Python práctico

Herramientas, conceptos y técnicas

Desde **www.ra-ma.es** podrá descargar material adicional.

Alfredo Moreno Muñoz • Sheila Córcoles Córcoles

Ra-Ma®

eduJ®

Python práctico

Herramientos, conceptos y técnicas

Alfredo Moreno Muñoz
Sheila Córcoles Córcoles

Ra-Ma®
edU
Conocimiento a su alcance
Bogotá - México, D.F.

Moreno Muñoz, Alfredo, *et al*

Python práctico / Alfredo Moreno Muñoz y Sheila Córcoles Córcoles --. Bogotá: Ediciones de la U, 2020
320 p. ; 24 cm
ISBN 978-958-792-168-7 e-ISBN 978-958-792-169-4
1. Informática 2. Programación 3. Conceptos 4. Técnicas. 5. Bases de datos Tít. 621.39 ed.

Edición original publicada por © Editorial Ra-ma (España)
Edición autorizada a Ediciones de la U para Colombia

Área: Informática
Primera edición: Bogotá, Colombia, mayo de 2020
ISBN. 978-958-792-168-7

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: DGP Editores SAS
Calle 63 #70D-34, Pbx (57+1) 3203510

Impreso y hecho en Colombia
Printed and made in Colombia

A nuestra hija Valeria,
no dejes nunca de aprender cosas nuevas,
tal y como llevas haciendo desde que naciste.

Te queremos

ÍNDICE

AGRADECIMIENTOS

Escribir el libro no ha sido tarea únicamente nuestra, por suerte estamos rodeados de muchas personas que nos han apoyado en este proyecto y que nos lo han puesto muy fácil para que podamos realizarlo.

En primer lugar, querríamos agradecérselo a nuestra familia, que en la distancia nos han apoyado y ayudado incondicionalmente, sabiendo que escribir el libro era importante para nosotros.

En segundo lugar, querríamos agradecérselo a nuestros amigos, por el tiempo que han dedicado a escucharnos hablar sobre Python y por todas las veces que nos han preguntado: ¿de dónde sacáis el tiempo?

Pero, el mayor agradecimiento de todos es para nuestra hija. ¡Ni te imaginas lo que hemos pensado en ti cada vez que escribíamos pensando en que algún día, tal vez, leas un libro escrito por tus padres!

¡GRACIAS!

SOBRE LOS AUTORES

ALFREDO

Ingeniero Informático, Máster en Arquitectura de Software, Certificación de Arquitecto de Software por IASA (International Association of Software Architects), Certificación en Adaptación Pedagógica y diferentes programas de liderazgo de equipos.

Actualmente trabaja como Software Solutions Architect en el Área de Investigación y Desarrollo de una multinacional social y sanitaria. Anteriormente desempeñó los puestos de Technical Lead y de Ingeniero de Software en la misma multinacional, y de Desarrollador en la Universidad de Extremadura, donde comenzó su carrera laboral.

A lo largo de su carrera ha liderado y participado en proyectos relacionados con Teleasistencia, Telemedicina y Domótica, creando proyectos innovadores en los diferentes sectores y utilizando tecnologías de vanguardia en el ámbito del desarrollo de software.

En su tiempo libre se dedica a escribir artículos relacionados con el desarrollo de software, investigar sobre tecnologías emergentes y a realizar proyectos personales con Arduino y Raspberry Pi. Creador del blog Time Of Software, que administra junto con Sheila:

http://www.timeofsoftware.com

SHEILA

Ingeniera Técnico Forestal, Máster en Dirección de Proyectos Informáticos, Certificación en Adaptación Pedagógica, múltiples certificaciones como docente en Robótica Educativa: Arduino, Crumble, Scratch, Lego, We Do, EV3, Inkscape y Raspberry Pi.

Actualmente trabaja como Técnico Informático en el Departamento de Informática de una multinacional social y sanitaria. Anteriormente ha ejercido como docente de secundaria y bachillerato y formadora de formadores, tanto en el sector público como en el privado, impartiendo diferentes asignaturas dentro del Área de Tecnología, incluyendo entre ellas Informática, Robótica y Electrónica.

Toda su carrera ha estado ligada a la tecnología, desde la Topografía Electrónica hasta la Robótica Educativa, pasando por la Informática de Sistemas, Electricidad y Electrónica.

En su tiempo libre se dedica a realizar proyecto personales relacionados con Arduino y Raspberry Pi, además, administra junto a Alfredo el Blog Time of Software:

http://www.timeofsoftware.com

INTRODUCCIÓN

Variables, programar, bucles, programación orientada a objetos, funciones, bases de datos, recursividad, booleanos, terminal, integración de aplicaciones, listas, Python, código fuente, excepciones, IDLE, tuplas, entorno de programación, pruebas, diccionarios, ficheros, pilas, colas, intérprete, comentarios de código…

¿Cuántos de estos términos te resultan familiares?

En estos días la programación se están convirtiendo en algo cada vez más a la orden del día. Está demostrado que programar tiene una serie de beneficios y es por ello por lo que está siendo introducida como asignatura en los colegios. Entre los beneficios de la programación podemos destacar:

- Enseña a seguir pasos o indicaciones.
- Aumenta la capacidad de resolución de problemas.
- Aumenta la creatividad.
- Mejora la capacidad de atención y concentración.
- Mejora el orden.
- Aumenta las capacidades de cálculo y lógica.

El objetivo del libro no es otro que crear una base sólida de programación para que puedas desenvolverte ante cualquier problema de programación. Para conseguir este objetivo vamos a utilizar el lenguaje de programación **Python**.

Python es un lenguaje de programación poderoso y fácil de aprender cuya sintaxis facilita el aprendizaje del lenguaje y de los conceptos de programación.

Nosotros creemos en el aprendizaje práctico y por ello hemos diseñado un libro en el que te explicamos todos los conceptos teóricos de programación

apoyándonos en ejercicios prácticos para afianzar dichos conceptos. Estamos seguros de que, si nos acompañas hasta el final del libro, se te van a ocurrir una cantidad grande de ideas de proyectos de programación, ya que cuantos más conocimientos vas aprendiendo, mas curiosidad desarrollarás y más ideas te irán surgiendo.

Te animamos a que comiences a adentrarte en este mundo y disfrutes con cada proyecto. No desesperes si no lo consigues a la primera, ya que seguro que de cada error aprendes algo que te sirve para seguir avanzando. Esto es solo el comienzo.

Te proponemos un reto, cuando acabes el libro vuelve a esta página y vuelve a leer el primer párrafo y pregúntate: ¿Cuántos términos me suenan? Si la respuesta es "Todos" habremos conseguido nuestro objetivo con el libro, enseñarte a programar y enseñarte Python.

¡Disfruta!

ENFOQUE DEL LIBRO

El libro se encuentra claramente centrado en un modo de aprendizaje que ha dado y da resultados en periodos de tiempo reducidos. El modo de aprendizaje no es otro que el **aprendizaje práctico**.

El libro se encuentra dividido en tres bloques claramente diferenciados que irán sumergiéndote de manera progresiva en el aprendizaje de la programación mediante el lenguaje de programación **Python**. Los bloques del libro son los siguientes:

- Bloque 1: Conceptos teóricos.
- Bloque 2: Puesta en marcha.
- Bloque 3: Aprendizaje práctico.

En el primer bloque vamos a exponerte conceptos teóricos básicos que debes conocer a la hora de aprender a programar. También en el bloque número 1 se explica el lenguaje de programación Python, su historia, sus ventajas y el por qué es el mejor lenguaje de programación para personas que quieren aprender a programar.

En el segundo bloque vamos a explicarte todo lo que necesitas instalar en tu ordenador para aprender a programar con Python. También harás tu primer programa con Python, el famoso "*Hola Mundo*" con el que todos los programadores empezamos en su día (no todos tuvimos la suerte de empezar con Python).

El tercer bloque del libro es el núcleo del mismo, el más importante ya que es el objetivo para el que se ha escrito el libro: enseñar a programar en Python. El bloque está estructurado con contenido que se presenta en forma de dificultad creciente, es decir, empezaremos con conceptos teóricos y prácticos básicos, cuya dificultad irá creciendo a medida que avanzas con el libro. El objetivo de todos los

capítulos del bloque es claro: presentar uno o más conceptos de programación de forma teórica y después plasmar el aprendizaje en una serie de ejercicios prácticos.

Por último, sin pertenecer a ningún bloque en concreto, se han incluido una serie de anexos junto con un glosario de términos que te facilitarán el aprendizaje de cada uno de los conceptos que aquí te presentamos.

Sin más dilación…

¡EMPECEMOS!

BLOQUE 1

CONCEPTOS TEÓRICOS

1

ANTES DE EMPEZAR

En este capítulo vamos a explicarte una serie de conceptos que debes de tener claros antes de empezar a aprender a programar con Python.

Empezaremos con la explicación de qué es un programa, continuaremos con qué es programar y por último explicaremos qué es el software libre.

1.1 ¿QUÉ ES UN PROGRAMA?

Un programa es el primer concepto que tienes que entender, y no es otra cosa que un conjunto de instrucciones que se le dan a un ordenador de forma secuencial para que realice una tarea específica.

Normalmente los programas reciben una serie de datos por parte del usuario de los mismos y obtienen otros datos de salida una vez terminan de ejecutar la tarea que se les ha ordenador realizar mediante programación.

El flujo normal de un programa es el siguiente:

- El programa recibe datos de entrada, normalmente introducidos por los usuarios de éste.
- Ejecuta las instrucciones especificadas por el programador.
- El programa obtiene como resultado un conjunto de datos de salida.

La siguiente imagen muestra lo que sería un programa desde un punto de vista de alto nivel, es decir, lo que ve un usuario relativo a un programa:

1.2 ¿QUÉ ES PROGRAMAR?

El siguiente concepto a entender es lo que significa programar, que debes de entenderlo como la acción de decirle a un ordenador exactamente qué tiene que hacer y cómo lo tiene que hacer.

La forma de decirle esto a un ordenador es mediante los lenguajes de programación, que básicamente son lenguajes de comunicación entre las personas y los ordenadores que permiten a los programadores transformar la idea que tienen del programa en un conjunto de instrucciones que el ordenador es capaz de ejecutar.

En el libro aprenderemos a utilizar el lenguaje de programación Python. En el siguiente capítulo te vamos a explicar las bondades de dicho lenguaje.

1.3 SOFTWARE LIBRE

Antes de explicar lo que es el software libre debemos de explicar lo que es el software. A grandes rasgos, el software es el conjunto de programas, instrucciones y reglas informáticas que permiten ejecutar distintas tareas en una computadora.

Decimos que un software es libre cuando los usuarios tienen la libertad de ejecutar, copiar, distribuir, estudiar, modificar y mejorar con total libertad. Gracias a dicha libertad, el control del software recae en los propios usuarios.

En la otra cara de la moneda se encuentra el software privativo, es decir, aquel software sobre el que el usuario no tiene las libertades expuestas en el párrafo anterior. En este caso, el programa es quien controla a los usuarios, aunque si somos

concretos, es la empresa o equipo de desarrollo que hay detrás del software el que controla a los usuarios.

Tal y como puedes observar, parece clara la diferenciación entre software libre y software privativo, pero, existen una serie de puntos que un software tiene que reunir para considerarse software libre. Son las llamadas "libertades esenciales", que son:

- **Libertad 0**: Libertad de ejecución del programa como se desea y con cualquier propósito.
- **Libertad 1**: Libertad de estudiar cómo funciona el programa y la posibilidad de cambiarlo para que haga lo que uno quiere. Es indispensable para cumplir esta libertad que el código fuente del programa sea accesible.
- **Libertad 2**: Libertad de redistribuir copias para ayudar a su prójimo.
- **Libertad 3**: Libertad de distribuir copias de las versiones modificada por el usuario para uso y disfrute de la comunidad. Al igual que en la libertad número 2, el acceso al código fuente es indispensable.

Un software se considerará libre si cumple todas las libertades de manera adecuada, en caso contrario, no se considerará software libre.

Existe un supuesto que tienes que tener en cuenta si estás pensando en distribuir software libre y es que todo software que quiera ser distribuido como software libre deberá ser libre en su conjunto, es decir, que además del propio software que queremos distribuir como libre deberán ser libres también aquellos programas que son necesarios para el correcto funcionamiento del mismo.

A continuación, te ofrecemos una serie de puntos que te ayudarán a aclarar las libertades que debe cumplir un software para ser considerado como software libre:

- Cualquier persona u organización es libre de usar el software.
- El uso del software no tiene que ser comunicado al desarrollador del mismo.
- Si distribuyes software libre a terceros, ellos tendrán total libertad de uso y podrán ejecutarlo para lo que necesiten.
- El acceso al código fuente es una condición necesaria para considerar el software como software libre.

- Cualquier modificación que se haga del código fuente deberá incluir la posibilidad de acceso al código fuente para seguir considerando la versión nueva como software libre.
- Cualquier modificación que se haga y en la que se incluya un copyright impedirá que la versión nueva sea considerada como software libre.
- Los usuarios pueden hacer modificaciones del código fuente para su uso sin necesidad de publicarlo en ningún sitio.
- Los usuarios tienen la libertad de distribuir copias originales o modificadas de forma gratuita o cobrando una tarifa por la distribución.
- Las modificaciones que se hagan del software libre jamás podrán limitar las futuras modificaciones que se quieran realizar.
- Los manuales del software deben ser libres ya que son considerados como parte del software.

A la hora de distribuir el software libre puedes aplicar reglas siempre y cuando no entren en conflicto con las libertades expuestas en este punto. Este es el caso del **Copyleft**, que implica que una distribución de software libre no puede incluir restricciones que impidan a los demás tener acceso a las libertades principales.

Existe una licencia, **GPL** (General Public License), que garantiza a los usuarios finales la libertad de usar, estudiar, modificar y compartir el software. Su objetivo es el siguiente:

- Declarar el software como Software Libre.
- Protegerlo utilizando Copyleft.

2

PYTHON

En este capítulo vamos a explicarte qué es Python y su historia, qué características tiene como lenguaje de programación, por qué debes aprender Python y por último te vamos a presentar la filosofía del lenguaje de programación.

2.1 SU HISTORIA

Antes de contarte la historia de Python consideramos que tienes que leer la definición que hacemos del lenguaje en una sola frase y que refleja la realidad del mismo:

Python es un lenguaje de programación de propósito general muy sencillo y fácil de aprender, a la vez que poderoso y flexible.

Python es un lenguaje de programación relativamente joven, surgió a finales de la década de los 80s y principio de los 90s y su creador es Guido Van Rossum (Holanda). El nombre del lenguaje de programación viene del grupo *Monty Python*, grupo del que el creador era fiel seguidor.

Guido Van Rossum ideó el lenguaje a finales de los 80s y empezó su implementación en diciembre de 1989, publicando la primera versión del lenguaje (v0.9.0) en febrero de 1991. Python surgió como un pasatiempo para darle continuidad al lenguaje que utilizaba Guido Van Rossum en su puesto de trabajo dentro de un centro de investigación holandés. Guido Van Rossum ha dirigido el desarrollo del lenguaje de programación desde su creación hasta principios de 2019. En Julio del 2018 anunció que dejaría de dirigir el lenguaje y que se pondría en marcha un consejo director de cinco miembros que serían los encargados de llevar las directrices del desarrollo del lenguaje, con la restricción de que únicamente podrían publicar una

versión de Python y que posteriormente se tendría que volver a elegir ese consejo director.

Las versiones de Python siguen una nomenclatura X.Y.Z con el siguiente significado:

- **X**: indica la versión mayor de Python. Las diferentes versiones mayores son incompatibles entre ellas.
- **Y**: indica una versión importante dentro de la versión mayor, pero manteniendo las compatibilidades dentro de la versión mayor.
- **Z**: indica una versión menor en la que se corrigen únicamente errores o fallos de seguridad.

En la historia de Python han existido tres versiones mayores:

- **Python 1.0.0**: publicada en enero de 1994.
- **Python 2.0.0**: publicada en octubre del 2000.
- **Python 3.0.0**: publicada en diciembre de 2011.

