible, ya que, dependiendo de la complejidad del diagrama de herencia, puede ser muy complicado establecer su orden y se pueden producir errores no deseados en tiempo de ejecución. Por otro lado, si mantenemos una relación sencilla, la herencia múltiple es un útil aliado a la hora de representar objetos y situaciones de la vida real.

POLIMORFISMO

En el ámbito de la orientación a objetos el *polimorfismo* hace referencia a la habilidad que tienen los objetos de diferentes clases a responder a métodos con el mismo nombre, pero con implementaciones diferentes. Si nos fijamos en el mundo real, esta situación suele darse con frecuencia. Por ejemplo, pensemos en una serpiente y en un pájaro. Obviamente, ambos pueden desplazarse, pero la manera en la que lo hacen es distinta. Al modelar esta situación, definiremos dos clases que contienen el método *desplazar,* siendo su implementación diferente en ambos casos:

```
class Pajaro:
    def desplazar(self):
        print("Volar")

class Serpiente:
    def desplazar(self):
        print("Reptar")
```

A continuación, definiremos una función adicional que se encargue de recibir como argumento la instancia de una de las dos clases y de invocar al método del mismo nombre. Dado que ambas clases contienen el mismo método, la llamada funcionará sin problema, pero el resultado será diferente:

```
>>> def mover(animal):
...      animal.desplazar()
>>> p = Pajaro()
>>> s = Serpiente()
>>> p.desplazar()
Volar
>>> s.desplazar()
Reptar
>>> mover(p)
Volar
>>> mover(s)
Reptar
```

Como hemos podido comprobar, en Python la utilización del polimorfismo es bien sencilla. Ello se debe a que no es un lenguaje fuertemente tipado. Otros lenguajes que sí lo son utilizan técnicas para permitir y facilitar el uso del

polimorfismo. Por ejemplo, Java cuenta con las *clases abstractas*, que permiten definir un método sin implementarlo. Otras clases pueden heredar de una clase abstracta e implementar el método en cuestión de forma diferente.

INTROSPECCIÓN

La *instrospección* o *inspección interna* es la habilidad que tiene un componente, como una instancia o un método, para examinar las propiedades de un objeto externo y tomar decisiones sobre las acciones que él mismo puede llevar a cabo, basándose en la información conseguida a través de la examinación.

Python posee diferentes herramientas para facilitar la introspección, lo que puede resultar muy útil para conocer qué acciones es capaz de realizar un objeto determinado. Una de las más populares de estas herramientas es la función integrada *dir()* que nos devuelve todos los métodos que pueden ser invocados para una instancia concreta. De esta forma, sin necesidad de invocar ningún método, tenemos información sobre un objeto. Por ejemplo, definamos una clase con un par de métodos y lancemos la función *dir()* sobre una instancia concreta:

```
>>> class Intro:
        def __init__(self, val):
            self.x = val
        def primero(self):
            print("Primero")
        def segundo(self):
            print("Segundo")
...
>>> i = Intro("Valor")
>>> dir(i)
['__class__', '__delattr__', __dict__', '__doc__', __eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'primero', 'segundo', 'x']
```

Como el lector habrá descubierto, la salida de la sentencia *dir(i)* devuelve una lista de *strings* con el nombre de cada uno de los métodos que pueden ser invocados. Los tres últimos son los métodos de instancia y el atributo que hemos creado, mientras que el resto son métodos heredados de la clase predefinida *object.*

Además de *dir(),* otras dos prácticas funciones de introspección son

isinstance() y *hasattr().* La primera de ellas nos sirve para comprobar si una variable es instancia de una clase. En caso afirmativo devolverá *True* y en otro caso *False.* Continuemos con el ejemplo de la clase anteriormente definida y probemos este método:

```
>>> isinstance(i, Intro)
True
```

Por otro lado, *hashattr()* devuelve *True* si una instancia contiene un atributo. Esta función recibe como primer parámetro la variable que representa la instancia y como segundo el atributo por el que deseamos preguntar. Dada nuestra clase anterior, escribiremos el siguiente código para comprobar su funcionamiento:

```
>>> if (i, 'x'): print("Instancia: i. Clase: Intro. Atributo: x")
Instancia: i. Clase: Intro. Atributo: x
>>> if (i, 'z'): print("Atributo z no existe")
Atributo z no existe
```

Si utilizamos la herencia, las funciones de introspección pueden sernos muy útiles, además nos permiten descubrir cómo los métodos y atributos son heredados. Si creamos una nueva clase que herede de la anterior, pero con un nuevo método, al llamar a *dir()* veremos cómo los atributos y métodos están disponibles directamente:

```
>>> class Hijo(Intro):
def tercero(self):
print("Tercero")
...
>>> h = Hijo()
>>> dir(h)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__, 'primero',
'segundo', 'tercero', 'x']
```

De igual forma, podemos emplear la función *hasattr()* para comprobar si el atributo *x* existe también en la clase *h:*

```
>>> if hasattr(h, 'x'): print("h puede acceder a x")
h puede acceder a x
```

Obviamente, aunque las clases *Intro* e *Hijo* estén relacionadas son diferentes. Sin embargo, la función *isinstance()* nos devolverá *True* si pasamos como primer argumento una instancia de cualquier de ellas y como segundo el nombre de una de las dos clases:

```
>>> if isinstance(h, Intro): print("h es instancia de Intro")
h es instancia de Intro
```

Entonces, ¿cómo podemos averiguar si una Instancia pertenece a la clase padre o hija? Fácil, basta con emplear la función Integrada *type()*, tal y como muestra el siguiente ejemplo:

```
>>> type(i)
<class '__main__.Intro'>
>>> type(h)
<class '__main__.Hija'>
```

Otro método relacionado con la introspección es *callable()*, el cual nos devuelve *True* si un objeto puede ser llamado y *False* en caso contrario. Por defecto, todas las funciones y métodos de objetos pueden ser llamados, pero no las variables. Gracias a este método podemos averiguar, por ejemplo, si una determinada variable es una función o no. Como ejemplo, echemos un vistazo al siguiente código:

```
>>> cad = "Cadena"
>>> callable(cad)
False
>>> def fun():
...   return("fun")
>>> callabe(fun)
True
```

Siguiendo la misma explicación, una instancia de objeto devolverá *False* cuando invocamos a *callable():*

```
>>> callable(i)
False
```

Echando un vistazo al resultado de la ejecución de la sentencia previa *dir(h)*, descubriremos métodos que pueden ser llamados a través de una instancia y que también sirven para la instrospeccl6n. Nos referimos a *__getattribute__()* y *__setattr__()*. El primero nos sirve para obtener el valor de un atributo a través

de su nombre, el segundo realiza la operación inversa, es decir, recibe como parámetro el nombre del atributo y el valor que va a serle fijado. Habitualmente, estos métodos no son necesarios, ya que podemos acceder directamente a un atributo de instancia directamente usando el nombre de la variable de instancia, seguido de un punto y del nombre del atributo en cuestión. Sin embargo, los métodos *__getattribute__()* y *__setattr__()* pueden ser muy útiles, cuando, por ejemplo, necesitamos leer los atributos de una lista o diccionario. Supongamos que tenemos una clase con tres atributos de instancia definidos y deseamos fijar su valor empleando para ello un diccionario. Comenzaremos definiendo una nueva clase:

```
class Test:
    def__init__(self):
        self.x = 3
        self.y = 4
        self.z = 5
```

Ahora crearemos un diccionario y una instancia de clase:

```
>>> attrs = {"x": 1, "y": 2, "z": 3}
>>> t = Test()
```

Seguidamente, pasaremos a asignar los valores del diccionario a nuestros atributos de clase:

```
>>> for k, v in attrs.items():
...   t.__setattr__(k, v)
```

Para comprobar que los atributos han sido fijados correctamente, emplearemos la función *__getattribute__()*, tal y como muestra el siguiente código:

```
>>> for k in attrs.keys():
...   print("{0}={1}".format(k, t.__getattribute__(k))
...
y = 2
x = 1
z = 3
```

Debemos tener en cuenta que *attrs* es un diccionario y como tal es una estructura de datos no ordenada, por lo cual la salida del orden de los atributos puede diferir durante sucesivas ejecuciones de la iteración.

PROGRAMACIÓN AVANZADA

INTRODUCCIÓN

En este capítulo nos adentraremos en una serie de aspectos avanzados que Python pone a nuestra disposición. Los conceptos explicados en este capítulo, aun no siendo imprescindibles para el aprendizaje del lenguaje, sí que son muy interesantes de cara a tener un amplio conocimiento del mismo y a poder escribir código de forma elegante. Asimismo, nos ayudarán a sacar mayor provecho del lenguaje.

En concreto, en este capítulo, comenzaremos descubriendo qué son y cómo utilizar los *iterators* y *generators.* Después pasaremos a explicar la técnica del empleo de *closures* y continuaremos centrándonos en los útiles *decorators.* Dado que la programación funcional va ganando adeptos día a día, también hemos creído conveniente dedicar un apartado a la misma. Finalmente, serán las expresiones regulares y la ordenación eficiente de elementos los apartados que cerrarán el presente capítulo.

ITERATORS Y GENERATORS

Python cuenta con dos mecanismos específicos que nos permiten iterar sobre un conjunto de datos: los *iterators* y los *generators.*

Iterators

Hasta el momento hemos visto una serie de ejemplos que nos han mostrado cómo iterar sobre una serie de objetos, habitualmente empleando un bucle *for* y a través del operador *in.* Para refrescar cómo puede llevarse a cabo una iteración, basta con echar un vistazo al siguiente código:

```
>>> lista = [1, 2, 3]
>>> for ele in lista:
...   print(ele)
...
1
2
3
```

En realidad, Python nos permite iterar directamente sobre ciertos objetos, como son las listas y los diccionarios, incluso es posible iterar sobre un *string*, tal y como podemos ver en el siguiente código:

```
>>> for letra in "hola":
...   print(letra)
...
h
o
l
a
```

Este mecanismo de iteración sobre objetos es bastante potente en Python y puede sernos muy útil para realizar tareas que son más complejas en otros lenguajes, como, por ejemplo, la lectura de las líneas que forman un fichero de texto. Esta operación en Python es muy sencilla gracias a que un fichero es un objeto *iterable,* al igual que lo son las listas, los diccionarios y los *strings,* tal y

como hemos comentado previamente. Aunque contamos con un capítulo dedicado a cómo trabajar con distintos tipos de ficheros, adelantaremos un sencillo ejemplo de código que nos enseña cómo leer todas las líneas de un fichero:

```
>>> for linea in open("fich.txt"):
...   print(linea)
```

En realidad, la iteración es posible en Python gracias a que el intérprete y el lenguaje disponen de una técnica que hace posible su implementación y ejecución a través de un determinado protocolo. Así pues, a bajo nivel, cuando empleamos la sentencia *for,* lo que realmente ocurre es que se realiza una llamada a un *iterator* determinado que se encarga de realizar la función de iteración sobre el objeto concreto.

En general, cualquier tipo de objeto en Python que implemente unos métodos especiales, que forman parte del mencionado protocolo, es un *iterator.* En concreto, para construir un objeto de este tipo, deberemos implementar al menos dos métodos: *__iter__()* y *__next__()*. El primero de ellos es empleado para devolver unareferencia a la Instancia de la clase que ¡mplementa el *iterator,* mientras que el segundo debe implementar la funcionalidad que será ejecutada cuando se lleva a cabo la iteración sobre cada elemento. Supongamos que necesitamos Iterar sobre un valor que será pasado como parámetro, para Imprimir todos los valores desde ese mismo hasta llegar a cero, es decir, se trata de implementar una sencilla cuenta atrás. Nuestra clase *iterator* quedaría de la siguiente forma:

```
class CuentaAtras(object):
    def __init__(self, start):
        elf.count = start
    def __iter__(self):
        return self
    def __next__(self):
        if self.count <= 0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r
```

El constructor de nuestra clase recibirá el valor inicial y lo asignará a una variable de instancia llamada *count.* Al ejecutar el método *__next__()*

comprobaremos el valor de dicha variable y si es igual o inferior a cero deberemos detener la iteración. Para ello, contamos con la excepción *StopIteration*, que será lanzada justo en ese momento. Si esta condición no se cumple, disminuiremos el valor de la variable de Instancia y devolveremos el valor de la misma antes de que se lleve a cabo el decremento. Una vez que tenemos nuestro objeto podemos emplearlo junto a un bucle *for* y al operador *in*, tal y como muestra el siguiente ejemplo:

```
>>> for ele in CuentaAtras(5):
...   print(ele)
...
5
4
3
2
1
```

De igual forma, podríamos trabajar con más de un argumento. Este sería el caso, por ejemplo, de necesitar iterar sobre un rango de números. Veamos cómo hacerlo construyendo un nuevo *iterator:*

```
class Rango:
    def __init__(self, low, high):
        self.current = low
        self.high = high
    def __iter__(self):
        return self
    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        self.current += 1
        return self.current - 1
```

La invocación a nuestro *iterator* sería como sigue:

```
>>> for ele in Rango(5,9)
...   print(ele)
...
5
6
7
8
9
```

FUNCIONES INTEGRADAS

A diferencia de las versiones 2.x de Python, la serie 3 de este lenguaje cuenta con una serie de funciones integradas que devuelven objetos *iterators* en lugar de listas. Entre estas funciones encontramos a *range(), map(), zip()* y *filter().* Por ejemplo, *map()* aplica una función determinada a cada uno de los valores que se le indiquen como parámetro. En Python 3, la llamada a esta función nos devolverá un objeto de tipo *map* que es un *iterator.* A continuación, el código de ejemplo que ilustra este hecho:

```
>>> map(abs, (-1, 2, -3)
<map object at 0x00000000026484E0>
```

Si deseamos obtener una lista a aplicar la función *map(),* tal y como ocurre en Python 2.x, deberemos invocar explícitamente a *list(),* como muestra el siguiente ejemplo:

```
>>> list(map(abs, (-1, 2, -3))
[1, 2, 3]
```

Nótese que la función *abs()* calcula el valor absoluto de un valor dado que recibe como parámetro.

Similar comportamiento encontramos en la función *zip()* que combina valores de los argumentos que recibe. Echemos un vistazo al siguiente código para comprobar su funcionalidad:

```
>>> list(zip (("123", "abc")))
[("1", "a"), ("2", "b"), ("3", "c")]
```

Relacionados con los *iterators* encontramos a los *generators,* de los que nos ocuparemos en el siguiente apartado.

Generators

Si tuviéramos que definir qué es un *generator* podríamos decir que es una función que devuelve una secuencia de resultados en lugar de un único valor. La relación entre los *iterators* y los *generators* es muy estrecha, ya que en la mayoría de las ocasiones, en lugar de crear un objeto *iterator,* podemos crear una

simple función que lleve a cabo la misma funcionalidad. De esta forma conseguimos minimizar las líneas de código y hacemos que el código sea más legible. Para comprobar este hecho, vamos a reescribir nuestro primer ejemplo de *iterator* empleando una simple función que hará de *generator:*

```
def cuenta_atras(ele):
    while ele > 0:
        yield n
        n -= 1
```

Si ahora realizamos la siguiente llamada, observaremos cómo el resultado es idéntico a emplear el *iterator* al que hemos llamado *CuentaAtras*:

```
>>> for ele in cuenta_atras(5):
...   print(ele)
```

Como el lector habrá podido observar, en nuestro nuevo ejemplo hemos introducido el uso una nueva función llamada *yield().* Esta es la encargada de detener la ejecución de nuestra función, producir un nuevo resultado y reanudar la ejecución de la función. Así pues, cada vez que se alcanza *yield* se procede a ejecutar el código que aparece dentro de nuestro bucle *for,* cuando esta acción termina, continúa la ejecución de *cuenta_atras().* Internamente, el intérprete de Python crea un objeto *iterator* cada vez que se llama a la función *generator.*

Un *generator* no tiene por qué ser forzosamente una función, también puede ser una expresión. De hecho, cuando explicamos la comprensión de listas, en el capítulo 2, vimos un ejemplo de una expresión que es un *generator.* Un ejemplo similar es el siguiente código donde, a través de una expresión, construimos un objeto *generator*:

```
>>> lista = [1, 2, 3]
>>> (7*ele for ele in lista)
<generator object <genexpr> at 0x00000000022AD750>
```

Si asignamos nuestro nuevo objeto a una variable, podremos emplear la misma para realizar iteraciones, tal y como demuestran las siguientes sentencias:

```
>>> gen = (7*ele for ele in lista)
>>> for ele in gen:
...   print(ele)
...
7
14
```

```
21
```

A diferencia de otros objetos, la iteración sobre un *generator* solo se puede hacer una única vez. Si deseamos hacerla más veces, deberemos volver a crear el *generator* primero. Esto no ocurre, por ejemplo, con las listas. Una vez definida es posible iterar *n* veces sobre la misma sin necesidad de volver a declarar. Teniendo en cuenta este hecho y tomando como base nuestro último ejemplo de código, si volvemos a ejecutar el bucle *for,* comprobaremos cómo no se imprime ningún número por la salida estándar.

CLOSURES

En Python es posible anidar una función dentro de otra. Una función no es más que un objeto del lenguaje, que, en realidad, es una sentencia ejecutable y como tal, puede aparecer en cualquier lugar del script. Cuando se anidan funciones debe ser tenido en cuenta el espacio de nombres de las mismas, ya que cualquier variable definida en la función que contiene a otra puede ser accedida desde la función anidada. El siguiente ejemplo muestra este hecho:

```
>>> def principal():
...   x = 8
...   def contenida():
...      print(x)
...   contenida()
...
>>> principal()
8
```

El ejemplo de código anterior muestra cómo la variable x es accesible a través de la función *contenida*, que a su vez se encuentra anidada dentro de *principal*. Estas reglas de acceso del espacio de nombres se mantienen incluso si la función *principal* devuelve la función *contenida* como resultado de su ejecución. Si modificamos el código anterior y lo sustituimos por el siguiente, comprobaremos cómo *x* es accesible:

```
def principal():
    x = 7
    def anidada():
        print(x)
    return anidada
```

Ahora es la función *principal* la que va a devolvernos como resultado otra función, que, en este caso, es la función anidada que contiene. Dado que una función queda definida por su nombre y el intérprete pasa a referenciar el espacio de memoria que necesita a través del nombre, es posible devolver una función como si de una variable cualquiera se tratase. De hecho, es simplemente eso, un *objeto* que Python referencia a través de un identificador. Teniendo esto en mente, podemos invocar a la función *principal()* y asignar su resultado a una

variable:

```
>>> res = principal()
```

Si ahora invocamos a la función a través de la nueva variable, observaremos cómo la variable x es accesible desde la función *anidada:*

```
>>> res ()
7
```

Debe ser tenido en cuenta que ya no podemos invocar directamente a *principal(),* sino que debemos hacerlo a través de la nueva variable *res.* En realidad, al devolver una función, aunque el ámbito de *principal()* ya no se encuentra en memoria sí que es posible acceder a la funcionalidad que la misma contiene, dado que su interior ha sido creada una función y devuelta como resultado de la ejecución de *principal().* Es decir, esto es lo que ocurre si invocamos ahora a *principal():*

```
>>> principal()
<function anidada at 0x0000000002577748>
```

A esta técnica se la conoce como *closure* o *factory function.* Esta última nomenclatura hace referencia a que una clase es capaz de crear otra diferente, es decir, funciona como una factoría de funciones.

En términos generales, un *closure* es una técnica de programación que permite que una función pueda acceder a variables que, a priori, están fuera de su ámbito de acceso. Son muchos los lenguajes de programación que soportan esta técnica, como por ejemplo Ruby, que utiliza el concepto de *blocks* para aplicarla.

Los *closures* suelen ser empleados cuando es necesario disponer de una serie de acciones que debe llevarse a cabo como respuesta a un evento que maneja otra función y que ocurre en tiempo de ejecución. Por ejemplo, pensemos en una aplicación que cuenta con una interfaz de usuario. Habitualmente, estas funciones disponen de *manejadores de eventos,* es decir, de funciones o métodos que responden a cierto tipo de eventos, como puede ser el pulsar un botón o hacer clic sobre una caja de texto. Supongamos que tenemos un evento que se encarga de controlar y responder cuando una caja de texto pierde el enfoque. Además, necesitamos realizar una serie de acciones en función del texto que la caja contenga. Dado que no sabemos a priori el texto que contendrá, ya que este

cambiará en tiempo de ejecución, necesitamos otra función que, dentro del manejador del evento, sea capaz de realizar ciertas acciones en función del contenido del texto de la caja.

Aunque hemos descrito un sencillo ejemplo de *closure*, obviamente, las funciones que forman parte del mismo pueden recibir y trabajar con parámetros. Veamos un sencillo ejemplo donde sumaremos el valor de los argumentos que reciben tanto la función contenedora como la anidada:

```
>>> def contenedora(y):
...   def anidada(z):
...      return y + z
...   return anidada
...
>>> fun = contenedora (9)
>>> fun(12)
21
>>> fun(14)
23
```

En este punto, si creamos una nueva variable pasando un valor diferente a *contenedora(),* apreciaremos cómo el resultado varía:

```
>>> otr = contenedora(1)
>>> otr(2)
3
```

Además, podemos seguir invocando, tanto a *fun(),* como a *otr(),* las veces que deseemos. Si comparamos estos ejemplos de código con el caso de la interfaz de usuario, anteriormente comentado, comprobaremos que existe una analogía entre ellos. En concreto, la función *contenedora()* sería la responsable de responder al evento, mientras que *anidada()* se encargaría de procesar el valor del campo de texto y devolver un resultado diferente en función del mencionado valor.

DECORATORS

En el capítulo anterior, hemos descubierto cómo utilizar un *decorator,* para por ejemplo, indicar que un método es de clase. En este apartado describiremos qué es un *decorator,* cómo se utilizan y cómo implementar los nuestros propios.

Patrón *decorator,* macros y Python *decorators*

Antes de describir qué es un *decorator* en Python, conviene entender las diferencias y similitudes entre tres conceptos diferentes: el patrón *decorator,* las macros y los *decorators* de Python.

Aquellos desarrolladores acostumbrados a implementar *patrones de diseño,* seguro que conocen y utilizan el patrón conocido como *decorator.* Básicamente, este patrón nos permite modificar dinámicamente el comportamiento de un objeto. En la práctica, el *decorator pattern* puede ser considerado como una alternativa a declarar una clase que hereda de otra, es decir, a la herencia simple. En los lenguajes compilados, como C++, el cambio de comportamiento se hace en tiempo de ejecución, no en tiempo de compilación.

Por otro lado, las *macros* son utilizadas en programación para modificar una serle de elementos del lenguaje. Los programadores de C estarán acostumbrados a trabajar con macros de preprocesador, Java y *C#* emplean las *annotations,* que es una técnica muy similar a las mencionadas macros.

Los *decorators* de Python no deben ser confundidos con el patrón de diseño *decorator,* ya que su funcionalidad está más cerca de las macros. De esta forma, un *decorator* o *decorador* de Python nos permite inyectar código para modificar el comportamiento de la ¡mplementaclón original de un método, función o clase. Por ejemplo, supongamos que deseamos realizar una serle de operaciones antes y después de ejecutar una determinada función. Para este caso podemos escribir funciones implementadas por decoradores y aplicar estos directamente a nuestra función. Además, esto presenta la ventaja de que las funciones implementadas pueden ser utilizadas como decoradores en otras funciones de nuestro programa, consiguiendo así reutilizar nuestro código eficientemente.

Declaración y funcionamiento

Cuando deseamos aplicar un decorador, por ejemplo, a una función, basta con escribir el nombre del mismo, precedido de la arroba (@), en la línea justamente anterior a la que declarara la función. Un sencillo ejemplo que nos permite aplicar el decorador *firstDecorator* a una función llamada *función*, sería el siguiente:

```
@firstDecorator
def funcion():
    print("Ejecutando la función")
```

Básicamente, la arroba es un operador que indica que un objeto función debe ser pasado a través de otra función, asignando el resultado a la función original. Esta sintaxis es más elegante que utilizar la misma funcionalidad realizando diferentes llamadas a las funciones. Así pues, el ejemplo anterior es similar a declarar una nueva función y realizar una llamada empleando el resultado obtenido previamente:

```
def firstDecorator(obj):
    pass
res = firstDecorator(funcion)
```

En realidad, los decoradores de Python siguen la idea de aplicar código a otro código, como hacen las macros, pero a través de una construcción y sintaxis dada directamente por el lenguaje.

Los *decorators* ofrecen bastante flexibilidad, ya que cualquier objeto, como una clase o función de Python puede ser utilizada como decorador sobre otro objeto, siendo habitualmente este último una función.

Para entender en la práctica cómo funcionan los decorados, nos centraremos en aplicar los mismos empleando clases y funciones. En el siguiente apartado, comenzaremos por las clases.

Decorators en clases

Para ilustrar el funcionamiento de la aplicación de decoradores utilizando clases, vamos a partir de un ejemplo, donde declararemos una clase que

posteriormente será utilizada como decorador en una función.

En primer lugar, vamos a declarar una clase en la que implementaremos y sobrescribiremos los métodos *__init__()* y *__new__()*:

```
class Decorador(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        print("inicio", self.f.__name__)
        self.f()
        print("fin", self.f.__name__)
```

El siguiente paso será definir una función a la que aplicaremos nuestra clase como decorador:

```
@Decorador
def funcion():
    print("soy función")
```

Si ahora procedemos a llamar a nuestra función, podremos observar el siguiente resultado que nos lanza el intérprete:

```
>>> funcion()
inicio funcion
soy función
fin funcion
```

La explicación al resultado obtenido puede explicarse fácilmente. Cuando llamamos a la función, se procede a modificar su comportamiento original aplicando el decorador. Así pues, dado que este es una clase, pasa a ejecutarse el constructor de la misma, que recibe como parámetro la función invocada. El constructor simplemente asigna la misma a un atributo de clase llamado *f*. Posteriormente, se llama automáticamente al método *__call__()* que imprime el primer mensaje en la salida estándar y después invoca a la función original a través del atributo de clase definido previamente. Dado que esta función originalmente imprime el mensaje *soy función*, esto es simplemente lo que ocurre. Continuando con la ejecución de *__call__()*, se imprime el último mensaje. Tal y como habremos observado, *__name__* es un atributo que nos permite acceder al nombre de la función, en este caso. De esta forma, estamos aprovechando las funcionalidades de introspección que Python nos ofrece. Queda demostrado que el comportamiento original de la función llamada *funcion*

ha sido modificado gracias al decorador que hemos creado previamente y aplicado automática y posteriormente.

Por convención, el nombre del decorador suele comenzar por minúscula. Sin embargo, en nuestro ejemplo hemos utilizado la mayúscula. Esto se debe a que los nombres de las clases, también por convención, comienzan por mayúscula. Si se desea puede optarse por cambiar la nomenclatura de nuestro ejemplo, renombrando el nombre de la clase para que la primera letra comience con minúscula. De esta forma, el nombre del decorador también deberá empezar por minúscula. Lo importante es que mantengamos la correspondencia entre nombres, en cualquier otro caso el intérprete lanzará un error sintáctico.

Funciones como *decorators*

Si en el apartado anterior hemos descubierto cómo aplicar clases como decoradores a una función, en éste nos ocuparemos de crear una función como decorador que será aplicado a otra.

Siguiendo con el ejemplo anterior, implementaremos una funcionalidad similar pero con funciones en lugar de con instancias de clases.

Lo primero que deberemos tener en cuenta es que vamos a trabajar con un *closure*. Este concepto hace referencia a que una función puede ser evaluada en un contexto que puede, a su vez, uno o más componente dependiendo de otro entorno diferente. En nuestro caso, tendremos una función declarada dentro de otra, devolviendo la principal el resultado de la ejecución de la función contenida en la misma. Aunque parece un poco confuso, es mejor entenderlo echando un vistazo al siguiente código:

```
def principal(f):
    def nueva():
        print("ini", f.__name__)
        f()
        print("fin", f.__name__)
    return nueva
```

El siguiente paso es definir una nueva función a la que llamaremos *para_decorar*. Será esta nueva función a la que aplicaremos como decorador la función *principal():*

```
©principal
def decorada() :
    print("decorada")
```

Ahora procederemos a invocar a la función *decorada()* y observaremos la salida dada por el intérprete del lenguaje:

```
>>> decorada()
ini decorada
decorada
fin decorada
```

Cuando invocamos a *decorada(),* se evalúa el decorador y se invoca a la función *principal().* Esta simplemente define otra función cuyo resultado es devuelto al finalizar la ejecución de *principal().* Dado que dentro de *nueva()* se invoca a la función *decorada(),* el mensaje "decorada" es impreso entre "ini decorada" y "fin decorada".

Debemos tener en cuenta que el objeto que hace de decorador debe recibir como parámetro del objeto que lo invoca. En este caso, cuando decimos *objeto* nos referimos a uno interno de Python, como puede ser una clase o una función. No se trata de un objeto en el sentido de instancia de una clase. Así pues, en el ejemplo del apartado anterior, era el constructor (*__init__()*) de la clase el encargado de recibir la función como parámetro. En el ejemplo del presente apartado, es la misma función (*principal()*) la que recibe como parámetro la otra función (*(decorada()*) pasada como tal.

Utilizando parámetros

Hasta el momento hemos utilizado los decoradores de Python sin tener en cuenta el paso de parámetros. Sin embargo, es posible invocar a una función decorada pasando a su vez un número determinado de parámetros. Es más, se pueden dar dos casos diferentes, uno donde solo la función decorada recibe parámetros y otro donde es el decorador el que admite ciertos parámetros. En este apartado veremos ambos casos, comencemos pues, por el primero de ellos.

DECORADOR SIN PARÁMETROS

Volviendo a nuestro ejemplo inicial en el que el objeto decorador era una clase, podemos modificarlo para añadirle parámetros en la función decorada. Para ello, bastará con cambiar el método *__call__()*, añadiendo un nuevo parámetro, que en este caso será ***argumentos.* Tal y como vimos en el capítulo 3, vamos a emplear la técnica para desempaquetado de argumentos. Así pues, podremos recibir *n* parámetros. Pasando directamente al código, nuestra clase quedaría reescrita de la siguiente forma:

```
class Decorador(object):
    def init (self, f):
        self.f = f
    def call (self, *argumentos):
        print("inicio", self.f.__name__)
        self.f(*argumentos)
        print("fin", self.f.__name__)
```

Por otro lado, la función decorada va a recibir, por ejemplo, tres parámetros diferentes. De esta forma, el nuevo código para la función decorada sería el siguiente:

```
@Decorador
def funcion(p1, p2, p3):
    print("soy función con argumentos", p1, p2, p3)
```

La invocación a la función decorada con tres parámetros diferentes y su correspondiente resultado sería como sigue:

```
>>> funcion("uno", "dos", "tres")
inicio funcion
soy función con argumentos uno dos tres
fin funcion
```

Como habremos podido comprobar, este paso de parámetros es sencillo. Simplemente hemos tenido en cuenta los mismos al igual que en una función convencional.

DECORADOR CON PARÁMETROS

Otro caso diferente al anteriormente tratado es cuando el decorador contiene diferentes parámetros durante su invocación. Es decir, al utilizar el decorador es posible también que este pueda trabajar con diferentes argumentos.

Para ilustrar el paso de parámetros en el *decorator* vamos a trabajar con funciones. Tendremos dos principales: el decorador y la decorada. Esta última contendrá a su vez otras dos anidadas que nos ayudarán a tratar con los argumentos del decorador y de la función decorada. Comencemos por la función decorada, que contendrá el siguiente código:

```
@decorator_args(1, 2, 3)
def fun(a, b):
    print ("fun args:", a, b)
```

En nuestro ejemplo, la función decorada recibirá dos parámetros, siendo su funcionamiento original la impresión de los mismos por la salida estándar.

Por otro lado, declararemos la función decorador empleando el siguiente código:

```
def decorator_args(arg1, arg2, arg3):
    def wrap(f):
        print( "ini wrap()")
        def wrapped_f(*args):
            print( "ini wrapped_f")
            print( "decorator args:", arg1, arg2, arg3)
            f(*args)
            print( "fin wrapped_f")
        return wrapped_f
    return wrap
```

En este punto, estamos en condiciones de proceder a la invocación de la función decorada con diferentes valores pasados como parámetros, siendo la salida la que aparece a continuación:

```
>>> fun('p1", "p2")
ini wrap()
ini wrapped_f
decorator args: 1 2 3
fun args: p1 p2
fin wrapped_f
```

En la salida anterior comprobaremos cómo los parámetros que utiliza el decorador y la función decorada son diferentes.

Si estamos trabajando con el intérprete, al terminar de definir la función decorada, veremos cómo, automáticamente, el intérprete llama al decorador. Posteriormente, podemos invocar a la función decorada con sus parámetros. Este sería el resultado utilizando directamente el intérprete de Python:

```
>>> @decorator_args(1, 2, 3)
... def fun(a, b) :
...   print ("fun args:", a, b)
...
ini wrap()
```

A pesar de haber definido y aplicado un único decorador por función, también es posible aplicar *n* decoradores sobre la misma. De esta forma podemos ganar en reutilización de código y aumentar el número de operaciones realizadas que sustituyen a la funcionalidad implementada en la función original.

Hasta aquí nuestro recorrido por los *decorators.* A pesar de haber mostrado sencillos ejemplos para ilustrar el funcionamiento básico de los mismos, la flexibilidad que nos ofrecen es significativa a la vez que potente. Sin duda, los *decorators* son uno de los aspectos más interesantes que nos ofrece Python.

PROGRAMACIÓN FUNCIONAL

Tanto la orientación a objetos como la programación procedural, son dos de los paradigmas más utilizados por los programadores de Python. Sin embargo, también es posible emplear la programación *funcional* con Python. Básicamente, este paradigma se fundamenta en el empleo de funciones aritméticas sin tener en cuenta cambios de estado. En este tipo de programación el valor generado por una función depende exclusivamente de los argumentos que recibe y no del estado del resto del programa cuando la misma es invocada. Un programa escrito utilizando este paradigma únicamente contiene funciones, pero no en el sentido de las *funciones* de la programación procedural, sino en el sentido matemático de expresión. Otra característica de la programación funcional es que no utiliza la asignación de variables. Tampoco se emplean construcciones secuenciales como la iteración, por el contrario se hace un uso intensivo de la recursión.

Para aplicar la programación funcional con Python contamos con varias técnicas y recursos como son: las funciones *lambda*, los *iterators* y *generators*, la *comprehensión* de listas y una serie de funciones integradas. Dado que solo nos queda ocuparnos de estas últimas, dedicaremos las siguientes líneas a este propósito.

En concreto, entre las funciones integradas que Python pone a nuestra disposición para escribir código utilizando programación funcional, destacan *map(), filter()* y *reduce().* La primera de ellas permite aplicar una función sobre un conjunto de elementos. Por ejemplo, supongamos que tenemos una lista con varias cadenas de texto y necesitamos generar una nueva lista con las mismas cadenas pero en minúsculas. Haremos uso del método *lower()*, junto con *map()* y la función *list()* para obtener una lista como resultado. Para ilustrar nuestro ejemplo, construiremos una pequeña función que hará de *wrapper* sobre el mencionado método *lower():*

```
>>> def lower(t) : return t.lower()
...
>>> texto = ["HOLA", "mUNdO", "AdiOS"]
>>> list(map(lower, texto))
["hola", "mundo", "adios"]
```

El mismo ejemplo podría haber sido realizado utilizando la comprehensión de listas:

```
>>> [cad.lower() for cad in texto]
```

Efectivamente, la función *map()* implementa la misma funcionalidad que las expresiones que producen *generators*, que es, en realidad, lo que hemos hecho con la sentencia anterior de código.

Por otro lado, la función *filter()* devuelve un *iterator* sobre todos los elementos de una secuencia que cumplen una determinada condición. Para comprobar esta condición se suele emplear un *predicado* que es, básicamente, una función que solo devuelve el valor *True* cuando se cumple una condición. Por ejemplo, supongamos que necesitamos solo los valores pares de una lista determinada. En primer lugar construiremos la función con nuestro *predicado*:

```
def par(val):
    return (val%2) == 0
```

Seguidamente pasamos a utilizar directamente la función *filter()*, junto con *list()*, sobre una determinada lista:

```
>>> lista = [0, 3, 5, 6, 8, 9]
>>> list(filter(par, lista))
[0, 6, 8]
```

Por último, *reduce()* aplica una función tomando como argumentos los valores correlativos de una secuencia, devolviendo un único resultado. Un ejemplo sencillo sería definir una función que suma los valores pasados como parámetros. A través de *reduce()* podemos ir sumando los valores de una lista de forma secuencial. Nuestra función quedaría como sigue:

```
def sumar(x, y):
    return x+y
```

Después podríamos invocar a *reduce()* sobre una lista con contiene valores del 1 al 4, donde *sumar()* se encargaría de realizar las siguientes operaciones:

```
1+2=3+3=6+4=10
```

Los números en negrita representan los elementos de la lista y el código completo para realizar la operación sería el siguiente:

```
>>> from functools import reduce
>>> lista = [1, 2, 3, 4]
>>> reduce(sumar, lista)
10
```

Seguro que el lector ha observado cómo hemos importado la función *reduce()* de un módulo llamado *functools*. Este es un módulo Integrado en Python3 y que ofrece varios métodos para generar *iterators* y que, por lo tanto, pueden utilizarse para la programación funcional. En la versiones 2.x de Python la función *reduce()* existía como tal en la librería estándar, no como parte del mencionado módulo.