2.2 CARACTERÍSTICAS

En este apartado vamos a mostrarte las características principales del lenguaje de programación:

- **Simplicidad:** ¡La gran fortaleza de Python!
- **Sintaxis clara:** la sintaxis de Python es muy clara, es obligatoria la utilización de la indentación en todo el código que se escribe. Gracias a esta característica todos los programas escritos en Python tienen la misma apariencia.
- **Propósito general:** se pueden crear todo tipo de programas, incluyendo páginas web. Muchas personas han considerado durante mucho que Python era un lenguaje de scripting y no un lenguaje de propósito general.
- **Lenguaje interpretado:** al ser un lenguaje interpretado no es necesario compilarlo, lo que te ahorrará tiempo a la hora de desarrollar. También implica que su ejecución sea más lenta, ya que los programas son ejecutados por el intérprete de Python en vez de ejecutados por la máquina donde lo arrancas.

- **Lenguaje de alto nivel:** no es necesario que te preocupes de aspectos de bajo nivel como puede ser el manejo de la memoria del programa.
- **Lenguaje orientado a objetos:** lenguaje construido sobre objetos que incorporan datos y funcionalidades.
- **Lenguaje multiparadigma:** además de programación orientada a objetos, Python ofrece programación estructurada, programación imperativa y programación funcional.
- **Fuertemente tipado:** esto implica que el tipo de valor no cambia de forma repentina, es decir, una cadena de texto que únicamente contiene número no se convertirá en un número sin realizar una conversión de tipos previa.
- **Tipado dinámico:** esto implica que el tipo de valor que contienen las variables puede cambiar en tiempo de ejecución, es decir, es posible tener una variable que durante la ejecución de un programa ha almacenado una cadena de texto y posteriormente un entero.
- **Open Source:** Python ha sido portado a los diferentes sistemas operativos, por lo que puedes usarlo en el que más te guste. Otra característica de ser Open Source es que es un lenguaje de programación gratuito.
- **Extensas librerías:** Facilitan la programación al incorporar mediante librerías una gran cantidad de funcionalidades.
- **Incrustable:** Es posible añadir programas escritos en Python a programas escritos en C y C++.

2.3 ¿POR QUÉ PYTHON?

La primera pregunta que se hace la gente que quiere aprender a programar es: ¿Qué lenguaje de programación debería aprender?. La verdad que existe un abanico de lenguajes de programación muy grande donde elegir y decantarse por uno o por otro puede ser difícil. Nosotros tenemos la opinión de que el mejor lenguaje de programación para prender a programar es Python, y en este apartado te explicaremos las razones de nuestros pensamiento.

Aunque existen multitud de lenguajes de programación, todos son muy parecidos entre ellos (Java, C#, C++…). En muchos casos, lo único que cambia entre ellos es simplemente la sintaxis utilizada para programar, lo cual es muy bueno, ya que aprendiendo uno de esos lenguajes te costará muy poco aprender los otros, simplemente cambiar la sintaxis.

Nosotros no pensamos únicamente que Python sea un lenguaje de programación perfecto para personas que acaban de comenzar, sino que es un lenguaje de programación que tiene una serie de características que le encantarán a cualquier persona que ya sepa programar.

Veamos las razones de porqué aprender Python:

- **Simplicidad**: la característica principal es Python es que es un lenguaje simple, reduce considerablemente el número de líneas de código en comparación con otros lenguajes y provee herramientas para realizar operaciones de forma más simple que como se realizan con otros lenguajes.

- **Resultados rápidos**: aprendiendo Python vas a estar haciendo programas a los pocos días, o incluso horas, verás como avanzas casi sin esfuerzo a gran velocidad. Este punto es muy importante, a cualquier persona que está aprendiendo le gusta ver resultados en un periodo corto de tiempo.

- **Comunidad**: la comunidad que hay detrás de este lenguaje de programación es inmensa, lo que provoca que el lenguaje no quede obsoleto y vaya recibiendo actualizaciones. Otro punto fuerte de la comunidad que tiene detrás es la creación de frameworks, módulos, extensiones y multitud de herramientas que facilitan el desarrollo con este lenguaje. Los desarrolladores en Python son los primeros interesados en que haya más gente que programe con Python, ya que, de esta forma, el número de herramientas/frameworks que facilitan el desarrollo será mayor.

  Una de las cosas más importantes para alguien que empieza con un lenguaje de programación es la ayuda que ofrece la comunidad que tiene alrededor el lenguaje de programación. Si te animas a aprender Python verás como podrás encontrar sin dificultad la resolución de tus preguntas/dudas/problemas.

- **Punto de partida**: Python es un lenguaje muy completo, no pienses que por ser simple es un lenguaje básico. Con Python vas a aprender todos los conceptos existentes en el mundo de la programación, como por ejemplo puede ser la programación orientada a objetos (POO), hilos… Python abarca todos los campos existentes dentro de la programación.

- **Ordenado y limpio**: el orden que mantiene Python es de las cosas que más gusta a los desarrolladores, es fácil de leer y por tanto fácil de mantener.

- **Multitud de librerías**: Python es un lenguaje poderoso. A medida que te vas familiarizando con el lenguaje y vas aprendiendo y manejando todas las funcionalidades descubres que Python dispone de un conjunto de librerías y módulos muy extenso que te permiten realizar cualquier tipo de proyecto, con total independencia de su naturaleza.
- **Portable**: puede ser utilizado en todos los sistemas operativos (Mac OS X, Microsoft Windows, Linux).
- **Desarrollo web**: el desarrollo web está de moda, y como no, Python posee una multitud de frameworks para desarrollar páginas web, entre ellos destaca Django.
- **Raspberry Pi**: Python es el lenguaje principal de programación de Raspberry.
- **Demanda laboral alta**: Python es utilizado por las grandes empresas tecnológicas del mundo… Saber Python implicará tener más posibilidades de encontrar ese trabajo que siempre has querido tener.

2.4 LA FILOSOFÍA DE PYTHON

La filosofía del lenguaje Python está plasmada en el documento escrito por Tim Peters que puedes encontrar en su página web. A continuación, encontrarás los mantras de Python traducidos al castellano:

- Hermoso es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Sencillo es mejor que anidado.
- Escaso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son lo suficientemente especiales para romper las reglas.
- Lo práctico le gana a la pureza.

- Los errores no deben pasar en silencio.
- A menos que sean silenciados.
- Respecto a la ambigüedad, rechazar la tentación de adivinar.
- Debe haber una – y preferiblemente sólo una – manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia en un primer momento a menos que seas holandés.
- Ahora es mejor que nunca.
- Aunque "nunca" es a menudo mejor que "ahora mismo".
- Si la aplicación es difícil de explicar, es una mala idea.
- Si la aplicación es fácil de explicar, puede ser una buena idea.
- Los espacios de nombres son una gran idea ¡hay que hacer más de eso!

BLOQUE 2

PUESTA EN MARCHA

3

INSTALACIÓN

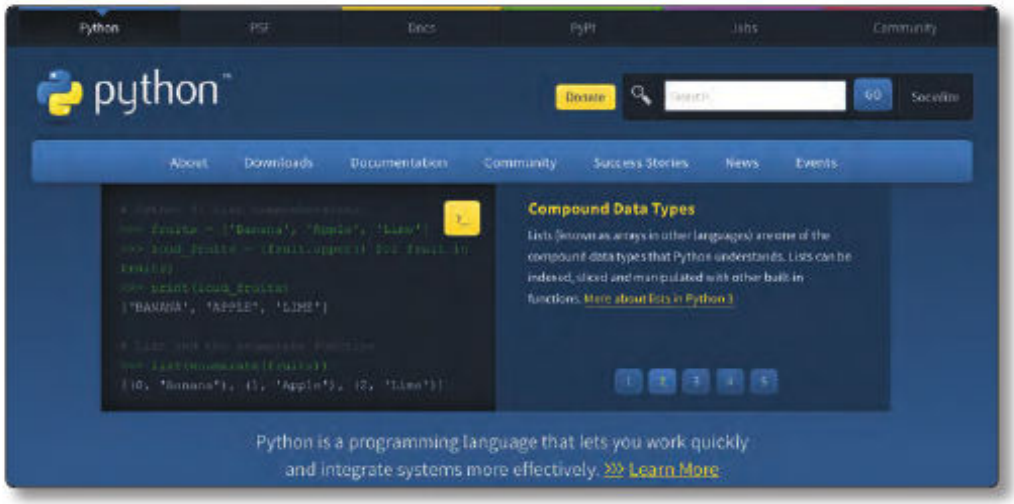
En este capítulo vamos a explicarte cómo realizar la instalación de Python en tu ordenador.

Al instalar Python instalas también un entorno de desarrollo básico que lleva incluido en el paquete de instalación desde la versión 1.5 de Python. El entorno de desarrollo se llama IDLE (**I**ntegrated **D**eve**L**opment **E**nvironment o **I**ntegrated **D**evelopment and **L**earning **E**nvironment).

Tal y como hemos visto en el bloque anterior, Python es multiplataforma, por lo tanto es posible instalarlo en cualquier sistema operativo. En el libro vamos a explicarte la instalación en los tres sistemas operativos más comunes:

- Mac OS X
- Microsoft Windows
- Linux

El paquete de instalación se descarga desde la página web de Python (*https://www.python.org*). La siguiente imagen muestra la página principal de la página web de Python:



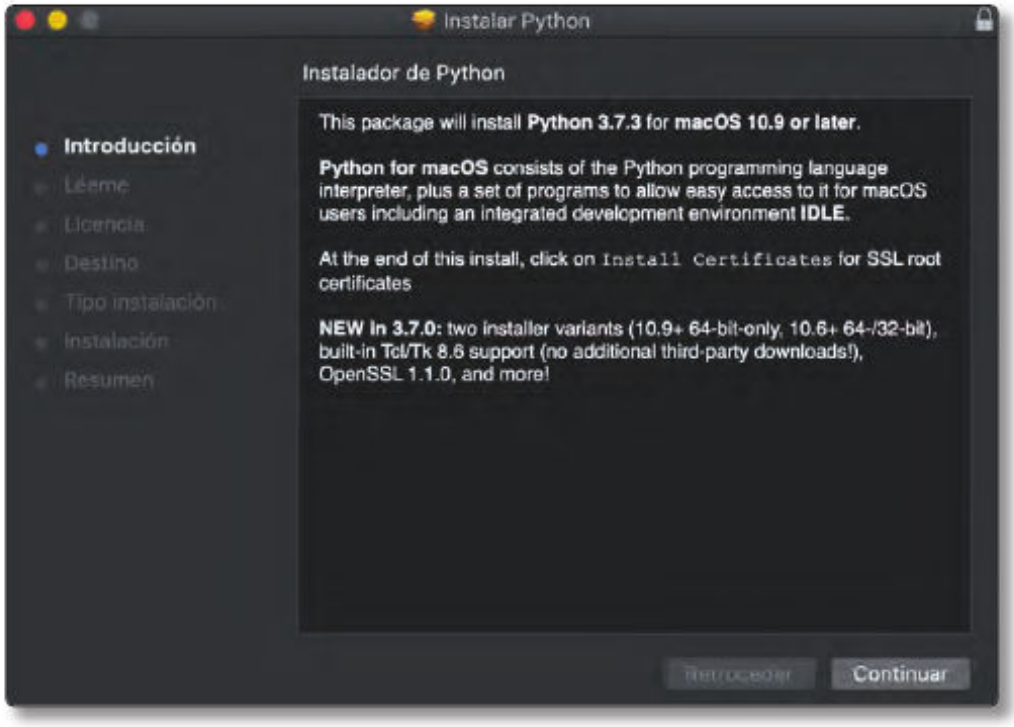
Una vez estés dentro de la web de Python, tienes que navegar a la sección *Downloads*. Por defecto, te aparecerá para descargar la versión que se corresponde con el sistema operativo de tu ordenador. Descarga la versión presionando el botón "*Download Python 3.7.3*".

Una vez se haya completado la descarga es el momento de instalar Python en tu ordenador. Veamos en detalle cómo se instala en cada uno de los sistemas operativos comentados anteriormente.

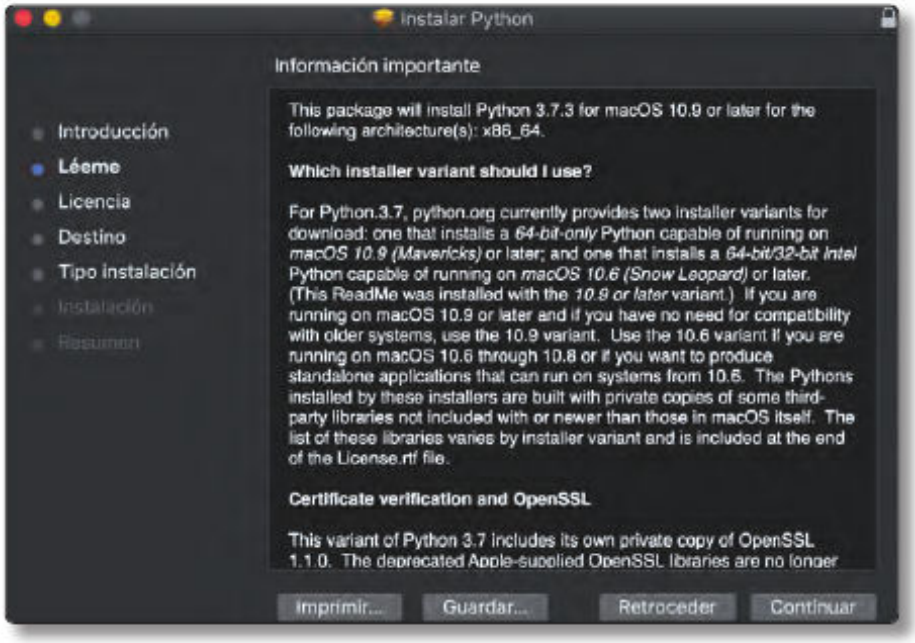
3.1 MAC OS X

La instalación en Mac OS X se realiza ejecutando el paquete que acabamos de descargar.

En la primera pantalla que verás cuando lo ejecutes podrás leer una pequeña introducción a lo que vas a instalar. La pantalla es la siguiente:

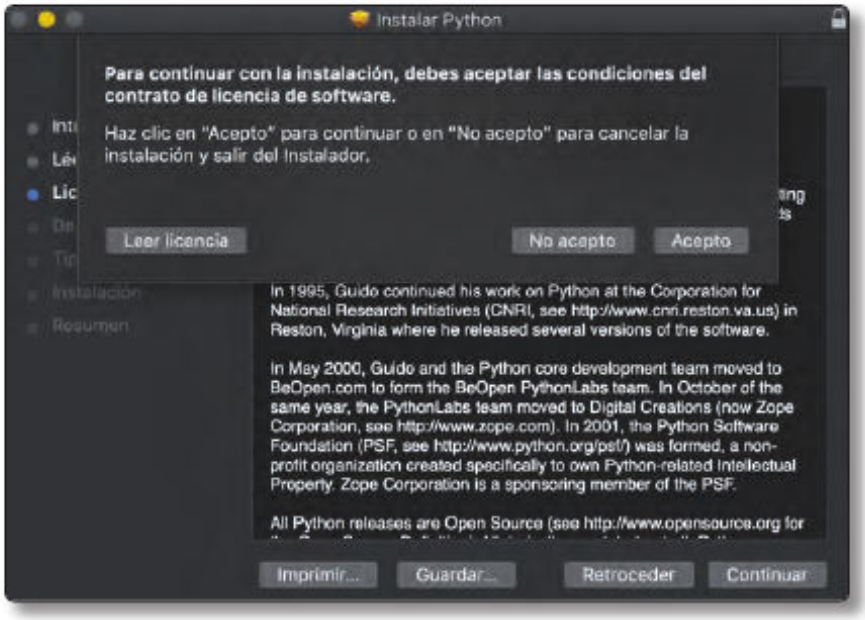


Una vez leída la introducción deberás presionar el botón "*Continuar*", esto te llevará a una pantalla con bastante información sobre Python y compatibilidades con el sistema operativo, te aconsejamos que la leas. Es posible imprimir la información o almacenarla en otro formato digital. Para pasar al siguiente paso presiona el botón "*Continuar*". La pantalla "*Leéme*" es la siguiente:



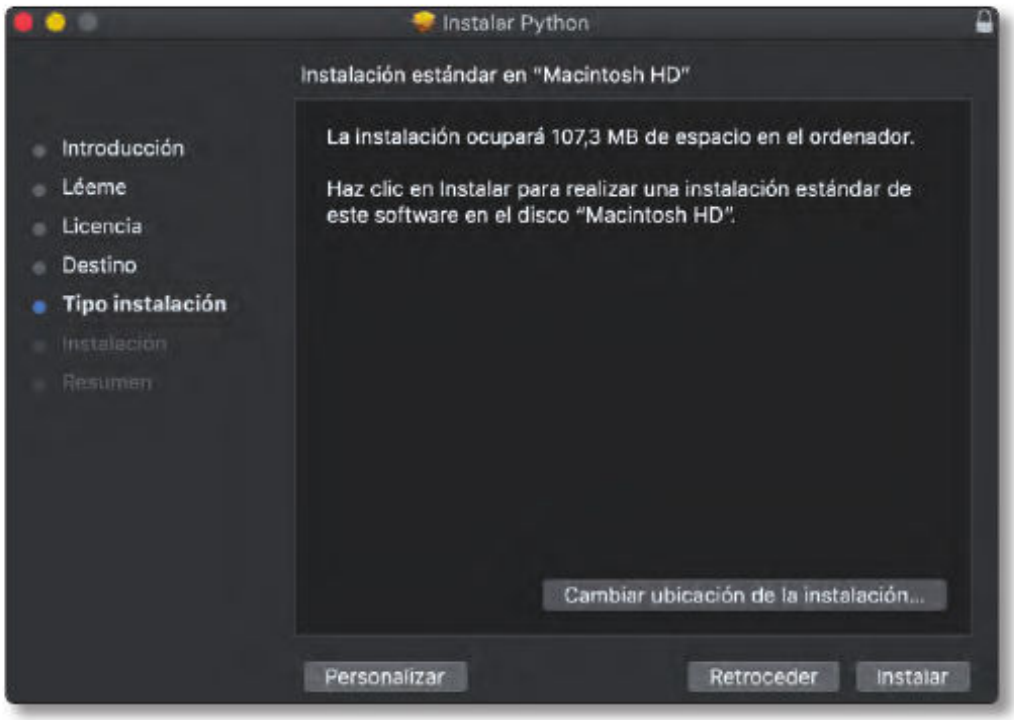
La siguiente pantalla es la licencia de uso del software que vas a instalar. Al igual que en la anterior pantalla, puedes imprimir o guardar en otro formato digital la licencia. Para pasar al siguiente paso tienes que presionar el botón *Continuar*. La pantalla de *Licencia* es la siguiente:

Si no has leído la licencia hasta el final recibirás el siguiente mensaje que te solicitará si la aceptas o no o si quieres volver a la pantalla anterior para leerla.

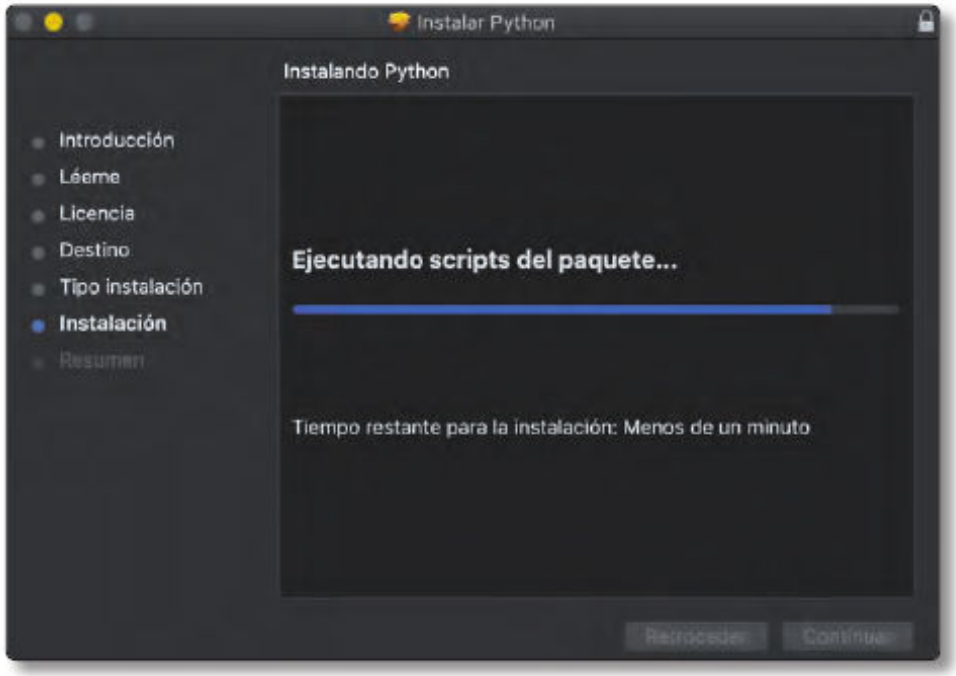


En la siguiente pantalla podrás seleccionar el disco duro dónde instalar Python. Una vez tengas seleccionado el disco presiona el botón *Continuar*. La pantalla de selección del disco duro es la siguiente:

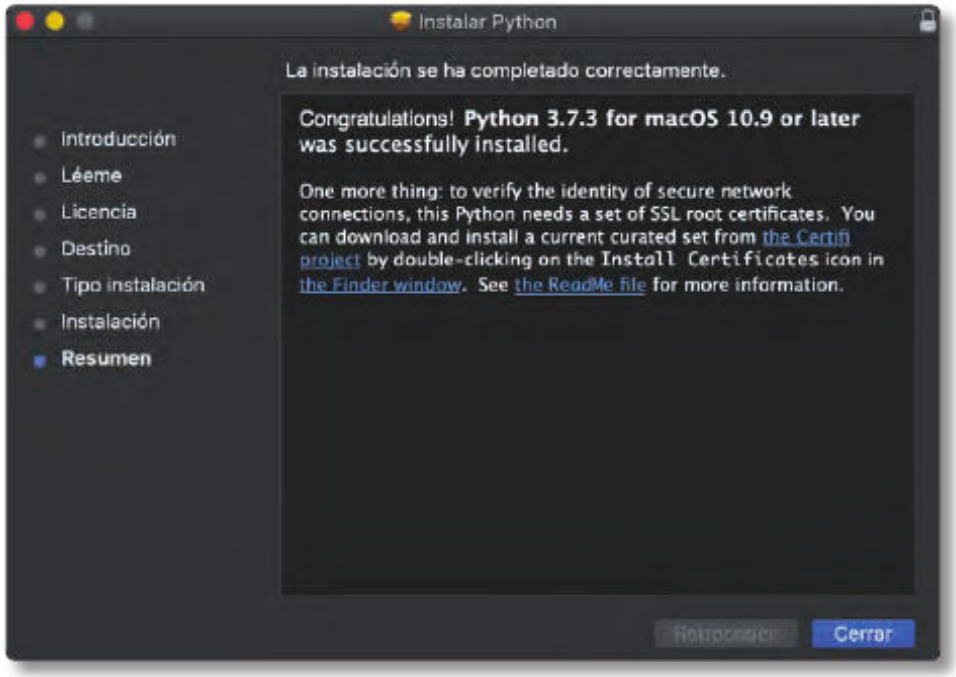
Llega el momento de seleccionar el tipo de instalación y la ubicación. Es posible especificar la ruta en la que se instala Python en lugar de la ruta de instalación por defecto, en caso de que quieras cambiarla deberás utilizar el botón "*Cambiar ubicación de la instalación...*". También es posible cambiar el tipo de instalación que se realiza mediante el botón *Personalizar*, si eres un usuario avanzado te recomendamos que uses esta opción de instalación, en caso contrario (o por ser lo que quieres) no toques nada e instala por defecto. La pantalla de selección de ubicación y tipo de instalación es la siguiente, para iniciar la instalación presiona el botón "*Instalar*":



La siguiente imagen muestra una captura del proceso de instalación:



Una vez acaba el proceso de instalación el instalador abre la pantalla de "*Resumen*", presiona "*Cerrar*" para completar el proceso:

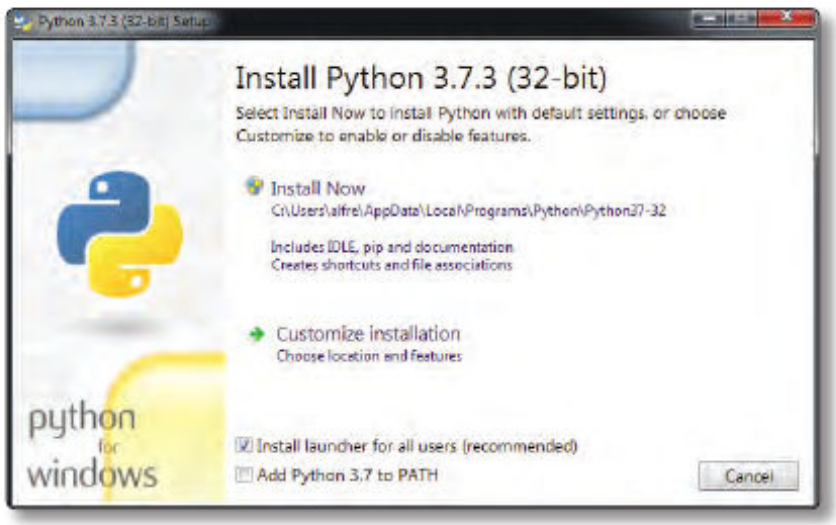


A partir de este momento Python estará instalado en tu Mac y podrás acceder a IDLE a través del *Launchpad*:

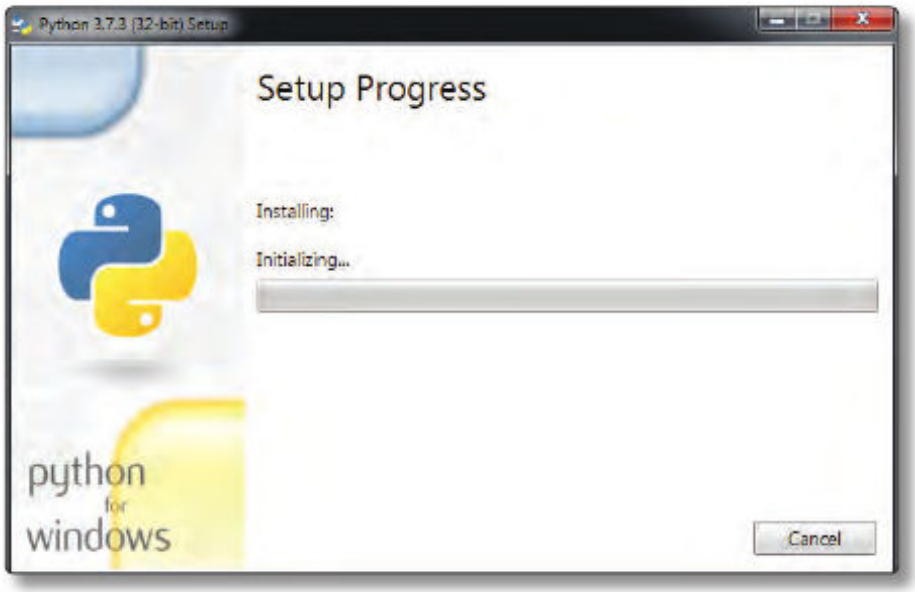
3.2 MICROSOFT WINDOWS

La instalación en Microsoft Windows se realiza ejecutando el paquete que acabamos de descargar.

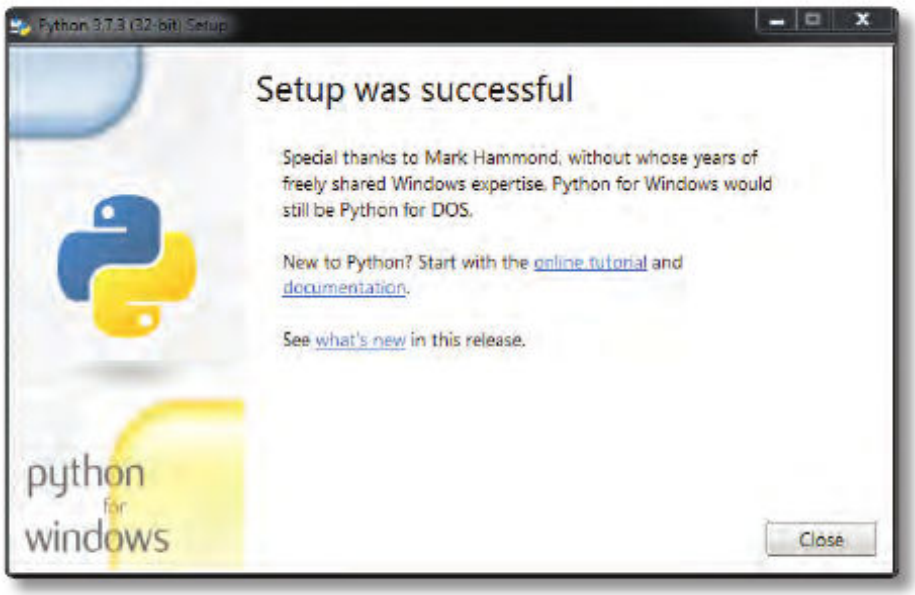
Una vez ejecutado el instalador se abre la primera pantalla, en la que podrás seleccionar una instalación rápida ("*Install Now*") en la que no seleccionas ni la ruta ni los componentes que se instalan o podrás seleccionar también una personalización de la misma ("*Customize installation*") para usuarios con conocimientos avanzados. En la pantalla es posible seleccionar el realizar la instalación para todos los usuarios y añadir al path del sistema Python. La pantalla inicial es la siguiente:



La siguiente imagen muestra la pantalla que verás durante el proceso de instalación:



Una vez acaba la instalación verás la siguiente pantalla indicándote que la instalación ha terminado satisfactoriamente, presiona el botón "*Close*" para finalizar el proceso:

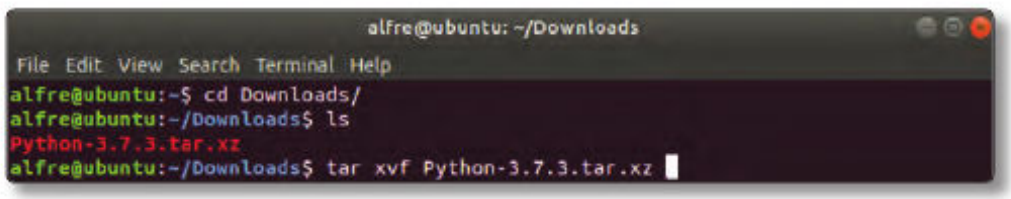


El acceso a IDLE de Python está dentro del menú inicio.