EXPRESIONES REGULARES

Las *expresiones regulares* definen una serie de patrones que se utilizan para trabajar con cadenas de texto. Podemos considerar a las mismas como un lenguaje específico diseñado para localizar texto basándonos en un determinado patrón previamente definido. Básicamente, las expresiones regulares son utilizadas con dos propósitos a menudo relacionados: la búsqueda y reemplazo de texto.

Habitualmente, a las expresiones regulares simplemente se las llama *regex* y son muchos los programadores que prefieren utilizar este término. Además, como nomenclatura, algunos desarrolladores emplean el prefijo *regex* para designar variables que representan a una expresión regular.

Al igual que la mayoría de los lenguajes de programación, Python incluye un completo soporte para las expresiones regulares. Este lenguaje dispone de un *módulo* específico, llamado *re*, que permite realizar búsquedas, sustituciones y obtener parte de una cadena de texto empleando expresiones regulares como patrones.

El módulo *re* de Python utiliza las reglas definidas en el modelo *NFA* tradicional (autómata finito no determinista), el cual se basa en la comparación de cada elemento de la expresión regular con la cadena de texto de entrada. Este modelo se encarga de mantener un seguimiento de las posiciones que referencian a la selección entre dos opciones dentro de la expresión regular. Si una de estas opciones falla, se busca la última entre las más recientes cuya posición ha sido guardada. Para más detalles sobre este modelo podemos consultar la página (ver referencias) de *Wikipedia* al respecto.

Patrones y metacaracteres

Para componer una expresión regular, debemos emplear una serie de patrones que son considerados como básicos. La tabla 5-1 muestra los más utilizados dentro de esta categoría y el significado asociados a ellos.

Patrón	Significado
	Cualquier carácter excepto el que representa una nueva línea
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador horizontal
\w	Cualquier carácter que represente un número o letra en minúscula
\W	Cualquier carácter que represente un número o letra en mayúscula
\s	Espacio en blanco (nueva línea, retorno de carro, espacio y cualquier tipo de tabulador)
\S	Cualquier carácter que no es un espacio en blanco
\d	Número entre 0 y 9 Inclusive
\D	Cualquier carácter que no es un número
^	Inicio de cadena
$	Fin de cadena
\	Escape para caracteres especiales
[]	Rango. Cualquier carácter que se encuentre entre los corchetes
^[]	Cualquier carácter que no se encuentre entre los corchetes
\b	Separación entre un número y/o letra

Tabla 5-1. Patrones básicos para definir expresiones regulares en Python

Por otro lado, cualquier carácter se representa a sí mismo, teniendo en cuenta que algunos constituyen por sí mismo un patrón básico. Por ejemplo, el carácter *$* indica el final de una cadena; por lo tanto, no podemos emplearlo, cuando, por ejemplo, buscamos este carácter tal cual en una cadena de texto. Si necesitamos buscar el carácter como tal, podemos emplear el patrón básico de *escape* (\). De esta forma, para busca el carácter *$* deberemos construir una expresión regular que utilice la secuencia *\$*.

Aparte de los patrones básicos, también debemos conocer cuáles son los caracteres que representan las *repeticiones* que pueden darse. Supongamos que estamos buscando en una cadena de texto un patrón que corresponde a un determinado número de veces que aparece en la misma un carácter. Para ello, recurriremos a los metacaracteres de repetición, los cuales aparecen explicados en la tabla 5-2.

Metacarácter	Significado
+	Una o más veces
*	Cero o más veces
?	Cero o una vez
{n}	El carácter se repite *n* veces

Tabla 5-2. Metacarácteres de repetición en Python

Una vez que conocemos los patrones básicos y los metacaracteres de repetición, podemos comenzar a definir sencillas expresiones regulares. Por ejemplo, la siguiente expresión regular representa a que un número puede aparecer una o más veces:

```
[\d]+
```

En los siguientes apartados nos centraremos en la funcionalidad que Python ponemos a nuestra disposición para realizar búsquedas y sustituciones de texto basándonos en expresiones regulares.

Búsquedas

La función básica de la que dispone el módulo *re* de Python es *search()*. Esta función recibe como argumento una expresión regular, representada por un patrón, y una cadena donde realizar la búsqueda. El objetivo es buscar una cadena dentro de otra, quedando la cadena que deseamos buscar definida por una expresión regular. Nuestro primer ejemplo consistirá en buscar la letra o en la cadena de texto *hola*. El siguiente código muestra cómo hacerlo:

```
>>> import re
>>> re.search(r"o", "hola")
<_sre.SRE_Match object at 0x00000000021A15E0>
```

Efectivamente, la simple cadena de texto "o" es por sí misma una expresión regular. Nótese que nuestra cadena va precedida por la letra "r", esto le indica al intérprete que se trata de un expresión regular. Por otro lado, el segundo argumento de *search()* es la cadena donde vamos a buscar.

Si reescribimos la última línea de código asignado el resultado a una

variable, podremos acceder a los diferentes métodos del objeto que representa los resultados obtenidos al realizar la búsqueda. Por ejemplo, el método *group()* nos devolverá la subcadena que coincide con nuestro patrón de búsqueda. Veamos un sencillo ejemplo:

```
>>> m = re.search(r"o", "hola")
>>> m.group()
"o"
```

Como habremos podido comprobar, la cadena devuelta es el propio carácter que buscamos. Pasemos a un ejemplo un poco más complicado, supongamos que necesitamos encontrar cualquier subcadena que contenga justamente tres números seguidos. En este caso buscaremos nuestro patrón en tres cadenas de texto diferentes:

```
>>> m = re.search(r"\d\d\d", "hola402adios").group()
402
>>> m = re.search(r"\d\d\d", "986hola").group()
986
>>> m = re.search(r"\d\d\d", "adios123").group()
123
```

Si pensamos utilizar la misma expresión regular para búsquedas en diferentes cadenas de texto, es aconsejable *compilar* la expresión regular. Esto se puede hacer directamente a través de la función *compile*. Gracias a ello ganamos en eficiencia y simplificamos nuestro código. A continuación, aplicaremos esta función tomando como base los ejemplos anteriores:

```
>>> patron = re.compile("\d\d\d")
>>> patron. search ("hola402adios").group()
402
>>> patron.search("986hola")
986
>>> patron.search("adios123")
123
```

¿Qué ocurre si el patrón no coincide con ninguna parte de la cadena donde se realiza la búsqueda? Sencillo, el método nos devolverá *None:*

```
>>> m = re.search(r"\d\d\d", "98x65adios")
>>> if m is None: print("No existen coincidencias")
```

Obsérvese, que en este último caso, no hemos utilizado directamente el método *group(),* dado que el resultado es *None*, si aplicáramos directamente la

llamada, obtendríamos un error en tiempo de ejecución. Es decir, se produciría una excepción.

En realidad, *group()* es capaz de agrupar resultados. De esta forma, podemos utilizar paréntesis en nuestra expresión regular con el objetivo de agrupar la subcadenas encontradas. Gracias a ello, es posible tener más control sobre el resultado. Supongamos que necesitamos buscar un patrón que representa una serie de números seguidos de un guión y de otro conjunto de letras. Además, queremos agruparlo en dos grupos, uno para los números y otro para las letras. Nuestra expresión regular quedaría de la siguiente forma:

```
>>> regex = re.compile(r"(\d+)-([A-Za-z]+")
```

Ahora pasaremos a buscar en una cadena concreta:

```
>>> m = regex.search("23-cDb")
```

La variable *m* representa el resultado de nuestra búsqueda. Dado que la cadena cumple el patrón, podremos llamar directamente al método *group():*

```
>>> m.group(1)
23
>>> m.group(2) cDb
```

Por otro lado, si llamamos a *group()* utilizando el índice 0, obtendremos la cadena encontrada completa:

```
>>> m.group(O)
23-cDb
```

Hasta ahora no hemos tenido en cuenta el principio y fin de cadena, simplemente estamos buscando un patrón en toda la cadena. Observemos las siguientes sentencias y su resultado:

```
>>> re.search(r" ^hola$", "adiosholaadios")
>>> re.search(r"hola", "adiosholaadios")
<_sre.SRE_Match object at 0x00000000021A15E0>
```

En el primer caso, no encontramos coincidencias, ya que, estamos indicando que la cadena debe comenzar y terminar con los caracteres que se encuentra entre el ¡nielo (^) y el fin (*$*).

Otra de las funciones de búsqueda con las que cuenta el módulo *re* es *findall()*, que puede ser utilizada junto con los grupos para obtener una lista de

tuplas que representa todas las coincidencias encontradas. El caso más sencillo sería utilizar *findall()* sin emplear grupos. Por ejemplo, supongamos que necesitamos una lista con todos los valores de una cadena de texto que cumplen el patrón, que nuestro caso será cuando aparezca exactamente tres números seguidos. Para ello, podemos emplear el código que viene a continuación:

```
>>> re.findall(r"\d{3}", "345abcd78ghx678")
[345, 678]
```

Si combinamos la función anterior con grupos, el resultado será una lista de tuplas, tal y como hemos avanzado previamente:

```
>> re.findall(r"([a-z]+)-(\d+)", "345abcd-78ghx-678")
[(abcd", "78"), ("ghx", "678"]
```

En el ejemplo anterior hemos podido observar cómo los valores de la lista contienen una tupla con cada uno de los valores de la cadena que coinciden con los grupos del patrón.

Sustituciones

La principal función que nos ofrece el módulo *re* para realizar reemplazos de texto en una determinada cadena, empleando para ello expresiones regulares, es *sub*. Como argumentos, esta función recibe cuatro argumentos diferentes. El primero de ellos es una expresión regular que indica el patrón que será buscado. El segundo argumento es la cadena de texto que se utilizará como reemplazo de la coincidencia indicada por el primer argumento. El tercero de los argumentos es la cadena de texto donde se llevará a cabo el reemplazo. El cuarto y último argumento es opcional e indica el número de ocurrencias que deben ser reemplazadas, por defecto se reemplazarán todas las que se encuentren. La función *sub()* devuelve el resultado de aplicar el reemplazado, quedan sin modificar los argumentos recibidos por la misma.

Supongamos que contamos con una cadena de texto que contiene letras y números y deseamos reemplazar todos los números que contenga por guiones. El siguiente código nos muestra cómo hacerlo:

```
>>> re.sub(r"\d", "-", "345abcdef987")
```

```
---abcdef---
```

Si en el código anterior utilizáramos el cuarto parámetro que acepta la función *sub()*, por ejemplo usando el valor 2, solo se reemplazarían los tres primeros números, tal y como podemos apreciar en el siguiente ejemplo:

```
>>> re.sub(r"\d", "-", "345abcdef987", 3)
---abcdef987
```

Por otro lado, *sub()* también no permite utilizar grupos y sustituir cada uno de ellos por un valor diferente. Un ejemplo de ello podría ser el que ilustra el siguiente código:

```
>>> re.sub(r"(\w+)@(\w+)", r"\1@123", "abc@efg")
abc@123
```

En el ejemplo anterior los caracteres *\1* del segundo parámetro indican que el primer grupo del patrón debe conservarse, siendo sustituido el segundo grupo por el valor indicado después de *\1*. Si sustituimos el segundo parámetro indicando un 2 en lugar de un 1, el resultado sería diferente:

```
>>> re.sub(r"(\w+)@(\w+)", r"\2@123", "abc@efg")
efg@123
```

Tanto para las búsquedas como para las sustituciones, Python 3 puede trabajar directamente con caracteres *Unicode*. No olvidemos, que por defecto, todos los *strings* son de este tipo en esta versión del intérprete del lenguaje. De esta forma, la siguiente sentencia es completamente válida y funcional:

```
>>> re.sub(r"ñ", "n", "niño")
niño
```

Separaciones

Interesante es la también función *split()* ofrecida por el módulo para expresiones regulares de Python. Básicamente esta función sirve para obtener una lista donde cada elemento es el resultado de utilizar como criterio de separación el patrón dado por la expresión regular. Esta funcionalidad es más fácil de entender utilizando un ejemplo:

```
>>> cad = "Uno<a>Dos<b>Tres<c>Cuatro"
["Uno", "Dos", "Tres", "Cuatro"]
```

Tal y como el lector habrá podido deducir, esta funcionalidad es similar al método del mismo nombre de los *strings*, con la diferencia que el incluido en el módulo *re* admite expresiones regulares.

Modificadores

Cuando compilamos una expresión regular utilizando la función *compile()* podemos emplear una serie de *modificadores* que alteran el comportamiento del patrón. Estos modificadores son muy útiles, por ejemplo, para cuando deseamos ignorar mayúsculas y minúsculas (case *insensitive).* En concreto, los modificadores más comunes son los siguientes:

- IGNORECASE: No tiene en cuenta mayúsculas ni minúsculas.
- DOTALL: Permite que el metacarácter punto (.) tenga en cuenta las líneas en blanco.
- MULTILINE: Sirve para que se puedan utilizar los caracteres de inicio (^) y fin (*$*) de línea en una cadena que contiene más de una línea.

A continuación, veamos un ejemplo de sustitución de cadena con independencia de si son mayúsculas o minúsculas:

```
>>> regex = re.compile(r"abc", re.IGNORECASE)
>>> regex.search("124AbC")
AbC
```

Para ver cómo funcionan las búsquedas utilizando el modificador *MULTILINE*, buscaremos un determinado patrón entre una cadena de texto que incluye el carácter salto de carro *(\n).* En realidad, esta cadena estará compuesta por varias líneas, tal y como ocurre, por ejemplo, cuando leemos de un fichero, donde cada línea es una cadena en sí misma. El código mostrado a continuación nos enseña cómo usar el mencionado modificador en estos casos:

```
>>> regex = re.compile(r"^Text: (\d+)$", re. MULTILINE)
>>> cad = "Text: 34\nText: 35\nText: aa\nText: 24\n"
```

```
>>> regex.search(cad).group(1)
Text: 34
```

Patrones para comprobaciones cotidianas

En este apartado vamos a mostrar una serie de expresiones regulares que suelen ser empleadas asiduamente para realizar comprobaciones y validaciones. Por ejemplo, es común en muchas aplicaciones web comprobar si una determinada cadena de texto introducida por el usuario es, sintácticamente, una cuenta de correo electrónico. La tabla 5-3 muestra un conjunto de este tipo de expresiones regulares.

Expresión regular	Funcionalidad
^d{1,6}$	Números desde el 0 hasta el 999999
^#([a-fA-FO-9]){3}(([a-fA-FO-9]){3})?$	Código hexadecimal para HTML
^\d\d/\d\d/\d\d\d\d$	Fecha en formato dd/mm/yyyy
^.*\//	UNIX path
^([0-9afA-F]{2}:){5}[0-9afA-F]{2}$	Dirección MAC
^[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z_+])*@([0-9a- zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z{2,9}$	Cuenta de correo electrónico
(https?):\/\/([0-9a-zA-Z][-\w+]*[0-9a-zA- Z]\.)+[a-zA-Z]{2,9})(:\d{1,4})?([-\w/\#~:?+=&%@~]*)	HTTP URL
^(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[0-1]?\d\d|2[0-4]\d|25[0-5])\.(\d[01]?\d\d|2[0-4\d|25[0-5])$	Dirección IPv4

Tabla 5-3. Expresiones regulares útiles para validaciones

ORDENACIÓN DE DATOS

En el presente apartado nos centraremos en aprender cómo ordenar datos. La función más sencilla que Python nos ofrece es la llamada *sorted()*, que básicamente, recibe como argumento una lista y devuelve otra con los elementos de la original ya ordenados. En principio, es el intérprete quien decide los criterios de ordenación, siendo esto trivial cuando los elementos de la lista son números o caracteres. Observemos el siguiente ejemplo, donde ordenaremos una lista de números:

```
>>> lista = [5, 1, 9, 8, 3]
>>> sorted(lista)
[1, 3, 5, 8, 9]
```

De forma análoga, podemos ordenar una serie de caracteres o cadenas de texto:

```
>>> cads = ["ca", "dc", "cb", "ab", "db"]
>>> sorted(cads)
["ab", "ca", "cb", "db", "dc"]
```

Python también tiene en cuenta el orden si las cadenas de texto utilizan mayúsculas y minúsculas, teniendo preferencia las primeras sobre las segundas, tal y como podemos comprobar en el siguiente código:

```
>>> cads = ["ca", "Cc", "cb", "aB", "db"]
>>> sorted(cads)
["Cc", "aB", "ca", "cb", "db"]
```

Adicional y optativamente, la función *sorted()* puede utilizar un segundo parámetro para indicar si el orden debe hacerse ascedente o descendentemente. Este parámetro se llama *reverse* y por defecto su valor es *False.* Así pues, para ordenar nuestra primera lista en orden descendente, basta con ejecutar la siguiente sentencia:

```
>>> sorted(lista, reverse=True)
[9, 8, 5, 3, 1]
```

En caso de que deseemos indicar cuál será el criterio de ordenación que debe

emplear la función *sort()*, podremos hacerlo gracias al parámetro *key* que acepta esta función. El valor de este parámetro deberá ser una función, que bien, podemos implementar nosotros mismos o una ya integrada en el intérprete. Por ejemplo, supongamos que deseamos ordenar una lista en función del número de elementos que contienen. En este caso, basta con utilizar la función integrada *len()*, tal y como muestra el siguiente ejemplo:

```
>>> lista = ["abc", "de", "fghij"]
>>> sorted(lista)
["abc", "de", "fghij"]
>>> sorted(lista, key=len)
["de", "abc", "fghij"]
```

Método *itemgetter()*

Para ordenaciones más complejas puede resultar muy útil emplear el método *itemgetter()* del módulo integrado *operator*. Este método permite generar un objeto que utilizará el valor pasado como parámetro para devolver el elemento que ocupa la posición que indica el mencionado valor cuando el objeto es invocado pasando como parámetro una serie de elementos. En realidad, es más fácil de ver con un sencillo ejemplo de código que entender la explicación que acabamos de ofrecer:

```
>>> from operator import itemgetter
>>> fun = itemgetter(2)
>>> fun( [1, 2, 3, 4])
3
```

El valor devuelto es 3 porque este es el valor que ocupa la posición dos en la lista pasado como parámetro, recordemos que el índice de los elementos de una lista comienza por cero y no por uno.

Si combinamos la función *sorted()* junto con el método *itemgetter()*, podemos realizar complejas ordenaciones. Supongamos que contamos con una lista de tuplas, donde cada uno de los elementos de cada tupla es un número y una cadena de texto. Dada esta estructura de datos, necesitamos ordenar teniendo en cuenta el segundo valor de la tupla, que en nuestro caso será un número. Veamos cómo hacerlo a través del ejemplo que viene a continuación:

```
>>> lista = [("Madrid", 4), ("Barcelona", 1), ("Sevilla", 5),
("Valencia", 3)]
[("Madrid", 4), ("Barcelona", 1), ("Sevilla", 5), ("Valencia", 3)]
>>> sorted(lista, key=itemgetter(1))
[("Barcelona", 1), ("Valencia", 3), ("Madrid", 4), ("Sevilla", 5)]
```

Tal y como podemos apreciar, al ordenar se ha tenido en cuenta el segundo valor de cada tupla, es por ello que hemos utilizado el valor *1* como parámetro para el método *itemgetter()*. No olvidemos que en las tuplas los índices también comienzan por cero, al igual que en las listas.

El método *itemgetter()* puede recibir más de un argumento. imaginemos que tenemos una lista similar a la anterior que queremos ordenar, primero teniendo en cuenta el segundo valor y luego el primero de cada una de las tuplas que forma la lista. Bastaría con pasar un argumento adicional al mencionado método, tal y como muestra el siguiente ejemplo:

```
>>> lista = [("a", 2), ("c", 2), ("b", 3), ("d", 3), ("z", 1), ("a",
1), ("d", 1)]
>>> sorted(lista, key=itemgetter(0,1))
[("a", 1), ("a", 2), ("b", 3), ("c", 2), ("d", 1), ("d", 3), (" z",
1) ]
```

De forma análoga, podemos invertir el orden de los parámetros para conseguir una ordenación diferente:

```
>>> sorted(lista, key=itemgetter(1,0))
[("a", 1), ("d", 1), ("z", 1), ("a", 2), ("c", 2), ("b", 3), ("d",
3)]
```

Funciones *lambda*

Una técnica habitual para ordenar a través de la función *sorted()* es emplear funciones *lambda()* como valor para el parámetro *key*. Gracias a esta técnica es fácil ordenar, por ejemplo, una lista que contiene una serie de objetos con unos valores determinados. Vamos a trabajar con una clase que representará al empleado de una empresa, posteriormente construiremos una lista de objetos *Empleado* y finalmente la ordenaremos en función del atributo que corresponde a sus apellidos. Comenzamos declarando nuestra clase:

```
class Empleado:
```

```
    def __init__(self, apellidos, puesto, edad):
        self.apellidos = apellidos
        self.puesto = puesto
        self.edad = edad
    def __repr__(self):
        return repr((self.apellidos, self.puesto, self.edad))
```

Llegamos al momento de crear nuestra lista con diferentes objetos de la clase *Empleado*:

```
>>> empleados = [Empleado("Fernández", "Administración", 25),
...              Empleado("Dominguez", "Finanzas", 38),
...              Empleado("Amaro", "Contabilidad", 21)]
...
>>> print(sorted(empleados, key=lambda empleado: empleado.apellidos))
[("Amaro", "Contabilidad", 21), ("Dominguez", "Finanzas", 38),
("Fernández", "Administración", 25)]
```

De forma análoga podemos emplear otro atributo como criterio de ordenación, por ejemplo, la edad:

```
>>> print(sorted(empleados, key=lambda empleado: empleado.apellidos)
)
[("Amaro", "Contabilidad", 21), ("Fernández", "Administración", 25)m
("Dominguez", "Finanzas", 38)]
```

FICHEROS

INTRODUCCIÓN

Los sistemas operativos almacenan los datos de forma estructurada en unas unidades básicas de almacenamiento a las que llamamos *ficheros o archivos.* Tarde o temprano, los programadores necesitan trabajar con ficheros, ya sea para leer datos de los mismos o para crearlos. Es por ello que los lenguajes de programación suelen incluir librerías que contienen funciones para el tratamiento de ficheros. En Python contamos con una serie de funciones básicas, incluidas en su librería estándar, para leer y escribir datos en ficheros. A estas funciones dedicaremos el primer apartado del presente capítulo.

Una vez que aprendamos todo sobre lo básico sobre el manejo de ficheros en Python, pasaremos a adentrarnos en el concepto de *señalización*, el cual está directamente relacionado con los ficheros. Descubriremos qué herramientas nos ofrece el lenguaje y qué módulos de la librería estándar pueden ser empleados para serializar datos en ficheros. Además, continuaremos haciendo un repaso a los tres principales formatos utilizados para serializar: XML, JSON y YAML. Estos formatos no solo se usan para serializar, sino que también suelen ser empleados para guardar información estructurada y para el intercambio de la misma. Por ejemplo, son muchas las aplicaciones que utilizan el formato YAML para guardar información sobre una determinada configuración.

Dado que el formato CSV es uno de los más sencillos y populares para guardar información estructurada en ficheros, dedicaremos un apartado específico a ver cómo puede Python manejar este tipo de ficheros. Aplicaciones como *MS Office* y *LibreOffice* pueden exportar e importar datos en formato CSV, de forma que necesitamos desarrollar una aplicación que trabaje con este formato y que pueda intercambiar datos con este tipo de programas, podremos emplear Python como lenguaje de programación.

Python cuenta con un módulo específico que permite leer ficheros en formato INI. Este es muy popular en los sistemas Windows y suelen ser empleado para guardar información relativa a una determinada configuración. También en este capítulo aprenderemos lo básico para poder trabajar con ficheros del mencionado

tipo.

Hoy en día es muy común trabajar con ficheros comprimidos que nos permiten almacenar más información en menos espacio. Formatos como ZIP, RAR o gzip se han convertido en formatos *de facto* y la mayoría de los sistemas operativos permiten utilizar herramientas para comprimir o descomprimir ficheros. Teniendo en cuenta estos factores, parece útil disponer de funcionalidades que permitan trabajar con este tipo de ficheros desde un lenguaje de programación. En Python contamos con varios módulos de su librería estándar que nos facilitan el trabajo, de ellas nos ocuparemos en el último apartado de este capítulo.

OPERACIONES BÁSICAS

Python incorpora en su librería estándar una estructura de datos específica para trabajar con ficheros. Básicamente, esta estructura es un *stream* que referencia al fichero físico del sistema operativo. Dado que el intérprete de Python es multiplataforma, el manejo interno y a bajo nivel que se realiza del fichero es transparente para el programador. Entre las operaciones que Python permite realizar con ficheros encontramos las más básicas que son: apertura, creación, lectura y escritura de datos.

Apertura y creación

La principal función que debemos conocer cuando trabajamos con ficheros se llama *open()* y se encuentra integrada en la librería estándar del lenguaje. Esta función devuelve un *stream* que nos permitirá operar directamente sobre el fichero en cuestión. Entre los argumentos que utiliza la mencionada función, dos son los más importantes. El primero de ellos es una cadena de texto que referencia la ruta *(path)* del sistema de ficheros. El otro parámetro referencia el *modo* en el que va a ser abierto el fichero. Es importante tener en cuenta que Python diferencia entre dos tipos de ficheros, los de texto y los binarios. Esta diferenciación no existe como tal en los sistemas operativos, ya que son tratados de la misma forma a bajo nivel. Sin embargo, a través del *modo* podemos indicarle a Python que un fichero es un tipo u otro. De esta forma, si el *modo* del fichero es texto, los datos leídos serán considerados como un *string*, mientras que si el *modo* es binario, los datos serán tratados como *bytes*.

Antes de comenzar a ver un ejemplo de código, abriremos nuestro editor de textos favorito, añadiremos una serie de líneas de texto y lo salvaremos, por ejemplo, con el nombre *fichero.txt*. Seguidamente, ejecutaremos el intérprete de Python desde la interfaz de comandos y escribiremos la siguiente línea:

```
>>> fich = open("fichero.txt")
```

En nuestro ejemplo, hemos supuesto que el fichero creado se encuentra en la

misma ruta desde la que hemos lanzado el intérprete, de no ser así es necesario indicar el *path* absoluto o relativo al fichero. Por otro lado, no hemos indicado ningún modo, ya que, por defecto, Python emplea el valor *r* para ficheros de texto. Antes de continuar, debemos tener en cuenta que los sistemas operativos suelen emplear diferentes caracteres como separadores entre los directorios que forman parte de una ruta del sistema de ficheros. Por ejemplo, supongamos que nuestro fichero se encuentra en un directorio llamado *pruebas*. Si estamos trabajando en Windows, nuestra línea de código para abrir el fichero sería la siguiente:

```
>>> fich = open("pruebas\fichero.txt")
```

Sin embargo, para realizar la misma operación en Linux necesitaríamos esta otra línea de código:

```
>>> fich = open("pruebas/fichero.txt")
```

Dado que Python es multiplataforma, es deseable que el mismo código funcione con independencia del sistema operativo que lo ejecuta, pero, en nuestro ejemplo, este código es distinto. ¿Cómo resolver este problema? Es sencillo, para ello Python nos facilita una función que se encuentra integrada en el módulo *os* de su librería estándar. Se trata de *join()* y se encarga de unir varias cadenas de texto empleando el separador adecuado para cada sistema operativo. De esta forma, sería más práctico escribir el código anterior para la apertura del fichero de la siguiente forma:

```
>>> from os import path
>>> ruta_fich = path.join("pruebas", "fichero.txt")
>>> fich = open(ruta_fich)
```

Por otro lado y como complemento a la función *join()*, también existe la variable sep, que también pertenece al módulo *os*. Esta representa el carácter propiamente dicho que cada sistema operativo emplea para la separación entre directorios y ficheros.

Respecto al valor que se puede indicar para el modo de apertura del fichero, Python nos permite utilizar un valor determinado en función de la operación (lectura, escritura, añadir y lectura/escritura) y otro para señalar si el fichero es de texto o binario. Obviamente, ambos tipos de valores se pueden combinar, para, por ejemplo, indicar que deseamos abrir un fichero binario para solo

lectura. En concreto, el valor "r" significa "solo lectura"; con "w" abriremos el fichero para poder escribir en él; para añadir datos al final de un fichero ya existente emplearemos "a" y, por último, si vamos a leer y escribir en el fichero, basta con utilizar "+". Por otro lado, con "b" señalaremos que el fichero es binario y con "t" que es de texto. Como hemos comentado previamente, por defecto, la función *open()* interpreta que vamos a abrir un fichero de texto como solo lectura. Tanto el valor para realizar una operación como para indicar el tipo de fichero debe indicarse como argumentos del parámetro *mode.* Así pues, para abrir un fichero binario para lectura y escritura basta con ejecutar el siguiente comando:

```
>>> open("data.bin", "b+")
```

Además de los mencionados parámetros de la función *open()*, existen otros que podemos utilizar. En concreto, contamos con cinco más. El primero de ellos es *buffering* y permite controlar el tamaño del *buffer* que el intérprete utiliza para leer o escribir en el fichero. El segundo parámetro es *encoding* y permite indicar el tipo de codificación de caracteres que deseamos emplear para nuestro fichero. Otro de los parámetros es *errors* que indica cómo manejar errores debidos a problemas derivados de la utilización de la codificación de caracteres. Para controlar cómo tratar los saltos de línea contamos con *newline.* Por último, *closefd* es *True* y si cambiamos su valor a *False* el descriptor de fichero permanecerá abierto aunque explícitamente invoquemos al método *close()* para cerrar el fichero.

Hasta el momento hemos hablado de abrir ficheros, utilizando para ello diferentes parámetros. Pero ¿cómo podemos crear un fichero en nuestro sistema de archivos desde Python? Basta con aplicar el modo de *escritura* y pasar el nombre del nuevo fichero, tal y como muestra el siguiente ejemplo:

```
>>> f_nuevo = open(nuevo.txt", "w")
```

Debemos tener en cuenta que, pasando el valor *w* sobre un fichero que ya existe, este será sobrescrito, borrando su contenido.

Ahora que ya sabemos cómo abrir y crear un fichero, es hora de aprender cómo leer y escribir datos.

Lectura y escritura

La operación básica de escritura en un fichero se hace en Python a través del método *write()*. Si estamos trabajando con un fichero de texto, pasaremos una cadena de texto a este método como argumento principal. Supongamos que vamos a crear un nuevo fichero de texto añadiendo una serie de líneas, bastaría con ejecutar las siguientes líneas de código:

```
>>> fich = open(texto.txt", "w")
>>> fich.write("Primera línea\n")
14
>>> fich.write("Segunda línea\n")
14
>>> fich.close()
```

El carácter "\n" sirve para indicar que deseamos añadir un retorno de carro, es decir, crear una nueva línea. Después de realizar las operaciones de escritura, debemos cerrar el fichero para que el intérprete vuelque el contenido al fichero físico y se pueda también cerrar su descriptor. Si necesitamos volcar texto antes de cerrar el fichero, podemos hacer uso del método *flush()* y, posteriormente, podemos seguir empleando *write(),* sin olvidar finalmente invocar a *close()*. En nuestro ejemplo, observaremos cómo, después de cada operación de escritura, aparece un número. Este indica el número de bytes que han sido escritos en el fichero.

Adicionalmente, el método *writelines()* escribe una serie de líneas leyéndolas desde una lista. Por ejemplo, si deseamos emplear este método en lugar de varias llamadas a *write(),* podemos sustituir el código anteriormente mostrado por este otro:

```
>>> lineas = ["Primera línea\n", "Segunda línea\n"]
>>> fich.writelines(lineas)
```

Ahora que ya tenemos texto en nuestro fichero, es hora de pasar a la operación inversa a la escritura, hablamos de la lectura desde el fichero. Para ello, Python nos ofrece tres métodos diferentes. El primero de ellos es *read(),* el cual lee el contenido de todo el fichero y devuelve un única cadena de texto. El segundo en cuestión es *readline()* que se ocupa de leer línea a línea del fichero. Por último, contamos con *readlines()* que devuelve una lista donde cada elemento corresponde a cada línea que contiene el fichero. Los siguientes

ejemplos muestran cómo utilizar estos métodos y su resultado sobre el fichero que hemos creado previamente:

```
>>> fich = open(texto.txt", "w")
>>> fich.read()
'Primera línea\nSegunda línea\n'
>>> fich.seek(0)
0
>>> fich.readlines(fich)
['Primera línea\n', 'Segunda línea\n']
>>> fich.seek(0)
0
>>> fich.readline()
'Primera línea\n'
```

Seguro que al lector no le ha pasado desapercibido el uso de un nuevo método. Efectivamente, *seek()* es otro de los métodos que podemos usar sobre el objeto de Python que maneja ficheros y que sirve para posicionar el puntero de avance sobre un punto determinado. En nuestro ejemplo, y dado que deseamos volver al principio del fichero, hemos empleado el valor *0*. Debemos tener en cuenta que cuando se hace una operación de escritura, Python maneja un puntero para saber en qué posición deberá ser escrita la siguiente línea con el objetivo de no sobrescribir nada. Sin embargo, este puntero avanza de forma automática y el método *seek()* sirve para situar el mencionado puntero en una determinada posición del fichero.

Para leer todas las líneas de un fichero no es necesario emplear *seek()* tal y como muestra el ejemplo anterior de código. Existe un método más sencillo basado en emplear un *iterator* para ello:

```
>>> for line in open "texto.txt"):
    print(line)
...
Primera línea
Segunda línea
```

Además, también es práctica habitual emplear *with* para leer todas las líneas de un fichero:

```
>>> with open(texto.txt") as fich:
...    print (fich.read())
...
Primera línea
Segunda línea
```

Hasta el momento, nuestros ejemplos se han referido en exclusiva a ficheros de texto, pero, obviamente, es posible crear, abrir, leer y escribir en ficheros binarios. La forma de llevar a cabo estas operaciones es similar a como hemos visto previamente para los ficheros de texto. Por ejemplo, podemos crear un fichero binario que solo contiene una secuencia de bytes, en concreto, vamos a escribir en el fichero tres bytes representados por tres números en hexadecimal. El código necesario para ello sería el siguiente:

```
>>> fich = open("test.bin", "bw")
<_io.BufferedWriter name='test.bin'>
>>> fich.write(b"\x33\xFA\x1E")
3
>>> fich.close()
```

El número *3* que el intérprete devuelve al escribir nuestra secuencia de bytes corresponde, efectivamente, a los tres bytes que hemos escrito en el fichero. Recordemos, que cada número en formato hexadecimal ocupa justamente un byte. Para leer el contenido que acabamos de escribir, volveremos a abrir el fichero, esta vez en modo de solo lectura:

```
>>> fich = open("test.bin", "br")
>>> fich.read(3)
b'\x33\xFA\x1E'
```

En este caso, el parámetro pasado al método *read()* indica el número de bytes que deseamos leer. Si hubiéramos utilizado *1* en lugar de *3*, entonces, el resultado hubiera sido *b'\x33'*. El fichero que hemos creado puede ser leído por un editor que soporte la lectura y edición de este tipo de ficheros.

El método *seek()* suele ser utilizado con frecuencia para desplazarse por un fichero binario, a diferencia de los de texto, que habitualmente suelen ser leídos línea a línea. El mencionado método soporta dos tipos de parámetros diferentes, el primero de ellos indica el número de bytes que debe moverse el puntero interno de señalización del fichero y, el segundo, marca el punto de referencia desde el que debe ser movido dicho puntero. Este segundo parámetro puede tomar tres valores: 0 (comienzo del fichero), 1 (posición actual) y 2 (fin de fichero). Por defecto, si no se indica este parámetro, el valor es 0. De esta forma, para leer el último byte de nuestro fichero, emplearíamos las siguientes sentencias:

```
>>> fich.seek(-1, 2)
```

```
2
>>> fich.read(1)
b'\xle'
```

Cuando se emplea como punto de referencia el final del fichero, se deben indicar valores negativos para el desplazamiento, tal y como habremos podido comprobar en el ejemplo anterior de código.

Relacionado con la creación, lectura y escritura de ficheros binarios se encuentra el concepto de *serialización* de objetos del que nos ocuparemos en el siguiente apartado.

SERIALIZACIÓN

La *serialización* es un proceso mediante el cual una estructura de datos es codificada de un modo específico para su almacenamiento, que puede ser físicamente en un fichero, una base de datos o un *buffer* de memoria. Habitualmente, la serialización se lleva a cabo utilizando instancias de objetos, aunque son muchos los lenguajes de programación que permiten aplicar este proceso a cualquier estructura de datos que cumpla ciertas condiciones. El propósito de este proceso, además del almacenamiento propiamente dicho, suele ser el transporte de datos a través de una red o, simplemente para crear una copia exacta de un objeto determinado. En computación distribuida es muy común señalizar objetos para su transporte entre diferentes máquinas, también es habitual emplear la serialización en protocolos como *CORBA (Common Object Request Broker Architecture),* que permite invocar a métodos y funciones escritos en un lenguaje determinado desde otro diferente. En general, el término de *serialización* es conocido como *marshalling* en el ámbito de las ciencias de la computación.

Diferentes lenguajes de programación emplean diferentes algoritmos para señalizar. Python soporta la serialización de objetos, a través de dos módulos diferentes de su librería estándar: *pickle* y *marshal.* Dado que los algoritmos empleados por estos módulos son diferentes, es necesario escoger uno de ellos a la hora de señalizar datos en Python. Al proceso de convertir un objeto cualquiera en un conjunto de bytes es llamado en Python *pickling*, siendo el proceso inverso llamado *unpickling.* Es por ello, que habitualmente, el módulo *pickle* es el más utilizado para la serialización de objetos en Python.

Debemos tener en cuenta que el módulo *marshal* señaliza de una forma que no es compatible entre Python 2.x y Python 3; sin embargo, *pickle* nos garantiza la portabilidad del código entre versiones del intérprete. Por otro lado, *marshal* no puede ser utilizado para señalizar las clases propias definidas por el programador, mientras que esto sí que es posible con *pickle.* Dado que el formato en el que son señalizados los datos, a través de *pickle,* es específico de Python, no es posible deserializar aquellos objetos señalizados con este módulo empleando otros lenguajes de programación.

La versión 3 de Python incluye cuatro protocolos diferentes con los que puede trabajar *pickle.* Cada uno de ellos es identificado por un número entre 0 y 3. El primero de ellos es compatible con versiones anteriores a Python 3 y serializa utilizando un formato *human-redeable.* El segundo de los protocolos emplea un formato binario y es compatible con las primeras versiones de Python. El tercero, identificado por el número 2, fue incluido por primera vez en Python 2.3 y es mucho más eficiente que sus antecesores. Por último, contamos con uno nuevo incluido en Python 3.0, que no puede ser utilizado por la serie 2.x de Python para la deserialización y que es actualmente el recomendado para la serie 3.x del lenguaje. Además, el mencionado módulo incluye dos constantes diferentes: *HIGHEST_PROTOCOL*, que se identifica a la versión más alta disponible y *DEFAULT_PROTOCOL,* que actualmente está asociada al protocolo 3.

Respecto a los tipos de datos que pueden ser señalizados en Python con *pickle,* debemos tener en cuenta que son los siguientes:

- Aquellos que toman los valores *True*, *False* y *None.*
- Números enteros, complejos y reales.
- Cadenas de texto, *bytes* y listas de bytes.
- Funciones definidas en el nivel superior de un módulo.
- Funciones integradas en el intérprete que residan en el nivel superior de un módulo.
- Tuplas, listas, conjuntos y diccionarios que contengan elementos que pertenezcan a los tipos anteriores.
- Clases definidas por el programador que contengan elementos que pertenezcan a los tipos anteriores.

Como complemento a *pickle,* Python pone a nuestra disposición otro módulo llamado *pickletools,* que contiene una serie de herramientas para analizar los datos en el formato generado al serializar con *pickle.*

Ejemplo práctico

Básicamente, para serializar objetos en Python, a través del módulo *pickle*, emplearemos dos funciones diferentes: *dump()* para serializar y *load()* para la operación inversa. Para nuestro ejemplo práctico vamos a crear una clase, similar a la del capítulo anterior que representaba a un empleado. En este caso vamos a definir una clase que representa a un alumno, tal y como muestra el siguiente código:

```
>>> class Alumno:
...   def __init__(self, nombre_completo, titulación, edad):
...      self.apellidos = nombre_completo['apellidos']
...      self.nombre = nombre_completo['nombre']
...      self . titulación = titulación
...      self.edad = edad
...   def __repr__(self):
...      return repr(self.nombre, self.apellidos, self.titulación)
```

Como el lector habrá podido adivinar, vamos a emplear cadenas de texto, enteros y un diccionario como tipos de datos que contendrá la Instancia de nuestra clase. Dado que todos cumplen con las condiciones para señalizar, no tendremos problema para llevar a cabo este proceso. Antes de ello, crearemos una Instancia que será la señalizada y, posteriormente, deserializada:

```
>>> nombre_completo = {"apellidos": "Rodríguez", "nombre": "Lucas"}
>>> alumno = Alumno(nombre_completo, "Grado en Derecho", 21)
```

El siguiente paso será llevar a cabo la serialización, previamente importaremos el módulo *pickle* y posteriormente Invocaremos a la función *dump().* Los datos de nuestra Instancia serán señalizados en un fichero binario llamado *alumnos.bin.* Efectivamente, los ficheros binarlos suelen ser los empleados para llevar a cabo este proceso, de ahí la relación entre este tipo de ficheros y la serialización, tal y como habíamos avanzado en el apartado anterior del presente capítulo. En cuanto al código, el siguiente nos muestra cómo crear el fichero y aplicar *dump():*

```
>>> import pickle
>>> with open(alumnos.bin", "wb") as fich:
pickle.dump(alumno, fich)
```

Para llevar a cabo el proceso inverso, es decir, la deseríalízacíón, es necesario invocar al método *load().* Así pues, y siguiendo con nuestro ejemplo, para leer los datos que acabamos de señalizar, basta con ejecutar las siguientes líneas de

código:

```
>>> with open(alumnos.bin", "rb") as fich:
...   pickle.load(fich)
Lucas Rodríguez Grado en Derecho 21
```

Es importante tener en cuenta que cuando se emplea la función *load()* para deseríalízar los datos de una clase, esta debe ser accesible en el mismo ámbito. En nuestro ejemplo y dado que hemos utilizado la consola de comandos del intérprete, cumplimos con esta condición.

FICHEROS XML, JSON Y YAML

Con la *serialización* se encuentran relacionados una serie de formatos de ficheros que suelen ser utilizados para guardar datos en un determinado formato y, que pueden compartirse entre diferentes máquinas y tratarse en distintos lenguajes de programación. Entre estos formatos, destacan tres: XML, JSON y YAML. En este apartado descubriremos cómo podemos trabajar con estos formatos desde Python. Comenzamos por uno de los más populares, el XML.

XML

Estas tres siglas vienen de *eXtensible Markup Language* y referencian a un lenguaje de etiquetas desarrollado por el WC3 *(World Wide Web Consortium).* En realidad, este lenguaje no es más que una adaptación del original SGML *(Standard Generalized Markup Language)* y fue diseñado con el objetivo de tener una herramienta que fuera capaz de ayudar a definir lenguajes de marcas y etiquetas, adaptados a diferentes necesidades. Además, también puede ser utilizado como un estándar para el intercambio de datos estructurados. De hecho, es esta una de las aplicaciones más usuales del formato XML. Dado que es posible definir de forma personalizada una estructura concreta de datos, estos pueden ser almacenados en un simple fichero de texto, haciendo el mismo totalmente portable entre máquinas y lenguajes de programación. En la práctica, podemos almacenar en un fichero XML desde información relativa a una configuración específica, hasta datos que habitualmente podrían almacenarse en una base de datos. Es más, comúnmente el formato XML es empleado para serializar datos y existen multitud de librerías que permiten, dada una definición de un objeto, volcarlo directamente a un fichero en este formato. En la actualidad, XML es un formato estándar *de facto,* empleado por multitud de aplicaciones web y de escritorio.

Los datos en formato XML pueden ser accedidos a través de dos interfaces diferentes: DOM *(Document Object Model)* y SAX *(Simple API for XML).* La primera de ellas se basa en utilizar y tratar los datos basándose en una estructura

de árbol, donde se definen nodos y una jerarquía que permite acceder a los diferentes elementos que forman parte de los datos contenidos en XML. Por otro lado, SAX está orientado a eventos y permite trabajar en un sección del documento XML en lugar de hacerlo en todo el conjunto, tal y como necesita DOM.

Son muchos los lenguajes de programación que ofrecen herramientas para trabajar con XML y Python se encuentra entre ellos. En concreto, el módulo *xml* de la librería estándar pone a disposición de los programadores diferentes analizadores sintácticos para leer y escribir información en formato XML Tres son los submódulos que permiten interactuar con XML de formas diferentes. El primero de ellos es *xml.sax.handler* que utiliza el módulo SAX para analizar sintácticamente (*parsear*). El segundo se denomina *xml.dom.minidom* que ofrece una implementación ligera para el modelo DOM. El último de los tres es *xml.etree.ElementTree* y básicamente ofrece la misma funcionalidad que *xml.dom,* solo que empleando una forma específica de Python para *parsear* el formato.

La estructura de un documento XML puede ser muy compleja; sin embargo, para nuestros ejemplos prácticos definiremos un sencillo documento y veremos cómo puede ser accedido empleando los tres módulos diferentes que pone Python a nuestra disposición. El lector interesado en profundizar en todas y cada una de las opciones existentes de estos módulos, puede echar un vistazo a la documentación oficial de Python al respecto (ver referencias).

Comenzamos definiendo un documento XML con el contenido que representa a la información de un varios álbumes musicales de un artista determinado:

```
<albums>
    <interprete>U2</interprete>
    <titulo>Achtung Baby</titulo>
    <titulo>Zooropa</titulo>
    <titulo>The Joshua Tree</titulo>
    <genero>rock</genero>
</albums>
```

Una vez definida la estructura, podemos guardar el contenido en un nuevo fichero llamado *albums.xml*. Seguidamente, veremos cómo mostrar la información relativa a los diferentes títulos que contiene, empleando para ello los diferentes módulos de Python para análisis de XML. Comenzamos por

ElementTree:

```
>>> from xml.etree.ElementTree import parse
>>> xml_doc = parse('albums.xml')
>>> for ele in xml_doc.findall('titulo'):
...   print(ele.text)
...
Achtung Baby
Zooropa
The Joshua Tree
```

La función *parse()* se encarga de cargar y la estructura del fichero en memoria y prepararla para poder recorrerla y acceder a su información. El método *findall()* localiza todas las etiquetas que contiene el nombre indicado, y, finalmente, para acceder a la información de los títulos en cuestión, utilizamos *text* que es un atributo que contiene el valor en cuestión.

De funcionalidad análoga al ejemplo anterior tenemos el siguiente código que emplea las herramientas de *minidom* para acceder e imprimir los títulos de nuestro fichero XML:

```
>>> from xml.dom.minidom import parse, Node
>>> xmltree = parse('albums.xml')
>>> for nodo in xmltree.getElementsByTagName('titulo'):
...   for nodo_hijo in nodo.childNodes:
...      if nodo_hijo.nodeType == Node.TEXT_NODE:
...         print(nodo_hijo.data)
...
Achtung Baby
Zooropa
The Joshua Tree
```

Observando el código anterior, descubriremos cómo pasando el parámetro *titulo al* método *getElementsByTagName()* podemos declarar un bucle para recorrer uno a uno todos los nodos que contiene que cumplen con la condición especificada. Además, anidamos otro bucle para recorrer todos los hijos de estos nodos. Dentro de este último bucle *for* debemos comprobar si el nodo es de tipo *texto*, en cuyo caso accederemos al atributo *data* que contiene el valor que buscamos. La constante *TEXT_NODE* representa el tipo texto que contiene una etiqueta determinada. En nuestro caso, corresponde al título en cuestión de cada disco.

El último de nuestros ejemplos muestra cómo emplear el método SAX para recorrer nuestro fichero e imprimir el título de cada álbum. Para procesar el

fichero vamos a crear una clase que se encargará de responder a los diferentes eventos que se producen cuando se utiliza el *parser* de SAX. Así pues, nuestra clase estará definida por el siguiente código:

```
import xml.sax.handler
class AlbumSaxHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
        self.in_title = False

    def startElement(self, name, attributes):
        if name == 'titulo':
            self.in title = True
    def characters(self, data):
        if self.in_title:
            print(data)

    def endElement(self, name):
        if name == 'titulo':
            self.in_title = False
```

La clase *AlbumSaxHandler* hereda de una clase incluida en el módulo que Python incluye para trabajar con SAX. Los métodos *startElement()* y *endElement()* son los encargados de llevar a cabo una serie de acciones cuando se detecta el principio y el fin, respectivamente, de cada etiqueta del fichero XML. Por otro lado, *characters()* comprobará si el atributo de clase *in title* es *True*, en cuyo caso imprimirá el contenido del elemento referenciado por *<titulo>*. Una vez que tenemos nuestra clase, solo nos queda invocar al *parser*, para ello necesitaremos el siguiente código:

```
>>> import xml.sax
>>> parser = xml.sax.make_parser()
>>> sax_handler = AlbumSaxHandler()
>>> parser.setContentHandler(sax_handler)
>>> parser.parse('albums.xml')
Achtung Baby
Zooropa
The Joshua Tree
```

La elección de la técnica y módulo de Python para utilizar a la hora de *parsear* y trabajar con XML's depende de varios factores. Dos de los más importantes son el tamaño del fichero y el tipo de estructura que presenta. Estos tienen gran impacto sobre la capacidad de procesamiento y memoria requerida para su análisis.

JSON

El formato *JSON* se ha convertido en uno de los más populares para la serialización e intercambio de datos, sobre todo en aplicaciones web que utilizan AJAX. Recordemos que esta técnica permite intercambiar datos entre el navegador cliente y el servidor sin necesidad de recargar la página. JSON son las siglas en inglés de *JavaScript Object Notation* y fue definido dentro de uno de los estándares (ECMA- 262) en los que está basado el lenguaje JavaScript. Es en este lenguaje donde, a través de una simple función (*eval()),* podemos crear un objeto directamente a través de una cadena de texto en formato JSON. Este factor ha contribuido significativamente a que sean muchos los servicios web que utilizan JSON para intercambiar datos a través de AJAX, ya que el análisis y procesamiento de datos es muy rápido. Incluso, son muchas las que han sustituido el XML por el JSON a la hora de intercambiar datos entre cliente y servidor.

Para estructurar la información que contiene el formato, se utilizan las llaves *({})* y se definen pares de nombres y valor separados entre ellos por dos puntos. De esta forma, un sencillo ejemplo en formato JSON, para almacenar la información de un empleado, sería el siguiente:

```
{"apellidos": "Fernández Rojas", "nombre": "José Luis",
"departamento": "Finanzas", "ciudad": "Madrid"}
```

JSON puede trabajar con varios tipos de datos como valores; en concreto, admite cadenas de caracteres, números, *booleanos*, arrays y *null.*

La mayoría de los modernos lenguajes de programación incluyen API y/o librerías que permiten trabajar con datos en formato JSON. En Python disponemos de un módulo llamado *json* que forma parte de la librería estándar del lenguaje. Básicamente, este método cuenta con dos funciones principales, una para *codificar* y otra para *descodificar.* El objetivo de ambas funciones es *traducir* entre una cadena de texto con formato JSON y objetos de Python. Obviamente, utilizando las funciones de ficheros de Python podemos leer y escribir ficheros que contengan datos en este formato. No obstante, debemos tener en cuenta que las funciones contenidas en *json* no permiten trabajar directamente con ficheros, sino con cadenas de texto.

Volviendo a nuestro ejemplo de fichero XML, los mismos datos pueden ser

codificados en formato JSON de la siguiente forma:

```
{"albums": {"titulos": ["Achtung Baby", "Zooropa", "The Joshua
Tree"], "genero": "Rock"}}
```

La anterior cadena puede ser salvada en un fichero llamado *albums.json*, siendo este el que utilizaremos para nuestros posteriores ejemplos.

Para leer los datos de nuestro nuevo fichero, basta con ejecutar las siguientes líneas de código:

```
>>> import json
>>> fich = open('albums.json')
>>> line = fich.readline()
>>> fich.close()
>>> data = json.loads(line)
```

La variable *data* contendrá un diccionario de Python que se corresponde con la estructura de nuestro fichero JSON. Así pues, para obtener los títulos de nuestros álbumes basta con ejecutar:

```
>>> data['albums'] ['titulos']
['Achtung Baby', 'Zooropa', 'The Joshua Tree']
```

Realizar el paso inverso es sencillo, partiendo del nuevo diccionario *data*, podemos directamente invocar a *dumps()*, pasando como argumento nuestro diccionario, de esta forma obtendremos una cadena de texto JSON, que posteriormente, puede ser guardada en un fichero. El siguiente código realizaría todas estas operaciones, incluyendo la modificación de la cadena origina, añadiendo un nuevo título:

```
>>> data['albums'] ['titulos'].append("Rattle and Hum")
>>> cad = json.dumps(data)
>>> new_fich = open('albums_new.json', 'w')
>>> new_fich.writeline(cad)
>>> new_fich.close()
```

Si abrimos el nuevo fichero *albums_new.json* y echamos un vistazo a su contenido, encontraremos el siguiente texto en formato JSON:

```
{"albums": {"títulos": ["Achtung Baby", "Zooropa", "The Joshua Tree",
"Rattle and Hum"], "genero": "Rock"}}
```

Además del módulo *json*, también existen otros módulos para trabajar con

JSON. Uno de ellos es *simplejson* (ver referencias), que no forma parte de la librería estándar pero que puede ser instalado fácilmente. En el capítulo *instalación y distribución de software* nos ocuparemos de cómo instalar módulos adicionales que pueden ser utilizados por nuestros propios programas.

YAML

El último de los formatos de ficheros relacionados con la serialización de datos es YAML, acrónimo de *YAML Ain't Markup Language*. Este formato fue diseñado con el objetivo de disponer de un formato de texto para serializar datos que fuese sencillo y fácil de leer para las personas. A pesar de no ser tan popular como JSON y XML, es interesante conocer este formato, ya que puede ser procesado rápidamente y resulta muy fácil de leer. El conocido *framework* web *Ruby on Rails* emplea el formato YAML para definir determinados ficheros de configuración, como es, por ejemplo, el que declara los datos para la conexión a las diferentes bases de datos de cada entorno.

Básicamente, YAML utiliza pares *clave: valor* para almacenar y estructurar los datos. Además, la indentaclón es empleada para separar datos que pertenecen a otros de una jerarquía superior. Un ejemplo de una cadena YAML podría ser la siguiente, donde se definen tres entornos diferentes y cada uno de ellos cuenta con una dirección IP y puerto diferentes:

```
desarrollo:
    IP: 127.0.0.1
    puerto: 8000
staging:
    IP: 192.168.1.2
    puerto: 8002
producción:
    IP: 192.168.1.2
    puerto: 8003
```

Python no dispone de ningún módulo en su librería estándar para trabajar con ficheros YAML; sin embargo, existen varias librerías *third party* para ello. Una de las más populares es *PyYAML* (ver referencias). Su instalación puede ser llevada a cabo directamente a través de su código fuente, el cual puede ser descargado desde la correspondiente página web (ver referencias). En cuanto

descarguemos el fichero *zip* con el código fuente, pasaremos a descomprimirlo. Seguidamente, desde la línea de comandos y desde el directorio creado al descomprimir, ejecutaremos el siguiente comando:

```
python setup.py install
```

En cuanto termine la ejecución del comando anterior, tendremos disponible un nuevo módulo para Python llamado *yaml*. Esto implica que, desde el intérprete de Python, podemos invocar al nuevo módulo con la siguiente sentencia:

```
>>> import yaml
```

Para la parte práctica partiremos de la cadena YAML que hemos definido previamente, creando un nuevo fichero denominado *servidores.yml*. De forma similar al módulo *json*, *yaml* cuenta con dos funciones principales, una para leer una cadena de texto a través de un fichero y transformarla en un diccionario de Python, y otra que realiza la operación Inversa, es decir, obtiene una cadena de texto a partir de un diccionario de Python. La primera de ellas se denomina *load()* y la segunda *dump()*. Así pues, para leer nuestro fichero YAML, basta con ejecutar la siguiente línea de código:

```
>>> data = yaml.load(open('servidores.yml'))
```

La variable *data* contiene ahora un diccionario donde las claves son *desarrollo*, *staging* y *producción*. Además, cada una de estas claves da acceso a otro diccionario cuyas claves son *IP* y *puerto* que da a su vez acceso a los valores de nuestro fichero. Dada esta estructura, para, por ejemplo, acceder al puerto del entorno de producción, basta con ejecutar la siguiente línea:

```
>>> data['producción'] ['puerto']
8003
```

Por otro lado, la estructura puede ser modificada y crear con ella un nuevo fichero YAML. Supongamos que necesitamos añadir un nuevo entorno llamado *pruebas*, junto con los datos de un nuevo servidor. En primer lugar, crearíamos un nuevo diccionario y lo asignaríamos a *data*, tal y como muestra el código a continuación:

```
>>> data['pruebas'] = {'IP': '192.168.2.8', 'puerto': 8004}
```

Seguidamente, invocamos a la función *dump* pasando como argumento nuestro diccionario original:

```
>>> yaml.dump(data)
'desarrollo: {IP: 192.168.2.8, puerto: 8004}\nproduccion: {IP:
192.168.1.2, puerto: 8003}\npruebas: {IP: 192.168.1.8, puerto:
8004}\n'\nstaging: {iP: 192.168.1.2, puerto: 8002}\n'
```

Ahora podemos guardar nuestra cadena YAML en un nuevo fichero:

```
>>> fich = open('new_servidores', 'w')
>>> fich.write(yaml.dump(data))
>>> fich.close()
```

FICHEROS CSV

El formato CSV *(Comma Separated Values)* es uno de los más comunes y sencillos para almacenar una serle de valores como si de una tabla se tratara. Cada fila se representa por una línea diferente, mientras que los valores que forman una columna aparecen separados por un carácter concreto. El más común de los caracteres empleados para esta separación es la coma, de ahí el nombre del formato. Sin embargo, es habitual encontrar otros caracteres como el signo del dólar (*$*) o el punto y coma (*;*). Gracias al formato CSV es posible guardar una serle de datos, representados por una tabla, en un simple fichero de texto. Además, software para trabajar con hojas de cálculo, como, por ejemplo, *Microsoft Excel*, nos permiten importar y exportar datos en este formato.

Dentro de su librería estándar, Python incorpora un módulo específico para trabajar con ficheros CSV, que nos permite, tanto leer datos, como escribirlos. Son dos los métodos básicos que nos posibilitan realizar estar operaciones: *reader()* y *writer()*. El primero de ellos sirve para leer los datos contenidos en un fichero CSV, mientras que el segundo nos ayudará a la escritura.

Supongamos que tenemos un sencillo fichero CSV, llamado *empleados.csv*, que contiene las siguientes líneas:

```
Martínez,Juan,Administración,Barcelona
López,María,Finanzas.Valencia
Rodríguez,Manuel.Ventas,Granada
Rojas,Ana,Dirección,Madrid
```

Cada línea de nuestro fichero identifica a un empleado, donde, el primer valor es el primer apellido, el segundo es el departamento donde trabaja y por último, el tercer valor es la ciudad en la que se encuentra. Para leer este fichero y mostrar la información indicando el nombre de cada campo y su valor asociado, bastaría con ejecutar el siguiente código:

```
>>> import csv
>>> with open('empleados.csv') as f:
...   reader = csv.reader(f)
...   for row in reader:
...      print("Apellido: {O}; Nombre: {1}; Departamento: {2};
Ciudad: {3}".format(row[0], row[1], row[2], row[3]))
```

```
...
Apellido: Martínez; Nombre: Juan; Departamento: Administración;
Ciudad: Barcelona
Apellido: López; Nombre: María; Departamento: Finanzas; Ciudad:
Valencia
Apellido: Rodríguez; Nombre: Manuel; Departamento: Ventas; Ciudad:
Granada
Apellido: Rojas; Nombre: Ana; Departamento: Dirección; Ciudad: Madrid
```

Efectivamente, el método *reader()* es de tipo *iterable* y nos da acceso a todas las filas del fichero, es por ello, que en nuestro ejemplo, utilizamos un bucle *for* para iterar sobre el objeto devuelto. Por otro lado, cada elemento *iterable* es una lista que contiene tantos elementos como valores separados por el carácter de separación tenga cada línea de nuestro fichero. En caso de que empleemos un carácter de separación diferente de la coma, deberemos indicarlo a través del parámetro *delimiter.* Si cambiamos nuestro fichero *empleados.csv* y reemplazamos la coma por, por ejemplo, el símbolo del dólar, tendríamos que emplear *delimiter* de la siguiente forma:

```
>>> reader = csv.reader(f, delimiter='$')
```

La operación de escritura en ficheros de tipo CSV se lleva a cabo a través de los métodos *writer()* y *writerow().* Este último escribe directamente una fila en el fichero y como argumento debemos pasar una lista donde cada elemento representa cada valor de la columna correspondiente a cada fila. Volvamos a tomar un par de líneas de nuestro ejemplo anterior y creemos un nuevo fichero a partir de ellas:

```
>>> fich = open('nuevo.csv', 'w')
>>> fich_w = csv.writer(fich, delimiter='$')
>>> empleados = [['Martínez', 'Juan', 'Administración', 'Barcelona'],
['López', 'María', 'Finanzas', 'Valencia']]
>>> for empleado in empleados:
...   fich_w.writerow(empleado)
...
>>> fich.close()
```

En esta ocasión hemos creado un nuevo fichero *(nuevo.csv)* que contiene solo los datos de un par de empleados; además, el carácter de separación es el símbolo del dólar. Si abrimos el fichero recién creado por código, veremos que tiene el siguiente contenido:

```
Martinez$Juan$Administracion$Barcelona
Lopez$Maria$Finanzas$Valencia
```

Finalizamos en este punto nuestro recorrido por el tratamiento de ficheros CSV con Python. El lector interesado en descubrir más sobre el módulo *csv* puede echar un vistazo a la página web oficial del mismo (ver referencias).

ANALIZADOR DE FICHEROS DE CONFIGURACIÓN

En los sistemas Windows es muy popular el formato INI, que permite almacenar una serie de datos en formato *propiedad = valor.* Además, se pueden agrupar los datos por secciones, siendo representada cada una de ellas por un nombre entre corchetes (*[]*). El registro de Windows emplea un formato ligeramente diferente al INI, pero inspirado claramente en el mismo.

Estos tipos de ficheros no son exclusivos de Windows, ya que son muchas las aplicaciones que las utilizan, sobre todo, para guardar datos relativos a cierta configuración. Por ejemplo, supongamos que tenemos que guardar la información de una serie de servidores, incluyendo el usuario y puerto por defecto empleados para la conexión. Podríamos utilizar una sección por servidor y dos propiedades diferentes, una para el usuario y otra para el puerto. El fichero INI, por ejemplo, *conf.ini*, sería como sigue:

```
[server1.dominio1]
user = first
port = 22

[server2,dominio2]
user = second
port = 88
```

En lugar de guardar datos de configuración en una base de datos o de tenerlos directamente en el código *(hardcoded)*, los ficheros INI son muy prácticos para almacenar y actualizar valores sin necesidad de cambiar el código fuente o de acceder a una base de datos.

Para el manejo de estos ficheros INI, Python pone a nuestra disposición un módulo llamado *configparser,* que forma parte de su librería estándar. Este módulo incorpora una clase base denominada *ConfigParser* que nos permite tanto leer como crear ficheros del mencionado tipo.

Nuestra práctica va a comenzar importando el módulo y cargando el fichero que hemos creado previamente:

```
>>> from configparser import ConfigParser
>>> config = ConfigParser()
>>> config.read("config.ini")
```

Ahora podemos, por ejemplo, obtener una lista con todas las secciones de nuestro fichero:

```
>>> config.sections()
['server1.dominio1', 'server2.dominio2']
```

Una vez que tenemos nuestra instancia *config* de la clase *ConfigParser,* podemos acceder a los valores de cada sección. Para ello, bastará con tener en cuenta que cada nombre de sección es la clave de un diccionario que, a su vez, nos proporciona acceso a otro diccionario donde cada clave es el valor de la propiedad. Al acceder a esta segunda clave dispondremos del valor correspondiente de la correspondiente propiedad de la sección. De esta forma, para acceder al valor *port* de la segunda sección de nuestro fichero de ejemplo, bastará con ejecutar la siguiente sentencia:

```
>>> config = ['server2.dominio2']['port']
88
```

Análogamente, si deseamos obtener el valor de la propiedad *user* de la primera sección del fichero, emplearíamos el siguiente código:

```
>>> config['server1.dominio1'] ['user']
first
```

Además de leer las propiedades, obviamente, también podemos escribir otras nuevas. Supongamos que tenemos que añadir una nueva sección con una serie de propiedades y valores diferentes. El primer paso sería crear un nuevo diccionario vacío y posteriormente añadirle las nuevas propiedades y valores, tal y como muestran las siguientes líneas:

```
>>> config['server3.dominio3'] = {}
>>> config['server3.dominio3'] ['protocol'] = ’ssh'
>>> config['server3.dominio3'] ['timeout'] = '30'
```

Seguidamente, deberemos abrir el fichero original y escribir en él a través del método *write()* que pertenece a la clase *ConfigParser.*

```
>>> with open ('config.ini', 'w') as fich:
...   config.write(fich)
...
```

Si ahora abrimos nuestro fichero INI con cualquier editor de textos,

comprobaremos cómo contiene las siguientes líneas:

```
[server1.dominio1]
ser = first
port = 22

[server2.dominio2]
user = second
port = 88

[server3.dominio3]
protocol = ssh
timeout = 30
```

Por otro lado, y dado que cada sección está representada por un diccionario, también es posible obtener un listado con las propiedades que tiene una determinada sección. El siguiente código nos muestra las propiedades para la sección que hemos creado anteriormente:

```
>>> for propiedad in config['server3.dominio3']:
...   print(propiedad)
...
protocol
timeout
```

Aunque hemos trabajado con una estructura básica de ficheros INI, estos pueden contener otra ligeramente diferente. Para averiguar cómo está definida esta estructura, aconsejamos visitar la página oficial de la documentación de Python al respecto (ver referencias).

COMPRESIÓN Y DESCOMPRESIÓN DE FICHEROS

Python nos ofrece la opción de trabajar con distintos formatos de compresión para archivos. Es decir, es posible crear ficheros comprimidos que contengan uno o varios ficheros a su vez, y descomprimir un archivo para extraer su contenido. En concreto, la librería estándar de Python cuenta con cuatro módulos diferentes que permiten trabajar con los formatos estándar de compresión ZIP (.zip), *tarball* (.tar), *gzip* (.gz) y *bunzip* (.bz2). A continuación, veremos una serie de ejemplos para comprimir y descomprimir ficheros utilizando cada uno de estos formatos.

Formato ZIP

El nombre del módulo de la librería estándar que nos permite trabajar con este formato se llama *zipfile*. En concreto, podemos crear, leer, escribir, añadir y listar el contenido de un fichero con este formato. Respecto a la encriptación, aún no es posible crear un fichero encriptado y, aunque la desencriptación es posible, no es recomendable, dado que el proceso requiere de mucho tiempo de procesamiento. El módulo *zipfile* también soporta la creación de ficheros ZIP de tamaño superior a 4 GB.

Dentro del mencionado módulo, la principal clase es *Zipfile* y encapsula la mayoría de operaciones que pueden ser llevadas a cabo sobre los ficheros ZIP. Adicionalmente, existe otra clase llamada *PyZipFile*, muy similar a la anteriormente mencionada, pero con una serie de mejoras sobre ella. Por otro lado, cuando invocamos a los métodos de *Zipfile*, que nos dan información sobre el contenido de un fichero comprimido, estos nos devuelven como resultado un objeto de tipo *ZipInfo*, el cual encapsula la respuesta ofrecida por los métodos en cuestión.

Comenzaremos nuestra práctica creando un fichero ZIP que contendrá una serie de ficheros. Podemos elegir varios ficheros que ya tengamos en nuestro disco, tanto binarios como de texto. El siguiente código muestra cómo crear el fichero *first.zip* con tres ficheros diferentes:

```
>>> import zipfile
>>> with zipfile.ZipFile('first.zip', 'w') as fzip:
...      fzip.write('empleados.csv')
...      fzip.write('fich.txt')
    fzip.write('test.dat')
```

Si ahora abrimos el recién creado fichero *first.zip* con una herramienta como *WinZip*, *Winrar* o similar, comprobaremos cómo, efectivamente el fichero comprimido contiene los tres ficheros que hemos elegido y comprimido desde nuestro código Python. Además, dado que es posible, listar el contenido desde Python, será este nuestro siguiente paso, para el que emplearemos el siguiente código:

```
>>> fzip = zipfile.ZipFile('first.zip')
>>> fzip.printdir()
Filename       Modified             Size
empleados.csv  2012-01-07 21:47:14  133
```

```
fich.txt       2012-02-07 22:32:02    9
test.dat       2012-03-07 20:24:54    1
```

En este último ejemplo de código, el método *Zipfile* solo recibe un único argumento, el nombre del fichero, ya que, por defecto, el fichero se abre en modo de solo lectura. Sin embargo, cuando hemos invocado al mismo método para crear el fichero, el valor *w* ha sido pasado como parámetro para indicar que el fichero debe ser creado. Este *modo* es similar al que utiliza la función *open(),* tal y como hemos aprendido previamente.

Relacionado con el método *printdir(),* encontramos a *namelist(),* el cual nos devuelve una lista que contiene un elemento por fichero contenido en el ZIP original. Podemos invocarlo directamente:

```
>>> fzip.namelist()
['empleados.csv', 'fich.txt', 'test.dat']
```

La operación inversa a la compresión es la descompresión y, para ello, la clase *Zipfile* cuenta con los métodos *extract()* y *extractall()*. El primero de ellos extrae solo uno de los ficheros que se encuentran comprimidos, mientras que el segundo se encarga de extraer todo el contenido del fichero ZIP. Por ejemplo, si ejecutamos el siguiente código, comprobaremos cómo se crean los tres ficheros de los que consta el fichero ZIP que hemos creado previamente:

```
>>> fzip.extractall()(path="..")
```

El paramétro *path* sirve para indicar en qué directorio del sistema de ficheros queremos que sean descomprimidos los ficheros. En nuestro ejemplo, hemos elegido el directorio padre desde el que se está ejecutando la consola del intérprete de Python. Adicionalmente, si el fichero hubiera sido creado con una *password*, podemos indicar la misma a través del parámetro *pwd*.

Como hemos mencionado previamente, una instancia de la clase *ZipInfo* es devuelta por los métodos que nos ayudan a obtener información del fichero comprimido; en concreto, dichos métodos son *infolist()* y *getinfo()*. Ambos devuelven el mismo tipo de información, la diferencia reside en que el primer método lo hace sobre todos y cada uno de los ficheros que forman parte del ZIP y el segundo necesita recibir como parámetro uno de los ficheros para devolver la información relativa al mismo. Entre la información que puede ser obtenida desde la clase *ZipInfo*, destacamos el nombre de cada fichero, la fecha y hora de la última modificación, el tipo de compresión y el tamaño que ocupa cada

fichero antes y después de la compresión. Sirvan como ejemplo las siguientes líneas de código que nos ofrecen información relativa al nombre, fecha de última modificación y tamaño de cada fichero comprimido:

```
>>> info = fzip.infolist()
>>> for arch in info:
...   print(arch.filename, arch.date_time,
        arch.compress_size)
...
empleados.csv (2012, 2, 7, 21, 47, 14) 133
fich.txt (2012, 2, 8, 18, 48, 26) 9
test.dat (2012, 2, 8, 18, 51, 6) 1
```

Respecto a la fecha y hora, en realidad, el valor devuelto es una tupla de seis elementos que indican año, mes, día del mes, hora, minutos y segundos.

Para más información sobre otros métodos ofrecidos por el módulo *zipfile*, aconsejamos visitar la documentación oficial del mismo (ver referencias).

Formato gzip

El módulo *gzip* de Python permite comprimir y descomprimir ficheros tal y como lo hacen las herramientas *gzip* y *gunzip*. Ambos programas pertenecen a GNU y son muy populares en el mundo UNIX/Linux. De hecho, la gran mayoría de los sistemas operativos basados en uno u otro incluyen ambas herramientas.

Técnicamente, el módulo *gzip* es una simple interfaz sobre las mencionadas herramientas de GNU, ya que, en realidad, el módulo *zlib* es el que proporciona la compresión real de datos.

La principal clase de *gzip* es *GzipFile* y simula la mayoría de los métodos disponibles para el tipo de dato fichero de Python. De esta forma, los mismos valores para abrir y crear un fichero que emplea la función *open()*, son aplicables al constructor de la mencionada clase.