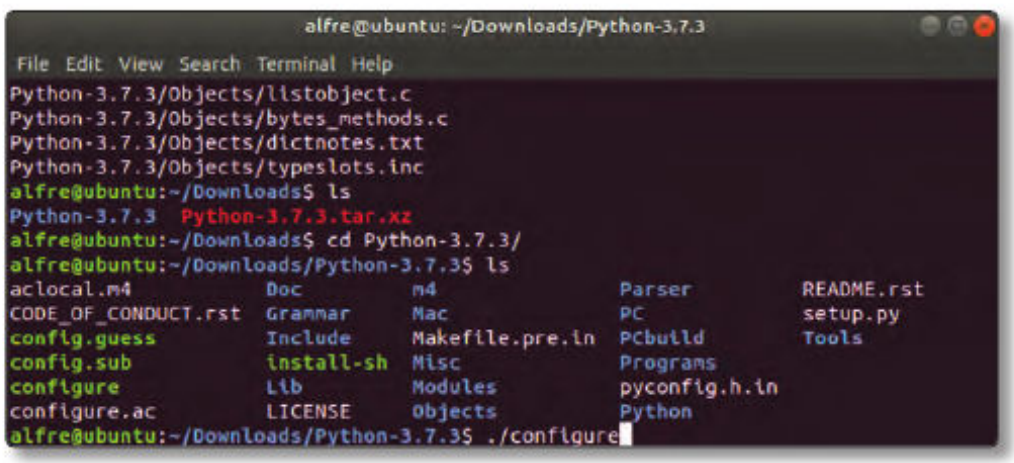
3.3 LINUX

La instalación en Linux está dividida en dos partes, por una se instala Python y por otra IDLE. El paquete que descargas desde la web de Python no incluye IDLE, por lo que se instala a posteriori.

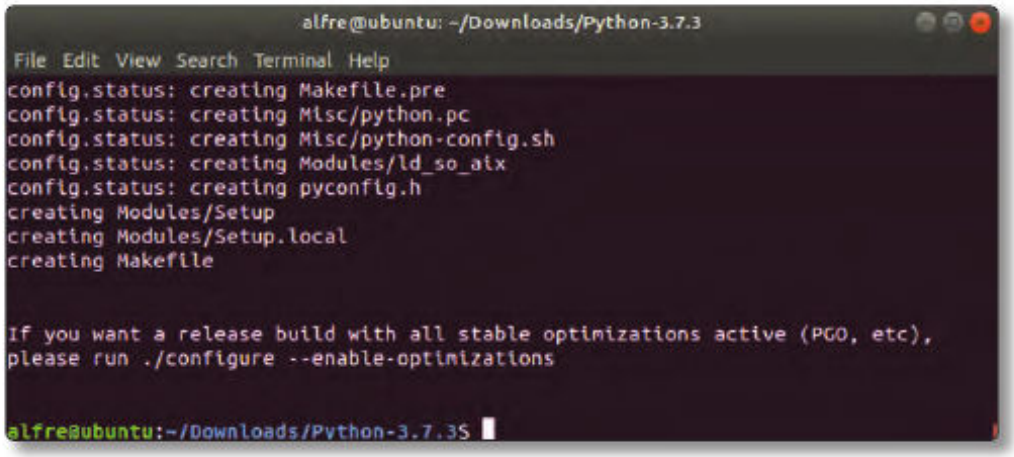
Para la instalación del paquete es necesario abrir un *Terminal* y navegar hasta la ruta en la que lo has descargado. El primer paso es ejecutar el siguiente comando "*tar xvf NombrePaquete*", el nombre del paquete variará en función de la versión de Python que te descargues. La siguiente imagen muestra los pasos a seguir de forma gráfica:



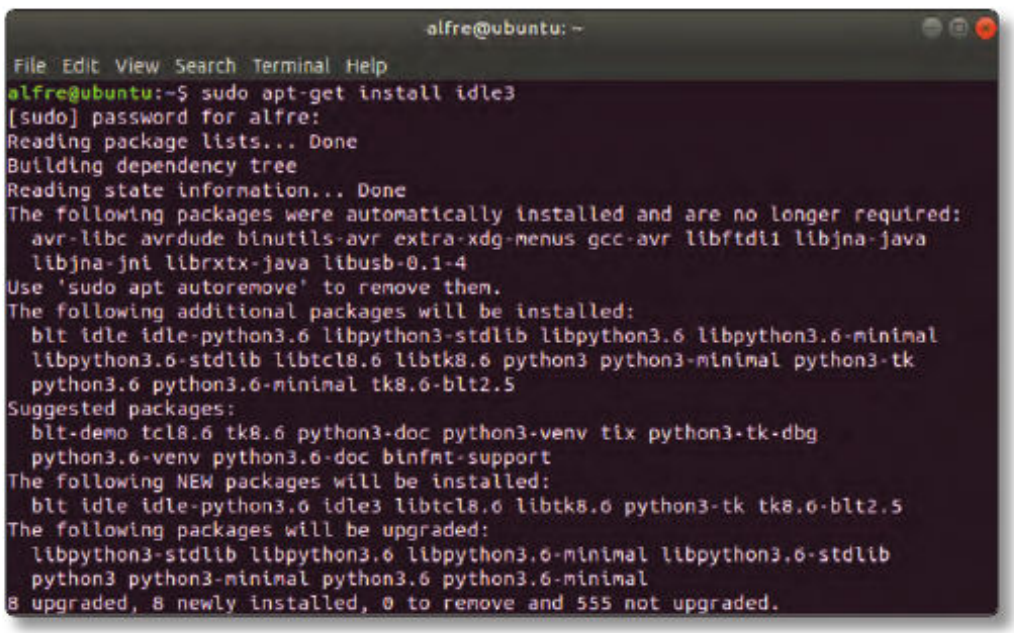
Una vez termina el primer paso tienes que entrar en la carpeta que se ha creado y ejecutar el comando "*./configure*". La siguiente imagen muestra los pasos a seguir de forma gráfica:



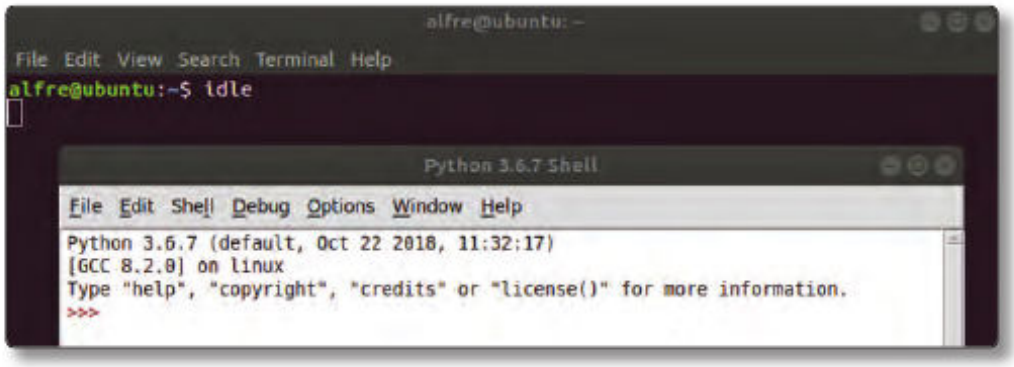
La siguiente imagen muestra la finalización del paso anterior.



En este punto ya tenemos instalado Python, por lo que el siguiente paso es instalar IDLE. La instalación se realiza desde el *Terminal* que has utilizado en los pasos anteriores, pero esta vez tienes que ejecutar el comando "*sudo apt-get Install idle3*". Durante la instalación vas a recibir preguntas de si instalar ciertos módulos, dale a todo "*Y*". La siguiente imagen muestra gráficamente el comando de instalación:



Una vez termina la instalación la apertura de IDLE la puedes realizar ejecutando el comando *"idle"* tal y como muestra la siguiente imagen:



4

ENTORNO DE DESARROLLO

En este capítulo vamos a introducirnos de lleno en los entornos de desarrollo, primero vamos a explicarte qué es un entorno de desarrollo, qué componentes tiene y qué beneficios conlleva su uso, posteriormente nos adentraremos en el entorno de desarrollo de Python, IDLE, explicando todas y cada una de las funcionalidades que aporta.

4.1 ¿QUÉ ES UN ENTORNO DE DESARROLLO?

IDE son las siglas que se corresponden con Entorno de Desarrollo Integrado, "**I**ntegrated **D**evelopment **E**nvironment".

Un IDE es un programa informático que contiene integradas todas las herramientas, utilidades y funcionalidades necesarias para facilitar la tarea de desarrollo de software.

Las tareas de desarrollo de software que integran los IDE son las siguientes:

- Creación y modificación de proyectos de desarrollo de software.
- Implementación de código fuente.
- Compilación del código fuente.
- Ejecución del programa.
- Depuración del programa.

Un aspecto muy importante que incluyen los IDE es que son entornos de programación amigables, cuidan la usabilidad de los mismos y la experiencia de usuario de los desarrolladores. Además, todos los IDE tienen más o menos la misma apariencia, lo que hace que los desarrolladores sean capaces de manejar diferentes IDE de forma fácil y sencilla para ellos.

Las características principales que presentan los IDE son las siguientes:

- **Multiplataforma**: están disponibles para los diferentes sistemas operativos existentes.
- **Múltiples idiomas**: están disponibles en diferentes idiomas para maximizar el alcance de los mismos.
- **Soporte para diversos lenguajes de programación**: permiten el desarrollo de software ofreciendo la posibilidad de utilizar diferentes lenguajes de programación. No todos los IDE soportan más de un lenguaje de programación. Existen IDE que únicamente están destinados a un único lenguaje.
- **Reconocimiento de Sintaxis**: son capaces de reconocer el conjunto de reglas que debe seguir el desarrollador al escribir un programa de forma correcta en un lenguaje concreto.
- **Extensiones y Componentes para el IDE**: permiten la posibilidad de añadir funcionalidades extras al IDE con el fin de mejorar el proceso del desarrollo de software por parte del desarrollador.
- **Integración con Framework populares**: permiten la integración de los programas desarrollados con los frameworks de desarrollo de terceros de forma dinámica y fácil.
- **Integración con Sistemas de Control de Versiones**: ofrecen la posibilidad de utilizar un sistema de control de versiones para el código fuente.
- **Depurador**: poseen herramientas para realizar depuraciones de código fuente.
- **Importar y Exportar proyectos**: permiten trabajar con proyectos que han sido creados con el mismo IDE pero en instalaciones diferentes.
- Incluyen Manual de Usuarios y Ayuda

4.1.1 Componentes de un IDE

Los componentes principales que incluyen los IDEs son los siguientes:

- **Editor de texto**: herramienta que sirve para escribir el código fuente del programa.

- **Compilador**: herramienta que permite traducir de código fuente a lenguaje máquina.
- **Depurador**: herramienta que permite probar y eliminar errores en el código fuente.
- **Editor gráfico**: herramienta que permite crear y diseñar las interfaces gráficas con las que interactuará el usuario de la aplicación.
- **Intérprete**: herramienta que es capaz de ejecutar el código fuente sin necesidad de utilizar lenguaje máquina, únicamente interpretando el lenguaje en el que está escrito el programa.

Normalmente, los IDE suelen incorporar muchos más componentes, es más, tal y como te hemos expuesto en el punto anterior, una de las características más relevantes que poseen es que tienen que permitir la posibilidad de incluir extensiones y componentes externos.

4.1.2 Ventajas de uso

La utilización de un IDE a la hora de programar te va a proporcionar una serie de ventajas respecto a no utilizarlo. A continuación, te presentamos una lista de las ventajas más importantes:

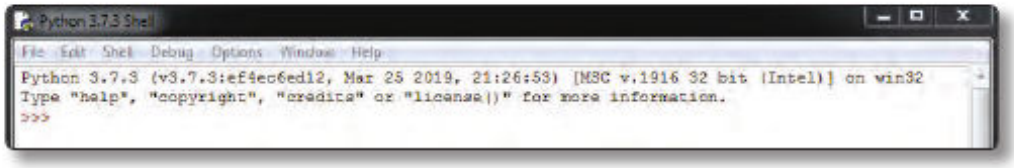
- Facilidad a la hora de codificar.
- Curva de aprendizaje muy baja.
- Herramienta óptima para personas sin experiencia.
- Creación y edición de código.
- Depuración de código fuente.
- Funciones automatizadas que permiten ahorrar tiempo:
  - Renombrar variables.
  - Renombrar clases.
  - Renombrar métodos y funciones.
  - Sangría automática.
- Búsqueda básica y avanzada de texto.
- Reemplazar texto.
- Advertencias y errores de sintaxis en pantalla.
- Herramientas de refactorización del código fuente.
- Herramientas para la creación de los componentes gráficos.
- Previsualización de los componentes gráficos.

- Posibilidad de incluir herramientas o programas externos que incrementan las funcionalidades de los IDE:
  - Emuladores.
  - Sistemas de control de código fuente.
- Integración con otras plataformas relacionadas con el desarrollo de software.

4.2 MANUAL DE USUARIO DE IDLE

En este apartado vamos a explicarte todas las funcionalidades incluidas en el entorno de desarrollo integrado en el paquete de Python, IDLE.

Lo primero que nos encontramos al abrir IDLE es la siguiente pantalla:



La pantalla que se abre es el Shell o Consola de Python. Se trata de una consola interactiva en la que puedes ejecutar código directamente y también es la consola en la que se ejecutarán los programas que ejecutes con IDLE.

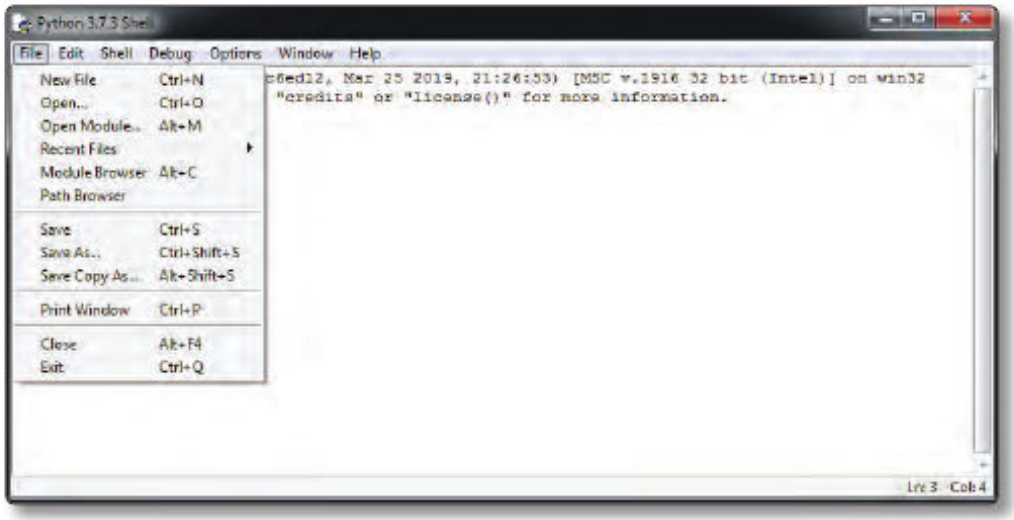
Tal y como puedes observar, la consola muestra la versión de Python instalada, en este caso la 3.7.3. La parte superior tiene un menú con todas las funcionalidades.

IDLE, además del Shell de Python, está compuesto por el editor de código fuente, que nos permitirá escribir y manipular código fuente. El menú superior del editor de código fuente es diferente al del Shell de Python, aunque varios de sus menús son compartidos entre el Shell y el Editor.

¡Vamos a ver los diferentes menús que existen en IDLE!