Para crear un fichero comprimido a partir de un fichero original, basta con abrir el fichero original, invocar al método *open()* del módulo *gzip*, y posteriormente invocar al método *writelines()*, tal y como muestra el siguiente código:

```
>>> import gzip
>>> with open('fich.dat', ' rb') as f_original:
```

```
with gzip.open('fich.txt.gz', 'wb') as fich:
...   fich.writelines(f_original)
...
```

Es importante tener en cuenta que el módulo *gzip* solo trabaja con datos en binario, es decir, que las cadenas de texto no están soportadas a través de los métodos de lectura y escritura. Esta es la razón por la cual nuestro ejemplo ha utilizado el valor *b* para marcar el tipo de modo.

Si necesitamos crear un fichero *gz* a partir de una serie de datos binarios, también podemos hacerlo, en este caso, a través del método *write():*

```
>>> datos_binarios = b"Este string es binario"
>>> with gzip.open('nuevo.gz', 'wb') as fich:
...   fich.write(datos binarios)
...
```

Por otro lado, el módulo *gzip* de Python también nos ofrece la opción de comprimir cadenas de texto, sin necesidad de trabajar con ficheros. De este modo, el siguiente código nos permitiría disponer de una cadena comprimida:

```
>>> datos_binarios = b"Comprimiendo cadena"
>>> comprimidos = gzip.compress(datos_binarios)
```

El método que realiza la operación contraria a la compresión se denomina *decompress()* y funciona de forma inversa a como lo hace *compress().*

Formato bz2

Para trabajar con ficheros en formato *bunzip,* Python nos ofrece el módulo llamado *bz¡p2,* que, básicamente, cuenta con la clase *BZ2File.* Esta representa a un fichero con este tipo de compresión y dispone de métodos para crear, leer, comprimir y descomprimir datos. Desde el punto de vista técnico, este módulo es una interfaz a la librería de compresión *bz2.* A pesar de que este formato no es tan popular como el *tarball* y el ZIP, cada vez son más los que lo emplean, gracias a que es capaz de ajustar bastante el tamaño final de los ficheros comprimidos.

La compresión y descompresión de datos se puede realizar fácilmente a través de los métodos *compress()* y *decompress(),* respectivamente. Por ejemplo,

para comprimir una cadena binaria:

```
>>> import bz2
>>> cad = b"Cadena binaria"
>>> cad_comprimida = bz2.compress(cad)
```

La operación inversa, es decir, la descompresión, se realizaría de forma similar:

```
>>> cad_descomprimida = bz2.decompress(cad_comprimida)
```

De forma similar a la que hemos visto previamente en los ejemplos de *gzip*, podemos crear un fichero comprimido en formato *bz2*, a partir de uno original sin compresión. Para ello, observemos el siguiente código:

```
>>> with open('fich.dat', 'rb') as f_original:
...      with bz2.BZ2File('fich.dat.bz2', 'wb') as fich:
...         fich.writelines(f_original)
```

Nótese que, a diferencia de *gzip*, el módulo *bz2* dispone del método *open()* a través de la clase *BZ2File*.

Formato tarball

El formato *tar* es uno de los más populares para compresión de archivos en sistemas UNIX. En la actualidad, la gran mayoría de las distribuciones de Linux Incorporan por defecto utilidades para trabajar con este formato. El mismo hecho puede ser aplicado para Mac OS X, mientras que en Windows, el formato más popular sigue siendo el ZIP.

Es común encontrar ficheros *tar* a los que, además, se les ha añadido compresión empleando *gzip o bz2*. En realidad, *tar* no comprime por sí mismo, simplemente *agrupa* ficheros. Los ficheros que utilizan *tar* junto a *gzip o bz2* son llamados *tarball* y pueden tener la extensión *tar.gz o tar.bz2*.

El módulo de Python que permite trabajar con *tarballs* se llama *tarfile* y permite comprimir, descomprimir y listar el contenido de este tipo de ficheros. La clase base del módulo se denomina *Tarfile* y funciona deforma similar a *Zipfile*.

Para crear un fichero *tarball* a partir de una serie de ficheros, podemos

emplear el siguiente conjunto de sentencias:

```
>>> import tarfile
>>> ftar = tarfile.open('first.tar.gz', 'w:gz')
>>> ftar.add('empleados.csv')
>>> ftar.add('fich.txt')
>>> ftar.add('test.dat')
>>> ftar.close()
```

Si abrimos el nuevo fichero *first.tar.gz*, veremos que efectivamente contiene los ficheros que hemos añadido. Por ejemplo, para comprobar que la operación se ha realizado correctamente, en la interfaz de comandos de Linux podemos ejecutar:

```
$ tar -zvtf first.tar.gz
```

Como el lector habrá podido observar, el método *open()* ha pasado como parámetro, además del nombre del nuevo fichero, el valor *w:gz*, el cual indica que el fichero debe crearse utilizando *gzip* para comprimir. Si solo indicáramos el valor *w,* simplemente se crearía un fichero *tar,* pero sin compresión. Para evitar errores, es conveniente que los ficheros que no utilicen compresión lleven la extensión *tar* en lugar de *tar.gz.*

Al igual que *ZipFile*, la clase *TarFile* cuenta con los métodos *extract()* y *extractall()* que nos permite extraer todo el contenido del *tarball o* de un fichero concreto que forma parte del mismo. Si deseamos extraer todos los ficheros basta con ejecutar:

```
>>> ftar = tarfile.open('first.tar.gz', 'r:gz')
>>> ftar.extractall()
```

En caso de necesitar listar el contenido del *tarball*, basta con invocar al método *list():*

```
>>> ftar.list()
rw-rw-rw- 0/0  133 2012-01-07 21:47:14  empleados.csv
rw-rw-rw- 0/0    9 2012-02-07 22:32:0   fich.txt
rw-rw-rw- 0/0    1 2012-03-07 20:24:54  test.dat
```

Efectivamente, la primera columna de la salida de *list()* muestra información relativa a los permisos que existen sobre el fichero. Esta es una de las habilidades de los *tarballs*, ya que permiten agrupar y comprimir una serie de ficheros manteniendo la jerarquía de permisos que existe sobre ellos.

Relacionado con el método anteriormente comentado, también disponemos de otro denominado *getnames()*, que nos devuelve una lista con el nombre de los ficheros que pertenecen al *tarball*. Siguiendo con nuestro ejemplo, podemos invocarlo tal y como muestra la siguiente línea de código:

```
>>> ftar.getnames()
['empleados.csv', 'fich.txt', 'test.dat']
```

Hasta aquí todo lo relativo a los ficheros y a su tratamiento. El siguiente capítulo lo dedicaremos a otro aspecto relacionado con el almacenamiento de datos: las bases de datos.

BASES DE DATOS

INTRODUCCIÓN

No cabe duda de que las bases de datos juegan un papel muy importante en el mundo del software. Muchas de las aplicaciones corporativas que se ejecutan diariamente en todo el mundo serían inconcebibles sin el uso de base de datos. Incluso los servicios web que usamos a diario cuentan con una base de datos para almacenar la información que utilizan.

Tradicionalmente, las bases de datos han sido identificadas directamente con un tipo específico, las llamadas *bases de datos relacionales o* RDBMS (*Relational Database Management System).* En realidad, estas denominaciones se refieren al *motor* empleado para gestionar la obtención y almacenamiento de información en ficheros físicos en disco duro. Sin embargo, existen varios tipos de bases de datos, además de las relacionales, como son, por ejemplo, las orientadas a objetos y las *NoSQL*. Estas últimas han ganado gran popularidad en los últimos años y motores como *Cassandra* y *MongoDB* son utilizados por grandes empresas para manejar gran cantidad de información.

En este capítulo nos centraremos en los dos tipos de bases de datos que más se utilizan en la actualidad: las relacionales y las NoSQL. Descubriremos cómo interactuar, desde Python, con varios de los motores más populares de cada uno de estos tipos. En concreto, trabajaremos con MySQL, PostgreSQL, Oracle y SQLite3, en lo que a los relacionales se refiere, y con Cassandra, Redls y MongoDB en el caso de los NoSQL.

Entre las principales operaciones, aprenderemos a conectarnos y desconectarnos de una base de datos, realizar operaciones de consultar e inserción y a mostrar resultados obtenidos como consecuencia de una operación de consulta.

Por otro lado, dentro de la categoría de las bases de datos relacionales, existen una serie de componentes a los que se les conoce como *ORM (Object-Relational Mapping),* que, básicamente, permiten interactuar con una base de datos con independencia de cuál sea su motor. Esto permite desarrollar código portable, ya que, en lugar de utilizar el propio lenguaje SQL, se utilizan una serie

de objetos y métodos que actúan como *wrappers*. Además, los ORM permiten establecer una relación directa entre objetos de nuestro programa y tablas de una base de datos. Ello se logra a través de la técnica conocida como *mapping*. Incluso, las relaciones establecidas entre diferentes clases se corresponden a relaciones entre tablas a través de, por ejemplo, *claves foráneas*. En Python, dos son los módulos más populares que se emplean como ORM: *SQLObject* y *SQLAchemy*. De cómo trabajar con ambos nos ocuparemos también en este capítulo.

Suponemos que el lector está familiarizado con el lenguaje SQL y, al menos, tiene la experiencia básica de trabajar con bases de datos relacionales. Igual suposición haremos para las bases de datos NoSQL. Ello se debe a que no explicaremos fundamentos de ninguno de ambos tipos de bases de datos, sino que nos centraremos en cómo interactuar con ellas desde Python. Además, para ejecutar y comprobar los ejemplos de código de este capítulo, recomendamos que previamente se lleve a cabo la instalación, en la máquina local, de cada motor de base de datos con los que vamos a trabajar. Si esto no es posible, otra opción podría ser la de acceder a una máquina que cuenta previamente con cada motor de base de datos previamente instalado y configurado. Para consultar información al respecto, aconsejamos echar un vistazo a los enlaces referenciados en las referencias para el presente capítulo.

RELACIONALES

Previamente a descubrir cómo interactuar con los gestores de bases de datos relacionales anteriormente mencionados, describiremos una serie de datos que pueden ser almacenados en una tabla y en una base de datos manejada por cualquiera de estos gestores. Posteriormente, utilizaremos el ejemplo de tabla descrito en este apartado como base para trabajar e interactuar con Python.

Para nuestro ejemplo utilizaremos una *entidad* que nos permita representar una serie de países que almacenen información sobre su población, en millones de habitantes, el continente al que pertenece y la moneda que se utiliza en el mismo. Así pues, necesitaremos crear una tabla llamada *países* que cuenta con cuatro atributos principales *(nombre, habitantes, moneda* y *continente*), más uno adicional *(id)* que es un número y que nos servirá para identificar de forma unívoca a cada país. De esta forma, nuestra tabla contendrá varios países, tal y como representamos a continuación:

ID	País	Continente	Habitantes	Moneda
1	España	Europa	47	Euro
2	Alemania	Europa	82	Euro
3	Canadá	América	34	Dólar canadiense
4	China	Asia	1340	Yuan
5	Brasil	América	204	Real

Tabla 7-1. Tabla con información sobre países

Antes de continuar, es conveniente crear una nueva base de datos a la que llamaremos *prueba.* Seguidamente, crearemos una tabla, en la nueva base de datos, utilizando para ello la siguiente sentencia SQL:

```
CREATE TABLE países(
    'id' INT(11) NOT NULL AUTOINCREMENT,
    'nombre' VARCHAR(1OO) NOT NULL,
    'continente' VARCHAR(20) NOT NULL,
    'habitantes' INT(11) NOT NULL,
    'moneda' VARCHAR(30) NOT NULL,
    PRIMARY KEY ('id’))
```

Debemos tener en cuenta que la anterior sentencia SQL puede variar en función del gestor de base de datos. Basta con comprobar la sintaxis para asegurarnos de que el campo 'id' será la *clave* de nuestra tabla y de que los tipos de cada uno de los otros campos coinciden con los indicados por la nuestra sentencia SQL.

Ahora que ya tenemos tanto la nueva base de datos, como la tabla de ejemplo de países, podemos pasar a los siguientes apartados donde aprenderemos a conectarnos a la base de datos, insertar registros en la nueva tabla y a extraer datos de la misma.

MySQL

La interacción de Python con MySQL puede ser llevada a cabo a través de un módulo llamado *MySQLdb*. Dado que este no forma parte de la librería estándar, necesitaremos instalarlo en nuestro sistema. Esta instalación puede ser llevada a cabo a través del gestor *pip*, del que nos ocuparemos en el capítulo 9 *Instalación y distribución de paquetes.* De momento, daremos por supuesto que tenemos instalado este gestor de paquetes para Python. Así pues, para la instalación de *MySQLdb*, bastará con ejecutar el siguiente comando desde una terminal:

```
pip install MySQLdb
```

Una vez que tengamos nuestro módulo instalado, podemos invocarlo como si de un módulo de la librería estándar se tratara. De esta forma, dentro del intérprete de Python ejecutaremos:

```
>>> import MySQLdb
```

Ya estamos listos para ¡nteractuar con el gestor de base de datos. Nuestra primera operación será obtener una conexión, para ello lanzaremos la siguiente sentencia:

```
>>> con = MySQLdb.connect('localhost', 'usuario', 'password',
'prueba')
```

La función *connect()* recibe como parámetros el nombre de la máquina que está ejecutando el gestor de base de datos, el usuario para conectarnos a nuestra

base de datos, su contraseña y el nombre de la base de datos en cuestión. Si el servidor de base de datos corre en nuestra máquina local, el valor *localhost* debe ser el indicado. Por otro lado, *usuario* es el valor de ejemplo que hemos elegido para el nombre del usuario en cuestión y *password* el valor para su contraseña. Obviamente, debemos sustituir estos valores por los reales de nuestra base de datos. Como respuesta a la invocación de *connect()* obtendremos un objeto que representa una conexión a la base de datos, a partir de la cual podremos realizar diversas operaciones.

Tanto para ejecutar una sentencia SQL para actualizar, Insertar o borrar datos, como para consultar datos, necesitaremos un objeto específico llamado *cursor.* Así pues, antes de realizar cualquiera de estas operaciones, procedemos a la obtención del mismo:

```
>>> cursor = con.cursor()
```

Ahora que ya disponemos del objeto *cursor,* podemos, por ejemplo, insertar un registro:

```
>>> sql = 'INSERT INTO(nombre, continente, habitantes, moneda) VALUES
('España', 'Europa', 47, 'Euro')'
>>> cursor.execute(sql)
```

Los demás países de nuestro ejemplo pueden ser insertados de la misma forma, cambiando, lógicamente, la sentencia SQL. El procedimiento para ejecutar sentencias SQL de tipo *UPDATE o DELETE* es similar, basta con escribir la sentencia en cuestión e invocar a *execute().* Sin embargo, si empleamos una sentencia de tipo *SELECT* deberemos manejar el resultado devuelto a través de la llamada a métodos adicionales. Supongamos, por ejemplo, que necesitamos obtener todos los países de Europa y mostrar toda la información relativa a los mismos. El código necesario para ello sería el siguiente:

```
>>> sql = 'SELECT * FROM paises WHERE continente='Europa''
>>> cursor.execute(sql)
>>> rows = cursor.fetchall()
>>> for row in rows:
...   print(row)
...
(1, 'España', 'Europa', 47, 'Euro')
(2, 'Alemania', 'Europa, 82, 'Euro')
```

El método *fetchall()* extrae todos los registros que cumplen con la condición especificada, si asignamos su resultado a la variable *rows*, podemos iterar y obtener la información de todos los registros. En realidad, obtendremos una tupla donde cada uno de los valores se corresponden con el de los campos de la tabla. Así pues, si quisiéramos obtener solamente el valor correspondiente a los millones de habitantes de cada país, bastaría sustituir la sentencia *print(row)* de nuestro bucle *for*, por la siguiente:

```
print(row[3])
```

Dado que el número índice para acceder a cada campo no es muy intuitivo, es posible emplear un tipo especial de *cursor* que nos permitirá utilizar como índice el nombre de cada campo de los registros de la tabla. Para ello, tendremos que emplear el método *cursor* pasando un parámetro específico, tal y como muestra la siguiente sentencia:

```
>>> cursor = con.cursor(MySQLdb.cursors.DictCursor)
```

Seguidamente, invocaremos al método *fetchall()*, tal y como hemos hecho previamente y cambiaremos nuestro bucle *for* por el siguiente, donde solo vamos a imprimir el nombre de país y su moneda:

```
>>> for row in rows:
...   print('País: {o}. Moneda: {1}'.format(row['nombre'],
row['moneda']))
...
País: España. Moneda: Euro
País: Alemania. Moneda: Euro
```

Otro método interesante que podemos invocar desde el objeto *cursor* es *fetchone()*, el cual solo obtiene un registro. Este método es especialmente útil cuando necesitamos obtener la información relativa a un único registro. Por ejemplo, supongamos que debemos leer la moneda de China. Dado que solo puede haber un país con ese nombre, ejecutaríamos:

```
>>> sql = 'SELECT * FROM países WHERE nombre='China''
>>> cursor = con.cursor()
>>> cursor.execute(sql)
>>> cursor.fetchone()
(4, 'China', 'Asia', 1340, 'Yuan')
```

Si el método *fetchone()* es lanzado sobre un *cursor* que devuelve más de un

resultado, solo el primero será obtenido por este método, quedando el resto de registros inaccesibles a través del *cursor* en cuestión.

Las transacciones también pueden ser manejadas por el módulo *MySQLdb*, siempre y cuando nuestro gestor de MySQL esté configurado para este propósito. Las operaciones básicas sobre transacciones que puede lanzar el módulo en cuestión, son *commit* y *rollback*. La primera de ellas realizaría las operaciones que forman la transacción, mientras que la segunda desharía todas si se produce algún error. Habitualmente, se suele utilizar un bloque *try/except* para ejecutar transacciones, tal y como muestra el siguiente ejemplo:

```
>> try:
...      cursor.execute(sql)
...      cursor.commit()
...   except MySQLdb.Error:
...      con.rollback()
...
```

Cuando una sentencia SQL en la que solo cambian los valores se ejecuta repetidamente, por cuestiones de eficiencia, es conveniente utilizar los llamados *prepared statements*. Un típico ejemplo es cuando debemos lanzar varias sentencias *SELECT* donde solo cambia el valor de un cierto campo del *WHERE*. Eso es bastante habitual en aplicaciones web, donde diferentes usuarios simultáneamente acceden a una determinada página que lanza una consulta. Además, en este tipo de aplicaciones los *prepared statement* evitan un posible ataque por inyección SQL. En la práctica, basta con parametrizar los valores que cambian en la sentencia SQL y pasarlos como un argumento adicional. En el caso de *MySQLdb* podemos hacerlo utilizando una tupla con los valores en cuestión, pasado como segundo parámetro del método *execute()*. El siguiente ejemplo de código muestra cómo emplear un *prepared statement* para realizar una consulta donde solo cambia el continente:

```
>>> sql = 'SELECT moneda, nombre FROM países WHERE continente=%s'
>>> cursor.execute(sql, ('Europa'))
>>> cursor.execute(sql, ('Asia'))
```

No olvidemos que es importante cerrar la conexión a la base de datos en cuanto finalicemos nuestro trabajo con ella. En caso contrario, estaremos consumiendo recursos innecesariamente y el rendimiento de la aplicación bajará considerablemente. Para cerrar una conexión que previamente tenemos abierta,

basta con invocar al método *close(),* en nuestro ejemplo ejecutaríamos la siguiente sentencia:

```
>>> con.close()
```

El módulo *MySQLdb* incluye otras clases y métodos que también pueden ser utilizadas para interactuar con MySQL. Una vez que hemos aprendido los fundamentos de utilización de ese módulo, dejamos al lector como ejercicio que eche un vistazo a la documentación de este módulo para complementar lo expuesto en este apartado.

PostgreSQL

El módulo más popular para trabajar con bases de datos PostgreSQL es *psycopg2* y, al igual que en el caso de *MySQLdb,* este no forma parte de la librería estándar de Python, por lo que hay que instalarlo para poder ser utilizado. Contando con que tenemos *pip* instalado, bastaría con ejecutar desde la línea de comandos:

```
pip install psycopg2
```

En cuanto lo tengamos instalado, podremos cargarlo desde el intérprete de Python:

```
>>> import psycopg2
```

La conexión a una base de datos es sencilla y similar a como hemos visto previamente para *MySQLdb,* basta con invocar al método *connect()* pasando como parámetros de conexión el nombre del servidor donde se está ejecutando la base de datos, el nombre de la base de datos en cuestión, el usuario y su contraseña. Como ejemplo, echemos un vistazo a la siguiente sentencia:

```
>>> con = psycopg2.connect('dbname=paises user=usuario
password=password host=localhost')
```

Para lanzar una sentencia SQL, también necesitamos contar con el objeto *cursor,* así pues, obtendremos una instancia ejecutando:

```
>>> cursor = con.cursor()
```

El método *execute()* es el encargado de lanzar las sentencias SQL, sirva como ejemplo una actualización del número de habitantes de Brasil:

```
>>> sql = 'UPDATE países SET habitantes=205 WHERE nombre='Brasil''
>>> cursor.execute(sql)
```

Para leer la información obtenida a través de sentencias *SELECT*, contamos con dos métodos principales homónimos a los de *MySQLdb: fetchone()* y *fetchall()*. Su funcionamiento es indéntico al mencionado módulo de MySQL. Adicionalmente, el método *fetchmany()* puede ser utilizado para obtener un determinado número de filas, basta con pasar como argumento el número de las que necesitamos, tal y como muestra el siguiente ejemplo:

```
>>> sql = 'SELECT nombre FROM países WHERE habitantes > 80'
>>> cursor.execute(sql)
>>> cursor.fetchmany(2)
('Alemania, 'Europa', 87, 'Euro')'
('China', 'Asia', 1340, 'Yuan)'
```

Otros métodos comunes a *MySQLdb* que también existen en *psycopg2* son *close(), commit()* y *rollback()*, teniendo estos análoga funcionalidad. Además, este módulo para interactuar con PostgreSQL ofrece la opción de trabajar con *prepared statement* de la misma forma que *MySQLdb*, es decir, parametrlzando con una tupla los valores de la sentencia SQL

El lector interesado puede ampliar su conocimiento sobre *psycopg2* consultando la documentación oficial (ver referencias) que puede ser encontrada en el sitio web del módulo.

Oracle

En el mundo empresarial *Oracle* es uno de los gestores de bases de datos relacionales más utilizados. Este gestor es conocido por su capacidad de procesamiento, en especial cuando trata con elevados conjuntos de datos. A diferencia de MySQL y PostgreSQL, Oracle no es *open source* y debe ser utilizado bajo previo pago de la correspondiente licencia de software.

Desde Python también podemos trabajar con bases de datos gestionadas por Oracle, para ello contamos con el módulo llamado *cx Oracle*. Dado que este no

existe en la librería estándar, también debe ser Instalado manualmente. Esta operación puede ser realizada, como hemos visto en casos anteriores, a través del gestor de paquetes *pip:*

```
pip install cx_Oracle
```

Para importar el módulo desde el intérprete o desde un script, escribiremos:

```
import cx_Oracle
```

La conexión a la base de datos se realiza a través de una cadena de texto con los datos necesarios para la conexión. Esta cadena debe ser pasada a la función *connect(),* tal y como muestra el siguiente ejemplo:

```
>>> cad_con = 'usuario/password@localhost/paises'
>>> con = cx_Oracle.connect(cad_con)
```

A partir de que obtengamos nuestro objeto de conexión y, de forma similar a como hemos aprendido en *MySQLdb* y *psycopg2*, necesitamos un *cursor* sobre el que ejecutar nuestras sentencias SQL. El método para obtener el cursor es similar al de los módulos anteriormente mencionados y el código para ello sería el siguiente:

```
>>> cursor = con.cursor()
```

Para obtener una lista de tuplas donde cada elemento representa un registro de la tabla, contamos con el método *fetchAII(),* el cual puede ser invocado directamente sobre el cursor. Por otro lado, si solo necesitamos un número determinado de registros, el método *fetchmany()* podrá ayudarnos. Este método debe pasar como parámetro el número de filas que deseamos obtener; para ello, fijaremos el valor de *numRows,* tal y como muestra el siguiente ejemplo:

```
>>> rows = cursor.fetchmany(numRows=3)
```

A la hora de utilizar *prepared statements,* el módulo *cxOracle* lo hace de forma diferente a *MySQLdb* y *psycopg2*, ya que emplea un método llamado *prepare()* que recibe la consulta y se vale de *execute()* para pasar los parámetros necesarios. De esta forma, el siguiente ejemplo muestra cómo parametrizar una sentencia *SELECT:*

```
>>> cursor = con.cursor ()
```

```
>>> sql = 'SELECT * FROM países WHERE moneda= :moneda')
>>> cursor.prepare(sql)
>>> cursor.execute(None, {'moneda': 'Euro'})
```

En el caso de *cx_0rade* se emplean diccionarios para las consultas parametrizadas, donde cada clave corresponde a la variable que va a ser parametrizada y su valor se corresponde con el que va a ser pasado para ejecutar la sentencia SQL.

Oracle permite trabajar con *procedimientos almacenados* que suelen estar escritos en un lenguaje propio llamado *PI/SQL*. Este tipo de componentes permiten ganar en eficiencia y es habitual que las bases de datos Oracle hagan uso de ellos, sobre todo para complicadas consultas y operaciones. Desde Python podemos invocar a estos procedimientos almacenados gracias al método *callfunc()* del objeto *cursor.* Supongamos que nuestra base de datos cuenta con uno de estos procedimientos al que hemos llamado *miproc*. Este debe recibir como argumento el valor para una variable, llamada x, que es de tipo entero. La invocación a través de *cx_Oracle* sería como sigue:

```
>>> cursor = con.cursor()
>>> res = cursor.callfunc('miproc', cx_Oracle.NUMBER, ('x', 33) )
```

A continuación, nos ocuparemos de los ORM más populares compatibles con Python 3.

SQLite3

SQLite3 es un gestor de bases de datos relacionales caracterizado por su motor, el cual se encuentra contenido en una librería de C. Gracias a este hecho, no es necesaria la instalación y configuración del gestor, basta con instalar la correspondiente librería en el sistema. Una de sus principales ventajas es que ocupa menos de 300 KB, lo que la hace muy portable, especialmente para dispositivos que cuentan con recursos hardware limitados. Por otro lado, una base de datos SQLite3 está compuesta por un único fichero binario.

A diferencia de los casos anteriormente explicados para Oracle, MysQL y PostgreSQL, para conectarnos a SQLite3 no necesitaremos ningún módulo adicional, debido a que Python incluye uno en su librería estándar. Así pues, para

su utilización, bastará con importarlo directamente, tal y como muestra la siguiente sentencia:

```
>>> import sqlite3
```

Para establecer la conexión a nuestra base de datos, lanzaremos el siguiente comando:

```
>>> con = sqlite3.connect('países.db')
```

De forma análoga a los ejemplos anteriores, para la ejecución de sentencias SQL necesitaremos valernos de un *cursor,* la obtención del mismo se realiza así:

```
>>> cursor = con.cursor()
```

En cuanto tengamos el cursor, podremos, por ejemplo, consultar nuestra tabla *países*, empleando para ello el método *execute():*

```
>>> cursor.execute('select * from paises')
>>> for pais in cursor:
...   print(pais)
```

Si lanzamos una sentencia que provoca un cambio en la base de datos, por ejemplo, una sentencia de tipo *insert* o *delete*, deberemos invocar al método *commit()* para que sea efectiva. El siguiente fragmento de código muestra cómo hacerlo:

```
>>> query = 'INSERT INTO pais(nombre, continente, habitantes, moneda)
VALUES ('Japón, 'Asia', 127, 'Yen)'
>>> cursor.execute(query)
>>> cursor.commit()
```

Una vez finalizado el recorrido por los gestores de bases de datos, llega el turno de los ORM, de los que nos ocuparemos en el siguiente apartado.

ORM

Dos son los ORM más utilizados en Python: *SQLAlchemy* y *SQLObject.* Ambos son similares en funcionalidad y, básicamente, ofrecen una serie de técnicas para interacturar con una base de datos con independencia del gestor. Es

decir, el código escrito que utiliza un ORM para trabajar con una base de datos puede ejecutarse en MySQL, PostgreSQL, Oracle y cualquier otro motor soportado por el ORM en cuestión. Además, los ORM proporcionan una interfaz basada en objetos, donde sus tablas son representadas por clases, los registros por instancias de estas clases y los campos de las tablas por los atributos de instancia. Así pues, es muy sencillo establecer una correspondencia directa en los componentes de una base de datos y una serie de objetos de Python.

Otra de las ventajas de emplear un ORM en las aplicaciones Python que necesitan interactuar con una base de datos, es que no es necesario escribir código SQL, ya que los ORM proporcionan una serie de métodos que hacen de interfaz a diferentes sentencias SQL. Sin embargo, son muchos los ORM que también permiten escribir directamente código SQL para trabajar con la base de datos. Algunos programadores prefieren emplear esta técnica para tener un mayor de control sobre el SQL que es ejecutado.

En la actualidad, los ORM son empleados por diferentes lenguajes y tecnologías y su uso se ha popularizado y, podríamos decir *estandarizado*, gracias a que son muchos los modernos *frameworks* web, que, o bien integran uno propio, o bien ofrecen facilidades para integrar el que el programador desee.

Comenzaremos centrándonos en *SQLAlchemy*, uno de los primeros y más populares ORM para Python.

SQLAlchemy

La primera versión de este ORM fue liberada en 2006 y rápidamente se convirtió en uno de los más aceptados por la comunidad de desarrolladores de Python. Es *open source* y se distribuye bajo la *MIT license.*

Entre las funcionales de *SQLAlchemy* encontramos las que nos permiten conectarnos a una base de datos para insertar, leer, modificar y borrar datos. También es posible crear diferentes clases estableciendo relaciones entre ellas que luego serán traducidas en la base de datos empleando claves foráneas. Además, las operaciones *join* se realizan automáticamente por el ORM cuando es necesario extraer información de diversas tablas que están relacionadas entre sí.

SQLAlchemy soporta diferentes gestores de bases de datos relacionales, como son, por ejemplo, *MySQL, PostgreSQL, SQLite, Oracle* y *MS SQL Server.* Para

los ejemplos que veremos en este apartado nos centraremos en *MySQL*, aunque, lógicamente, son totalmente extrapolares a otro gestor. Básicamente, lo único dependiente del gestor en sí, será la cadena de conexión que empleemos en la función *create_engine()*, tal y como veremos más adelante.

A pesar de que *SQLAlchemy* incluye una amplia funcionalidad, en este apartado nos centraremos en descubrir aquellas funcionalidades que son consideradas como básicas. Si el lector está interesado en explotar todo lo que este módulo tiene que ofrecernos, recomendamos consultar la documentación oficial del mismo (ver referencias).

Es necesario instalar *SQLAlchemy* en nuestro equipo, ya que este módulo no forma parte de la librería estándar de Python. Tal y como hemos aprendido previamente, este módulo puede ser instalado directamente a través de *pip*; para ello, basta con ejecutar el siguiente comando desde un terminal:

```
pip install sqlalchemy
```

El primer paso para poder comenzar a utilizar nuestro nuevo módulo será *importarlo*, de lo que se encargará la siguiente sentencia:

```
>>> import sqlalchemy
```

Por simplicidad y, dado que vamos a emplear varias clases de *SQLAlchemy*, en nuestro ejemplo comenzaremos cargando aquellas que necesitamos. Para ello, bastará con ejecutar las siguientes sentencias:

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy import Column, Integer, String, create_engine
```

Seguidamente, estableceremos una conexión con nuestra base de datos. Como hemos comentado previamente, utilizaremos *MySQL* como ejemplo:

```
>>> cad_con = 'mysql://usuario:password@localhost/prueba'
>>> engine = create_engine(cad_con)
```

Las clases que vayan a ser creadas para representar a las tablas de la base de datos deben heredar de una clase propia de *SQLAlchemy*, llamada *Base*. Esta debe ser obtenida a través de la llamada a una función concreta:

```
>>> Base = declarative_base()
```

SQLAlchemy requiere que cada clase, que vaya a declararse para que se

corresponda con una determinada tabla, tenga al menos dos métodos principales. El primero de ellos será un constructor que recibirá una serie de parámetros que deben corresponder con los valores de un registro determinado. Este constructor se debe encargar de asignar estos valores a cada uno de los atributos de clase. Por otro lado, el segundo método en cuestión será utilizado para identificar a cada instancia de clase. Con sobrescribir el método especial *__repr__()*, será suficiente. Teniendo en cuenta estos requisitos, nuestra clase quedaría de la siguiente forma:

```
class Pais(Base):
    __tablename__ = 'país'
    id = Column(Integer, primary_key=True)
    nombre = Column(String(100))
    continente = Column(String(20))
    habitantes = Column(Integer)
    moneda = Column(String(30))

    def __init__(self, nombre, continente, habitantes, moneda):
        self.nombre = nombre
        self.continente = continente
        self.habitantes = habitantes
        self.moneda = moneda

    def __repr__(self):
        return '<Pais('%s')>' % (self.nombre)
```

Ahora es el momento de crear la tabla físicamente en la base de datos, a partir de la definición de nuestra clase *Pais*. Esta acción es sencilla y puede ser llevada a cabo a través de la siguiente sentencia:

```
>>> Base.metadata.create_all(engine)
```

Conectándonos a la base de datos podremos comprobar que, efectivamente, la nueva tabla ha sido creada. Sin embargo, las operaciones para interactuar con ella deben ser ejecutadas a través de una instancia específica de la clase *Session*. Esta debe ser creada empleando el siguiente código:

```
>>> Session = sessionmaker(bind=engine)
>>> Session = sessionmaker()
>>> Session.configure(bind=engine)
>>> session = Session()
```

Dado que nuestra nueva tabla no contiene ningún registro, el siguiente paso

será añadir uno nuevo:

```
>>> p = Pais('Tailandia', 'Asia', 65, 'Baht')
>>> session.add(p)
>>> session.commit()
```

Justo después de lanzar la última sentencia, se ejecutará el comando *INSERT* que insertará nuestro nuevo registro. Podemos comprobar que, efectivamente, ha sido así, lanzando una consulta que recorra toda la tabla y muestre el nombre del país, tal y como muestra el siguiente código:

```
>>> rows = session.query(Pais, Pais.nombre) .all ()
>>> for row in rows:
...   print(row.nombre)
...
Tailandia
```

SQLAlchemy incluye varios métodos, llamados *filters*, para realizar consultas que contengan una cláusula *WHERE*. Por ejemplo, si deseamos obtener todos los países cuya moneda es el euro, podemos lanzar la siguiente sentencia:

```
>>> res = session.query(Pais).filter(Pais.moneda == 'Euro')
```

Para la ordenación podemos emplear el método *order_by*, por ejemplo, para obtener todos los países ordenados por su *id:*

```
>>> session.query(Pais).order_by(Pais.id)
```

La actualización de un valor de un determinado campo de una tabla es muy sencilla, basta con asignar un nuevo valor al correspondiente atributo de instancia e invocar al método *commit()*, tal y como muestra el siguiente código:

```
>>> p.habitantes = 67
>>> session.commit()
```

Una vez aprendido lo básico sobre la utilización de *SQLAlchemy*, es hora de comenzar a descubrir *SQLObject.*

SQLOBJECT

Al igual que *SQLAlchemy, SQLObject* es otro de los más populares ORM disponibles para Python. Se distribuye bajo la licencia libre LGPL y también

puede ser instalado a través del gestor de módulos *pip*. A pesar de que incluye las mismas funcionalidades básicas que *SQLAlchemy,* este último cuenta con un mayor número de clases, métodos y funciones que aportan mayor funcionalidad. Sin embargo, *SQLObject* es perfectamente utilizable para la gran mayoría de aplicaciones que requieren de un ORM rápido y sencillo de utilizar.

La primera versión liberada de *SQLObject* fue en mayo 2002 y, en la actualidad, la sintaxis y la forma de realizar la correspondencia entre tablas y objetos son muy similares a la de *Active Record,* el popular ORM empleado por el *framework* web *Ruby on Rails.* Si el lector tiene experiencia previa con este ORM, seguro que no tiene ninguna dificultad en familiarizarse rápidamente con *SQLObject.* Por otro lado, es este el módulo elegido por *Turbogears,* conocido *Framework* web para Python, como ORM por defecto.

SQLObject puede trabajar con diferentes gestores de bases de datos relacionales, como *MySQL, PostgreSQL, MS SQL Server, SQLite* y *Sybase.* Al igual que en el caso de *SQLAlchemy* vamos a trabajar con una base de datos *MySQL* como ejemplo.

Nuestra práctica comienza instalando el correspondiente módulo en nuestro sistema:

```
pip install sqlobject
```

A partir de la correcta instalación del módulo podemos importarlo como tal, como muestra la siguiente sentencia:

```
>>> import sqlobject
```

Al igual que hemos hecho con *SQLAlchemy,* comenzaremos creando nuestra clase que será la que represente a, la ya popular, tabla *paises.* La definición de la clase en cuestión es como sigue:

```
>>> class Pais(sqlobject.SQLObject):
...   nombre = sqlobject.StringCol(length=100)
...   continente = sqlobject.StringCol(length=20)
...   habitantes = sqlobject.IntCol()
...   moneda = sqlobject.StringCol(length=30)
```

El siguiente paso será establecer una conexión con nuestra base de datos *pruebas*; para ello, nos apoyaremos en el método *connectionForURI()* y asignaremos el valor que nos devuelva el mismo a una variable llamada

processConnection. Con esta asignación nos aseguramos de que todas las clases que definamos interactuarán con la conexión previamente establecida. El código para realizar estas operaciones es el siguiente:

```
>>> db_cad = 'mysql://cruceros:cruceros@localhost/cruceros'
>>> con = sqlobject.connectionForURI(db_cad)
>>> sqlobject.sqlhub.processConnection = con
```

Ahora estamos ya listos para crear la tabla *pais* partiendo directamente de la clase anteriormente definida:

```
>>> Pais.createTable()
```

Si consultamos la base de datos *pruebas*, veremos cómo tenemos la nueva tabla *pais*, que, por el momento, está vacía. Así pues, pasaremos a crear un nuevo país en la misma a través de una nueva instancia de nuestra clase:

```
>>> Pais(nombre='Francia',continente='Europa',moneda='Euro',
habitantes=66)
```

Simplemente, ejecutando la sentencia anterior, ya tendremos un nuevo registro en nuestra tabla. Para comprobar este hecho, utilizaremos el método *get()* que recibirá como argumento el valor *1*, indicando que deseamos obtener aquel cuyo *id* coincide con dicho valor:

```
>>> p = Pais.get(1)
>>> p.nombre
Francia
```

Si ahora deseamos modificar uno de los campos del registro recién creado, bastará con invocar al método *set()*, tal y como muestran las siguientes líneas:

```
>>> p.set(habitantes=67)
>>> p = Pais.get(1)
>>> p.habitantes
67
```

La consulta de registros puede ser llevada fácilmente a cabo gracias al método *selectBy()* que puede recibir como argumento el nombre del atributo y su valor que correspondería con la cláusula *WHERE* de SQL. Por ejemplo, para obtener todos los países de Europa, ejecutaríamos:

```
>>> paises = Pais.selectBy(continente='Europa')
```

```
>>> from pais in paises:
...   print(pais.nombre)
...
Francia
```

El módulo *SQLObject* nos permite representar relaciones *1-N* y *N-M* de tablas. Además, ofrece una serie de sencillos métodos para obtener valores de las tablas relacionadas, haciendo automáticamente cuántas operaciones *JOIN* sean necesarias.

Comparado con *SQLAlchemy, SQLObject* es más sencillo e intuitivo, aunque sí es cierto que el primero ofrece más funcionalidades. La elección de uno u otro dependerá de varios factores, aunque si buscamos un ORM que consuma pocos recursos y sea rápido, *SQLObject* puede ser una buena elección.

NOSQL

Las bases de datos NoSQL *(Not Only SQL)* son diferentes a las bases de datos relacionales tanto en estructura como en el tipo de relaciones que se establecen y en la forma de interactuar con los datos. Si en las relacionales el lenguaje SQL es el encargado de trabajar directamente con los datos, en las NoSQL no se utiliza este tipo de lenguaje. Además, la mayoría de bases de datos NoSQL no soportan *JOINS*, ya que no se establecen las típicas relaciones *1-N* y *N-M*.

Frente a las bases de datos relacionales, las de tipo NoSQL son fácilmente escalables, ofrecen mínimos tiempos de consulta y pueden trabajar con grandes volúmenes de datos. Gracias a estas características se han vuelto muy populares para aplicaciones web de alto tráfico, como son las ofrecidas por empresas como Google, Facebook o Twitter.

Básicamente, una base de datos NoSQL almacena una serie de pares *claves:valor* y, en vez de hablar de *registros*, se habla de *documentos*. En la actualidad, son tres los gestores de bases de datos NoSQL más populares: *Redis*, *Cassandra* y *Redis*. De cómo interactuar con ellos desde Python nos ocuparemos en esta parte final del presente capítulo. Debemos tener en cuenta que el lector está familiarizado con el funcionamiento de estos gestores y conoce los fundamentos sobre ellos. Asimismo, vamos a aprender lo básico sobre la conexión y manejo de datos desde Python contra estos gestores. Si el lector está

interesado en profundizar en el tema, es aconsejable echar un vistazo a la documentación de los módulos de Python con los que vamos a trabajar para interactuar con cada uno de los mencionados gestores NoSQL.

Redis

Para trabajar con *Redis* contamos con varios módulos de Python. Uno de los más sencillos y fáciles de usar se llama *redis* y ofrece las funcionalidades básicas para fijar un par *clave:valor*, para leer registros y para borrarlos.

Dado que el mencionado módulo para trabajar con *Redis* se encuentra disponible a través de *pip*, su instalación es bastante sencilla, simplemente deberemos ejecutar el siguiente comando desde una terminal:

```
pip install redis
```

Si la instalación ha finalizado correctamente, el módulo podrá ser importado en cualquiera de nuestros *scripts*. Es más, podemos probar directamente en la interfaz de comandos del intérprete de Python:

```
>>> import redis
```

Fijar el valor de una clave es tan sencillo como conectarnos a una de las bases de datos de nuestro servidor *Redis*, e invocar al método *set()*. El primer paso será establecer la conexión, tal y como muestra la siguiente sentencia:

```
>>> con = redis.StrictRedis(host='localhost', port=6379, db=0)
```

Los tres parámetros pasados al constructor de la clase *StrictRedis* indican: el servidor al que vamos a conectarnos, el puerto en el que está corriendo y el número que identifica a la base de datos. Si no pasamos ninguno de estos valores, se utilizarán los que hemos empleado en nuestra línea ejemplo.

Redis permite simplemente fijar pares *clave:valor*, a diferencia de *MongoDB* y *Cassandra* que soportan el almacenamiento más complejo de valores básandose en la misma estructura. De esta forma, la inserción de un valor en nuestra base de datos *Redis*, sería como sigue:

```
>>> con.set('clave', 'valor')
```

La recuperación de un valor asociado a una clave, es decir, la lectura de un registro almacenado en la base de datos, se realiza invocando al método *get()*. El valor que hemos insertado previamente puede ser recuperado ejecutando la siguiente sentencia:

```
>>> con.get('clave')
valor
```

El registro insertado puede ser borrado sencillamente gracias al método *delete()*. Así pues, la siguiente línea de código eliminará el registro que hemos insertado previamente:

```
>>> con.delete('clave')
```

El módulo *redis* soporta trabajar con *pools* de conexiones, con lo que ganamos en eficiencia a la hora de establecer diferentes conexiones con la base de datos. La clase *ConnectionPool()* se encarga de realizar la conexión empleando el *pool* de conexiones y seleccionando aquella que se encuentre libre. Las siguientes dos líneas de código son necesarias para utilizar el *pool* y obtener una conexión libre que podamos emplear para nuestro trabajo:

```
>>> pool = redis.ConnectionPool()
>>> con = redis.Redis(connection_pool=pool)
```

Aprendido lo básico para interactuar con una base de datos *Redis* desde Python, continuaremos descubriendo cómo realizar operaciones similares con *MongoDB*.

MongoDB

El módulo más popular para trabajar con *MongoDB* desde Python es el llamado *PyMongo*. Este puede ser instalado fácilmente a través de *pip*, así pues basta con invocar a este gestor directamente desde la línea de comandos:

```
pip install pymongo
```

En cuanto finalice la instalación, podrá ser importado a través de la siguiente sentencia:

```
>>> import pymongo
```

Para nuestros ejemplos vamos a partir de que tenemos previamente creada una base de datos llamada *pruebas,* conectarnos a la misma es bastante sencillo. Primero debemos obtener una conexión y después podremos conectarnos a la base de datos. El código necesario para ello sería el siguiente:

```
>>> con = pymongo.Connection()
>>> db = con.pruebas
```

Hemos de tener en cuenta que si no pasamos ningún parámetro a la hora de realizar la conexión, por defecto se entenderá que deseamos conectarnos a *localhost* y al puerto estándar (27017) que emplea *MongoDB.* En caso contrario, podemos pasar dos parámetros diferentes, uno para el nombre o IP del servidor y otro para el puerto en cuestión.

Los datos son almacenados en *MongoDB* empleando el formato JSON; así pues, podemos crear un simple documento para representar un país en cuestión:

```
>>> pais = {'nombre': 'Alemania', 'habitantes': 82, 'continente':
'Europa', 'moneda': 'euro'}
>>> paises = db.paises
>>> paises.insert(pais)
```

En Python utilizaremos un diccionario para emular el formato JSON, tal y como nos muestra la primera línea del ejemplo anterior. Una vez que tenemos nuestro documento, creamos una *colección* y añadimos dicho documento a la misma.

Para comprobar que el documento ha sido insertado correctamente, basta con invocar al método *find_one()* que se encargará de consultar nuestra colección:

```
>>> paises.find_one({'nombre': 'Alemania})
{'nombre': 'Alemania', 'habitantes': 82, 'continente': 'Europa',
'moneda': 'euro'}
```

Si necesitamos hacer una consulta que devuelva más de un documento, podemos emplear el método *find()* e iterar sobre el resultado, ya que en este caso obtendremos una lista de diccionarios. Por ejemplo, supongamos que tenemos varios países que tienen el euro como moneda y queremos extraer los mismos de la base de datos. Para ello, bastaría con ejecutar las siguientes líneas:

```
>>> p_euro = paises.find({'moneda': 'Euro'})
```

```
>>> for pin p_euro:
...   print(p)
{'nombre': 'Alemania', 'habitantes': 82, 'continente': 'Europa',
'moneda': 'euro'}
{'nombre': 'España, 'habitantes': 47, 'continente': 'Europa',
'moneda': 'euro'}
{'nombre': 'Francia', 'habitantes': 67, 'continente': 'Europa',
'moneda': 'euro'}
```

Otro de los métodos útiles que nos ofrece *PyMongo* es *count()*, el cual nos devuelve el número de documentos que cumplen cierta condición. Así pues, si ejecutamos la siguiente sentencia, obtendremos los tres países que utilizan el euro y que se encuentran en nuestra base de datos:

```
>>> paises.find({'moneda': 'Euro'}).count()
3
```

Ahora que ya hemos aprendido lo básico sobre *PyMongo,* pasaremos a ocuparnos de cómo trabajar desde Python con *Cassandra.*

Cassandra

El gestor de bases de datos *Cassandra* almacena la información de forma diferente a como lo hace *MongoDB. Cassandra* se basa en los conceptos de *keyspace* y *column familiy* para almacenar la información de forma estructurada. Al fin y al cabo, también se emplea el concepto de *clave* y *valor,* solo que también se permite estructurar la información en *columnas* y en *supercolumnas.*

A pesar de que existen varios módulos para trabajar con *Cassandra* desde Python, uno de los más populares y fáciles de usar es *pycassa.* De forma análoga a como hemos instalado otros módulos que no forman parte de la librería estándar de Python, ejecutaremos desde la línea de comandos, la siguiente orden:

```
pip install pycassa
```

Para trabajar con el módulo que acabamos de instalar, basta con *importarlo*:

```
>>> import pycassa
```

La conexión a la base de datos se hará a través de un objeto específico que representa a un *pool* de conexiones, al que indicaremos el nombre del *keyspace*

que deseamos utilizar, más los datos de conexión del servidor:

```
>>> from pycassa.pool import ConnectionPool
>>> pool = ConnectionPool('mainspace')
```

En caso de que tengamos *Cassandra* corriendo en un puerto diferente al estándar o en un servidor que no sea la máquina local (*localhost*), podemos indicar estos valores a través de una lista, tal y como muestra la siguiente línea de código:

```
>>> pool = ConnectionPool('mainspace', ['192.168.0.3', '8080'])
```

Una vez que tengamos acceso a nuestro *keyspace*, es necesario indicar la *column family* a la que deseamos conectarnos. Esta operación puede ser llevada a cabo a través de una clase llamada *ColumnFamily*. Las siguientes líneas muestran cómo realizar esta operación:

```
>>> from pycassa.columnfamily import ColumnFamily
>>> col = ColumnFamily(pool, 'maincolumn')
```

Ahora ya estamos listos para insertar nuestro primer registro; para ello, emplearemos el método *insert()*, cuyo primer argumento referencia a la clave y el segundo es un diccionario con los pares *clave:valor* que deseamos insertar. Veamos cómo el siguiente código nos servirá para insertar un nuevo país:

```
>>> col.insert('Alemania', {'habitantes': 82, 'continente': 'Europa',
'moneda': 'euro'})
```

El recién insertado registro puede obtenerse fácilmente gracias al método *get()*. Las siguientes líneas muestran cómo utilizarlo:

```
>>> col.get('Alemania')
{'habitantes': 82, 'continente': 'Europa', 'moneda': 'euro'}
```

Como resultado de pasar la clave *Alemania*, obtenemos un diccionario de Python con los valores requeridos. Si necesitáramos dos registros diferentes en una única sentencia, podríamos hacerlo a través del método *multiget()*, tal y como muestra el siguiente ejemplo:

```
>>> col.multiget(['Alemania', 'Francia'])
{'Alemania': {'habitantes': 82, 'continente': 'Europa', 'moneda':
'euro'},
'Francia': {'habitantes': 67, 'continente': 'Europa', 'moneda' :
'euro'}}
```

Finalizamos así el capítulo dedicado a las bases de datos. Esperamos que el lector haya logrado una buena visión de conjunto que le permita trabajar, tanto con bases de datos relacionales, como con bases de datos NoSQL, desde Python.

INTERNET

INTRODUCCIÓN

En este capítulo nos ocuparemos de aquellos aspectos más relevantes relacionados con Python y la programación relativa a servicios de Internet. Ante el innegable auge de la red de redes, es muy interesante descubrir cómo interactuar, a través de la misma, empleando Python como lenguaje de programación para escribir aplicaciones que puedan hacer uso de servicios como el correo electrónico, la web, los *web Services* o la conexión directa a través de protocolos específicos como FTP.

Comenzaremos nuestro recorrido aprendiendo cómo realizar conexiones con servidores accesibles a través de Internet, empleando dos protocolos ampliamente utilizados, como son TELNET y FTP. Gracias a utilizar un lenguaje como Python, descubriremos cómo es posible automatizar muchas tareas que se realizan de forma manual trabajando con estos protocolos.

Seguiremos el camino ocupándonos de los servicios web que emplean el protocolo XML-RPC como estándar para la comunicación e intercambio de información entre el cliente y el servidor.

Al finalizar con los protocolos anteriormente mencionados, llegará la hora del correo electrónico. Aprenderemos cómo conectarnos e interactuar con servidores que empleen diferentes protocolos estándar como son IMAP4, POP3 y SMTP.

El último apartado del presente capítulo se ocupará de la web, donde nos centraremos en aprender lo básico sobre el manejo de protocolos como CGI y WSGI. Además, descubriremos qué herramientas podemos emplear para realizar *web scraping*, una de las técnicas más utilizadas en la actualidad para la obtención de información desde sitios web. Por último, llegará el turno de los *frameworks* web, también de rabiosa actualidad, gracias al conjunto de componentes y herramientas que ofrecen para desarrollar y mantener fácilmente complejas aplicaciones web.

Es conveniente tener en cuenta que este capítulo no pretende realizar un exhaustivo análisis de todos los módulos, herramientas y componentes existentes

y relacionados con Internet y disponibles para Python. El objetivo es otro, tratándose el mismo de repasar aquellos aspectos, técnicas y módulos que consideramos más interesantes para poder comenzar a desarrollar rápidamente aplicaciones en Python que puedan sacar provecho del potencial que nos ofrece Internet.

En la actualidad existe un mayor conjunto de módulos y componentes para interactuar con Internet en la versión 2 de Python que en la 3. No obstante, esta situación está cambiando y son muchos desarrolladores que se encuentran trabajando en la migración de sus componentes. A pesar de ese hecho, es fácil encontrar módulos que funcionan correctamente en la última versión del intérprete de Python. Una muestra de ellos son los que descubriremos y utilizaremos en el presente capítulo.

A continuación, comenzamos aprendiendo a trabajar con los protocolos FTP y TELNET desde Python.

TELNET Y FTP

Dos de los más utilizados protocolos en Internet para interactuar con servidores son FTP y TELNET. Ambos permiten comunicarnos con una máquina desde otra a través de una conexión a Internet. El protocolo FTP es ampliamente utilizado tanto para *subir* como para *bajar* ficheros. Si alguna vez hemos usado en servicio de *hosting*, seguro que hemos utilizado este protocolo a través de algún cliente que lo soportara. Por otro lado, gracias a TELNET podemos abrir un terminal y ejecutar comandos como si físicamente estuviéramos sentados enfrente de la máquina remota. Este protocolo se utiliza típicamente en servidores UNIX, Linux y Mac OS X Server. Aunque es menos frecuente, también algunas versiones de Windows lo soportan.

En este apartado descubriremos qué módulos tenemos disponibles en Python para poder trabajar con los mencionados protocolos para comunicarnos remotamente con otras máquinas configuradas como servidores. Comenzamos ocupándonos del protocolo TELNET.

telnetlib

Python cuenta en su librería estándar con un módulo llamado *telnetlib*, el cual encapsula la funcionalidad básica para conectarnos y desconectarnos a un servidor, enviar comandos y obtener respuestas, todo a través del protocolo TELNET.

El mencionado módulo pone a nuestra disposición una clase principal denominada *Telnet*. Al crear una instancia de la misma obtendremos un objeto que nos permitirá interactuar con el servidor. El método *open()* será el encargado de abrir una conexión a un servidor que acepte el protocolo TELNET. Como parámetros del constructor de la mencionada clase podemos pasar tres diferentes, uno que indique nombre o IP del servidor, otro para especificar el puerto de conexión y un tercero que servirá para fijar un número de segundos como máximo para esperar respuesta. El único de estos parámetros requeridos es el primero, ya que, por defecto, la conexión se realizará al puerto estándar (23)

para TELNET.

Antes de comenzar a ver los ejemplos de código de utilización de *telnetlib*, si no disponemos de conexión a ningún servidor mediante TELNET, es fácil instalar un servidor en nuestra máquina local. Si utilizamos Mac OS X bastará con ejecutar el servicio que corre un servidor de TELNET en el puerto 21. Para ello, abriremos un terminal y lanzaremos el siguiente comando:

```
$ sudo launchctl load -w /System/Library/LaunchDaemons/telnet.plist
```

La mayoría de las distribuciones de GNU/Linux cuentan con servidores de TELNET entre sus paquetes binarios. Así pues, por ejemplo, en Ubuntu podemos abrir una consola de comandos y lanzar el siguiente comando:

```
$ sudo apt-get install telnetd
```

Para instalar un servidor de TELNET en Fedora lanzaremos el siguiente comando:

```
$ sudo yum install telnet-server
```

En cuanto tengamos listo nuestro servidor local o los datos de acceso a uno remoto, podemos comenzar a practicar con el código ejemplo que veremos a continuación.

La primera operación será conectarnos a nuestro servidor, para ello, ejecutemos las siguientes sentencias:

```
>>> from telnetlib import Telnet
>>> tel = Telnet('localhost')
```

En lugar de invocar al método *open()* para abrir una conexión, vamos directamente a llamar al método *read_until(),* el cual se encargará de abrir la conexión por nosotros y comenzar a leer desde el servidor hasta que este ofrezca la respuesta que coincida con el argumento pasado al mencionado método. En nuestro ejemplo y dado que vamos entrar en la máquina empleando un *login* y *password* correspondientes a un usuario concreto, vamos pasar como parámetro la cadena binaria *"login: "*, tal y como muestra el siguiente código:

```
>>> tel.read_until(b'login: ')
'\r\r\nDarwin/BSD (yoda.local) (ttysOOO)\r\r\n\r\r\nlogin:'
```

En cuanto el servidor responda, deberemos mandar el nombre de usuario para

hacer *login,* para ello contamos con el método *write(),* el cual se encarga de mandar una serie de bytes al servidor. Esta será la sentencia que necesitaremos, donde *usuario* debe ser reemplazado por el nombre en cuestión del usuario que va a realizar *login* en el servidor:

```
>>> tel.write(b'usuario\n')
```

A través de la cadena *\n* indicamos que un retorno de carro debe ser lanzado justo después del nombre de usuario. Ello simularía la pulsación de la tecla *enter* si estuviéramos utilizando el comando *telnet* directamente desde una terminal. El siguiente paso será esperar a que conteste el servidor pidiéndonos una contraseña:

```
>>> tel.read_until(b'Password: ')
' usuario\r\nPassword:'
```

De la misma forma que el nombre de usuario, ahora mandaremos nuestra contraseña, a través del método *write():*

```
>>> tel .write (b'password\n' )
```

Si todo va bien, el login se realizará correctamente y podremos enviar otros comandos. Probemos con un sencillo listado del contenido del directorio del usuario que se ha conectado al servidor:

```
>>> tel.write(b'ls\n')
```

Seguidamente y, previo paso a leer los datos de respuesta enviados por el servidor, procederemos realizar el *logout* del servidor, para ello bastará con mandar el comando *exit,* soportado por el protocolo TELNET:

```
>>> tel.write(b'exit\n')
```

Para leer la salida mandada desde el servidor contamos con el método *read_all()* que nos devolverá todos los datos enviados como respuesta desde el servidor, incluyendo, tanto la salida del comando *ls,* como los que se obtienen al ejecutar el comando *exit:*

```
>>> tel. read_all ()
'\r\nLast login: Mon Feb 20 17:10:37 on ttys003\r\n[arturo@yoda

~\xlb[0;32m]\x1b[0;37m$ls\r\nApplications\t\t\tfile.txt\r\nDesk
```

```
top\t\t\t\t
ftdetect\r\nDocuments\t\t\tftplugin[arturo@yoda~\x1b[0;32m]\x1b [0;37m
       $\r [arturo@yoda ~\x1b[0;32m]\x1b[0;37m$ exit\r\nlogout\r\n'
```

Por último, solo nos queda cerrar la conexión al servidor. Acción que puede realizarse gracias al método *close()*, tal y como muestra la siguiente línea:

```
>>> tel.close()
```

El módulo *telnetlib* también puede ser utilizado para conectarnos a otros servicios que corren en máquinas remotas y que aceptan el protocolo TELNET. Como ejemplo de estos servicios están los servidores de correo electrónico que soportan SMTP y POP3, los servidores de caché como *memcached* y *redis*, el servidor de base datos NoSQL.

ftplib

Sin necesidad de recurrir a ningún módulo fuera de la librería estándar de Python, para el manejo de conexiones a través del protocolo FTP *(File Transfer Protocol)* contamos con el módulo *ftplib*. Este incluye una clase principal llamada *FTP* que se encarga de gestionar la conexión con otros servidores. Entre las acciones que la misma permite llevar a cabo, destacan las que nos permiten abrir y cerrar una conexión, autenticarnos, mandar cualquier comando aceptado por el protocolo y leer los resultados enviados como respuesta por el servidor. Utilizar esta clase desde Python nos permitirá automatizar sencillas tareas como la descarga o subida de ficheros e incluso aquellas más complejas, como pueden ser, por ejemplo, la creación y mantenimiento de un servidor *mirror*.

La obtención de una instancia de la clase principal *FTP* se realiza a través de su constructor, el cual recibe como parámetros la IP o nombre del servidor y el puerto donde realizar la conexión. Si no se pasa ningún entero para el puerto, se usará el empleado por defecto y convención en los servidores. De esta forma, para conectarnos a un servidor, crearemos previamente una instancia de la mencionada clase:

```
>>> from ftplib import FTP
>>> con = FTP('ftp.miservidor.com')
```

En cuanto tengamos creada la instancia podremos comenzar a interactuar con el servidor. Si necesitamos autenticarnos, invocaremos al método *login()*, tal y como muestra la siguiente sentencia:

```
>>> con.login('usuario', 'password')
'230 User usuario logged in.'
```

Son muchos los servidores FTP que aceptan un usuario especial llamado *anonymous*, cuya *password* coincide con el nombre de usuario. Es práctica habitual colocar la información pública que ofrecen los servidores FTP bajo el acceso de este usuario especial. Así pues, para conectarnos empleando el mismo bastará con invocar al método *login()* sin pasar ningún parámetro.

Una vez conectados al servidor mediante FTP, podemos, por ejemplo, echar un vistazo al contenido del directorio al que hemos accedido por defecto. Para ello invocaremos al método *retrlines()*, el cual obtiene el contenido de un fichero o lista los que forman parte de un determinado directorio. En nuestro ejemplo, vamos a mandar el comando *LIST* para obtener una cadena de texto con el contenido del directorio al que hemos accedido por defecto al realizar la conexión al servidor:

```
>>> con.retrlines('LIST')
total 234
drwxrwsr-x  5 usuario usuario 1520 Feb 16 09:22 .
dr-xr-srwt 15 usuario usuario 1520 Feb 16 09:25 ..
-rw-rw-rw- 20 usuario usuario 230  Feb 18 10:12 myfile.txt
```

Subir un fichero a un servidor es sencillo gracias al método *storbinary()*. Simplemente deberemos abrir el fichero en cuestión e invocar al mencionado método pasando dos parámetros. El primero de ellos será una cadena de texto que contiene el valor 'STOR', seguido del nombre que deseemos darle al fichero remoto. El segundo en cuestión será un objeto de tipo fichero. El siguiente código nos muestra cómo acceder a un fichero local llamado *prueba.json* y subirlo con el nombre *test.json:*

```
fich = open ('prueba.json', 'rb')
>>> con.storbinary('STOR test.json', fich)
'226 Transfer complete. '
>>> fich.close()
```

Efectivamente, para abrir nuestro fichero hemos empleado *b* como *flag* para indicar que el fichero, aunque sea de texto, debe ser tratado como binario. De

esta forma podremos utilizar su contenido sin problema a través del método *storbinary()*.

Análogamente, también es posible descargarnos un fichero desde el servidor FTP. Para esta acción contamos con el método *retrbinary()*, el cual también recibe dos parámetros: uno para indicar que se va a llevar a cabo la operación de descarga y otro para pasar el objeto fichero donde se almacenará el contenido del fichero descargado. Como ejemplo podemos descargarnos el fichero que hemos subido previamente. En esta ocasión nos bajaremos el fichero *test.json* y lo salvaremos con el nombre *miprueba.json*. El siguiente código nos muestra cómo hacerlo:

```
>>> fich = open('miprueba.json', 'wb')
>>> con.retrbinary('RETR test.json', fich.write)
'226 Transfer complete.'
>>> fich.close()
```

Es importante cerrar al final el fichero, en caso contrario el fichero no será creado. Igualmente ocurre en el caso de la subida, hasta que no cerremos el fichero, no será volcado su contenido al servidor remoto.

Para enviar un comando FTP válido existe el método *sendcmd()*, que recibe como cadena de texto el comando en cuestión que será ejecutado. Como resultado obtendremos la salida de la ejecución del mismo. Por ejemplo, para indicar que vamos a cambiar a modo binario, podríamos lanzar la siguiente sentencia:

```
>>> con.sendcmd('TYPE i')
```

Otro método interesante con los que cuenta la clase *FTP* es *size()* que sirve para averiguar cuánto ocupa un determinado fichero en disco. La invocación puede realizarse directamente, pasando como parámetro el nombre del fichero en cuestión, tal y como muestra el siguiente ejemplo:

```
>>> con.size('myfile.txt')
230
```

Para trabajar con directorios dentro del servidor existen diferentes métodos, como son *cwd(), pwd(), mkd()* y *rmd()*. El primero de ellos sería para cambiar a un directorio determinado, el cual será fijado como actual, para ello deberemos especificar como parámetro el nombre del directorio en cuestión. El segundo

simplemente nos devolverá el *path* del directorio actual. Por otro lado, *mkd()* creará un nuevo directorio, mientras que *rmd()* servirá para borrar aquel directorio que ya no necesitemos.

No olvidemos cerrar la conexión al servidor en cuanto finalicemos nuestro trabajo:

```
>>> con.quit()
```

Como habremos podido comprobar la interacción con servidores FTP es bastante sencilla desde Python, lo cual nos abre un interesante campo para crear scripts o aplicaciones que requieran de dicha interacción.

XML-RPC

El protocolo *XML-RPC* se basa en la utilización de la invocación remota a funciones o métodos a través de la codificación de los datos necesarios para la llamada en el formato XML. Para el transporte de datos se emplea el protocolo HTTP, es por ello que XML-RPC es uno de los protocolos más utilizados para la implementación de *web Services.*

En Python 3 contamos con dos módulos principales para trabajar con XML-RPC. El primero de ellos es *xmlrpc.server* y nos ofrece una serie de funcionalidades para desarrollar un servidor que pueda interactuar con un cliente a través del mencionado protocolo. Por otro lado, el segundo módulo en cuestión es *xmlrpc.client,* diseñado para implementar clientes que puedan interactuar con cualquier servidor que sea capaz de entender y trabajar con este protocolo. Ambos módulos forman parte de la librería estándar de Python, lo que significa que podemos utilizarlos directamente, sin necesidad de realizar ninguna instalación adicional.

A continuación, veremos una serie de ejemplos prácticos, tanto de cliente, como de servidor, para aprender lo básico para trabajar con XML-RPC en Python. Comenzamos por el módulo que se ocupa del servidor.

xmlrpc.server

Para ilustrar el funcionamiento de este módulo, vamos a implementar un sencillo servidor que contendrá dos funciones diferentes que podrán ser llamadas desde un cliente a través de, lógicamente, XML-RPC. En realidad, vamos a desarrollar un sencillo *web Service* que correrá en nuestra máquina local. El mismo código puede ejecutarse en un servidor accesible a través de Internet. Es decir, con *xmlrpc.server* podemos desarrollar cualquier *web Service* en Python capaz de interactuar con clientes, escritos o no en el mismo lenguaje.

Nuestro web Service contendrá una función llamada *say_bye(),* la cual recibirá como parámetro una cadena de texto y devolverá un sencillo mensaje, concatenando la cadena pasada como parámetro. Además, el mencionado *web*

service también contará con una clase con un método, al que llamaremos *say_hello()*, que devolverá el clásico *Hola Mundo!*. Tanto la función como el método en cuestión serán accesibles directamente como funciones desde un cliente XML-RPC.

Dos clases del módulo *xmlrpc.server* son fundamentales para la implementación de un servidor: *SimpleXMLRPCServer* y *SimpleXMLRPCRequestHandler.* Ambas clases contienen las funcionalidades necesarias para crear un servidor que pueda entender las peticiones que se realizan desde un cliente y responder a las mismas a través de las funciones previamente definidas.

El primer paso en la implementación de nuestro servidor será crear una clase que herede de *SimpleXMLRPCRequestHandler* y que contenga una variable indicando la ruta (*path*) a través del cual será accesible nuestro *web service.* El código Python para ello sería el siguiente:

```
>>> from xmlrpc.server import SimpleXMLRPCServer
>>> from xmlrpc.server import SimpleXMLRPCRequestHandler
>>> class RequestHandler(SimpleXMLRPCRequestHandler):
...   rpc_paths = ('/mywebservice',)
```

Seguidamente, crearemos una instancia de la clase *SimpleXMLRPCServer* a la que pasaremos dos parámetros diferentes. Uno de ellos será un tupla que contendrá el nombre o IP del servidor más el puerto donde correrá el *web service*. El segundo parámetro indicará cuál será la clase que manejará las peticiones recibidas. Dado que estamos trabajando con nuestra máquina, en local, indicaremos como servidor *localhost.* Como puerto vamos a elegir, por ejemplo, el 8080. La mencionada clase que gestionará las peticiones de los clientes será la que hemos definido previamente. Así pues, el código en cuestión para crear la instancia de nuestro *web service*, es el siguiente:

```
>>> servidor = SimpleXMLRPCServer(("localhost", 8080),
requestHandler=RequestHandler)
```

Ahora llega el momento de crear nuestra primera función invocable desde el cliente. Simplemente definiremos la función y la *registraremos* empleando el método *register_function().* Si no registramos la función, no será posible su invocación. No olvidemos este paso, pues es importante. A continuación, el código necesario que define nuestra función y que la registra en el servidor:

```
>>> def say_bye(name):
...   return("Bye, bye {0}".format(name))
...
>>> servidor.register_function(say_bye)
```

El módulo *xmrpc.server* no solo nos permite definir funciones, también es posible crear clases con sus correspondientes métodos y hacer estos accesibles a los clientes que interactúen con el *web service.* Para ello, deberemos seguir los mismos pasos que hemos visto previamente para las funciones, es decir, bastará con definir nuestra clase y *registrarla.* La principal diferencia es que emplearemos el método *register_instance(),* en lugar de *register_function().* El código en cuestión para ambas operaciones es el que viene a continuación:

```
>>> class MySer:
...   def say_hello(self):
...      return 'Hola Mundo!'
...
>>> servidor.register_instance(MySer())
```

Con el objetivo de que los clientes puedan tener acceso a las funciones que expone nuestro *web service,* vamos a invocar a un método determinado, tal y como muestra la siguiente línea de código:

```
>>> servidor.register_introspection_functions()
```

Una vez definidas y registradas la clase y la función de nuestro servidor, solo nos quedará lanzarlo para que los clientes puedan trabajar con él. Este paso se lleva a cabo a través de un método específico llamado *server_forever():*

```
>>> servidor. serve forever()
```

Ya tenemos nuestro servidor web desarrollado y ejecutándose, desde este momento, los clientes, escritos en cualquier lenguaje que permita escribir clientes que se ajusten a las especificaciones del protocolo XML-RPC, podrán conectarse a nuestro servidor y realizar las llamadas a las funciones expuestas por el mismo. En el siguiente apartado, vamos a escribir uno de estos clientes en Python de esta forma, descubriremos cómo el módulo *xmlrpc.client* puede ayudarnos a ello.

xmlrpc.client

En Python el módulo *xmlrpc.client* de su librería estándar nos ofrece una serie de funcionalidades para implementar sencillamente clientes que interactúen con servidores a través de XML-RPC. Como ejemplo, vamos a desarrollar un cliente que se pueda conectar al servidor que hemos creado en el apartado anterior y que invoque a sus funciones definidas.

La clase principal que permite establecer una conexión con el servidor se llama *ServerProxy.* A través de una instancia de esta clase podremos, directamente, invocar a los métodos que nos ofrezca el servidor. Así pues, el primer paso será realizar la mencionada conexión; sirva el siguiente código como ejemplo:

```
>>> from xmlrpc.client import ServerProxy
>>> url = 'http://localhost:8080/mywebservice'
>>> cli = ServerProxy(url)
```

Efectivamente, la variable *url* contiene tanto el nombre y puerto del servidor, como la ruta que han sido indicados previamente en nuestro servidor web. Por otro lado, la variable *cli* será la instancia que emplearemos para llamar a las funciones del servidor. Antes de continuar, invocaremos a un método, accesible a través de nuestra instancia de cliente, que nos mostrará las funciones expuestas en el servidor web:

```
>>> cli.system.listMethods()
['say_bye', 'say_hello', 'system.listMethods', 'system.methodHelp',
'system.methodSignature']
```

Como el lector habrá podido comprobar, los dos primeros valores de la lista devuelta por el método *listMethods()* corresponden a las funciones que pueden ser invocadas desde el cliente. El resto de valores hacen referencia a otras funciones accesibles, por defecto, a través de *system*, que a su vez tiene acceso desde la instancia de nuestro cliente. De esta forma, ya estamos listos para invocar a las funciones de nuestro web service, tal y como muestra el siguiente código:

```
>>> cli.say_hello()
Hola Mundo!
>>> cli.say_bye('Lucas')
Bye, bye Lucas
```

Si en lugar de emplear el intérprete de Python desde la línea de comandos,

creamos un fichero, copiamos el código y lo lanzamos desde un terminal, comprobaremos cómo cada petición desde el cliente origina un mensaje de *log* en la terminal donde se está ejecutando nuestro servidor. Por ejemplo, justo después de invocar al servidor desde el cliente, obtendremos una línea en la terminal, donde ha sido lanzado el servidor, como la siguiente:

```
localhost - - [20/Feb/2012 11:13:53] "POST /mywebserviceHTTP/1.1" 200
-
```

Tal y como hemos podido comprobar, a través de un sencillo ejemplo, Python pone a nuestra disposición todo lo básico para trabajar con servidores y clientes XML-RPC en dos sencillos módulos que, además, forman parte de su librería estándar. De esta forma, es muy fácil escribir tanto clientes, como servidores, ya que toda la funcionalidad y gestión a bajo nivel del protocolo, está encapsulada en las diferentes clases, métodos y funciones que nos ofrecen los mencionados módulos.

CORREO ELECTRÓNICO

De los servicios utilizados a través de Internet, sin duda el correo electrónico es uno de lo más utilizados diariamente por millones de personas. Para el envío y recepción de correo electrónico son varios los protocolos de red empleados. Entre los más populares se encuentran IMAP4, POP3 y SMTP. Este último se utiliza para el envío, mientras que IMAP4 y POP3 se encargan de la recepción.

Gracias a una serie de módulos contenidos en la librería estándar de Python, podemos escribir programas que se encarguen de trabajar con los protocolos anteriormente mencionados para enviar y recibir correo electrónico. Por ejemplo, podemos desarrollar un sencillo script que se conecte a un servidor SMTP y realice el envío de uno o varios correos. Incluso podríamos escribir nuestra propia aplicación cliente de correo, con funcionamiento similar a los populares *Thunderbird* y *Outlook*, para enviar y recibir correo electrónico.

En este apartado nos centraremos en tres módulos específicos que permiten interactuar con servidores de IMAP4, SMTP y POP3 desde Python. En concreto nos ocuparemos de *poplib*, *imaplib* y *smtplib*. Comenzaremos por el primero de ellos.

pop3

El módulo *poplib* es el encargado de manejar las operaciones necesarias, utilizando el protocolo POP3, para interactuar con servidores de correo electrónico. Básicamente, cuenta con dos clases principales, *POP3* y *POP3_SSL*. Ambas implementan la misma funcionalidad, con la diferencia de que la segunda permite manejar conexiones al servidor a través del protocolo seguro SSL. De hecho, *POP3_SSL* está implementada como una subclase de *POP3*.

Entre los métodos disponibles en las mencionadas clases encontramos los que nos permiten realizar una conexión y desconexión al servidor, pasar un usuario y contraseña, listar los mensajes disponibles para dicho usuario, obtener Información sobre un buzón y, cómo no, obtener los correos recibidos.

Para comenzar a trabajar con *poplib*, lo primero que deberemos hacer es

establecer una conexión con el servidor. Ello se lleva a cabo fácilmente a través de una instancia de la clase *POP3*, que recibirá dos parámetros principales: el nombre o IP del servidor más el puerto donde se está ejecutando. Por defecto, se utilizará el puerto 110, que es el empleado por convención por los servidores POP3. Adicionalmente, se puede emplear un tercer parámetro (*timeout*) para indicar el número de segundos que se deben esperar para obtener una respuesta del servidor, si en esos segundos no obtenemos respuesta, la conexión será fallida y no será establecida. Las siguientes líneas de código muestran cómo conectarnos a un servidor POP3:

```
>>> from poplib import P0P3
>>> servidor = POP3('miservidor.com')
```

Si la conexión se ha realizado satisfactoriamente, estaremos en condiciones de conectarnos al buzón de un determinado usuario. Para ello, emplearemos dos métodos diferentes, uno para pasar el nombre de usuario y otro para indicar su contraseña:

```
>>> servidor.user('nombre_de_usuario')
>>> servidor,pass_('password_de_usuario')
```

Ahora que ya tenemos acceso al buzón del usuario, obtengamos, por ejemplo, el número de correos que existen en su buzón. Esta acción es tan sencilla como invocar al método *list()*, que nos devolverá una lista con todos los correos recibidos, además la longitud del segundo elemento hará referencia al número de ellos. El siguiente código nos muestra un ejemplo de utilización del mencionado método:

```
>>> num = len(servidor.list()[1])
>>> "El usuario tiene {0} mensajes".format(num)
El usuario tiene 5 mensajes
```

El método *retr()* es el encargado de obtener el contenido de los mensajes que se encuentran en un buzón. Por ejemplo, para imprimir por pantalla todos los correos de nuestro usuario ejemplo, podríamos emplear el siguiente código:

```
>>> i = 0
>>> for i in range(num):
...   for mensaje in servidor.retr(i+1)[1]:
...      print(mensaje)
...
```

En cuanto terminemos de realizar las operaciones que necesitemos con el servidor P0P3, deberemos cerrar la conexión al servidor. Esta acción se lleva a cabo a través del método *quit()*, tal y como muestra la siguiente sentencia:

```
>>> servidor.quit()
```

En ocasiones es interesante obtener una traza de lo que está ocurriendo al interactuar con el servidor. Es por ello, que existe el método *set_debuglevel()* que imprime por pantalla información sobre la acción que produce en el servidor cualquier operación que ejecutemos a través de la clase *POP3*. El mencionado método para depuración admite como parámetro un entero que indica el nivel de log que deseamos sea mostrado. Cuanto mayor sea este número, mayor será la información que obtenemos.

Previamente hemos mencionado que la clase *POP3_SSL* cuenta con la misma funcionalidad que *POP3*. Como ejemplo de utilización de *POP3_SSL* vamos a conectarnos al servidor de *GMail*, que requiere este tipo de autenticación y, además, fijaremos el nivel de depuración a 2, con el fin de obtener información ofrecida por el servidor sobre lo que está ocurriendo cuando invocamos a los métodos de la clase empleada para la conexión. El siguiente código muestra cómo realizar la mencionada conexión y la información ofrecida por el servidor al pasar nuestro usuario:

```
>>> from poplib import P0P3_SSL
>>> ser = P0P3_SSL('pop.gmail.com', 995)
>>> ser.set_debuglevel(2)
>>> ser.user('arturofernandezm@gmail.com')
*cmd* 'USER arturofernandezm@gmail.com'
*put* b'USER arturofernandezm@gmail.com'
*get* b'+OK send PASS\r\n'
*resp* b'+OK send PASS'
b'+OK send PASS'
>>> ser.quit()
*cmd* 'QUIT'
*put* b'QUIT'
*get* b'+OK Farewell.\r\n'
*resp* b'+OK Farewell.'
b'+OK Farewell.'
```

No olvidemos que GMail emplea como usuario de correo el nombre seguido de la *arroba* más el dominio. Sin embargo, otros servidores de correo utilizan nombres de usuario que no contienen ni la *arroba* ni el dominio en cuestión.

Además, el puerto empleado por GMail no es el estándar, sino el 995, es por ello que hemos indicado el mismo al crear nuestra instancia de la clase *POP3_SSL*.

Ahora que hemos aprendido lo básico para conectarnos y acceder a la información de un determinado usuario en un servidor de correo electrónico compatible con POP3, es el momento de pasar a ver cómo trabajar con otro protocolo diferente que nos permita enviar correos, en lugar de tener acceso a los recibidos.

smtp

El protocolo de red para enviar correos más utilizado es SMTP (*Simple Mail Transfer Protocol).* Python cuenta en su librería estándar con el módulo llamado *smtplib,* el cual permite conectarse a cualquier servidor que cuente con un servidor de correo, que emplee el mencionado protocolo para enviar correos electrónicos.

De forma similar a *poplib,* el módulo *smtplib* cuenta con dos clases principales, una para realizar la conexión no segura y otra para emplear el protocolo seguro SSL. Los nombres de ambas clases son, respectivamente, *SMTP* y *SMTP_SSL*. La utilización de una u otra clase dependerá del servidor al que vayamos a conectarnos. Entre los métodos ofrecidos por ambas clases encontramos los que nos permiten conectarnos y desconectarnos de un servidor, pasar usuario y contraseña, y por supuesto, realizar el envío de un correo electrónico.

Para conectarnos a un servidor SMTP basta con crear una instancia de la mencionada clase *SMTP o SMTP_SSL,* en función de si el servidor trabaja con SSL o no. Como parámetros deberemos pasar el nombre o IP del servidor y el puerto de conexión, siendo obligatorio únicamente el primer parámetro, ya que, por defecto, si no se indica ningún valor, se empleará el puerto estándar para SMTP, que es el 25. El siguiente código establece la conexión a un servidor determinado:

```
>>> from smtplib import SMTP
>>> servidor.SMTP('miservidor.com')
```

Una vez que tenemos la conexión realizada, si el servidor así lo requiere, será

necesario pasar el nombre de usuario y contraseña del usuario en cuestión que va a realizar el envío del correo electrónico. Para ello, basta con invocar al método *login(),* tal y como podemos observar en la siguiente sentencia:

```
>>> servidor.login('usuario', 'password')
```

En este momento, estamos en condiciones de realizar el envío del correo electrónico en cuestión. El método que nos permitirá hacerlo se llama *sendmail()* y requiere de varios parámetros para su invocación. El primero de ellos hace referencia a la dirección desde la que se realizará el envío. Como segundo parámetro debe indicarse la dirección del destinatario. El tercer parámetro será el cuerpo del correo en cuestión. Opcionalmente se puede indicar también el asunto del mensaje como otro parámetro adicional. Así pues, para nuestro ejemplo vamos a crear tres variables diferentes antes de invocar al método que realiza el envío del correo electrónico propiamente dicho:

```
>>> email_from = 'miusuario@miservidor.com'
>>> email_to = 'destinatario@miservidor.com'
>>> email_body = 'Mensaje de prueba'
>>> servidor.sendmail(email_from, email_to, email_body)
```

En caso de que el método *sendmail()* falle, se lanzará una excepción diferente en función del error producido. Para estos casos, el módulo *smtplib* cuenta con cuatro excepciones diferentes que son: *SMTPRecipientRefused, SMTPHeloError, SMTPSenderRefused* y *SMTPError.* Capturando estas excepciones podremos indicarle al usuario qué tipo de error ha ocurrido y actuar en consecuencia.

Al igual que hemos visto previamente en el caso de *poplib,* en cuanto finalicemos la interacción con el servidor de SMTP deberemos cerrar la conexión que hemos creado al principio. Bastará con invocar al método *quit(),* tal y como muestra la siguiente sentencia:

```
>>> servidor.quit()
```

Otro método análogo al *set debuglevel()* de *poplib* es el homónimo que existe en *smtplib.* La invocación debe realizarse a través de la instancia creada de la clase *SMTP o SMTP SSL*

Por otro lado, el método *starttls* permite emplear TLS (*Transport Layer Security)* para aquellos servidores que lo requieran. Adicionalmente, *smtplib* cuenta también con un método llamado *helo()* que permite ejecutar el comando

del mismo nombre en el servidor.

imap4

IMAP son las siglas de *Internet Message Access Protocol*, un protocolo para el acceso a servidores de correo, consulta de buzones y descarga de mensajes. IMAP4 hace referencia a la versión 4 del mencionado protocolo, siendo esta la más reciente y sustituyendo a la versión 3, ampliamente utilizada durante muchos años.

Junto con POP3, IMAP4 es uno de los dos protocolos más utilizados en la actualidad para obtener correo electrónico desde un servidor. Además, la mayoría de clientes de correo son compatibles con ambos protocolos. Por otro lado, servidores que ofrecen correo electrónico de forma gratuita, como GMail, sorportan ambos protocolos además de, obviamente, la interfaz web.

En Python el módulo que ofrece la interacción con servidores IMAP4 se llama *imaplib* y cuenta con tres clases principales: *IMAP4*, *IMAP4_SSL* y *IMAP4_stream*. Las dos primeras son análogas, con la excepción de que la segunda permite el acceso a través de SSL. La tercera en cuestión soporta la utilización de la entrada y salida estándar como descriptores de ficheros, permitiendo la ejecución de comandos y salida de resultados empleando las mismas.

De forma similar a como hemos aprendido previamente, en el caso de POP3 y SMTP, *imaplib* ofrece métodos para conectarnos a un servidor, autenticarnos e interactuar con el mismo. La conexión al servidor es sencilla, basta con crear una instancia de la clase *IMAP4*, tal y como podemos apreciar en el siguiente ejemplo:

```
>>> from imaplib import IMAP4
>>> con = IMAP4()
```

A diferencia de las clases *SMTP* y *POP3*, al constructor de *IMAP4* no le hemos pasado ningún parámetro. En este caso y por defecto, el servidor utilizado será *localhost* y el puerto en cuestión será el 143, siendo este el que usan por convención los servidores. Obviamente, podemos pasar al mencionado constructor diferentes valores para estos parámetros, siendo el primero el

servidor y el segundo el puerto del mismo.

La autenticación se lleva a cabo a través del método *login()*, que necesita el usuario y contraseña como parámetros:

```
>>> con.login('usuario', 'password')
```

Una vez establecida la conexión y realizada la autenticación, en caso de que ello sea necesario, estaremos en disposición de interactuar con el servidor. Los mensajes de un determinado buzón pueden ser leídos a través del método *search()*, que se utiliza para buscar mensajes que cumplan un criterio dado. Antes de invocar a este método, debemos seleccionar el buzón que va a ser leído. Esta acción se ejecuta gracias al método *select()*, que recibe como parámetros el nombre del buzón y un *flag* para indicar si el acceso será de solo lectura o no. Si el mencionado método no recibe ningún parámetro, el buzón del usuario previamente autenticado será el utilizado. Así pues bastará con invocar a este método para seguir con nuestro ejemplo:

```
>>> con.select()
```

Ahora que tenemos seleccionado nuestro buzón, podemos obtener, por ejemplo, todos los mensajes que existen en el mismo. Esto es tan sencillo como ejecutar la siguiente sentencia:

```
>>> tipo, datos = con.search(None, 'ALL')
```

El anterior método devolverá una tupla donde el segundo elemento de la misma es una lista que contendrá los datos referentes a los mensajes encontrados y que cumplen con el criterio de búsqueda. Como parámetros del método *search()*, hemos utilizado *None* para indicar que vamos a emplear el conjunto de caracteres por defecto del servidor y la cadena de texto *ALL* para indicar que deseamos recuperar todos los mensajes del buzón. En caso, por ejemplo, de necesitar todos los mensajes enviados desde una dirección concreta, bastaría con pasar un par de parámetros adicionales. Supongamos que deseamos obtener todos los mensajes recibidos y que han sido enviados por la cuenta *prueba@miservidor.com*. En este caso, realizaríamos la siguiente llamada:

```
>>> tip, data = con.search(None, 'FROM',
                           '"prueba@miservidor.com"')
```

Volviendo a nuestra llamada anterior, cuando esta sea ejecutada y tengamos los valores *tipo* y *datos*, podremos escribir un bucle para, por ejemplo, imprimir por pantalla todos los mensajes obtenidos del buzón. Sirva para ello el siguiente código:

```
>>> for mensaje in datos [0] .split():
typ, data = con.fetch(mensaje, '(RFC822)')
print('Mensaje {O}: {1}'.format(mensaje, data[0][1]))
...
```

Efectivamente, el método *fetch()* es el responsable de obtener la información de cada mensaje, ya que *search()* nos devuelve la información que debemos procesar. Como primer argumento de *fetch()* hemos pasado, precisamente, información recogida gracias a *search()*. Por otro lado, el segundo argumento hace referencia al estándar *RFC822* que define el formato de los mensajes que deben tener los correos electrónicos.

No olvidemos que, al finalizar nuestra interacción con el servidor IMAP4, deberemos desconectarnos del mismo, siendo esto posible gracias al método *close()*. Además, si nos hemos autenticado previamente, deberemos hacer *logout*. Dado que así lo hemos hecho en nuestro ejemplo, para finalizar nuestro trabajo con *imaplib* invocaremos a ambos métodos:

```
>>> con.close()
>>> con.logout()
```

Aparte de las funcionalidades básicas que hemos comentado, *imaplib* pone a nuestra disposición otras más avanzadas, como, por ejemplo, la copia de un mensaje a un determinado buzón, el borrado completo de un buzón, el listado de los nombres de todos los buzones que cumplen una determinada condición o la posibilidad de trabajar con ACL *(Access Control List)*.

En contraste con POP3, la interacción con IMAP4 desde Python es más compleja. Ello se debe a que, de por sí, el protocolo IMAP4 utiliza un acceso diferente a los buzones y que, además, ofrece la posibilidad de ejecutar distintas acciones.

WEB

La web es uno de los servicios que más peso tiene en Internet, junto con el correo electrónico. Los sitios web ya se cuentan por millones y cada vez se encuentran más tipos de aplicaciones que ofrecen su funcionalidad a través de la web. Tanto es así que el desarrollo de aplicaciones web es uno de los campos a los que más profesionales se dedican dentro del ámbito de la ingeniería del software. Python es una buena opción a la hora de elegir un lenguaje de programación para desarrollar aplicaciones web, tal y como podremos comprobar a continuación. Una de las principales razones para ello es la cantidad de módulos, tanto dentro como fuera de la librería estándar, que existen para interactuar con la web, ya sea, a través de protocolos como CGI y WSGI, o empleando modernos *frameworks* web.

En este apartado también aprenderemos a aplicar la técnica del *web scraping* para acceder, de forma automática, a la información que ofrecen los sitios web.

CGI

Durante largos años, el estándar CGI *(Common Gateway Interface)* fue la técnica más utilizada para generar páginas web dinámicas. Esta técnica consiste en ejecutar un programa en el servidor que se encarga de construir una página web como respuesta a una petición que se realiza desde un cliente web, siendo el más habitual de estos un navegador. El lenguaje de programación Perl fue uno de los más utilizados para desarrollar estos programas de servidor a los que se les llamaba *Scripts CGI.* La forma de procesar la petición y generar la respuesta se realiza basándose en una serie de reglas que define el estándar CGI (RFC 3875). Floy en día, aún se utiliza el CGI para construir sitios web dinámicos, aunque de forma minoritaria. Ello es debido a que existen otras opciones que ofrecen ventajas significativas relacionadas con la seguridad y la eficiencia. Incluso se han llegado a desarrollar soluciones específicas para algunos lenguajes y tecnologías, como es el caso de los *servlets* de Java, *Rack* para Ruby, *mod_php* para PHP y *WSGI* para Python.

En ocasiones puede ser interesante escribir un script CGI para, por ejemplo, dotar de una interfaz web a un programa que ya tenemos desarrollado. Supongamos que tenemos un script escrito en Python que se encarga de analizar una serie de ficheros de log y mostrar información sobre los mismos. Adaptarlo para que pueda mostrar el resultado a través de una página web es fácil gracias a CGI. Además, no necesitaremos montar ningún servidor de aplicaciones ni ninguna compleja arquitectura de servidores web. Bastará con que el script se ejecute en cualquier servidor web que pueda ejecutar Scripts CGI. En la práctica, la mayoría de los modernos servidores web tienen esta capacidad, entre ellos *Apache, nginx* y *lighttpd.*

En la librería estándar de Python encontramos dos módulos que nos permitirán desarrollar scripts CGI, nos referimos a *cgitb* y a *cgi.* Ambos módulos nos ofrecen funcionalidades para analizar la petición enviada desde el cliente, generar contenido en formato HTML, leer valores de variables pasadas por el método *GET* y procesar formularios.

Para realizar pruebas es conveniente contar con un servidor web capaz de interpretar y gestionar CGI. No vamos a entrar en detalle cómo configurar el servidor y daremos por hecho que el lector posee los conocimientos básicos sobre el funcionamiento de CGI. Los usuarios de Mac OS X lo tienen sencillo, ya que, por defecto, Apache está instalado y configurado para ejecutar CGI. Los scripts CGI deben residir en el directorio */Library/WebServer/CGI-Executables/.*

Como ejemplo, vamos a escribir un sencillo script que lanzará un mensaje de bienvenida, leyendo el nombre de una persona que pasaremos como parámetro, a través del método *GET* de HTTP. La lectura de este tipo de variables y de aquellas enviadas por el método POST, como ocurre habitualmente con los formularios, se lleva a cabo a través de la clase *FieldStorage(),* la cual pertenece al módulo *cgi.* A continuación, reproducimos el código completo de nuestro primer CGI en Python:

```
#!/usr/local/bin/python3
import cgitb
import cgi

cgitb.enable()

vars = cgi.FieldStorage()
nombre = vars.getvalue('nombre')

print("Content-type: text/html")
```

```
print()
print("<html>")
print("<body>")
print("<h2>Bienvenid@ {0}</h2>".format(nombre))
print("</body>")
print("</html>")
```

Para probar nuestro script, deberemos copiar el código en un nuevo fichero al que llamaremos, por ejemplo, *hola.py.* Luego pasaremos a alojarlo en el directorio determinado del servidor web donde deben residir los CGI's según la configuración del mismo. Los permisos de ejecución del fichero deben estar activos, en otro caso obtendremos un error al realizar la petición desde el navegador. En cuanto estemos listo para la prueba, abriremos un navegador y nos dirigiremos a la siguiente URL:

```
http://localhost/cgi-bin/hola.py?nombre=Lucas
```

Si el script se ejecuta correctamente, veremos un mensaje como el que muestra la figura:

Figura. Página web generada por el script CGI

En la URL hemos pasado una variable llamada *nombre* con el valor *Lucas;* para recoger su valor, hemos creado una instancia de la clase *FieldStorage().* El método *getvalue()* de la mencionada clase nos da acceso a la variable pasada por GET.

Por otro lado, la primera línea de nuestro script es necesaria para indicar dónde se encuentra el intérprete de Python que debe utilizarse para procesar el código del script en cuestión. El método *enable()* se ejecuta para llevar a cabo una serie de inicializaciones necesarias para trabajar con CGI. Con la primera sentencia *print* comienza a generarse la salida HTML que será enviada al

navegador. Es por ello, que lo primero que hacemos es fijar el tipo de contenido, en este caso será, obviamente, HTML. También es posible indicar, por ejemplo, que la salida será texto o *json*, útil para responder a llamadas realizadas vía AJAX. Justamente después, necesitamos una llamada a *print()* sin parámetros para indicar que comienza a generarse lo que será el código HTML propiamente dicho. A partir de este punto construimos la página HTML de salida, empleando para ello una serie de etiquetas HTML y el valor de la variable *nombre* que hemos recogido gracias a la clase *FieldStorage().*

WSGI

Originalmente los *frameworks* para desarrollar aplicaciones web en Python presentaban la restricción de que cada uno de ellos necesitaba un determinado servidor web que implementara la interfaz necesaria para la comunicación entre ambos. De esta forma, por ejemplo, si un framework había sido desarrollado utilizando *mod_python* como interfaz, el servidor web Apache era requerido para la ejecución de aplicaciones. Con objetivo de eliminar este requisito restrictivo se desarrolló WSGI *(Web Server Gateway Interface),* una interfaz de bajo nivel que permite la comunicación entre diferentes frameworks y servidores web, haciendo portables las aplicaciones entre ambos componentes.

La interfaz WSGI define dos componentes diferenciados, uno es el *servidor* o *gateway* y otro es la aplicación o framework que la sirve. El *gateway* se encarga de gestionar el intercambio de información entre el cliente y la aplicación propiamente dicha.

Para trabajar con WSGI, Python pone a nuestra disposición el módulo *wsgiref* de su librería estándar, el cual incluye funcionalidades para procesar variables de entorno, procesar peticiones y generar respuestas para un cliente web.

Como ejemplo de utilización del módulo *wsgiref,* vamos a construir un sencillo servidor web capaz de manejar peticiones WSGI y responder a las mismas generando contenido que pueda ser leído por un navegador web. Nuestro sencillo servidor devolverá el famoso mensaje *Hola Mundo!* cuando un navegador invoque a una URL determinada a la que responderá nuestro servidor. Antes de comenzar y para probar nuestro servidor, todo el código que vamos a ir

mostrando y explicando debe salvarse en un fichero, el cual lanzaremos por la línea de comandos. La primera línea de código se encargará de importar la función necesaria para crear el mencionado servidor web:

```
from wsgiref.simple_server import make_server
```

Justo después definiremos una función que responderá a la URL raíz de nuestro servidor, devolviendo el mencionado mensaje:

```
def simple_app(environ, start_response):
    status = '200 OK'
    header = [('Content-type', 'text/html; charset=utf-8')]
    start_response(status, header)
    ret = b"<html><body><h2>Hola Mundo</h2></body></html>"
    return [(ret)]
```

La variable *status* devolverá el valor 200, que es el código HTTP estándar para indicar que la respuesta se ha generado correctamente. Por otro lado, la variable *header* Indicará que vamos a generar contenido HTML, siendo el conjunto de caracteres elegido el UTF-8. Por último, devolveremos el código HTML que generará en el cliente la página web de respuesta. Observemos cómo deberemos devolver una lista que contenga una tupla, en nuestro caso, con un único valor.

Ya solo nos queda crear nuestro servidor web y hacer que se ejecute. Estas acciones son llevadas a cabo por la función *make_server()* y por el método *serve_forever()*, respectivamente. Además, vamos a lanzar un mensaje que indique que el servidor ha sido iniciado. El código necesario para todo ello sería el siguiente:

```
if __name__ == '__main__':
httpd = make_server('', 8080, simple_app)
print("Lanzando servidor en puerto 8080...")
httpd.serve_forever()
```

Ahora solo nos queda invocar a nuestro script por línea de comandos, en cuanto sea lanzado apreciaremos que obtendremos un mensaje como el siguiente:

```
$python servidor_wsgi.py
Lanzando servidor en puerto 8080...
```

Si abrimos nuestro navegador y nos conectamos a la URL

http://localhost:8080, obtendremos como respuesta una sencilla página con el mencionado *Hola Mundo!* Volviendo a la terminal donde hemos lanzado nuestro servidor, comprobaremos que aparece una nueva línea:

```
localhost - - [21/Feb/2012 12:59:21] "GET / HTTP/1.1" 200 45
```

La anterior línea es similar a la lanzada por el servidor web Apache, la cual habitualmente se refleja en el fichero *access_log*.

Tal y como el lector habrá podido comprobar, con muy pocas líneas de código, Python nos ofrece todo lo necesario para construir un servidor WSGI capaz de responder a peticiones HTTP. Diferentes frameworks se basan en el módulo *wsgiref* para desarrollar sus propios servidores, ajustando así como manejar peticiones y respuestas y ofreciendo, por ejemplo, capas adicionales de *middleware*.

Web scraping

El *web scraping* es la técnica mediante la cual se extraen datos de una determinada página web. Habitualmente se utiliza el *web scraping* para simular el comportamiento de una persona en la navegación de un sitio web, a través de un programa. De esta forma, se automatiza la conexión a un determinado sitio web, la navegación sobre el mismo y la final extracción de la información que contiene. Los robots que se dedican al indexado de la web, conocidos como *bots*, hacen uso de esta técnica para poder leer el contenido de los sitios web. Otros ejemplos de aplicaciones que emplean el *scraping* son los comparadores de precios y aquellas que utilizan datos ofrecidos por sitios de terceros para integrarlos con sus propios sitios web.

Para realizar el *web scraping* necesitamos llevar a cabo varias tareas como la conexión a un sitio web, la autenticación de diferentes tipos, la gestión de *cookies*, la navegación a través de distintos enlaces incluidos en el propio sitio web y el análisis de la estructura HTML para obtener la información que contiene. Para esta última fase necesitaremos leer un documento HTML y ser capaces de extraer información analizando las distintas etiquetas que contiene. Existe un módulo para Python que podemos emplear para ello, su nombre es *Ixml*. Por otro lado, el resto de acciones comentadas que forman parte del

scraping pueden ser llevadas a cabo por un módulo de la librería estándar de Python denominado *urllib.request.* De ambos módulos nos ocuparemos a continuación.

URLLIB.REQUEST

El módulo *urllib.request* ofrece la funcionalidad necesaria para abrir conexiones HTTP y HTTPS, obteniendo el resultado ofrecido por un servidor web a través de estos protocolos. Dado que en este proceso puede ser necesario realizar diferentes tipos de autenticación, gestión de *cookies* y redirecciones, *urllib.request* pone a nuestra disposición un conjunto de clases y funciones para realizar estas acciones de forma fácil y eficiente.

La principal función que nos servirá para conectarnos a una URL determinada es *urlopen(),* la cual recibirá la URL en cuestión como parámetro. Supongamos que vamos a hacer *web scraping* para obtener la previsión del tiempo para la ciudad de Madrid para los próximos cinco días. El servicio *Weather* de *Yahoo!* ofrece esta información directamente en una página web concreta, cuya URL es *http://weather.yahoo.com/spain/madrid/madrid-766273/?unit=c*. Así pues, para conseguir esta información de forma automática, deberemos conectarnos a la mencionada URL, leer su contenido y analizarlo. El primer paso lo vamos a realizar empleando la mencionada función *urlopen(),* mientras que del segundo se encargará el módulo *Ixml,* tal y como veremos más adelante. De esta forma, procederemos a importar el correspondiente módulo y a invocar a la función en cuestión:

```
>>> import urllib.request
>>> url = 'http://weather.yahoo.com/spain/madrid/madrid- 766273/?
unit=c'
>>> pagina = urllib.request.urlopen(url)
```

Seguidamente, cargaremos el contenido leído en una cadena de texto para que su posterior análisis sea más sencillo. Para ello, basta con invocar al método *open()* del objeto *HTTPResponse,* que a su vez fue devuelto por la función *urlopen():*

```
>>> contenido = pagina.read()
```

Si en este momento hacemos un *print* de la variable *contenido,* observaremos

cómo tendremos todo el código HTML que forma la página web a la que hemos accedido por la URL en cuestión.

En caso de que la URL a la que nos conectemos requiera de autenticación básica HTTP, podemos emplear la clase *HTTPBasicAuthHandler()*, la cual se encargará de gestionar la autenticación. La siguiente línea muestra cómo instanciar la mencionada clase:

```
>>> auth = urllib.request.HTTPBasicAuthHandler()
```

En cuanto tengamos la instancia, el siguiente paso será añadir la información relativa al usuario y contraseña. Ello puede realizarse empleando el método *add_password()*, tal y como muestran las siguientes líneas:

```
>>> auth.add_password(user='usuario', passwd= 'password')
```

Después invocaremos a un par de métodos para gestionar la autenticación de forma transparente y llamar a la URL en cuestión de la misma forma que si esta no necesitara autenticación:

```
>>> opener = urllib.request.build_opener(auth)
>>> urllib.request.install_opener(opener)
>>> pagina = urllib.request.urlopen('http://localhost/login/')
```

En ocasiones es útil pasar una cabecera *(header)* en la solicitud de una página web. Por ejemplo, para indicar que deseamos utilizar un determinado *User-Agent*. Esto es práctico cuando necesitamos comunicarle al servidor que nuestra petición simula ser un navegador concreto. A través del método *addheaders()*, es posible enviar una cabecera HTTP determinada. Supongamos que vamos a pedir una página web pero queremos identificarnos como si fuéramos el navegador web *Safari*, ejecutado desde la versión 10.6.8 de Mac OS X. El código Python necesario para ello sería el siguiente:

```
>>> opener = urllib.request.build_opener()
>>> opener.addheaders([('User-agent',
                               Mozilla/5.0 (Macintosh; U; Intel Mac OS
X 10_6_8) ')])
>>> opener.open(url)
```

Efectivamente, al método *addheaders()* debemos pasarle una lista que contenga tantas tuplas como valores diferentes admitidos por una cabecera HTTP necesitemos.

En nuestro ejemplo, simplemente hemos indicado un único valor, el del mencionado *User-Agent.*

Cuando navegamos por un sitio web es común hacer uso de las *cookies* para guardar información única de cada sesión que se mantiene abierta por el navegador web. Dado este hecho, al emplear el *web scraping*, es habitual tener que pasar la información de las cookies entre diferentes páginas web del mismo sitio web. Esto es fácil en Python gracias a la clase *HTTPCookieProcessor,* que realiza el trabajo por nosotros de forma transparente. Esta clase acepta en su constructor, como parámetro, un objeto de tipo *CookieJar,* cuya clase se encuentra definida en el módulo *http.cookiejar*, también de la librería estándar de Python. Si la petición a una URL determinada requiere del manejo de cookies, podemos utilizar el siguiente código para realizar la gestión:

```
>>> import http.cookiejar
>>> cookie = http.cookiejar.CookieJar()
>>> handler = urllib.request.HTTPCookieProcessor(cookie)
>>> opener = urllib.request.build_opener(handler)
>>> opener.open(url)
```

Tal y como habremos podido comprobar, el módulo *urllib.request* ofrece lo básico para realizar conexiones y gestiones de cookies, cabeceras HTTP y autenticación. Siendo todas estas operaciones la parte inicial del *web scraping*, nos queda averiguar cómo extraemos los datos del código HTML. De ello nos ocuparemos en el siguiente apartado, que describe el funcionamiento básico del módulo *Ixml.*

LXML

A pesar de que su nombre puede confundir, el módulo *Ixml* permite analizar, tanto ficheros XML como HTML. Desde el punto de vista técnico, este módulo es un *binding* para Python de las populares librerías *libxml2* y *libxslt*, escritas ambas en el lenguaje de programación C. Haciendo uso de la estructura *ElementTree* y las correspondientes clases y funciones, es posible manejar con soltura la estructura de un documento XML y HTML, lo cual resultará muy práctico para el *web scraping.* Básicamente, la estructura *ElementTree* proporciona acceso a los elementos del documento como si las etiquetas y nodos del mismo estuvieran estructurados en forma de árbol. En la actualidad son

muchos los componentes software que se valen de este tipo de estructura para el análisis de documentos estructurados SGML, como lo son el formato XML y el HTML.

Dado que *lxml* no forma parte de la librería estándar de Python, deberemos acudir a un gestor de paquetes, por ejemplo *pip*, para su instalación. Así pues nos bastará con ejecutar la siguiente orden desde la línea de comandos:

```
pip install lxml
```

Las funcionalidades ofrecidas por el módulo *lxml* son extensas y entre ellas se encuentran las que nos permiten utilizar *XPath, XSLT* y diferentes tipos de *parsers*, como son *html5lib* y *BeautifulSoup*. Sin embargo, dado que estamos tratando el tema del *web scraping*, nos centraremos en la parte específica del análisis de HTML ofrecido por *lxml*. En concreto, las clases y métodos para este propósito se encuentran en el módulo *lxml.html*. Para ilustrar de forma sencilla el funcionamiento del mismo, nos centraremos en el ejemplo que empezamos en el apartado anterior. De esta forma, partiremos de que ya tenemos volcado el contenido de la página web del tiempo, ofrecida por *Yahoo!*, en una variable a la que hemos llamado *pagina*. Teniendo este hecho en cuenta, procederemos a buscar la información que necesitamos. Esta se encuentra en el interior de un elemento <*tr*> cuyo atributo *class* tiene el valor *fiveday-temps*. Partiendo de este dato, bastará con invocar al método *find_class()*, el cual nos devolverá un objeto que representa al elemento HTML en cuestión. Además, llamaremos a la función *fromstring()* que se encargará de leer una cadena de texto y formar un objeto que cumpla con la estructura *ElementTree*. Finalmente, solo nos quedará emplear el método *text_content()* para obtener el contenido del elemento HTML que buscamos dentro de la página web en cuestión. El código requerido para todas estas operaciones es el siguiente:

```
>>> from lxml.html import fromstring
>>> doc = fromstring(pagina)
>>> ele = doc.find_class('fiveday-temps')
>>> ele [0] .text_content()
'High: 12\xb0 Low: -3\xbOHigh: 14\xb0 Low: -3\xbOHigh: 17\xb0 Low:
-3\xbOHigh: 17\xb0 Low: -2\xbOHigh: 17\xb0 Low: 1\xb0'
```

Otro práctico método para acceder al contenido de un determinado nodo de un documento HTML es *get_element_id()*, el cual se basa en la búsqueda del atributo *id* de los elementos HTML.

Además de acceder a la información de un documento HTML, también podemos modificar su estructura. De ello se encargan dos métodos diferentes, *drop_tree()* y *drop_tag(),* que permiten borrar un elemento y todos sus hijos y borrar un elemento manteniendo sus hijos y el texto que el mismo contiene, respectivamente.

Para el trabajo con enlaces y formularios presentes en un documento HTML, el módulo *lxml.html* ofrece una serie de fundones adicionales que facilitan en gran medida el trabajo. Ejemplos de ellos son aquellas funciones que nos permiten acceder a todos los elementos de un formulario, las que nos devuelven el tipo de elemento *input* o las que nos permiten iterar directamente por todos los enlaces presentes en el documento.

El lector interesado en profundizar en todos los componentes de *lxml.html* puede consultar la documentación de referencia correspondiente (ver referencias).

Frameworks

Para el desarrollo de aplicaciones web complejas, los *frameworks* web ofrecen diversas herramientas y utilidades para agilizar el desarrollo y facilitar el mantenimiento. En los últimos años su uso se ha popularizado enormemente y en la actualidad es fácil encontrar multitud de ellos para lenguajes como Java, PHP, Ruby y, cómo no, Python. Uno de los más conocidos y utilizados para este último lenguaje es *Django,* el cual, en el momento de escribir estas líneas, aún no es compatible con Python 3. Sin embargo, contamos con otros que sí lo son, como es el caso de *Pyramid* y *Pylatte,* de los que nos ocuparemos en este apartado. Más que hacer un análisis exhaustivo de ellos, nos dedicaremos a ver qué puede ofrecernos cada uno en líneas generales.

PYRAMID

El proyecto *open source* llamado *Pylons Project* es el responsable del desarrollo y mantenimiento del *framework* web *Pyramid.* El desarrollo de este componente software nace con la idea de disponer de un *framework* web que sea fácil de utilizar, minimalista en cuanto al conjunto de componentes de los que se

compone y que sea rápido para la ejecución de aplicaciones web.

La primera versión de Pyramid fue liberada allá por diciembre de 2010, siendo la última versión la 1.3 totalmente compatible con Python 3. Como interfaz de comunicación emplea WSGI y se inspira en otros *frameworks* como *Zope, Pylons* y *Django.*

Pyramid se basa en el patrón de diseño MVC (*Model View Controller*), aunque no lo implementa de la manera tradicional. Es decir, no existen clases o componentes que actúen directamente con un controlador o un modelo. En cambio, una aplicación construida con este *framework* cuenta con un *árbol* de *recursos,* que representa la estructura del sitio web. Adicionalmente existen una serie de *vistas* y un conjunto de clases para representar los modelos de la aplicación. Las vistas son las encargadas de conectar y procesar las peticiones generadas en un cliente para ofrecer una salida basada en la información de los datos representados por los modelos. Sin embargo, y a diferencia de otros *frameworks,* Pyramid no ofrece facilidades para conectar los datos que existan en una base de datos con las clases que definen los modelos. No obstante, sí que ofrece facilidades, para, por ejemplo, integrar un ORM que realice este trabajo de conexión o *mapping.*

En comparación con otros *frameworks* web de tipo *full stack,* Pyramid es minimalista en el sentido de que ofrece lo básico para poder desarrollar aplicaciones web, siendo el programador el responsable de añadir módulos adicionales para implementar funcionalidades avanzadas, como es el caso, por ejemplo, de un ORM para interactuar con una base de datos.

Una de las características principales de Pyramid es que permite, haciendo gala de su carácter minimalista, la creación de aplicaciones web utilizando un único fichero. Lo cual resulta muy cómodo para realizar prototipos o para desarrollar pequeñas aplicaciones.

Pyramid incluye herramientas para que los programadores puedan depurar de forma fácil y rápida sus aplicaciones. Con este objetivo, la barra de herramientas de depuración *(debug toolbar)* ofrece información como, por ejemplo, sobre las variables enviadas en cada petición, la configuración actual del servidor e información sobre el rendimiento de la aplicación.

Gracias a una serie de componentes, la internacionalización y localización de aplicaciones es posible en Pyramid, utilizando como base *gettext* para la generación de cadenas de texto en diferentes idiomas.

Al igual que otros *frameworks,* Pyramid permite trabajar con sesiones HTTP,

definir fácilmente las URL de las rutas accesibles de la aplicación, servir ficheros estáticos y utilizar plantillas *(templates)* para la generación de páginas HTML.

A diferencia de otros, Pyramid ofrece la opción de utilizar eventos durante el ciclo de vida de las peticiones que recibe. De esta forma, es posible crear manejadores de eventos que respondan con una determinada acción cuando se produzca un evento determinado. Incluso es posible definir eventos propios creados por el programador. Ello nos da gran flexibilidad y control sobre la forma de responder a las diversas peticiones originadas desde un cliente web.

La funcionalidad de Pyramid puede ser extendida gracias a los llamados *add-ons,* que son componentes software que permiten utilizar funcionalidades no definidas originalmente en el *framework.* Entre estos componentes, contamos con algunos que permiten integrar diferentes librerías JavaScript, emplear JSON o utilizar un determinado sistema de plantillas.

La instalación de Pyramid puede ser llevada a cabo directamente a través del gestor de paquetes de Python *pip.* Para ello, bastará con ejecutar desde la línea de comandos la siguiente orden:

```
pip install pyramid
```

Durante el proceso de instalación, serán descargados y también instalados una serie de paquetes adicionales que Pyramid requiere para su funcionamiento. Dado que este proceso es transparente, no será necesario llevar a cabo ninguna acción adicional.

En cuanto finalice la instalación podremos comenzar a escribir nuestra primera aplicación. Como sencillo ejemplo, vamos a crear un simple script que responda a una determinada URL generando el mensaje *Hola Mundo!.* Para ello, comenzaremos creando un nuevo fichero, al que llamaremos *hola_mundo.py,* que contendrá las siguientes líneas iniciales de código:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response
```

Seguidamente, pasaremos a crear una función que será ejecutada como respuesta a una petición y que, simplemente, mostrará el mencionado mensaje inicial. El código para la función sería el que mostramos a continuación:

```
def hola_mundo(request):
```

```
    return Response('Hola Mundo!' % request.matchdict)
```

Ahora debemos escribir el código de entrada de nuestra aplicación Pyramid. Inicialmente, instanciaremos la clase que lee la configuración para la aplicación. Dado que estamos trabajando con un ejemplo mínimo, no es necesario crear ninguna configuración adicional. La línea de código que se encarga de ello será la siguiente:

```
config = Configurator()
```

El siguiente paso será establecer la ruta de la URL e indicar qué función debe ejecutarse como resultado de su petición. En nuestro caso, vamos a responder a la URL */hola*, devolviendo un mensaje a través de la función *hola_mundo()*, la cual ha sido previamente definida. El código en cuestión necesario para estas acciones es el siguiente:

```
config.add_route('hola', '/hola')
config.add_view(hola_mundo, route_name='hola')
```

Ya solo nos queda crear un servidor WSGI e indicar que nuestra aplicación debe ser servida por él mismo. Ello es bastante sencillo, siendo este el código que necesitamos:

```
app = config.make_wsgi_app()
server = make_server('0.0.0.0', 8081, app)
server.serve_forever()
```

A través del primer y segundo parámetro de la función *make_server()* hemos indicado que el servidor identificado por la IP 0.0.0.0, es decir, *localhost*, debe servir nuestra aplicación Pyramid en el puerto 8081. Al invocar a nuestra aplicación desde la interfaz de comandos, quedará automáticamente accesible a través del navegador web. Así pues, al pedir la URL *http://localhost/hola*, obtendremos el mensaje *Hola Mundo!*

Una vez descubierto lo básico sobre Pyramid es hora de continuar con *Pylatte* nuestro recorrido por los *frameworks* web para Python 3.

PYLATTE

Los *frameworks* web más populares y utilizados en la actualidad solo son

compatibles con Python 2.x. Este es el caso de *Django*, *Flask*, *Turbogears* y *web2py*. Es por ello que *Pylatte* nace con la idea de desarrollar un *framework* web específicamente diseñado para ser ejecutado con Python 3.

Pylatte es *open source* y la primera versión fue liberada en octubre de 2011, es por lo tanto un proyecto bastante joven. Sin embargo, su estado es estable, con lo que puede ser empleado sin ningún problema.

Una de las características principales de Pylatte es que emplea un formato específico llamado *pyl*, que contiene tanto código HTML como código Python. Internamente, un componente de Pylatte se encarga de analizar y procesar este tipo de ficheros y producir la correspondiente salida en HTML, la cual será enviada al cliente web.

Para la correspondencia entre URL y las acciones que deben ser llevadas a cabo, Pylatte utiliza ficheros XML donde se realiza esta configuración. Pylatte permite aplicar una serie de *filtros*, es decir, acciones que se llevan a cabo previa o posteriormente al procesamiento de una petición. Su configuración también se realiza a través de ficheros XML.

Dado que Pylatte no incluye un ORM como tal, sino una serie de facilidades para interactuar con la base de datos. De momento, solo MySQL está soportado, siendo necesaria la instalación del módulo *MySQLdb* para trabajar con el mencionado gestor de bases de datos relacionales.

Componentes incluidos el framework facilitan la obtención de parámetros recibidos por GET y POST, así como el uso de sesiones HTTP. Pylatte también incluye un servidor web para poder servir las aplicaciones que desarrollemos empleando este framework.

La instalación de Pylatte se puede realizar a partir de su código fuente, el cual se encuentra disponible en la página web de descargas del proyecto (ver referencias).

En comparación con otros frameworks web de Python, Pylatte ofrece menos componentes para facilitar el desarrollo web, siendo los mismos demasiado básicos. Por otro lado, no implementa como tal el patrón MVC y permite la mezcla de código Python y HTML en el mismo fichero, lo que, en términos generales, no es aconsejable. Por otro lado, es un framework totalmente pensado para trabajar con Python 3, lo que supone una ventaja significativa con respecto a sus competidores.

La elección de un determinado framework web no es una tarea sencilla, influyendo en la decisión diversos factores técnicos y humanos. Es conveniente

tener en cuenta aspectos como qué componentes necesitamos teniendo en cuenta la funcionalidad de la aplicación, el rendimiento en tipo de ejecución o cómo es la curva de aprendizaje. Animamos al lector a investigar sobre otros frameworks web para Python con el objetivo de elegir aquel que más se adapte a sus necesidades.

INSTALACIÓN Y DISTRIBUCIÓN DE PAQUETES

INTRODUCCIÓN

En capítulos anteriores nos hemos valido de ciertos módulos que no están presentes en la librería estándar de Python, para tener acceso a determinadas funcionalidades. Para emplear estos módulos, simplemente los hemos *instalado* a través de una herramienta llamada pip. Pero ¿qué es esta herramienta? ¿Cómo podemos instalarla y utilizarla? La primera parte de este capítulo se ocupará de los métodos de instalación de módulos, donde obtendremos respuestas a las preguntas anteriores y aprenderemos lo básico para poder instalar y trabajar con módulos desarrollados por otras personas y que no forman parte de la librería estándar de Python. En concreto, aprenderemos a utilizar el método estándar, proporcionado por el módulo *distutils* y a trabajar con dos *gestores* de paquetes: *pip* y *easy_install.*

Como desarrolladores, nosotros también podemos crear nuestros propios paquetes y distribuirlos para que puedan ser empleados por otras personas. De hecho, este proceso suele formar parte del desarrollo de software. Preparar nuestros programas para que puedan ser distribuidos es una tarea importante que no debe ser pasada por alto. Más aún si vamos a obtener por ello un beneficio económico, ya que nuestros clientes necesitarán instalar el software que hemos desarrollado para ellos. De la misma forma, en el ámbito del FOSS *(Free and Open Source software),* este proceso también es fundamental si queremos que otros puedan trabajar con nuestro software. En la segunda parte del presente capítulo nos ocuparemos del proceso de distribución de software, aprendido a empaquetar nuestros propios módulos.

Cuando hablamos sobre la distribución e instalación de software de terceros en Python, empleamos tanto el término *paquete* como *módulo.* Recordemos que, por convención, un *módulo* de Python puede ser un simple script escrito en este lenguaje, mientras que un *paquete* es un conjunto de ficheros Python que guardan entre sí una relación funcional. Sin embargo, para referirnos a un script

de Python no solemos emplear la palabra *módulo,* sino *programa* o, simplemente *script.* Es por ello que suele ser habitual emplear *módulo* cuando técnica y estrictamente deberíamos decir *paquete.* Este hecho nos lleva a emplear el término *módulo* y *paquete* indistintamente, tal y como hemos hecho a lo largo de los diferentes capítulos de este libro.

La parte final del capítulo estará dedicada a los *entornos virtuales,* los cuales nos permiten tener un control total sobre los módulos que instalamos en nuestro sistema. Explicaremos cómo crear diferentes entornos y veremos cómo ellos nos permiten tener diferentes versiones del mismo módulo instalados en la misma máquina.

INSTALACIÓN DE PAQUETES

La instalación en nuestro sistema de paquetes Python desarrollados por terceros puede ser llevada a cabo, básicamente, de dos formas diferentes. La primera de ellas es a través de un *gestor,* como *pip,* y la otra es directamente a partir del código fuente. La utilización de uno u otro método dependerá de qué método han decidido los desarrolladores para distribuir su software. Supongamos que necesitamos un determinado paquete, ofrecido como *open source* y cuyos desarrolladores han decidido distribuirlo exclusivamente utilizando el código fuente. En este caso no podremos recurrir a un gestor de paquetes, ya que los desarrolladores no han ofrecido la forma de instalarlo a través del mismo. En este caso la única opción es realizar la instalación desde el fuente. En otros casos, será posible utilizar ambos métodos, quedando a nuestra elección el que prefiramos.

A continuación, describiremos ambos métodos para la instalación de módulos en Python. Comenzamos aprendiendo cómo realizar la instalación a partir del código fuente.

Instalación desde el código fuente

Python nos ofrece un módulo en su librería estándar que incluye una serie de herramientas para poder *empaquetar* y distribuir el software que hemos desarrollado en este lenguaje. De esta forma, a partir del código fuente y utilizando estas herramientas es posible crear un fichero listo para su distribución e instalación. Este fichero estará comprimido y el formato del mismo dependerá del sistema operativo. Por ejemplo, en Mac OS X y Linux se emplea un *tarball,* mientras que en Windows es un ZIP. De los detalles de este proceso nos ocuparemos más adelante. El módulo en cuestión se llama *distutils* y la utilización del mismo está considerada como la forma estándar de crear y distribuir paquetes de Python.

Si necesitamos instalar un paquete creado con *distutils,* bastará con obtener el fichero en el que está siendo distribuido, descomprimir el mismo y ejecuta un

comando determinado. Por ejemplo, supongamos que nuestra máquina está corriendo Fedora Linux y que el paquete que vamos a instalar viene en un fichero llamado *hoia-1.O.tar.gz.* El primer paso será descomprimirlo, acción que llevaremos a cabo a través de un terminal:

```
$ tar -xzvf hola-1.0.tar.gz
```

Seguidamente, accederemos al directorio que ha sido creado como consecuencia de la descompresión del fichero y ejecutaremos un script llamado *setup.py,* que encontraremos en el nuevo directorio. A este script le pasaremos un argumento: *install.* Ambas acciones comentadas serán ejecutadas con los siguientes comandos:

```
$ cd hola-1.0
$ python setup.py install
```

En cuanto lancemos el último comando se procederá a la instalación automática del nuevo paquete. Al finalizar el proceso, el mismo quedará directamente accesible y podremos importarlo tanto en nuestros programas, como desde la línea de comandos del intérprete de Python. Esto se debe a que, por defecto, el script *setup.py* se ha encargado de copiar los ficheros Python al directorio donde, por convección, se almacenan los paquetes de terceros. Este directorio variará en función del sistema operativo. Por ejemplo, en Windows estará en *C:\Python3.2/Lib/site- packages.* En los sistemas Linux, el menionado directorio suele estar en */usr/lib/python/lib/site-packages.* Debemos tener en cuenta que esta ruta puede variar en función de la versión de Python que tengamos instalada o de si hemos escogido para la instalación un directorio diferente al que se emplea por defecto. En cualquier caso, el directorio *site-packages* suele encontrarse como subdirectorio de *lib.*

El script *setup.py* no solo se encarga de copiar los ficheros al directorio comentado anteriormente, sino que también realiza otras operaciones, como, por ejemplo, la compilación de código. Pensemos en el caso de un paquete que incluye código C o C++ que es invocado desde Python. Dado que estamos distribuyendo el código fuente y C/C++ necesita ser compilado, para la instalación del paquete será necesario realizar este proceso. De ello se ocupará automática y transparentemente el script *setup.py.* Sin embargo, existe un argumento que puede ser pasado al mencionado script para solo compilar el código fuente, tal y como muestra el siguiente comando:

```
$ python setup.py build
```

No olvidemos que deberemos invocar al mismo script pasando el argumento *install* para realizar la instalación, una vez que haya terminado la compilación. En caso contrario solo habremos compilado y el módulo no será instalado.

Si en lugar de realizar la instalación de nuevo módulo en el *path* por defecto preferimos hacerla en otro diferente, podemos pasar un tercer argumento al script *setup.py.* En concreto podemos emplear *--user,* para realizar la instalación en el directorio *home* del usuario que está llevando a cabo la instalación; *--home,* para un directorio determinado, y *--prefix,* utilizado en el caso de Windows, ya que el concepto de *home* suele ser exclusivo de sistemas UNIX. Por ejemplo, para que un módulo quede instalado en el directorio *c:\Temp\Python\Lib\site-packages* de una máquina Windows, ejecutaremos el siguiente comando:

```
python setup.py install --prefix=\Temp\Python
```

Puede que en un momento determinado nos interese tener un control sobre qué tipo de ficheros se instalan en qué directorios cuando invocamos a la instalación de un método. Para este caso contamos con una serie de parámetros que también pueden ser pasados a *setup.py.* De esta forma la personalización del esquema de instalación es completa. En concreto, contamos con argumentos para indicar dónde instalar los módulos puros escritos en Python; aquellos que son de *extensión,* escritos en C/C++; todos los módulos, sin diferencia de tipo; aquellos ficheros que son considerados scripts y que deben ser alojados en un directorio accesible a través de la variable de entorno *PATH;* los datos puramente dichos y las *headers* de los ficheros C. Por ejemplo, supongamos que vamos a instalar un módulo que solo contiene dos ficheros Python y que podrán ser invocados directamente por línea de comando. Es decir, consideraremos que son *scripts,* similares a los de *bash* y que, por tanto, queremos que se encuentren, por ejemplo, en el directorio */usr/local/bin/.* En este caso, bastará con ejecutar el siguiente comando:

```
$ python setup.py --install-scripts=/usr/local/bin/
```

Otro parámetro interesante, que también acepta *setup.py* para la instalación de módulos que contienen extensiones escritas en C o C++, es *-compiler.* Este nos sirve para indicar qué compilador de C/C++ deseamos utilizar, siendo los valores diferentes en función del sistema operativo donde estamos realizando la

instalación. Por ejemplo, en Windows podemos usar algunos diferentes como *cygwin*, *mingw32* y *Borland C++*. El siguiente comando sería utilizado, por ejemplo, para indicar que deseamos compilar solo con *cygwin*:

```
python setup.py build --compiler=cygwin
```

Obviamente, para utilizar cualquiera de los compiladores anteriormente mencionados, deberemos tenerlos instalados en nuestra máquina.

Ahora que hemos aprendido a instalar un paquete a partir de su código fuente, estamos en condiciones de aprender el segundo método, del que nos ocuparemos a continuación.

Gestores de paquetes

Un *gestor de paquetes* es un software que nos permite realizar diversas acciones como son, la consulta, la búsqueda y, por supuesto, la instalación de módulos de Python. Gracias a estos gestores podemos realizar la instalación de paquetes con solo un comando. También son prácticos para descubrir si existen algunos paquetes relacionados con un criterio específico. Los usuarios de Linux están acostumbrados a sistemas similares como son, por ejemplo, *apt-get* y *yum*, los cuales permiten instalar, desinstalar y buscar, entre otras operaciones, software para nuestro sistema operativo.

Tal y como hemos comentado previamente, para Python contamos con *easy install* y *pip*. Aunque son diferentes, ambos implementan la misma funcionalidad básica y, lo que es más importante, los dos utilizan la misma fuente para instalar los paquetes. ¿De dónde obtienen estos gestores la información sobre los paquetes que pueden ser instalados? La respuesta a esta pregunta es sencilla, ambos usan como origen un servicio denominado *Python Package Index (PyPi)*. Este servicio está accesible a través de Internet y es considerado el repositorio oficial de paquetes para Python. En otros lenguajes existen servicios similares como *CPAN* para Perl y *PEAR* para PHP. A través del sitio web de PyPi (ver referencias) podemos acceder a información como el número total de paquetes que hay, el listado completo de paquetes o los últimos 40 que han sido añadidos. Además, en el mismo sitio web, se ofrece información para aquellos desarrolladores interesados en publicar sus propios módulos para que queden

accesibles desde el repositorio. Actualmente, más de 19.000 paquetes pueden ser instalados a través de PyPi.

A continuación, descubriremos cómo emplear los gestores *easy_install* y *pip* para interactuar con PyPi.

EASY_INSTALL

Existe un módulo para Python llamado *distribute* que contiene, entre otros componentes, un script llamado *easy_install.* Este script será el que utilicemos para invocar al gestor de paquetes del mismo nombre. Dado que *distribute* no forma parte de la librería estándar de Python, recurriremos a su código fuente para realizar la instalación del mismo. Para ello, seguiremos una serie de sencillos pasos. El primero será descargarnos un script que se encargará de descargar el código necesario y de realizar automáticamente la instalación. Así pues, deberemos descargar el fichero *distribute_setup.py* que se encuentra accesible a través de la página web (ver referencias) de descargas del mencionado módulo para Python. En sistemas Linux y Mac OS X es habitual contar con herramientas como *wget* y *curl,* que permiten la descarga de ficheros desde la línea de comandos. Si tenemos uno de estos dos programas instalados, podremos descargarnos el mencionado fichero empleando, por ejemplo, el siguiente comando que hace uso de *curl*:

```
$ curl -O http://python-distribute.org/distribute_setup.py
```

En cuanto dispongamos del script de instalación en nuestra máquina, procederemos a la instalación propiamente dicha del módulo *distribute.* Para ello, solo necesitaremos ejecutar el siguiente comando desde un terminal:

```
python distribute_setup.py
```

Dado que *distribute* es un módulo empaquetado siguiendo el estándar para ello propuesto oficialmente por Python, es posible realizar la instalación del mismo directamente a través del *tarball* ofrecido por los desarrolladores del módulo en cuestión. Es decir, opcionalmente, en lugar de realizar la instalación como hemos visto anteriormente, podemos recurrir al método estándar. Para ello, en primer lugar, nos descargaremos el mencionado *tarball.* En el momento de escribir estas líneas, la última versión es la 0.6.24, así pues, será esta la que

empleemos para mostrar cómo realizar la instalación. Como ejemplo, partiremos de un sistema Linux, aunque el proceso es prácticamente igual para Mac OS X y Windows. Comenzamos con la descarga del fichero en cuestión:

```
$ curl -O
http://pypi.python.org/packages/source/d/distribute/distribute-
0.6.24.tar.gz
```

Después procederemos a la descompresión del fichero descargado:

```
$ tar -zxvf distribute-0.6.24.tar.gz
```

Ahora llega el momento de acceder al nuevo directorio creado e invocar a *setup.py* para que comience la instalación:

```
$ cd distribute-0.6.24
$ python setup.py install
```

Al finalizar la instalación, con independencia del método empleado, ya tendremos acceso al mencionado módulo y al script *easy_install*, a través del cual podremos instalar, buscar o consultar paquetes de Python que se encuentran registrados en PyPi. Para comprobar que la instalación se ha realizado correctamente y que, efectivamente tenemos acceso al gestor de paquetes, comprobaremos la existencia de un fichero llamado *easy_install.py,* en el caso de sistemas UNIX o de *easy_install.exe,* en el caso de Windows. Si estamos trabajando en este último sistema operativo y el directorio raíz de la instalación de Python es *C:\Python3.2,* podremos comprobar cómo existirá un subdirectorio llamado *Scripts* donde localizaremos el fichero *easy_install.exe.* Con independencia del sistema operativo que estemos utilizando, es interesante que el script *easy_install* sea accesible a través de la variable de entorno *PATH.* Si realizamos esta configuración, tanto en Mac OS X, como en Linux y Windows, tendremos acceso directo a *easy_install* desde la línea de comandos. Así pues, nuestra primera invocación al script será para echar un vistazo a las opciones que nos ofrece. Para realizar esta acción, por ejemplo en Linux, bastará con lanzar el siguiente comando:

```
$ easy_install --help
```

Al ejecutar el comando anterior observaremos cómo obtenemos información sobre las opciones y los argumentos adicionales que podemos pasar al script para

realizar diferentes acciones.

La principal acción que puede ser llevada a cabo con *easy_install* es, lógicamente, la instalación de paquetes para Python. Ello es tan sencillo como indicar como argumento el nombre del paquete en cuestión. Por ejemplo, supongamos que deseamos instalar el paquete *lxml*. El comando para ello será el siguiente:

```
$ easy_install lxml
```

Automáticamente, *easy_install* se encargará de acceder a PyPi, buscar el fichero de distribución dado por los desarrolladores, compilar el código C/C++ -en caso de que sea necesario- y proceder a la copia de los correspondientes ficheros al directorio *site-packages* de nuestro intérprete de Python. Debemos tener en cuenta que PyPi actúa como registro de los paquetes, pero en ocasiones no directamente como repositorio. Esto significa que algunos desarrolladores alojan el fichero de distribución de un módulo en un servidor distinto. Este es el caso, por ejemplo, de *lxml*. Durante el proceso de instalación del mismo comprobaremos cómo se van lanzando una serie de líneas que nos van informando de qué acción se está llevando a cabo en cada momento. Observando esta información, comprobaremos cómo la descarga del fichero de distribución de *lxml* se ha realizado desde la URL *http://lxml.de/files/lxml-2.3.3.tgz*.

En lugar del nombre del paquete, *easy_install* también acepta el paso como argumento de una URL que indique dónde se encuentra el fichero de distribución correspondiente al módulo en cuestión que deseamos instalar. Esto también permite instalar paquetes que no estén referenciados por PyPi.

Otra interesante funcionalidad referente a la instalación de paquetes es la opción de Indicar el número de versión que deseamos instalar de un determinado paquete. En este caso, deberemos indicar el número de versión concreto, tal y como muestra el siguiente ejemplo:

```
$ easy_install lxml==2.3.3
```

Además de la instalación, también es posible actualizar a la versión más reciente que exista un paquete que tengamos previamente instalado. Esta acción puede ser llevada a cabo empleando la opción —*upgrade*, tal y como muestra el siguiente ejemplo:

```
$ easy_install --upgrade lxml
```

Cuando ya no necesitemos un paquete podremos borrarlo directamente a través de *easy_install.* Para realizar esta acción, solo será necesario indicar el parámetro *-m* precedido por el nombre del paquete que deseemos desinstalar. Por ejemplo, el siguiente comando desinstalaría el paquete *lxml* de nuestro sistema:

```
$ easy_install -m lxml
```

Si por defecto deseamos emplear una fuente adicional a PyPi para la instalación de paquetes, podremos hacerlo a través de un fichero de configuración llamado *setup.cfg.* Este fichero, por defecto, residirá en el directorio desde donde estamos realizando la instalación. Adicionalmente, es posible también crear un fichero en el directorio *home* del usuario, al que llamaremos *pydistutils.cfg.* De esta forma, conseguiremos que la configuración indicada en este último fichero sea utilizada por *easy_install*, con independencia del directorio donde sea invocado. Las siguientes líneas, correspondientes al mencionado fichero de configuración, muestran cómo indicar que vamos a emplear un servidor adicional para la instalación de paquetes:

```
[easy_install]
find_links = http://miservidor/python/paquetes/
```

Hasta aquí todo lo básico para trabajar con *easy_install*, seguimos nuestro recorrido por los gestores de paquetes con *pip*.

PIP

El gestor de paquetes *pip* nació con la idea de ofrecer una alternativa a *easy_install*, que además, ofreciera funcionalidades adicionales. Además de la funcionalidad básica de instalación de paquetes, *pip* permite actualizaciones, búsquedas, desinstalaciones e incluso es capaz de mostrar información sobre todas las versiones específicas de cada paquete instalado en un sistema.

Para instalar *pip* deberemos previamente tener instalado *distribute*, si aún no lo hemos hecho, antes de continuar, deberemos realizar la instalación de este módulo, tal y como hemos explicado en el apartado anterior.

La instalación de *pip* se puede realizar de dos formas diferentes. Una de ellas es a través del script de instalación ofrecido por sus desarrolladores. La otra forma es empleando el fichero de distribución e invocando al script *setup.py.*

Ambas técnicas son sencillas, tal y como podremos comprobar. El primer método requiere de la descarga del instalador en cuestión, acción que puede ser realizada directamente a través de herramientas como *curl o wget.* En este ejemplo, utilizamos *wget* desde Fedora Linux:

```
$ wget https://raw.github.com/pypa/pip/master/contrib/get-pip.py
```

En cuanto tengamos descargado el script, procederemos a su ejecución:

```
$ sudo python get-pip.py
```

Para la instalación alternativa necesitaremos el *tarball* de distribución, el cual también puede ser descargado a través de *curl o wget.* En esta ocasión, vamos a invocar a *curl*, tal y como muestra el siguiente comando:

```
$ curl -O http://pypi.python.org/packages/source/p/pip/pip-1.0.tar.gz
```

El siguiente paso será la descompresión del *tarball* que acabamos de descargar:

```
$ tar -zxvf pip-1.0.tar.gz
```

Ahora es el turno de acceder al nuevo directorio y proceder a la invocación del script *setup.py,* tal y como muestran las siguientes sentencias:

```
$ cd pip-1.0
$ python setup.py install
```

La instalación de módulos en sistemas UNIX se realiza, por defecto, en un determinado directorio en el que solo el usuario *root* tiene permisos de escritura. Este hecho implica que para instalar paquetes, tanto con *easy_install,* como con *pip,* deberemos invocar a estos programas como el mencionado usuario *root.* En sistemas operativos como Ubuntu, esto es tan sencillo como ejecutar los scripts a través del comando *sudo.* En otro caso, es necesario cambiar a *root* antes de invocar a cualquiera de estos scripts. En Windows no tendremos este problema, ya que, por defecto, la instalación se realiza en un directorio donde el usuario tendrá acceso de escritura.

Una vez que la instalación de *pip* ha finalizado satisfactoriamente, podremos ejecutarlo, por ejemplo, para mostrar las opciones que nos ofrece. El siguiente comando se encargará de esta acción:

```
$ pip --help
```

Tal y como hemos aprendido en capítulos anteriores, la instalación de un paquete a través de *pip* es tan sencilla como invocar al comando *install*, seguido del nombre del paquete que deseamos instalar. Como ejemplo sirva el siguiente comando:

```
$ pip install lxml
```

La actualización de paquetes a la versión más reciente liberada también se realiza empleando el comando *install*, pasándole como parámetro *-U,* tal y como muestra el siguiente ejemplo:

```
$ pip install -U lxml
```

Al igual que *easy_install*, *pip* también admite la instalación desde un determinado *tarball* accesible a través de una URL Para este tipo de instalación bastará con pasar la URL en cuestión al comando *install.*

Muy interesante resulta la opción de poder instalar un paquete directamente a través de un sistema de control de versiones. En algunos casos, esto puede sernos muy útil, para, por ejemplo, realizar instalaciones desde nuestro propio sistema de control de versiones o para asegurarnos de obtener el último código que está siendo desarrollado. El parámetro *-e*, seguido de la URL en cuestión, será el que deberemos pasar al comando *install* para realizar este tipo de instalación. Por ejemplo, para instalar el paquete *prueba* que se encuentra en un sistema de control de versiones *git,* ejecutaríamos el siguiente comando:

```
$ pip -e git://miservidor.com/prueba.git#egg=prueba
```

A diferencia de *easy_install, pip* sí que nos permite realizar búsquedas de paquetes. Gracias a esta funcionalidad, es posible pasar como argumento un nombre de paquete y *pip* se encargará de ofrecernos una serie de resultados, consultando para ello PyPi y buscando todos los paquetes que coinciden con el criterio de búsqueda. Por ejemplo, supongamos que deseamos buscar paquetes relacionados con XML. El siguiente comando será el que emplearemos para realizar la búsqueda:

```
$ pip search xml
```

Por la salida estándar recibiremos un completo listado, indicando, tanto el

nombre del paquete, como una pequeña descripción. Si además uno de los paquetes listado coincide con que ya lo tengamos instalado, se nos informará de ello, mostrándonos información adicional sobre la versión más reciente que existe sobre el mismo. Como ejemplo, mostramos unas cuantas líneas obtenidas al invocar al comando anterior de búsqueda:

```
ll -xist   - Extensible HTML/XML generator, cross-platform templating
           language, Oracle Utilities and various other tools
lxml       - Powerful and Pythonic XML processing library combining
           libxml2/libxslt with the ElementTree API.
           INSTALLED: 2.3.3
           LATEST: 2.3beta1
Chameleon  - Fast HTML/XML Template Compiler.
           INSTALLED: 2.7.3 (latest)
generateDS- Generate Python data structures and XML parser from
           Xschema
archgenxml- UML to code generator for Plone
libxml2dom- PyXML-style API for the libxml2 Python bindings
bridge     - General purpose XML library for
           CPython and IronPython
Amara      - Library for XML processing in Python
```

La desinstalación de módulos puede ser ejecutada fácilmente gracias al comando *uninstall*, el cual se encargará de eliminar aquellos ficheros instalados a través de un comando *install.*

Consultar todos los paquetes instalados en nuestro sistema es posible gracias al comando *freeze*, el cual produce un listado con los nombres y versiones de todos ellos. Después del nombre de cada paquete aparecerán dos símbolos de igual, seguidos de la versión en cuestión. Por ejemplo, tras invocar al siguiente comando, obtendremos una serie de líneas como las que se indican a continuación:

```
$ pip freeze
wsgiref==0.1.2
wxPython==2.8.8.1
wxPython-common==2.8.8.1
wxaddons==2.8.8.1
xattr==0.5
zope.deprecation==3.5.0
zope.interface==3.8.0
```

En lugar de pasar diferentes opciones para cada comando, puede ser

interesante emplear un fichero de configuración donde indiquemos las opciones, con sus correspondientes valores que necesitemos. El fichero en cuestión se llama *pip.conf* o *pip.ini,* en el caso de Windows, y su formato se corresponde con un INI estándar, donde podemos tener diferentes secciones y donde cada línea indica *propiedad* igual a *valor.* El mencionado fichero puede ser encontrado en un subdirectorio, llamado *pip* o *.pip,* dentro del directorio *home* del usuario en cuestión que invoca a *pip.*

En un momento dado puede ser interesante obtener información sobre la sintaxis de cada uno de los comandos de *pip.* Para realizar esta acción, bastará con emplear otro comando llamado *help.* Así pues, por ejemplo, para consultar la información sobre el comando *search,* ejecutaremos la siguiente orden:

```
$ pip help search
```

Tanto *easy_install* como *pip* son capaces de manejar automáticamente las *dependencias* entre paquetes. Esto implica que, si un determinado paquete requiere de otros para su funcionamiento, estos serán instalados automáticamente. Esta funcionalidad es una gran ventaja y nos asegura que cada paquete será instalado y funcionará correctamente.

En este punto termina nuestro recorrido por la instalación de paquetes, en el siguiente apartado trataremos sobre cómo preparar nuestro software para que pueda ser distribuido e instalado por otros desarrolladores o usuarios.

DISTRIBUCIÓN

Para que otros desarrolladores puedan instalar y utilizar nuestros módulos de Python, deberemos preparar estos de una forma determinada. En la librería estándar de Python existen un módulo que nos ayudará a esta tarea, llamado *distutils*. Gracias al mismo podremos *empaquetar* nuestros módulos Python para que otros puedan instalarlos, por ejemplo, a través de *PyPi*.

El proceso general para crear un fichero de distribución y ponerlo a disposición de otras personas consta de los siguientes pasos:

- Creación del fichero *setup.py* con la información sobre el módulo.
- Creación del fichero para distribución, puede ser un *tarball o* ZIP.
- Registro del paquete en PyPi, si deseamos que pueda ser indexado.
- Subida del paquete al repositorio de PyPi. Este paso es opcional.

A través de un ejemplo básico, vamos a descubrir cómo aplicar los pasos anteriormente mencionados. Partiremos de la idea de que nuestro paquete estará compuesto por un solo fichero Python, el cual contendrá una clase. Nuestra clase contendrá un constructor y un método para imprimir el popular *Hola Mundo!*. En concreto el código de nuestro *hola.py,* será el siguiente:

```
class Hola:
    def __init__(self):
        pass
    def say_hello(self):
        print('Hola Mundo!")
```

Ahora necesitamos crear un directorio al que copiaremos nuestro nuevo fichero. Podemos llamarlo *hola* y dentro del mismo vamos a crear el fichero *setup.py.* El contenido del mismo estará formado por las siguientes líneas:

```
from distutils.core import setup
setup(name='hola', version='1.0', py_modules= [' hola'])
```

A través de la función *setup()* vamos a indicar diferente tipo de información sobre el paquete. Hemos elegido la información mínima para nuestro ejemplo,

indicando, simplemente, el nombre del paquete, la versión y los módulos Python de los que consta. Esta función acepta argumentos opcionales para indicar datos, como el tipo de licencia, una descripción sobre el paquete, el nombre y correo electrónico del autor y una lista de *clasificadores.* Esta lista sirve para indicar qué tipo de paquete es, a qué audiencia va dirigido o para qué versión de Python ha sido desarrollado. En general esta información va dirigida a poder clasificar el paquete, de forma que pueda ser fácilmente indexado por PyPi. Una lista completa de todos los clasificadores soportados puede consultarse en la página web (ver referencias) de PyPi dedicada a ello.