4.2.1 Menú File

El menú *File* nos va a permitir realizar las típicas funciones de un menú *Archivo*, pero obviamente orientadas a la programación con Python. El menú está disponible tanto en el Shell de Python como en el Editor. La siguiente imagen muestra el menú desplegado:

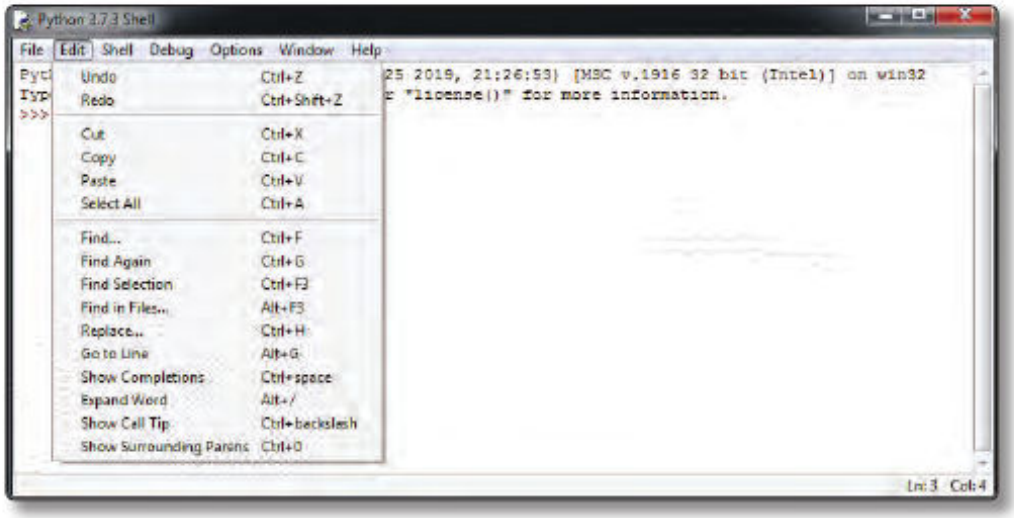


Veamos en detalle cada una de las opciones del menú:

- **New File**: crea un nuevo fichero en la ventana de edición.
- **Open...**: abre un archivo existente desde un cuadro de diálogo.
- **Open Module...**: abre un módulo existente desde un cuadro de diálogo.
- **Recent Files**: lista de ficheros reciente, se pueden abrir haciendo clic en ellos.
- **Module Browser**: abre un módulo existente junto con una ventana de navegación por él desde un cuadro de diálogo.
- **Path Browser**: muestra todo lo incluido de Python en el path del sistema.
- **Save**: guarda el fichero existente.
- **Save As...**: permite guarda el fichero existente como otro fichero.
- **Save Copy As...**: permite crear otro fichero idéntico al abierto pero con diferente nombre y ruta.
- **Print Window**: imprime la ventana actual por la impresora por defecto.
- **Close**: permite cerrar la ventana actual, en caso de no haber guardado los cambios pregunta si queremos guardar.
- **Exit**: cierra todas las ventanas y sale de IDLE, en caso de no haber guardado los cambios en los ficheros pregunta si queremos guardar.

4.2.2 Menú Edit

El menú *Edit* nos va a permitir realizar las típicas acciones de los menú *Edición* a la hora de manipular texto. El menú está disponible tanto en el Shell de Python como en el Editor. La siguiente imagen muestra el menú desplegado:



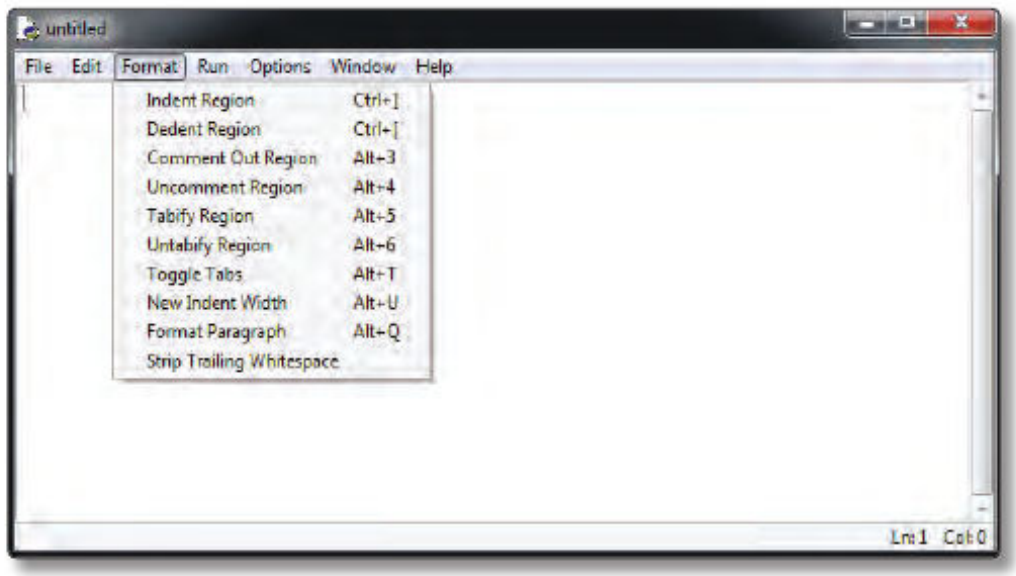
Veamos en detalle cada una de las opciones del menú:

- **Undo**: deshace el último cambio realizado sobre el fichero abierto.
- **Redo**: realiza de nuevo el último cambio deshecho sobre el fichero abierto.
- **Cut**: copia al portapapeles el texto seleccionado y lo borrar del fichero.
- **Copy**: copia al portapapeles el texto seleccionado.
- **Paste**: pega el contenido del portapapeles en la ventana abierta y en la posición en la que se encuentra el cursor dentro del fichero.
- **Select All**: selecciona todo el texto del fichero abierto.
- **Find…**: abre un cuadro de diálogo con múltiples opciones para realizar búsqueda en el fichero abierto.
- **Find Again**: realiza de forma automática la última búsqueda realizada en caso de que exista una última búsqueda.
- **Find Selection**: realiza una búsqueda de forma automática del texto seleccionado en el fichero en caso de existir texto seleccionado.

- **Find in Files...**: realiza una búsqueda en los ficheros seleccionados mediante el cuadro de diálogo.
- **Replace...**: abre un cuadro de diálogo con múltiples opciones para buscar y reemplazar cadenas de texto.
- **Go to Line**: permite navegar a una línea en concreto dentro del fichero abierto e introduciendo la línea en un cuadro de diálogo.
- **Show Completions**: muestra una lista de posibles palabras reservadas para introducir en la posición en la que se encuentra el cursos dentro del fichero abierto.
- **Expand Word**: expande un prefijo que se haya escrito en el fichero abierto para que coincida con un texto completo ya escrito.
- **Show Call Tip**: ayuda a completar los parámetros al usar una función.
- **Show Surrounding Parens**: muestra paréntesis redundantes.

4.2.3 Menú Format

El menú *Format* nos va a permitir manipular código fuente con funciones relacionadas con la programación (indentaciones, comentarios, etc.). El menú únicamente está disponible en el Editor. La siguiente imagen muestra el menú desplegado:

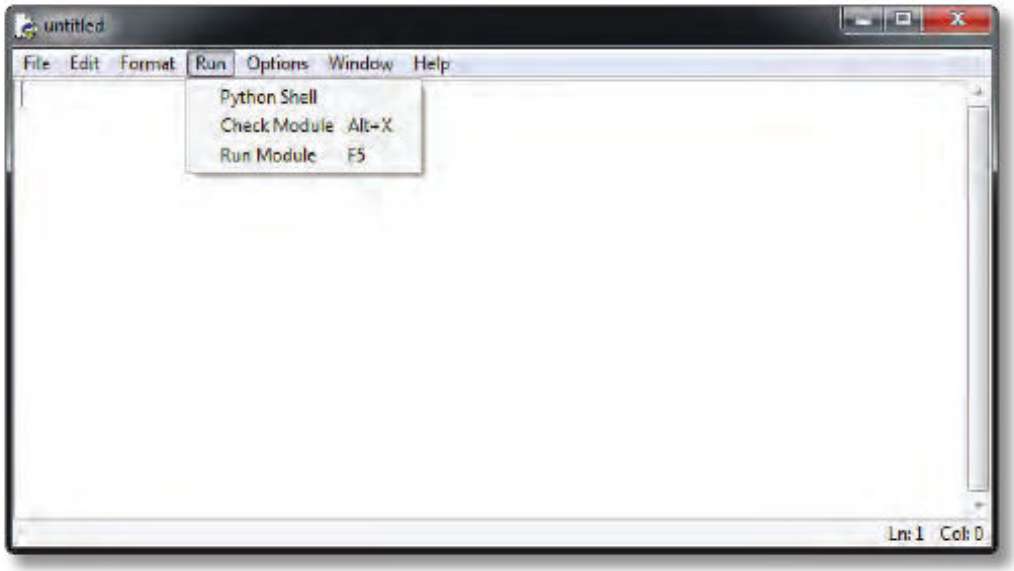


Veamos en detalle cada una de las opciones del menú:

- **Indent Region**: mueve las líneas de código fuente seleccionadas una indentación a la derecha.
- **Dedent Region**: mueve las líneas de código fuente seleccionadas una indentación a la izquierda.
- **Comment Out Region**: convierte en comentarios las líneas de código seleccionadas.
- **Uncomment Region**: elimina los comentarios de las líneas de código seleccionadas.
- **Tabify Region**: convierte los espacios en blanco en tabulaciones para indentar el texto seleccionado.
- **Untabify Region**: convierte las tabulaciones en espacios en blanco para indentar el texto seleccionado.
- **Toggle Tabs**: abre un cuadro de diálogo para para elegir el tipo de indentación: espacios en blanco o tabulaciones.
- **New Indent Width**: abre un cuadro de diálogo para seleccionar el número de espacios en blanco que componen una indentación. El valor recomendado es de 4 espacios en blanco.
- **Format Paragraph**: formatea el párrafo de texto seleccionado en columnas utilizando los espacios en blanco como separadores a la hora de crear las columnas. El número máximo de columnas es 72.
- **Strip Trailing Whitespace**: elimina los espacios en blanco al final de línea en caso de existir.

4.2.4 Menú Run

El menú *Run* nos va a permitir realizar una serie de acciones relacionadas con la ejecución y verificación del código fuente. El menú únicamente está disponible en el Editor. La siguiente imagen muestra el menú desplegado:

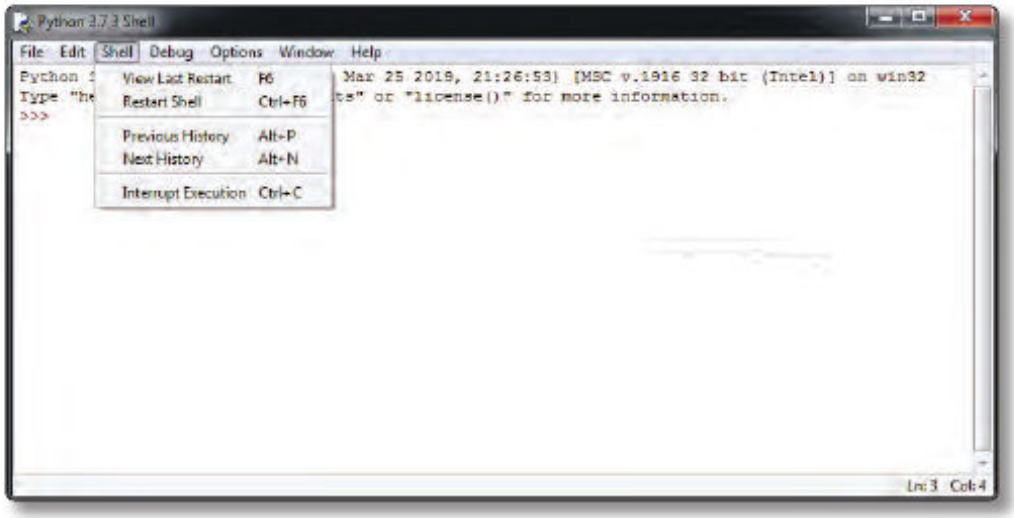


Veamos en detalle cada una de las opciones del menú:

- **Python Shell**: abre o trae al primer plano la ventana del Shell de Python.
- **Check Module**: realiza una comprobación del código fuente escrito buscando si la sintaxis es correcta. Es necesario guardar el fichero antes de ejecutar el chequeo del código fuente.
- **Run Module**: ejecuta el código fuente del fichero en el Shell de Python. Antes de ejecutar el código realiza un chequeo del código fuente (menú anterior). Es necesario guardar el fichero antes de ejecutar el fichero.

4.2.5 Menú Shell

El menú *Shell* nos va a permitir realizar una serie de funciones en la ventana del Shell de Python. Obviamente, el menú está únicamente disponible en el Shell de Python. La siguiente imagen muestra el menú desplegado:

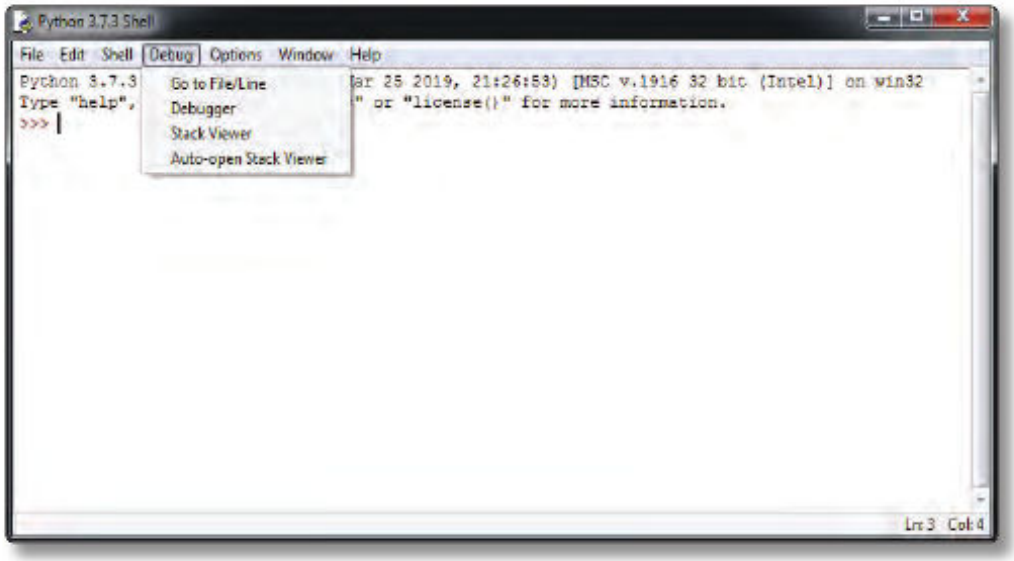


Veamos en detalle cada una de las opciones del menú:

- **View Last Restart**: mueve el Shell de Python hasta el último reinicio del mismo.
- **Restart Shell**: realiza un reinicio del Shell de Python.
- **Previous History**: busca en los comandos previamente ejecutados a la entrada actual si alguno coincide con la dicha entrada.
- **Next History**: busca en los comandos ejecutados posteriormente a la entrada actual si alguno coincide con dicha entrada.
- **Interrupt Execution**: detiene el programa en ejecución en el Shell de Python.

4.2.6 Menú Debug

El menú *Debug* nos va a permitir realizar una serie de operaciones relacionadas con la depuración de código. El menú está únicamente disponible en el Shell de Python. La siguiente imagen muestra el menú desplegado:

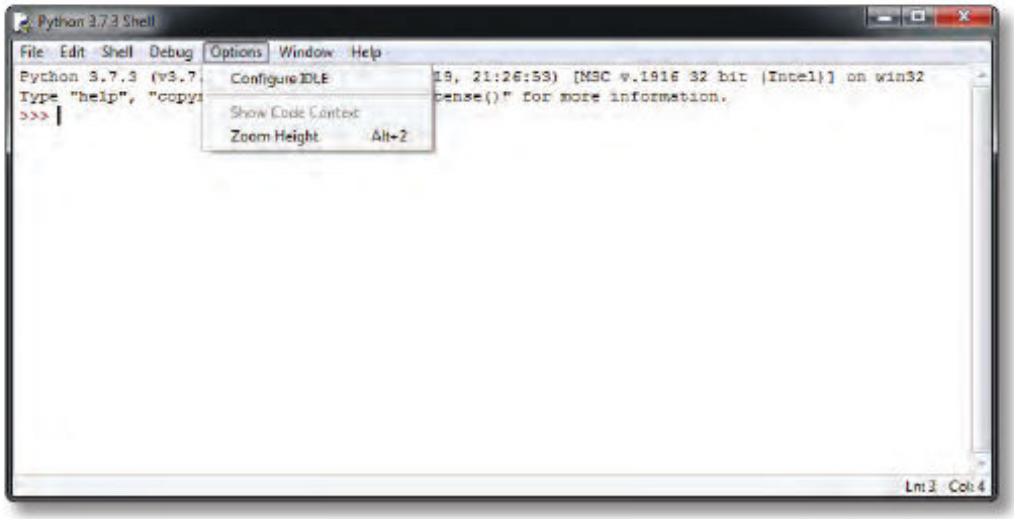


Veamos en detalle cada una de las opciones del menú:

- **Go to File/Line**: permite navegar hasta el fichero y la línea de código en la que se encuentra el cursor en caso de existir. Es muy útil para seguir trazas de código fuente.
- **Debugger**: permite activar la función de depuración del código fuente, siempre que se ejecute el código lo hará en modo depuración.
- **Stack Viewer**: permite visualizar la pila de llamadas en caso de producirse una excepción.
- **Auto-open Stack Viewer**: permite activar que se visualice siempre la pila de llamadas en caso de excepción no controlada.

4.2.7 Menú Options

El menú *Options* nos va a permitir configurar una serie de parámetros de IDLE. El menú está disponible tanto en el Shell de Python como en el Editor. La siguiente imagen muestra el menú desplegado:



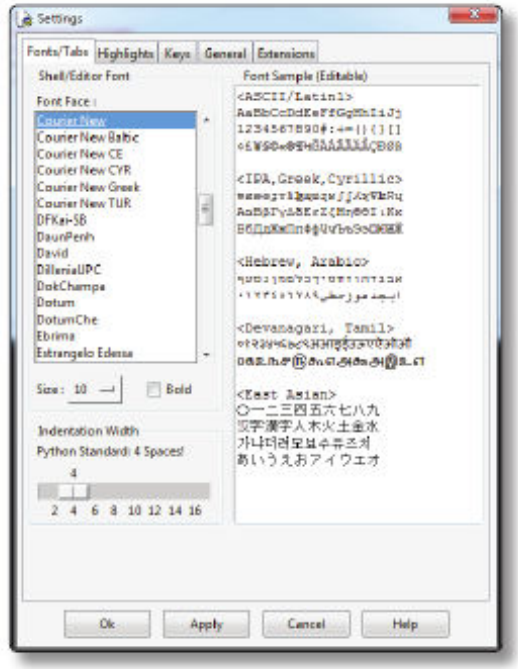
Veamos en detalle cada una de las opciones del menú:

- **Configure IDLE**: abre un cuadro de diálogo con diferentes opciones de configuración del IDLE. Las opciones las veremos en el siguiente subapartado.
- **Show Code Context**: abre en la parte superior del fichero un panel que muestra a qué bloque pertenece el código fuente; la funcionalidad es útil en aquellos casos en los que se tiene programas largos o bloques largos ya que te permite visualizar sin moverte la línea de apertura del bloque. Esta opción estará únicamente habilitada si se entra al menú *Options* desde el editor de código fuente.
- **Zoom Height**: permite aumentar el tamaño de la ventana o restaurar a su tamaño por defecto (40 líneas de 80 caracteres).

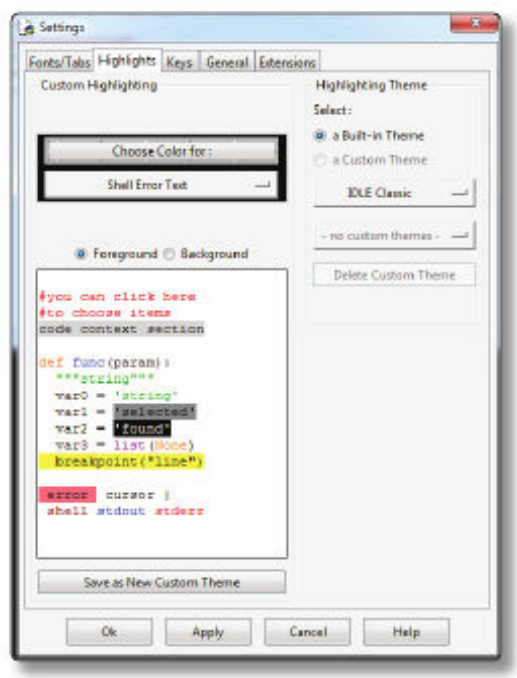
4.2.7.1 CONFIGURACIÓN IDLE

El menú de configuración de IDLE está dividido en 5 grupos según los parámetros que configuran cada uno de ellos.

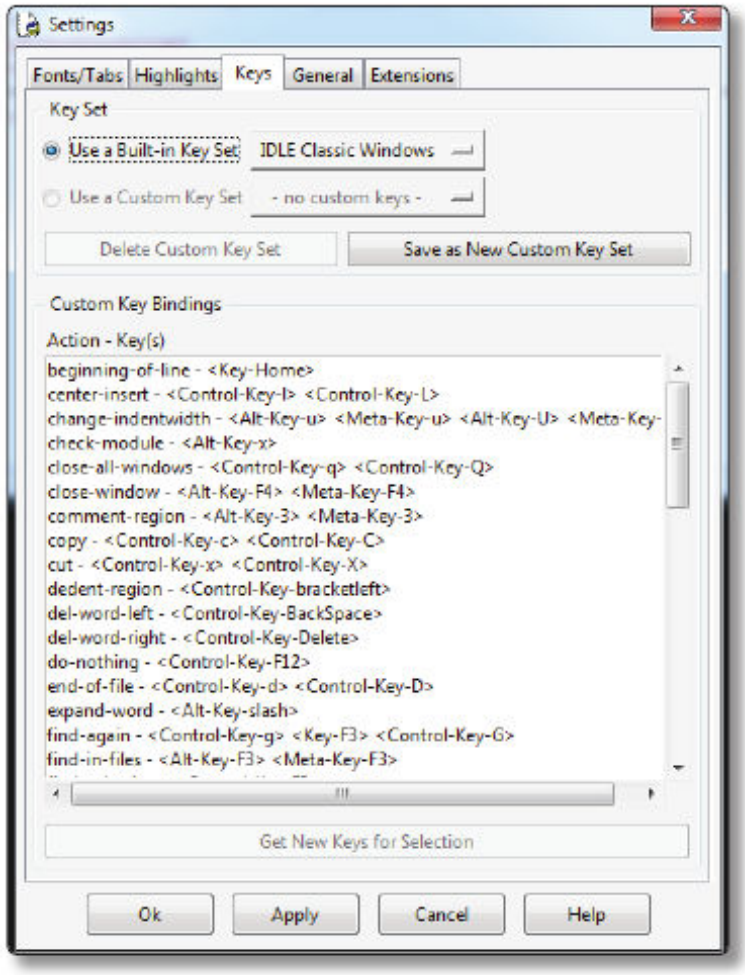
El primer grupo de opciones (*Fonts*/Tabs) que se pueden configurar se centra en la configuración del texto en el Editor. Se puede modificar el tipo de letra, el tamaño y si se quiere en negrita. También es posible establecer el número de espacios en blanco que componen una indentación. La siguiente imagen muestra una captura del menú:



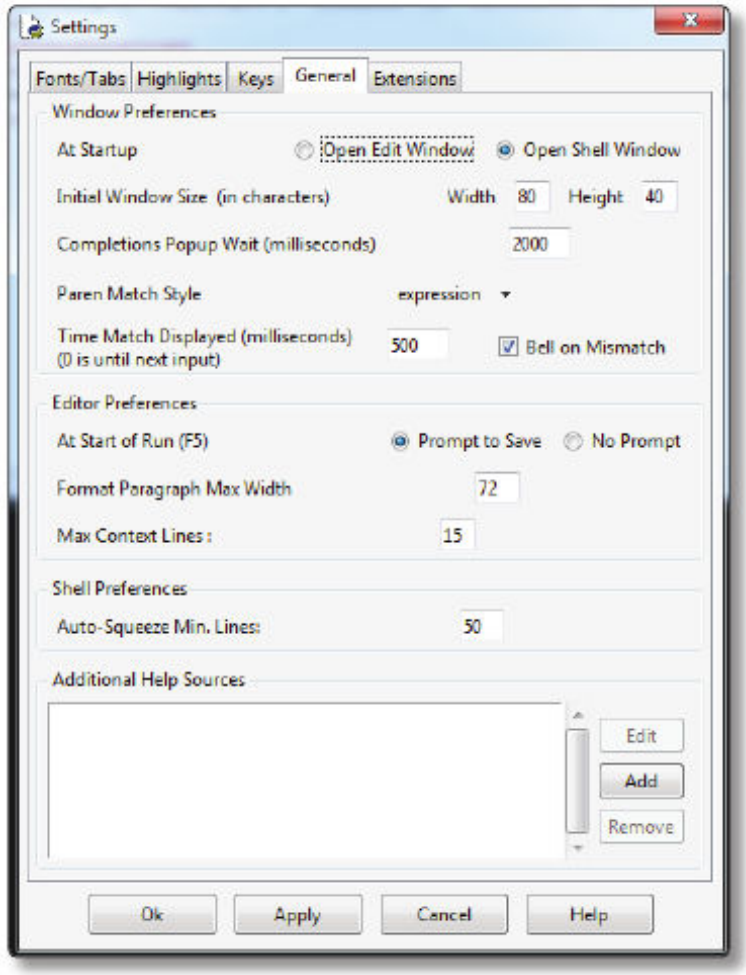
El segundo grupo de opciones (*Highlights*) se centra en la configuración de las diferentes opciones a la hora de resaltar código fuente. La siguiente imagen muestra una captura del menú:



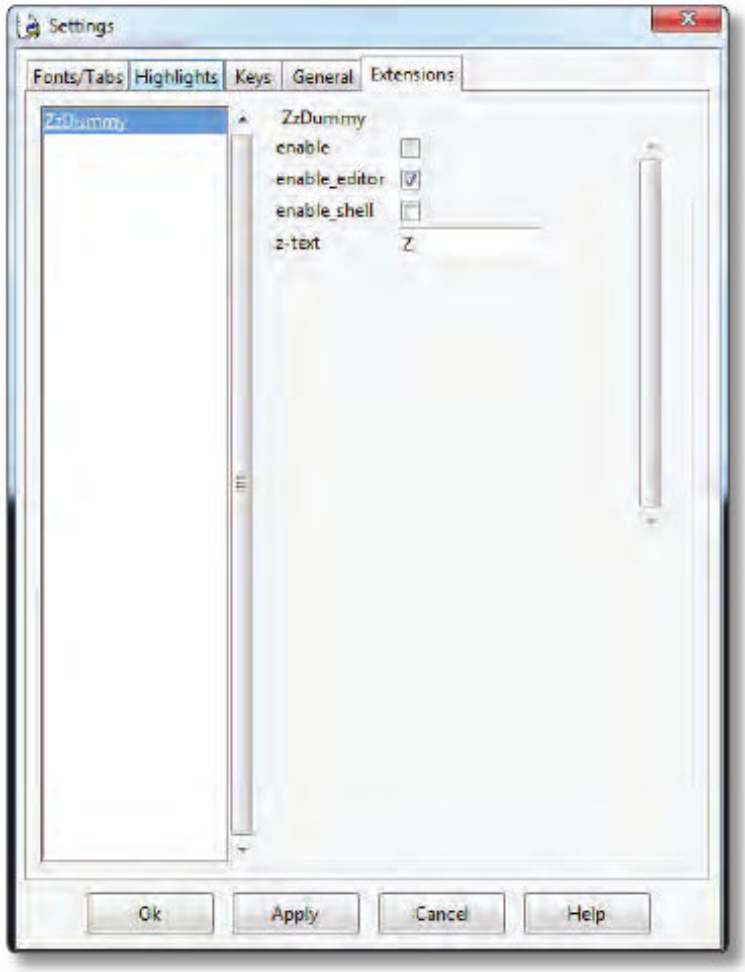
El tercer grupo de opciones (*Keys*) hace referencia a la configuración de teclas con funciones de los menús que tiene IDLE y con funciones rápidas de navegación en el Editor. La siguiente imagen muestra una captura del menú:



El cuarto grupo de opciones (*General*) agrupa una serie de opciones de carácter general dentro de IDLE. Las opciones a configurar van desde el tamaño por defecto de la ventana del Editor al número de columnas máximas por línea de código fuente. La siguiente imagen muestra una captura del menú:

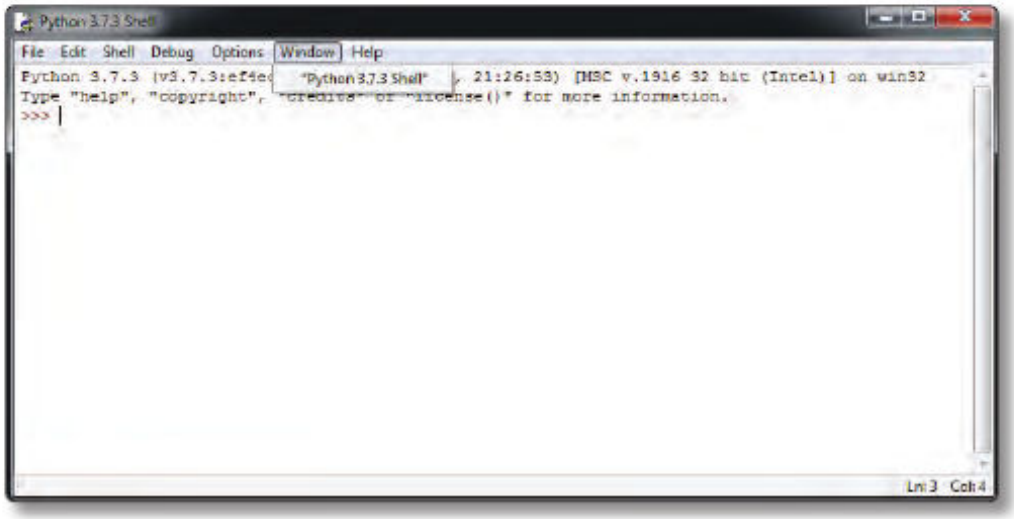


El quinto grupo de opciones (*Extensions*) permite añadir extensiones al Shell de Python y al Editor, o a uno de ellos únicamente. La siguiente imagen muestra una captura del menú:



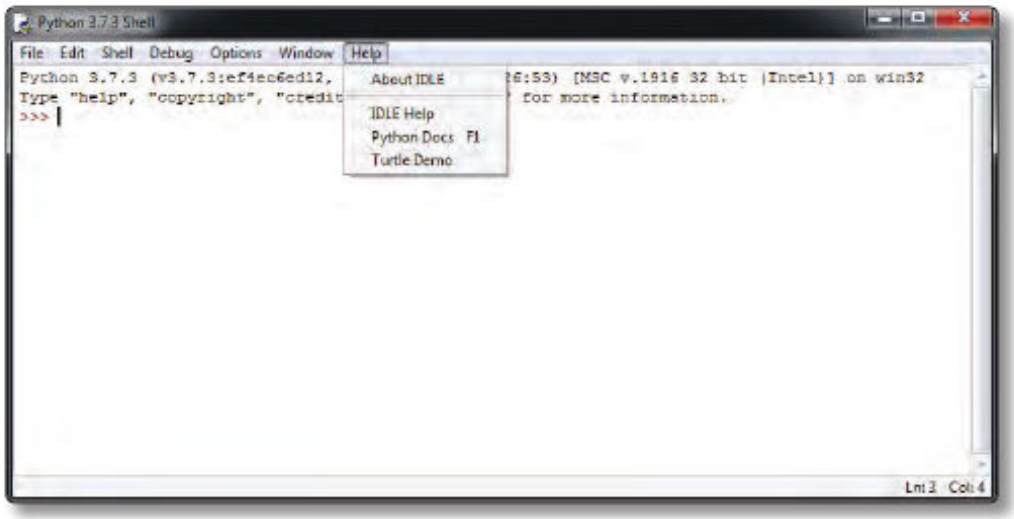
4.2.8 Menú window

El menú *Window* nos va a permitir la navegación entre todos los ficheros abiertos. En la siguiente imagen podrás ver cómo se despliega el menú, aunque en este caso únicamente se muestra el Shell de Python abierto:



4.2.9 Menú Help

El menú *Help* nos va a permitir acceder tanto a información sobre la versión de Python como a recursos documentales de IDLE y Python. La siguiente imagen muestra el menú desplegado:



Veamos en detalle cada una de las opciones del menú:

- **About IDLE**: muestra una ventana con la versión, el copyright, la licencia de uso, los créditos y mucho más.
- **IDLE Help**: accede a la ayuda de IDLE.
- **Python Docs**: accede a la ayuda de Python en caso de tenerla instalada.
- **Turtle Demo**: ejecuta la demo turtle con ejemplos de Python.