El siguiente paso será invocar a un comando que nos creará directamente el fichero de distribución. En Windows será un ZIP, mientras que en Mac OS X y Linux será un *tarball.* El comando en cuestión es el siguiente:

```
$ python setup.py sdist
```

Si echamos un vistazo a nuestro directorio *hola*, comprobaremos cómo se han creado un nuevo directorio, llamado *dist,* y un fichero con el nombre *MANIFEST.* El contenido de este fichero está formado por las siguientes líneas, que hacen referencia a una serie de información sobre el paquete:

```
Metadata-Version: 1.0
Name: hola
Versión: 1.0
Summary: UNKNOWN
Home-page: UNKNOWN
Author: UNKNOWN
Author-email: UNKNOWN
License: UNKNOWN
Description: UNKNOWN
Platform: UNKNOWN
```

En lo que al directorio *dist* respecta, comprobaremos cómo se ha creado el fichero de distribución, en nuestro caso, se llama *hola-1.O.tar.gz.* Ya estamos preparados para distribuir nuestro paquete *hola.* Podríamos hacer el fichero accesible desde un servidor web o FTP, para que pudiera ser descargado. Para poder instalarlo, habría que seguir el procedimiento estándar que hemos visto previamente, es decir, bajarse el fichero, descomprimirlo, entrar en el directorio y ejecutar *python setup.py install.*

Opcionalmente, podemos hacer que PyPi indexe nuestro paquete para que pueda ser buscado e instalado a través de gestores como *easy_install* y *pip.*

Antes de continuar y dado que PyPi requiere que el nombre y correo electrónico del autor de cada paquete sea indicado, deberemos modificar nuestro *setup.py* para añadir estos datos y volver a generar el *tarball*. Por otro lado, PyPi también necesita que el autor, que vaya a registrar y/o subir su paquete, esté registrado en su sistema. Cualquiera puede registrarse a través de un formulario web disponible en la correspondiente página web (ver referencias) de PyPi.

En cuanto cumplamos con la formalidad del registro esteremos listos para registrar nuestro nuevo paquete. Esto es tan sencillo con ejecutar el siguiente comando y seguir las instrucciones que se nos irán dando:

```
$ python setup.py register
```

Justo después de ejecutar el comando anterior, veremos cómo aparece un *prompt* que nos preguntará si deseamos utilizar un usuario de PyPi existente, crear uno nuevo o abortar el proceso. Daremos por hecho que ya contamos con un usuario, así pues, elegiremos la primera opción, introduciremos nuestro usuario y contraseña. Automáticamente se procederá al registro del paquete en cuestión.

Una vez registrado el paquete, también podemos subirlo al repositorio de PyPi, con el objetivo de que pueda ser descargado desde el mismo. En este caso, simplemente bastará con ejecutar el siguiente comando:

```
$ python setup.py sdist upload
```

Llegados a este punto, cualquiera podrá instalar nuestro paquete desde, por ejemplo, *pip*. Si el paquete es instalado, podrá ser importado en cualquier entorno virtual o sistema donde haya sido instalado. De hecho, podemos comprobar que funciona correctamente ejecutando las siguientes líneas de código:

```
>>> import hola
>>> h = hola.Hola()
>>> h.say_hello()
Hola Mundo!
```

Como el lector habrá podido comprobar, Python nos ofrece, a través de su módulo *distutils* y del sistema PyPi, un completo sistema para distribuir nuestros paquetes.

ENTORNOS VIRTUALES

Hasta el momento hemos aprendido a instalar paquetes en el directorio asignado por defecto para ello y que afectan a todo el sistema. Esto implica que si tenemos dos paquetes instalados que, a su vez, emplean un tercero, ambos tendrán que usar la versión instalada de este último. Pero ¿qué ocurre si actualizamos a una nueva versión de este tercer paquete? En principio, uno de los dos que lo utilizan podría no funcionar correctamente por los cambios aplicados en la nueva versión. Además, ¿qué pasaría si uno de estos paquetes sí que requiere de esta nueva versión? Por otro lado, ¿cómo podríamos utilizar exclusivamente una determinada versión de un módulo del que depende otro? ¿Qué pasaría si deseamos instalar una serie de módulos pero no tenemos acceso al directorio por defecto? Estas preguntas suelen hacérselas muchos administradores de sistemas cuando tienen que mantener un servidor y diferentes aplicaciones Python que corren en el mismo. También son muchos los desarrolladores que se enfrentan a estos escenarios, cuando por ejemplo, necesitan probar si una nueva versión de un determinado módulo es compatible con la aplicación ya desarrollada.

Dadas las preguntas anteriores y las situaciones que las originan, parece conveniente encontrar una solución que nos ayude a resolver el problema. Es aquí donde podemos contar con los *entornos virtuales.* Existen para Python un conjunto de herramientas que permiten crear entornos aislados, donde cada uno de ellos es responsable de sus propios paquetes. Es decir, es posible crear una *sandbox* en la que instalar diferentes módulos que serán independientes, tanto de los que existen a nivel de sistema (directorio *site-packages),* como unos de otros. De esta forma, por ejemplo, para comprobar si un determinado módulo es compatible con nuestra aplicación, bastará con crear un entorno virtual, instalar todos los paquetes, incluida la nueva versión del paquete en cuestión. Si la aplicación funciona correctamente, entonces podremos instalar esta nueva versión en nuestro entorno de producción.

virtualenv

Para la gestión de entornos virtuales en Python, necesitamos instalar un módulo llamado *virtualenv,* el cual está accesible a través de PyPi. Ello implica que podremos instalarlo con un gestor de paquetes como *easy_install* y *pip.* Por ejemplo, empleando este último ejecutaríamos el siguiente comando:

```
$ pip install virtualenv
```

Una vez instalado *virtualenv,* tendremos acceso a un script Python con el mismo nombre, el cual puede ser invocado directamente. Por ejemplo, en Linux podremos lanzar el siguiente comando:

```
$ virtualenv -help
```

Para crear un nuevo entorno virtual, bastará con pasar la ruta del directorio donde queremos que sea creado. El siguiente ejemplo creará un entorno virtual en el directorio /tmp/env de un sistema Linux:

```
$ virtualenv /tmp/env --no-site-packages
```

Observando la salida del comando anterior descubriremos cómo las herramientas proporcionadas por *distribute* y el gestor de paquetes *pip* son instalados también. Si nos fijamos en el directorio que hemos pasado como argumento al comando *virtualenv,* comprobaremos cómo se han creado tres subdirectorios diferentes. El primero de ellos es *bin* y contiene una serie de scripts que podemos utilizar desde nuestro entorno. Entre ellos encontraremos al mencionado *pip* y otros como *actívate,* del cual nos ocuparemos más adelante. Otro de los directorios es *include,* que contendrá el intérprete de Python y una serie de ficheros fuente que serán empleados, en caso de que algunos de los paquetes que instalemos requiera de ello, para compilar. El último directorio en cuestión es *lib,* que contendrá, entre otros ficheros y directorios, el subdirectorio *site-packages.* Serán en este directorio donde se instalen los paquetes que pertenezcan a nuestro entorno virtual.

Ya tenemos nuestro entorno creado, así que nuestro siguiente paso será *activarlo* para poder comenzar a trabajar con él. Esta acción se lleva a cabo a través del script *activate,* que se encuentra en el directorio *bin.* En Linux, ejecutaríamos los siguientes comandos para activar nuestro entorno recién creado:

```
$ cd /tmp/env
```

```
$ source bin/activate
(env)$
```

La última línea indica que el *prompt* de la línea de comandos habrá cambiado para mostrar, entre comas, el nombre del entorno que está activado. A partir de este momento, si instalamos un módulo lo haremos solo en el entorno virtual en cuestión, quedando el mismo accesible exclusivamente en dicho entorno. Debemos tener en cuenta que esto ha sido posible gracias a que pasamos el argumento *—no-site-packages* al crear nuestro entorno. Si no lo hubiéramos hecho, el entorno virtual tendría acceso también a los paquetes del sistema. Esto último podría ser interesante en algunos casos, pero generalmente, se suele emplear el mencionado argumento para lograr entornos virtuales completamente aislados.

Al lanzar el siguiente comando, comprobaremos cómo los paquetes listados son diferentes a los que obtendremos lanzando el mismo comando fuera de nuestro entorno virtual:

```
(env)$ ./bin/pip freeze
```

Esto nos demuestra que, efectivamente, los paquetes quedan aislados unos de otros, quedando nuestro entorno aislado del resto de paquetes Python instalados en el sistema. Cada vez que creemos un nuevo entorno virtual lo haremos en un directorio diferente del sistema de ficheros.

virtualenvwrapper

Con el objetivo de facilitar el manejo de *virtualenv* y los diferentes entornos virtuales de Pyton que pueden ser instalados en la misma máquina, se desarrolló otro paquete conocido como *virtualenvwrapper.* Como su propio nombre indica, este paquete contiene una serie de herramientas que actúan como *wrapper* sobre *virtualenv,* y facilitan su utilización. Entre las funcionalidades que incluye, encontramos la de crear diferentes entornos virtuales bajo el mismo directorio del sistema de ficheros, activar un entorno a través de un nombre dado, permitir fácilmente desactivar un entorno para activar otro diferente, borrar un entorno y copiar completamente un entorno con un nombre distinto.

La instalación de *virtualenvwrapper* es muy sencilla y puede ser llevada a

cabo a través de un gestor de paquetes como *pip*. De esta forma, bastaría con invocar al siguiente comando para realizar la instalación:

```
$ pip install virtualenvwrapper
```

Antes de comenzar a trabajar con las herramientas que ofrece *virtualenvwrapper* es conveniente modificar el fichero de configuración de la terminal *(.bashrc, .profile* o similar) para añadir un par de líneas que nos faciliten la gestión de los entornos. La primera de ellas indicará cuál será el directorio a partir del que se crearán los diferentes entornos. La segunda línea se encargará de invocar automáticamente al *shell* script *virtualenvwrapper.sh*, cada vez que abramos una terminal. Las líneas en cuestión son las siguientes:

```
export WORKON_HOME=$HOME/.virtualenvs
source /usr/bin/virtualenvwrapper.sh
```

En nuestro ejemplo hemos decidido que el subdirectorio *.virtualenvs,* que pertenecerá al directorio *home* de nuestro usuario, será el que albergará los diferentes entornos que sean creados. Por otro lado, la segunda línea de configuración hace referencia a la ruta donde, por defecto, existirá el *shell* script instalado por *virtualenvwrapper.*

Ahora nos encontramos en disposición de usar *virtualenvwrapper,* siendo nuestra primera acción la creación de un nuevo entorno llamado *prueba*, tal y como muestra el siguiente comando:

```
$ mkvirtualenv prueba --no-site-packages
```

Por defecto, el comando *mkvirtualenv* se encargará tanto de crear el entorno *prueba* como de activarlo. De esta forma, justo después de ejecutar el comando anterior, estaremos en condiciones de comenzar a instalar paquetes en nuestro nuevo entorno virtual.

Si tenemos varios entornos virtuales, podemos cambiar de uno a otro directamente a través del comando *workon.* Este comando se encargará tanto de desactivar el entorno actualmente activado, como de activar el nuevo. Si ningún entorno está activado, pasará a activar el indicado. Por ejemplo, supongamos que nos encontramos en el entorno *prueba* y deseamos pasar a otro denominado *test,* el comando para ello sería el siguiente:

```
(prueba)$ workon test
```

Un listado completo de todos los entornos virtuales que tenemos instalados en una máquina puede ser obtenido invocando al comando *workon*, sin pasar ningún argumento.

La desactivación del entorno actualmente activado se realiza con el comando *deactivate,* el cual puede ser utilizado directamente.

pip y los entornos virtuales

Anteriormente hemos aprendido a consultar los paquetes que existen, bien en el sistema, bien en un determinado entorno virtual. Es la opción *freeze* del gestor *pip* la que nos permite realizar esta acción. Además de listar los paquetes, esta opción puede sernos útil para crear un fichero con todos los paquetes que tenemos instalados. Bastaría con redireccionar la salida estándar a un fichero, tal y como muestra el siguiente comando:

```
$ pip freeze > paquetes.txt
```

Si tenemos el mencionado fichero, será posible crear un nuevo entorno a partir del mismo. Esto es muy útil cuando, por ejemplo, tenemos diferentes máquinas de desarrollo y tenemos que mantener los mismos paquetes en ambas. Existe un parámetro adicional que puede ser pasado al comando *install* de *pip*, que se encargará de leer un fichero de texto e Instalar los paquetes listados en el mismo. Gracias a este comando, es posible utilizar el fichero creado con *freeze* para realizar la instalación automática de los paquetes, bien en una nueva máquina, bien en un nuevo entorno. Continuando con nuestro ejemplo, la Instalación se realizaría ejecutando el siguiente comando:

```
$ pip install -r paquetes.txt
```

Tanto los gestores de paquetes, como los entornos virtuales, son herramientas muy útiles para el desarrollo, administración y mantenimiento de aplicaciones Python. En la actualidad son muchos los desarrolladores y administradores de sistemas que emplean estas herramientas diariamente en su trabajo.

PRUEBAS UNITARIAS

INTRODUCCIÓN

Una de las fases más importantes en el proceso del desarrollo de software es la de pruebas. Con ella pretendemos demostrar que el software desarrollado cumple con la funcionalidad para la que ha sido diseñado e implementado.

Existen diferentes tipos de pruebas para el software, como son, por ejemplo, las unitarias, las funcionales y las de integración. El primer tipo se emplea para demostrar que una serie de componentes cumplen su función correctamente. Por otro lado, las de integración nos aseguran que diversos componentes software que interactúan entre sí lo hacen de forma correcta. Las pruebas funcionales se encargan de comprobar que la funcionalidad general para la que ha sido diseñado un software específico es correcta. Por ejemplo, una prueba unitaria se encargaría de comprobar que una función suma dos valores de forma correcta. Otra prueba garantizaría que otra función también lo hace al llamar a la función de suma. Por último, una prueba funcional demostraría que, cuando un usuario introduce dos valores empleando la interfaz gráfica, la suma devuelve el valor esperado.

Las pruebas unitarias son muy importantes, ya que pueden considerarse como una *unidad* mínima y, además, son la base de la fase de pruebas en el proceso de desarrollo de software. Cada prueba unitaria comprobará que cada parte de código (función, método o similar) cumple su función por separado, es decir, de forma independiente.

Una de las principales características de las pruebas unitarias es que estas deben ser automáticas. Es decir, una vez diseñadas, podrán ser ejecutadas directamente a través de un comando o script. Esto, además, garantiza que las pruebas se pueden repetir sucesivas veces.

Entre las ventajas de emplear pruebas unitarias, destacan el hecho de simplificar la integración de componentes, el aislamiento y acotación de errores, la facilitación de la refactorización de código y la separación entre la interfaz de usuario y los detalles de implementación.

En los últimos años, la amplia aceptación de metodologías de desarrollo

como TDD (*Test-Driven Development)* y BDD *(Behavior Driven Development)*, han puesto de manifiesto el interés por llevar a cabo pruebas sobre el software y han demostrado la necesidad de las mismas para desarrollar software de calidad.

En este capítulo final nos ocuparemos de las herramientas que pueden ser utilizadas en Python para escribir y ejecutar pruebas unitarias. Comenzaremos explicando una serie de conceptos básicos y después pasaremos a centrarnos en los más utilizados *frameworks* de Python que existen para pruebas.

CONCEPTOS BÁSICOS

Para llevar a cabo las pruebas unitarias, es importante estar familiarizado con una serie de conceptos básicos, de los cuales nos vamos a ocupar a continuación. El primero de ellos es el llamado *fixture*, que referencia a aquellos datos iniciales que vamos a necesitar para nuestras pruebas. Por ejemplo, supongamos que vamos a probar una serie de acciones que tienen lugar entre un conjunto de usuarios. Para ello, previamente, necesitaremos unos cuantos usuarios, estos serían nuestros *fixtures*. Otro ejemplo, sería una serie de ficheros o directorios, en caso de que las pruebas requieran de ellos.

Probablemente, el concepto más importante en el ámbito de las pruebas unitarias sea el de *caso de prueba*, también conocido como *test case*. El caso de prueba representa aquella funcionalidad que va a ser probada. Se trata de comprobar una serie acciones en base a unos datos de entrada. Como resultado obtendremos éxito o fallo, lo que demostrará si las acciones probadas funcionan correctamente, es decir, tomando unos valores de entrada conseguimos un determinado resultado esperado.

Dado que una prueba unitaria puede constar de varios casos de prueba, se define la *suite de pruebas (test suite)* como un conjunto de casos de prueba que deben ser ejecutados de forma agregada para determinar el resultado final de una prueba.

Una vez que tenemos preparados nuestros casos de prueba, incluyendo o no suites de los mismos, y los *fixtures*, solo nos queda un componente que se encargue de lanzar las pruebas y obtener un resultado. A este será al que llamamos *test runner* y también será responsable de ofrecer el resultado de una forma fácil de interpretar. En ocasiones se utilizan gráficos y otras, simplemente texto; en cualquier caso debemos obtener información sobre el número de pruebas ejecutadas y si el resultado ha sido satisfactorio o no.

Para probar una aplicación se suelen crear diferentes casos de prueba, incluyendo varias suites de prueba. Debe tenerse en cuenta que cada funcionalidad, considerada como tal, necesitará de su propio caso de prueba. Por ejemplo, partamos de un sencillo ejemplo, un pequeño módulo Python que contiene tres funciones diferentes. En este caso se debería escribir un caso de

prueba para cada una de las funciones. No existe un criterio general para establecer una correspondencia entre los casos de prueba y los componentes de código. En algunas ocasiones, la elección de caso de prueba es trivial, como en el ejemplo previo, pero en otras necesitaremos definir qué funciones, clases u otros componentes intervienen en un único caso de prueba. Igual escenario ocurre con las suites de prueba, quedando a criterio de los programadores la decisión sobre la creación de las mismas para probar determinado código.

Las pruebas unitarias en Python pueden ser llevadas a cabo con la ayuda de diferentes *frameworks*, los cuales ofrecen una serie de herramientas para facilitarnos el trabajo. A continuación, nos ocuparemos de los *frameworks* más populares.

UNITTEST

Python incorpora un módulo en su librería estándar que permite escribir y ejecutar pruebas unitarias, su nombre es *unittest* y puede ser invocado de la siguiente forma:

```
>>> import unittest
```

Para crear casos de prueba contamos con la clase denominada *TestCase,* mientras que para escribir suites tenemos otra llamada *TestSuite.* La creación y/o carga de *fixtures* se realiza a través de un método especial *(setUp)* que pertenece a la clase *TestCase.* Por otro lado, si son necesarias realizar acciones al terminar cada caso de prueba, por ejemplo, el borrado de algún dato, estas pueden llevarse a cabo gracias al método *tearDown(),* también implementado en *TestCase().* Tanto *setUp(),* como *tearDown(),* deben ser sobrescritos en nuestra clase de prueba, en caso de que sean necesarias acciones de inicialización o finalización, respectivamente. Es importante saber que para cada método de nuestra clase de prueba, serán ejecutados *setUp()* y *tearDown().* El primero lo será justo antes de comenzar la ejecución de cada método y el segundo al terminar dicha ejecución.

Así pues, para escribir un script que ejecute una o varias pruebas unitarias, comenzaremos creando una clase de prueba que herede de *TestCase.* Seguidamente, escribiremos una serie de métodos, uno por funcionalidad y finalmente, invocaremos a la función *main()* del módulo *unittest,* que se encargará de ejecutar las pruebas cuando nuestro script sea invocado. ¿Cómo comprobamos si la prueba tiene éxito o no? Para ello contamos con varios métodos, de la clase *TestCase,* que comienza con el prefijo *assert.* De esta forma, por ejemplo, el método *assertEqual()* recibirá dos valores y, en caso de ser iguales, devolverá un valor que indicará que el resultado de la prueba es válido. Si deseamos comprobar que dos valores son diferentes, utilizaremos el método *assertNoEqual.* Otros métodos similares son *assertTrue()* y *assertFalse()* que comprueban si un valor es *True o False,* respectivamente. Interesantes son también los métodos *assertIsInstance()* y *assertNotIsInstance(),* útiles para comprobar si un objeto es instancia o no de una determinada clase.

Para ilustrar el funcionamiento de *unittest* vamos a escribir un sencillo

fichero en Python que contendrá una función para sumar todos los valores de una lista. Posteriormente, escribiremos un caso de prueba que comprobará si el valor obtenido es el esperado, en cuyo caso, la prueba habrá sido positiva. El código en cuestión para nuestro fichero *(lista.py)* de ejemplo sería el siguiente:

```
def suma(li):
    total = 0
    for ele in li:
        total += ele
    return total
```

El siguiente paso será escribir un nuevo script al que llamaremos *pruebas.py,* cuyo contenido será el siguiente:

```
import unittest
import lista

class ListaTestCase(unittest.TestCase):
    def setUp(self):
        self.li = [1, 3, 5, 7]
    def test suma(self):
        total = lista.suma(self.li)
        self.assertEqual(total, 16)
if __name__ == '__main__':
    unittest.main()
```

Por convención, los nombres de los métodos de la clase de pruebas comenzarán por el prefijo *test_,* seguido del método o función que van a probar. En nuestro ejemplo solo tenemos un método llamado *test_suma().* A través del método *assertEqual()* comprobaremos si el valor obtenido al invocar a la función *suma() es* 16, que es el correspondiente a la suma de todos los valores de nuestra lista de ejemplo, cuyo valor hemos prefijado, como un atributo de instancia, en el método de inicialización *setUp().* Al ejecutar nuestro script por línea de comandos, la función *main()* será invocada, procediendo al lanzamiento de las pruebas. Ahora solo nos queda lanzar el script de la siguiente forma:

```
python pruebas.py
```

Dado que el valor es correcto, obtendremos el siguiente resultado por la salida estándar:

```
..
```

```
Ran 1 test in 0.002s
OK
```

La información obtenida hace referencia al número de pruebas ejecutadas y la cadena de texto *string* significa que la prueba ha sido pasada correctamente, asegurando así que nuestra función *suma()* funciona correctamente.

Hasta aquí la funcionalidad básica del módulo *unittest*, los lectores interesados pueden completar la información recibida con la documentación existente en la página web oficial (ver referencias) del mencionado módulo.

DOCTEST

Además de *unittest*, Python nos ofrece en su librería estándar otro módulo para escribir y realizar pruebas unitarias. Su nombre es *doctest* y permite escribir pruebas *en línea*, es decir, como parte de la documentación de una función y método, podemos escribir directamente el caso de prueba. Para comprobar el funcionamiento de este módulo vamos a modificar la función *suma()* del ejemplo anterior, añadiendo nuestro caso de prueba como parte del comentarlo de la documentación de la función. De esta forma, la nueva función *suma()* del módulo *lista* quedaría como sigue:

```
def suma(li) :
    """Devuelve la suma de todos los elementos de la lista
    >>> suma([1, 2, 3])
    6
    """
    total = 0
    for ele in li:
        total += ele
    return total
```

Asimismo, para poder ejecutar el nuevo caso de prueba, será necesario añadir un par de líneas, las cuales serán ejecutadas cuando se invoque a *lista.py*, directamente desde la línea de comandos. Las líneas de código en cuestión serán las siguientes:

```
if __name__ == '__main__ :
import doctest
doctest.testmod()
```

Si ahora lanzamos el siguiente comando, comenzará la ejecución de nuestro caso de prueba:

```
python lista.py
```

Como resultado de la ejecución del comando anterior no obtendremos ninguna respuesta, ello significará que el test ha funcionado correctamente. No obstante, si quisiéramos ver un registro de las acciones llevadas a cabo, podríamos recurrir al parámetro *-v*, tal y como muestra el siguiente ejemplo:

```
python lista.py -v
```

Por otro lado, si la prueba fallara, obtendríamos información al respecto. Si cambiamos el valor que debe ser obtenido a otro cualquiera, la prueba dará un resultado erróneo, tal y como podemos comprobar en la salida que obtendríamos:

```
**********************************************************************

File "lista.py", line 3, in __main__.suma
Failed example:
    suma([1, 2, 3])
Expected:
    7
Got:
    6

**********************************************************************
    1 ítems had failures:
    of 1 in __main__.suma
***Test Failed*** 1 failures.
```

A pesar de que la funcionalidad y flexibilidad ofrecidas por *doctest* es bastante limitada con respecto a *unittest*, el primero es un módulo que nos permite escribir pruebas unitarias de una forma muy sencilla. En función de la complejidad que necesitemos podemos emplear uno u otro módulo.

OTROS FRAMEWORKS

Aparte de *doctest* y *unittest* existen otros *frameworks* para escribir pruebas unitarias en Python. Entre ellos, por ejemplo, se encuentran *nose, unittest2* y *pytest.* Ninguno de estos forma parte de la librería estándar, por lo que deben ser instalados, por ejemplo, a través de un gestor de paquetes como *pip*.

El módulo *unittest2* fue desarrollado con la idea de implementar las nuevas funcionalidades, que fueron añadidas a *unirttest* en la versión 2.7 de Python, para hacerlo compatible con las versiones de Python desde la 2.3 a la 2.6. Además, existe también una versión específica de *unittest2* para Python 3, de forma que aquellos que han escrito sus pruebas con *unittest,* puedan migrar su código a Python 3 sin problemas.

pytest incorpora una serie de componentes para escribir complejos casos de prueba. Entre sus funciones encontramos aquellas que permiten obtener información sobre los tests que tardan más de un tiempo determinado, continuar con la ejecución de todas las pruebas aunque algunas fallen y la opción es escribir *plugins* para personalizar nuevas pruebas funcionales que nos sean necesarias.

Por último, *nose* está basado en *unittest* solo que se han añadido una serie de funcionalidades para lograr que sea más sencillo escribir y ejecutar casos de prueba. Entre ellas se encuentra la opción de pasar una serie de argumentos, por línea de comandos, para ejecutar una lista de test concreta o para indicar un directorio por defecto donde se encuentran todos los scripts de prueba.

A
EL ZEN DE PYTHON

TRADUCCIÓN DE "EL ZEN DE PYTHON"

Bonito es mejor que feo.
Explícito es mejor que implícito.
Simple es mejor que complejo.
Complejo es mejor que complicado.
Sencillo es mejor que anidado.
Disperso es mejor que denso.
La legibilidad cuenta.
Los casos especiales no son lo suficientemente especiales para romper las reglas. La practicidad bate a la pureza.
Los errores no deben pasar inadvertidos.
A menos que sean explícitamente silenciados.
En caso de ambigüedad, rechazar la tentación de adivinar.
Debe haber una -y preferiblemente única-forma obvia de hacerlo.
Aunque esa única forma no sea obvia en un primer momento, a no ser que seas holandés.
Ahora es mejor que nunca.
Aunque nunca es mejor que *ahora* mismo.
Si la implementación es difícil de explicar, es una mala idea.
Si la implementación es fácil de explicar, debe ser una buena idea.
Los espacios de nombres son una idea genial -¡hay que hacer más de esos!

El original puede leerse desde el intérprete interactivo, ejecutando la

siguiente línea:

```
>>> import this
```

B

CÓDIGO DE BUENAS PRÁCTICAS

REGLAS

- Es importante estructurar correctamente un proyecto. Crear una adecuada jerarquía de directorios y ficheros en función de las necesidades de cada aplicación.
- Utilizar una guía de estilo para la codificación. *PEP 8* (*http://www.python.org/dev/peps/pep-0008/)* define una completa guía de estilo, considerada como un estándar para código Python. El módulo *pep8* puede comprobar automáticamente si un script es compatible con las reglas fijadas en *PEP 8.*
- Aplicar *El Zen de Python.*
- Documentar siempre el código. Unas simples líneas de documentación pueden ahorrar horas de trabajo en el futuro. Herramientas como *Sphinx (http://sphinx.pocoo.org/)* y formatos como *reStructuredText (http://docutils.sourceforge.net/rst.html)* pueden ayudarnos mucho.
- Los controles de versiones son una excelente herramienta para el desarrollo de software. Cualquier proyecto debería hacer uso de uno de ellos.
- Escribir pruebas unitarias. Las pruebas de integración y funcionales también son recomendables.

REFERENCIAS

CAPÍTULO 1. PRIMEROS PASOS

- Sitio web oficial de Python: http://www.python.org/
- Página web de descargas de Python: http://www.python.org/getit/
- Página web de descargas para Python 3.2.2: http://www.python.Org/getit/releases/3.2.2/
- Página web oficial sobre *IDLE:*http://docs.python.org/library/idle.html
- Página web sobre la comunidad de Python: http://www.python.org/community/
- Utilidad *py2exe:*http://www.py2exe.org
- Información para configurar *Emacs* para desarrollo con Python: http://www.emacswiki.org/emacs/?action=browse;old¡d=PythonMod e;id=PythonProgramminglnEmacs
- Scripts y plugins de Python para *Vim*\ http://vim.wikia.eom/wiki/Category:Python
- Sitio web de *TextMate*: http://www.macromates.com
- Sitio web de *gedit:*http://projects.gnome.org/gedit/
- Sitio web de *Notepad++:*http://notepad-plus-plus.org/
- Sitio web de *Eclipse*: http://www.eclipse.org/
- *PyDev:* Python + Eclipse: http://pydev.org/
- Sitio web de *NetBeans:*http://netbeans.org
- Soporte para Python en *NetBeans:*http://wiki.netbeans.org/Python
- fríe IDE: http://eric-ide.python-projects.org
- *PyCharm* IDE: http://www.jetbrains.com/pycharm/
- *Wingware* Python IDE: http://wingware.com
- Documentación oficial de *IPython:*http://ipython.org/ipython-doc/stable/index.html
- Documentación y comandos de *pdb:* http://docs.python.org/library/pdb.html
- Página web dedicada a *2to3*, la herramienta para migrar de Python 2.x a Python 3: http://docs.python.org/library/2to3.html
- Página oficial sobre las novedades de Python 3: http://docs.python.Org/py3k/whatsnew/3.0.html

CAPÍTULO 3. SENTENCIAS DE CONTROL, MÓDULOS Y FUNCIONES

- Página web oficial de la librería estándar de Python: http://docs.python.org/py3k/library/index.html

CAPÍTULO 4. ORIENTACIÓN A OBJETOS

- Página de Wikipedia sobre orientación a objetos: http://es.wikipedia.org/wiki/Programacion_orientada_a_objetos
- Principio de abstracción: http://es.wikipedia.org/wiki/Abstracción_(informática)

CAPÍTULO 5. PROGRAMACIÓN AVANZADA

- Modelo NFA tradicional: http://en.wikipedia.org/wiki/Nondeterministic_finite_automata

- Página oficial sobre el módulo *csv:* http://docs.python.org/py3k/library/csv.html
- Documentación oficial sobre el módulo *zipfile:* http://docs.python.org/py3k/library/zipfile.html
- Documentación oficial sobre el módulo *tarfile:* http://docs.python.org/py3k/library/tarfile.html
- Documentación oficial sobre el módulo *bz2:* http://docs.python.org/py3k/library/bz2.html
- Documentación oficial sobre el módulo *gzip:* http://docs.python.org/py3k/library/gzip.html
- Estructura para ficheros INI: http://docs.python.org/py3k/library/configparser
- Documentación sobre analizadores sintácticos de XML y HTML: http://docs.python.org/py3k/library/markup.html
- Sitio web de *PyYAML:*http://pyyaml.org/wiki/PyYAML
- URL para la descarga de la versión 3.10 de *PyYAML:* http://pyyaml.org/download/pyyaml/PyYAML-3.10.zip
- Sitio web oficial de JSON: http://www.jsong.org/
- Página web sobre el módulo *simplejson:* http://pypi.python.org/pypi/simplejson/

- Sitio web de MySQL: http://www.mysql.com
- Documentación de referencia de MySQLdb: http://mysql-python.sourceforge.net/MySQLdb.html#mysqldb
- Sitio web de *psycopg2*: http://initd.org/psycopg/
- Documentación oficial de *psycopg2:*http://initd.org/psycopg/docs/
- Sitio web de PostgreSQL: http://www.postgresql.org
- Sitio web de Oracle: http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html
- Sitio web de *cx_Oracle:*http://cx-oracle.sourceforge.net/
- Sitio web de *SQLAlchemy:*http://www.SQLAlchemy.org
- Documentación oficial de *SQLAlchemy*: http://docs.SQLAlchemy.org/en/latest/index.html
- Sitio web de *SQLObject*: http://www.sqlobject.org
- Documentación oficial sobre *SQLObject*: http://sqlobject.org/SQLObject.html
- Sitio web de *Cassandra*: http://cassandra.apache.org
- Documentación oficial sobre *pycassa*: http://pycassa.github.com/pycassa/api/index.html
- Sitio web de *Redis*: http://redis.io
- Sitio web de *redis-py* en *github:* https://github.com/andymccurdy/redis-py
- Sitio web de *MongoDB:*http://www.mongodb.org
- Documentación oficial sobre *PyMongo:* http://api.mongodb.org/python/current/

CAPÍTULO 8. INTERNET

- Sitio web de WSGI: http://www.wsgi.org
- Información de referencia del módulo *lxml.html:* http://lxml.de/lxmlhtml.html
- Sitio web de *Django:*http://www.djangoproject.com
- Sitio web oficial de *Pyramid:*http://www.pylonsproject.org
- Sitio web oficial de *Pylatte:*http://www.pylatte.org
- Página web de descarga de *Pylatte:* http://www.pylatte.org/subpage/download.html

CAPÍTULO 9. INSTALACIÓN Y DISTRIBUCIÓN DE PAQUETES

- Sitio web de *PyPi:*http://pypi.python.org/pypi
- Página web de descargas del módulo *distribute:*http://python-distribute.org/
- Página web de *distribute* en *PyPi:* http://pypi.python.org/pypi/distribute
- Página web de *pip* en *PyPi:*http://pypi.python.org/pypi/pip
- Página web de *virtualenv* en *PyPi:* http://pypi.python.org/pypi/virtualenv
- Página web de *virtualenvwrapper* en *PyPi:* http://pypi.python.org/pypi/virtualenvwrapper
- Sitio web de *virtualenvwrapper:* http://www.doughellmann.com/projects/virtualenvwrapper/
- Listado completo de clasificadores para *setup.py:* http://pypi.python.org/pypi7%3AactionHist_classifiers
- Formulario de registro para registrar y subir paquetes a *PyPi*: http://pypi.python.org/pypi?%3Aaction=register_form

CAPÍTULO 10. PRUEBAS UNITARIAS

- Documentación oficial del módulo *unittest:* http:// http://docs.python.org/py3k/library/unittest.html
- Página de *PyPi* sobre *unittest2:*http://pypi.python.org/pypi/unittest2
- Sitio web de *pytest:*http://pytest.org/latest/
- Documentación oficial de *nose:* http://readthedocs.org/docs/nose/en/latest/

ÍNDICE ALFABÉTICO

S

T

U

V

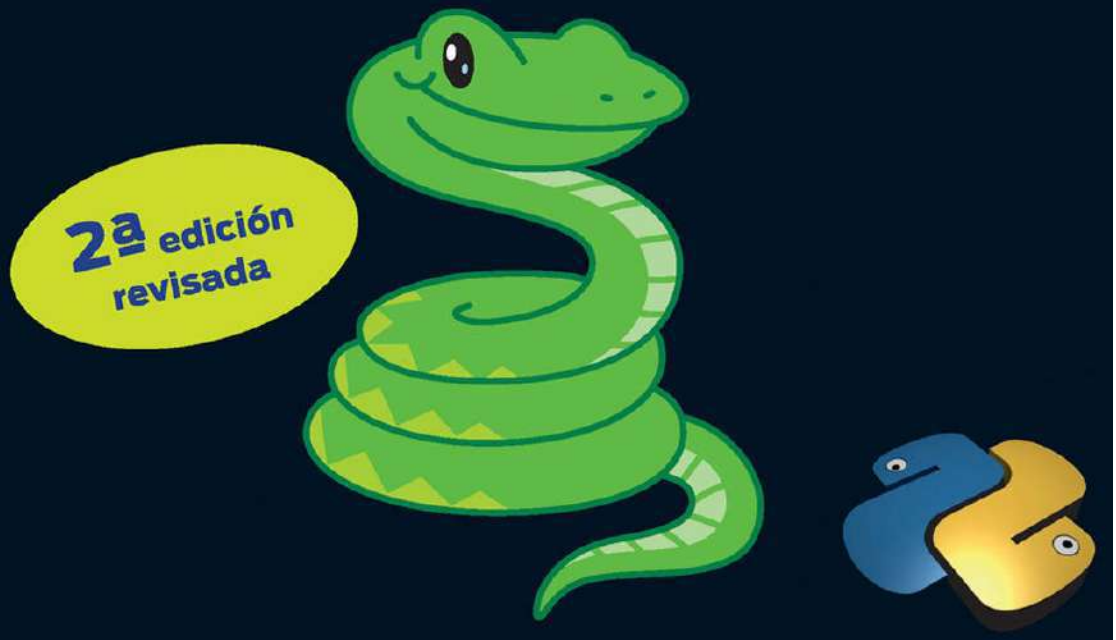
W

X

Y

Z

Python 3
al descubierto



Arturo Fernández Montoro

Alfaomega

libros R