4.2.10 Colores IDLE

IDLE resalta el código fuente con diferentes colores, a medida que avanzas en los ejercicios que te planteamos irás comprobando que cada "cosa" tiene su color. Los colores utilizados y para qué se utilizan están resumidos en la siguiente lista:

- **Naranja**: palabras reservadas de Python.
- **Verde**: cadenas de texto.
- **Azul**: resultado de ejecución de una sentencia.
- **Rojo**: mensajes de error.
- **Púrpura**: funciones.

5

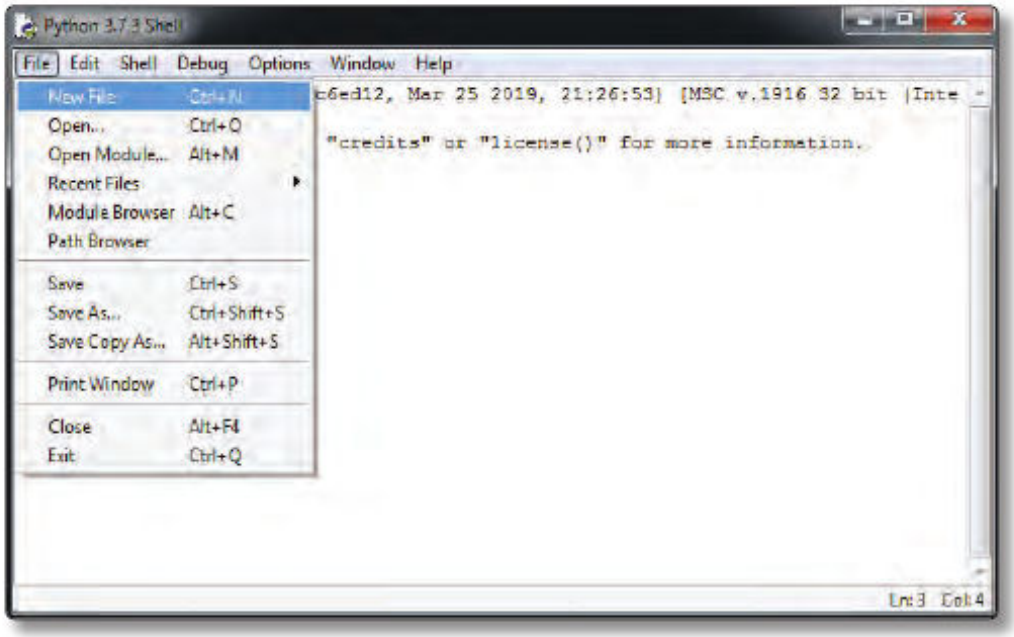
MI PRIMER PROGRAMA CON PYTHON

En este capítulo vamos a explicarte cómo realizar tu primer programa en Python paso a paso.

Es común en el mundo del desarrollo de software que el primer programa que se realiza cuando aprendes un lenguaje de programación es realizar el "*Hola Mundo*", un programa muy básico pero mediante el cual aprendes el proceso de creación y ejecución de programas. Nosotros no vamos a ser menos y vamos a hacer el "*Hola Mundo*" en Python como tu primer programa en este lenguaje.

- **PASO 1.** Abre el entorno de desarrollo IDLE. Dependiendo del sistema operativo que tengas lo podrás encontrar en el Launchpad, Inicio o ejecutarlo desde consola.

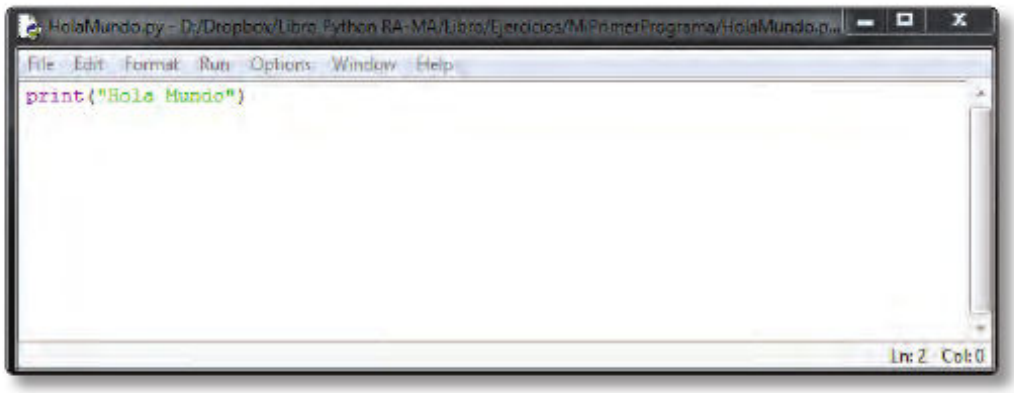
- **PASO 2.** Una vez tienes el entorno abierto lo que tienes que hacer es crear un nuevo fichero de Python en el que escribirás el programa, para ello tienes que entrar en el menú *File/New File*. Esto te abrirá una nueva pantalla en la que escribirás el código fuente de tus programas. En la siguiente imagen puedes ver dónde se encuentra dicho menú:



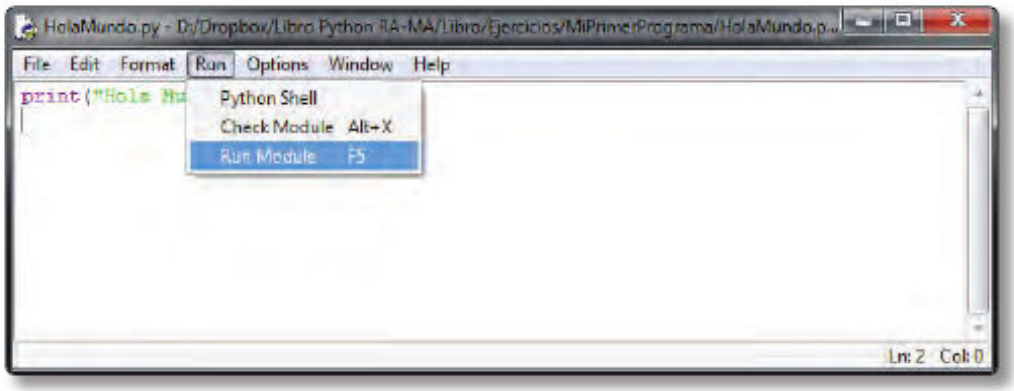
• **PASO 3.** En la nueva pantalla que se ha abierto escribe el siguiente código fuente:

print("Hola Mundo")

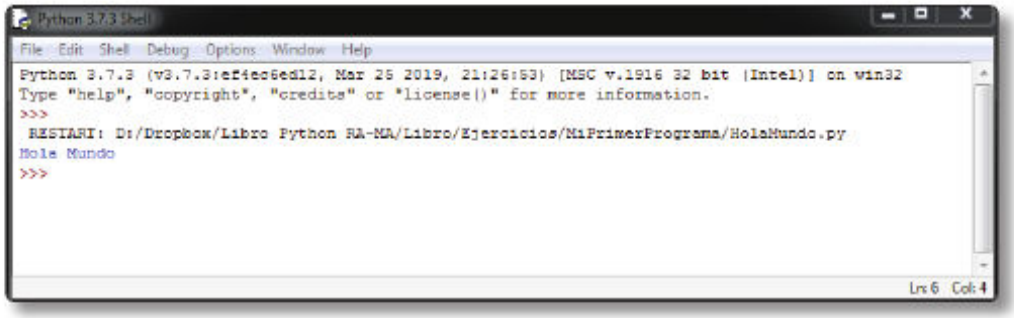
En este punto es en el que tienes que tener un acto de fe en lo que te contamos, escribe la sentencia tal y como la hemos puesto, en los próximos capítulos te explicaremos qué significa. Una vez tienes escrito el código fuente tienes que guardar el fichero. En la siguiente imagen te mostramos cómo quedaría el programa:



- **PASO 4.** Llega el momento de ejecutar tu primera aplicación escrita en Python. Para ejecutar el programa tienes que ir al menú *Run/Run Module* y el programa se ejecutará. En la siguiente imagen te mostramos dónde se encuentra el menú:



- **PASO 5.** El programa se ejecuta en la *Shell* de Python, la primera pantalla que se abrió al abrir IDLE. La siguiente imagen muestra la ejecución del programa que acabamos de realizar:



¡¡ENHORABUENA!!
¡¡Acabas de realizar tu primer programa en Python!!

Los ejercicios que veremos en el libro siguen el mismo proceso que acabamos de explicarte en este capítulo, obviamente, con una dificultad de código fuente mayor, pero no te preocupes, los iremos explicando paso a paso y están ordenados en dificultad creciente.

Para cada ejercicio deberás de hacer los siguientes pasos:

1. Crear un fichero nuevo.
2. Escribir el código fuente.
3. Guardar el fichero.
4. Ejecutar el programa.

6

SHELL / TERMINAL / CONSOLA

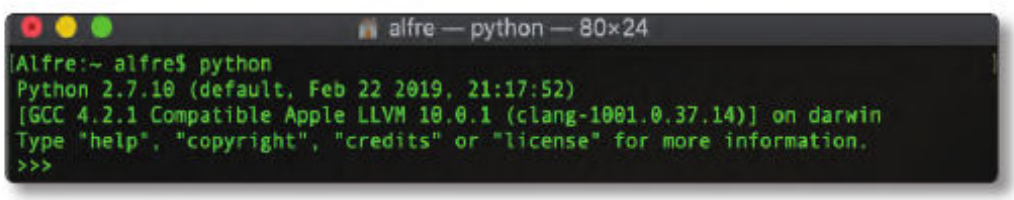
En el primer bloque del libro te hemos contado que Python es un lenguaje interpretado, es decir, no se compila para ser ejecutado, sino que el código es ejecutado línea a línea por el intérprete.

Python provee un terminal interactivo que es capaz de ejecutar sentencias de Python. Al abrir IDLE se abre el terminal integrado dentro del mismo. El terminal de Python puede abrirse sin necesidad de abrir IDLE. Los pasos para abrir el terminal de Python en todos los sistemas son los mismos:

1. Abre la consola del sistema operativo.

2. Ejecuta el comando de apertura: *Python*. En caso de tener más de una versión mayor de Python instalada tienes que especificar la versión en el comando, por ejemplo, para abrir la versión 3 tienes que usar el comando: *Python3*.

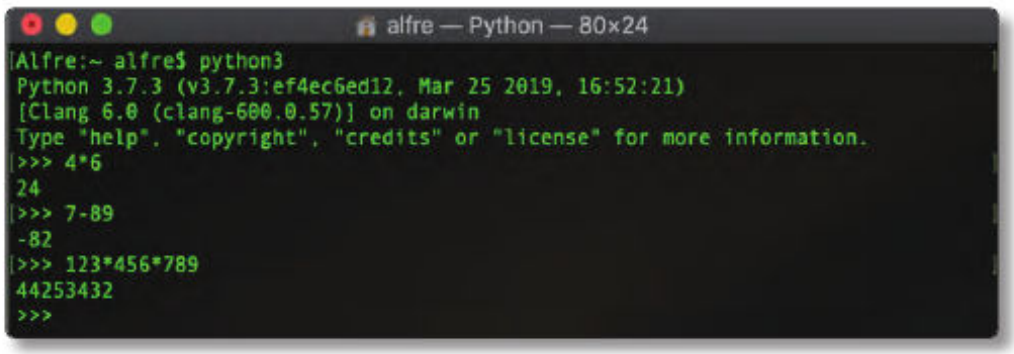
Una vez ejecutas el comando de apertura estarás dentro de la consola de Python. Veamos un ejemplo de apertura:



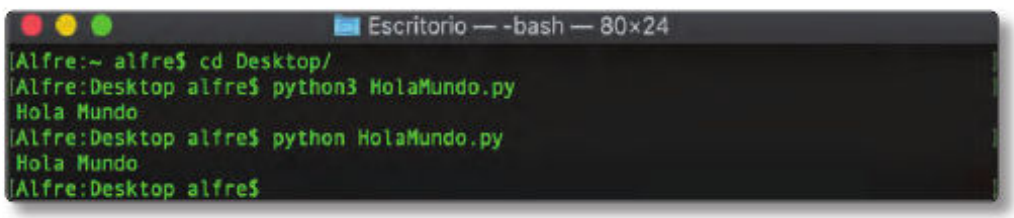
En la imagen puedes comprobar que el terminal de la versión de Python que se ha abierto es de la 2.7.10. Si quisieras abrir el terminal de Python3 tendrías

que ejecutar el comando *Python3*. En la siguiente imagen se muestra la apertura del terminal de Python3:

El terminal de Python permite ejecutar sentencias de Python directamente en el terminal, vamos a hacer una serie de operaciones aritméticas para que veas cómo se hace. La siguiente imagen muestra la ejecución de dichas operaciones aritméticas:



El terminal de Python no sólo permite ejecutar sentencias de Python, sino que también permite la ejecución de programas escritos en Python. La ejecución de programas desde el terminal se realiza de forma diferente a lo que hemos visto, se tiene que ejecutar el comando de apertura del terminal seguido de la ruta con el programa a ejecutar. En la siguiente imagen puedes comprobar cómo se ejecuta el programa que hemos escrito anteriormente ("*HolaMundo*") tanto en Python3 como en Python2:



BLOQUE 3

APRENDIZAJE PRÁCTICO

7

PROCESO DE APRENDIZAJE

El proceso de aprendizaje que hemos diseñado para este bloque de aprendizaje práctico está basado en un método progresivo diseñado para aprender de forma rápida y sencilla todos los conceptos de programación que te presentamos en cada uno de los capítulos.

Todos los capítulos siguen el mismo formato:

- **Parte 1**: introducción a todo lo que se va a tratar en el capítulo.
- **Parte 2**: presentación de los conceptos teóricos.
- **Parte 3**: para cada uno de los conceptos teóricos se realizan una serie de ejercicios para afianzar dichos conceptos. Cada ejercicio incluye lo siguiente:
  - Código fuente junto a su explicación en caso de ser necesario.
  - Imagen de una ejecución de un programa.

Los capítulos están ordenados de forma que aprendas con una metodología de aprendizaje progresivo de dificultad creciente, pero dentro de ese orden están mezclados capítulos que consideramos básicos y capítulos que consideramos de nivel avanzado. Los capítulos de nivel básico serían los siguientes:

- Variables y constantes.
- Entrada y salida estándar.
- Booleanos.
- Cadenas de texto.
- Listas, tuplas y diccionarios.
- Comentarios de código.
- Control de flujo.

- Bucles.
- Funciones.
- Módulos en Python.
- Manejo de ficheros.
- Control de excepciones.
- Programación orientada a objetos.
- Librería estándar.
- Python y las bases de datos.
- Test unitarios.

Los capítulos de nivel avanzado serían los siguientes:

- Colas y pilas.
- Recursividad.
- Programación paralela.

Nosotros te recomendamos que hagas los capítulos según te lo hemos planteado, pero puedes realizar primero los básicos y posteriormente los avanzados. Lo dejamos a tu elección.

La ejecución de los programas de todos los capítulos la hemos hecho desde IDLE, a excepción de una serie de ejercicios dentro del capítulo de programación paralela. Nosotros te recomendamos IDLE para escribir los ejercicios y para ejecutar, pero siéntete libre de utilizar cualquier entorno de desarrollo que tenga integrado el lenguaje de programación Python.

8

VARIABLES

Las variables son datos que utilizarás en todos los programas y que almacenarán información útil para su ejecución. Los datos de las variables son almacenados en la memoria del ordenador.

Las variables están compuestas por tres componentes:

- **Nombre**: identificador dentro del código fuente que utilizamos para usarlas.
- **Tipo**: tipo de dato que almacena la variable.
- **Valor**: valor que almacenan. Al declarar una variable tienes que indicarle un valor inicial, que puede verse modificado a medida que se va ejecutando el programa y según vayas necesitando, de ahí que se llamen variables.

Un ejemplo de uso de variables puede ser la necesidad de almacenar en tu programa algún tipo de información de los usuarios, por ejemplo, el teléfono. Para ello, crearías una variable con un nombre concreto y el tipo de datos que almacenarías en ella sería una cadena de texto.

Tal y como te explicamos al principio del libro, Python es un lenguaje de programación fuertemente tipado y con un tipado dinámico. Eso significa que las variables pueden cambiar de tipo de dato que almacenan durante la ejecución y que no cambiarán de forma repentina sin una conversión de tipos previa.

En Python las variables se definen utilizando las letras de la A a la Z, tanto en mayúsculas como en minúsculas, los números del 0 al 9 (excepto en el primer carácter del nombre) y utilizando el carácter "_". El lenguaje Python tiene palabras reservadas que no pueden ser utilizadas como nombres de variables. En los anexos del libro puedes encontrar la lista de palabras reservadas de Python y para qué se utiliza cada una de ellas.

Los valores son asignados a las variables con el operador asignación "=", lo que esté en la parte derecha del operador será asignado (almacenado) en la variable de la parte izquierda. Veamos unos ejemplos:

- edad = 35
- ciudad = "Alcalá de Henares"
- precio = 31,23

La primera variable, *edad*, almacenará el valor entero 35, la segunda variable, *ciudad*, almacenará la cadena de texto "Alcalá de Henares" y por último la tercera variable, "*precio*", almacenará el número real 31,23.

Tal y como puedes observar, en Python no se establece el tipo de dato a la hora de declarar una variable, realmente no existe una declaración propiamente dicha como existe en otros lenguajes, simplemente escribes el nombre que quieres que tenga y le asignas el valor correspondiente.

8.1 TIPOS DE DATOS

En informática, la información no es otra cosa que una secuencia de ceros y unos que se estructuran en bloques para facilitar el manejo de ésta.

Por lo tanto, toda la información que existe se encuentra tipificada por el tipo de información que es (tipo de dato). No es lo mismo una cadena de texto que un número entero, o decimal.

Los tipos de datos existentes en Python son los siguientes:

Tipo de dato	Descripción
Entero	Número sin decimales, tanto positivo como negativo, incluyendo el 0.
Real	Número con decimales, tanto positivo como negativo, incluyendo el 0.
Complejo	Número con parte imaginaria.
Cadenas de texto	Texto.
Booleanos	Pueden tener dos valores: True o False
Lista	Vector de elementos que pueden ser de diferentes tipos de datos.
Tuplas	Lista inmutable de elementos.
Diccionario	Lista de elementos que contienen claves y valores.

Tal y como hemos comentado desde el principio, las variables no tienen un tipo concreto en Python, puedes utilizar una variable para almacenar un número en una parte de tu programa y posteriormente puedes utilizarla para almacenar una lista de elementos.

9

ENTRADA Y SALIDA ESTÁNDAR

En este capítulo se va a explicar la interacción con el terminal para mostrar información por pantalla y para leer información introducida mediante el teclado. Para Python, la entrada estándar de información es el teclado y la salida estándar es el terminal de texto o consola.

La interactividad de las aplicaciones software está ligada al intercambio de información entre el aplicativo y el usuario de la misma, es por ello que estas dos operaciones que vas a aprender en este capítulo son de mucha utilidad dentro del desarrollo de software.

9.1 SALIDA POR PANTALLA

En este apartado vamos a explicarte la operación para mostrar información por pantalla a través del terminal, lo que te permitirá mostrar información al usuario de la misma.

Python dispone del comando *print* para mostrar información por pantalla a través del terminal. El comando sigue la siguiente estructura:

print(TextoAMostrar)

Veámoslo con un ejercicio. El código fuente es el siguiente:

```
print("Hola Python")
```

La ejecución del código fuente mostrará el mensaje *Hola Python* en el terminal:

```
Hola Python
```

El comando *print* tiene diferentes formas de usarse, una de ellas es que permite añadir varias cadenas de texto separadas por comas y mostrar un mensaje completo en una única instrucción, lo que simplifica el no tener que escribir una sentencia por cada mensaje que queramos mostrar por pantalla de forma secuencial. Por defecto, se introduce siempre un espacio en blanco como carácter de separación entre las diferentes cadenas.

En el siguiente ejercicio vamos a añadir diferentes cadenas al comando, el código fuente es el siguiente:

```
print("Estoy","aprendiendo","Python")
```

La ejecución del código fuente mostrará el mensaje en el terminal separando cada cadena entre si mediante el espacio en blanco:

```
Estoy aprendiendo Python
```

Hasta ahora, los ejercicios han mostrado cadenas de texto por pantalla, pero es posible añadir diferentes tipos de datos al comando. En el siguiente ejercicio vamos a mostrar por pantalla tres números enteros. El código fuente sería el siguiente:

```
print(1,2,3)
```

La ejecución del código fuente mostrará en el terminal los números separados por espacio en blanco:

```
1 2 3
```

A la hora de indicarle a *print* la información que tiene que mostrar por pantalla, es posible indicarle información que sea de diferentes tipos de datos. En el siguiente ejercicio vamos a mostrar una ejecución de *print* que muestra una cadena de texto y un entero en la misma instrucción. El código fuente es el siguiente:

```
print("Hola Python", 2019)
```

La ejecución del código fuente mostrará la información en el terminal añadiendo el espacio en blanco por defecto de separación entre ambos parámetros:

```
Hola Python 2019
```

El comando *print* permite mostrar estructuras de datos más complejas, pero serán abordadas en los siguientes capítulos del libro.

El comando *print* permite mostrar el valor de variables además de valores directos. Veámoslo en un ejercicio, el código fuente sería el siguiente:

```
texto = "Hola Python"
anio = 2019
print(texto,anio)
```

El ejercicio muestra la misma información que el ejercicio anterior, pero pasándole la información a mostrar a *print* mediante variables en vez de hacerlo de forma directa. La ejecución del código fuente tendrá la siguiente salida:

```
Hola Python 2019
```

9.1.1 Formateando la salida

El comando *print* dispone de dos parámetros que nos van a permitir darle formato al mensaje que se mostrará por pantalla. Los parámetros son los siguientes:

- **sep**: permite modificar el carácter por defecto de espacio en blanco que se introduce de forma automática como separador entre las diferentes cadenas que se muestran.
- **end**: permite añadir una cadena de texto como elemento final del conjunto de cadenas de texto que se han enviado para mostrar por pantalla.

El siguiente ejercicio consiste en modificar el espacio en blanco por defecto entre cadenas por una coma, mostrando dos mensajes por pantalla, primero sin usar el parámetro *sep* y posteriormente usándolo. El código fuente es el siguiente:

```
print(1,2,3)
print(1,2,3, sep=',')
```

La ejecución del código fuente tendrá la siguiente salida:

```
1 2 3
1,2,3
```

El siguiente ejercicio consiste en la utilización del parámetro *end* para establecer un carácter de terminación del mensaje que se muestra, en este caso añadiremos un punto al ejemplo del ejercicio anterior. El código fuente es el siguiente:

```
print(1,2,3)
print(1,2,3, sep=',', end='.')
```

La ejecución del código fuente tendrá la siguiente salida:

```
1 2 3
1,2,3.
```

9.1.2 Caracteres especiales

Existe un conjunto de caracteres que requieren una forma concreta a la hora de ser mostrados por pantalla, bien sea porque aportan funciones dentro de las cadenas de texto (tales como salto de línea o tabulador) o porque pertenecen a la sintaxis de Python (tales como comillas simples o dobles). Para poder utilizar estos caracteres es necesario utilizar un carácter dentro de la cadena para indicar que se trata de un carácter especial, dicho carácter se llama carácter de escape y en Python es una barra invertida \. La siguiente tabla muestra los diferentes caracteres especiales soportados en Python:

Carácter	¿Qué se muestra?
\n	Retorno de línea
\t	Tabulador
\\	Diagonal invertida
\"	Comilla doble
\'	Comilla simple
\xNN	Carácter hexadecimal NN en ASCII
\uNN	Carácter hexadecimal NN en Unicode

El siguiente ejercicio muestra un ejemplo de uso de algunos de los caracteres especiales vistos. El código fuente es el siguiente:

```
print("Primera línea: \"Texto entre comillas\"\nSegunda línea: \'\x24\'")
```

La ejecución del código fuente tendrá la siguiente salida:

```
Primera línea: "Texto entre comillas"
Segunda línea: '$'
```

9.2 ENTRADA DESDE TECLADO

En este apartado vamos a explicarte la operación de lectura de información introducida por teclado por el usuario de la aplicación.

Python dispone del comando *input* para leer información introducida por los usuarios de la aplicación mediante el teclado. Por defecto, la información introducida por el usuario mediante el teclado es retornada a la aplicación en formato de cadena de texto, aunque como veremos en los siguientes capítulos es posible transformarla al tipo de dato que necesitemos. En todos los casos, para leer la información, es necesario que el usuario de la aplicación presione la tecla *Enter*.

Veamos cómo funciona con un ejercicio en el que vamos a solicitar al usuario de la aplicación su nombre y sus apellidos mediante los comandos *print* e *input*. La información introducida por el usuario la vamos a almacenar en variables para mostrarla por pantalla antes de terminar la ejecución de la aplicación. El código fuente es el siguiente:

```
print("Introduzca su nombre:")
nombre = input()
print("Introduzca sus apellidos:")
apellidos = input()
print("Hola",nombre,apellidos)
```

La ejecución del código fuente tendrá la siguiente salida:

```
Introduzca su nombre:
Alfredo
Introduzca sus apellidos:
Moreno Muñoz
Hola Alfredo Moreno Muñoz
```

El ejercicio que acabamos de hacer puede realizarse de una forma más sencilla, ya que el comando *input* permite indicarle una cadena de texto para mostrar como mensaje de petición de información al usuario de la aplicación, es decir, el

texto será mostrado y posteriormente la aplicación se quedará esperando a que el usuario introduzca el texto. El código fuente simplificado sería el siguiente:

```
nombre = input("Introduzca su nombre:")
apellidos = input("Introduzca sus apellidos:")
print("Hola",nombre,apellidos)
```

La ejecución del código fuente tendrá la siguiente salida:

```
Introduzca su nombre:Sheila
Introduzca sus apellidos:Córcoles Córcoles
Hola Sheila Córcoles Córcoles
```

10

TIPOS DE DATOS NUMÉRICOS

En este capítulo vamos a explicarte los tipos de datos más básicos que existen, los números.

Los tipos de datos en Python relacionados con los números son tres:

- **Entero**: número entero con límite de valor.
- **Reales**: número compuesto por parte entera y parte decimal.
- **Complejo**: número compuesto por parte real y parte imaginaria.

Antes de entrar en detalle en cada uno de ellos, vamos a explicarte en el siguiente apartado los operadores aritméticos.

10.1 OPERADORES ARITMÉTICOS

Los operadores aritméticos son aquellos operadores que nos van a permitir realizar operaciones aritméticas con los datos. Python dispone de los siguientes operadores aritméticos:

- **Suma**: el operador "+" realiza la suma de dos valores de tipo numérico.
- **Resta**: el operador "-" realiza la resta de dos valores de tipo numérico.
- **Multiplicación**: el operador "*" realiza la multiplicación de dos valores numéricos.
- **División**: el operador "/" realiza la división de dos valores numéricos.

- **División entera**: el operador "//" realiza la división entera de dos valores numéricos.
- **Módulo**: el operador "%" realiza el módulo de dos valores numéricos, es decir, obtiene el resto de la división.
- **Exponente**: el operador "**" realiza el exponente de dos valores numéricos.
- **Negación**: el operador "-" asigna un valor negativo a un valor numérico.

Los operadores aritméticos tienen un orden de precedencia a la hora de ejecutar las operaciones, es el siguiente:

1. Se ejecuta la operación de exponente.
2. Se ejecuta la operación de negación.
3. Se ejecutan las operaciones de Multiplicación, División, División entera y Módulo.
4. Se ejecutan las sumas y restas.

Te recomendamos que a la hora de utilizar operadores aritméticos utilices paréntesis para establecer el orden concreto de resolución de las operaciones, así no tendrás que preocuparte por el orden de precedencia. La utilización de los paréntesis es una buena práctica ya que cada lenguaje de programación establece la resolución de forma diferente y puedes encontrar diferencias dependiendo del lenguaje de programación que utilices.

En los ejercicios dentro de los siguiente apartados practicaremos los diferentes operadores aritméticos disponibles en Python.

10.2 NÚMEROS ENTEROS

Los números enteros son aquellos números que no tienen decimales y que pueden ser positivos y negativos, incluyendo al cero. Los números enteros se representan en Python como *int*. En entornos de 32 bits pueden almacenar valores entre -2^{31} a $2^{31}-1$, o lo que es lo mismo, entre -2.147.483.648 y 2.147.483.647. En entornos de 64 bits pueden almacenar valores entre -2^{63} a $2^{63}-1$, o lo que es lo mismo, entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807.

En el capítulo anterior te explicamos que la lectura de la entrada estándar siempre es una cadena de texto, pero que es posible transformarla al tipo de dato

que necesitemos. En los ejercicios de este apartado vamos a transformar el valor de entrada en un entero de la siguiente forma:

```
int(input("texto"))
```

Mediante *int* lo que hacemos es transformar la cadena de texto leída a un valor entero.

Los ejercicios que vamos a realizar en este apartado siguen el mismo formato:

1. Petición del primer número entero.
2. Petición del segundo número entero.
3. Operación matemática utilizando los dos números.

En los ejercicios introduce únicamente valores enteros positivos o negativos, no introduzcas números con decimales, esos los veremos en el siguiente apartado. En caso de introducir un dato que no sea entero obtendrás un error en la consola, hay un capítulo más adelante en el que te enseñaremos a cómo controlar esos errores.

El primer ejercicio consiste en la realización de una suma con los dos valores leídos, el código fuente es el siguiente:

```
numero1 = int(input("Primer sumando: "))
numero2 = int(input("Segundo sumando: "))
print("Resultado: ", numero1 + numero2)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Primer sumando: 546
Segundo sumando: 875
Resultado:  1421
```

El segundo ejercicio consiste en la realización de una división entera con los dos valores leídos, el código fuente es el siguiente:

```
dividendo = int(input("Dividendo: "))
divisor = int(input("Divisor: "))
print("Resultado: ", dividendo // divisor)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Dividendo: 67
Divisor: 4
Resultado:  16
```

El tercer ejercicio consiste en la realización de un exponente con los dos valores leídos. El código fuente es el siguiente:

```
base = int(input("Base: "))
exponente = int(input("Exponente: "))
print("Resultado: ", base ** exponente)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Base: 2
Exponente: 8
Resultado:  256
```

10.2.1 Números enteros long

Los números enteros long son aquellos números que no tienen decimales y que pueden ser positivos y negativos, incluyendo al cero, y que se diferencian con los enteros normales en que el rango de valores es mucho mayor. Los números enteros long se representan en Python como *long*. Un número entero long puede contener valores tan grandes como permita la memoria de la máquina, eso si, por defecto, si el valor no llega a ser superior a un entero normal el valor será un entero normal, es decir, únicamente se utilizarán los enteros long en aquellos casos en los que el número no pueda almacenarse en un entero normal.

Los enteros long fueron suprimidos en Python 3, por lo que es un tipo de dato por el cual no preocuparse.

10.3 NÚMEROS REALES

Los números reales son aquellos números que tienen decimales. En Python se representan como *float* y la separación entre la parte entera y la parte decimal se realiza con un punto.

En el capítulo anterior te explicamos que la lectura de la entrada estándar siempre es una cadena de texto, pero que es posible transformarla al tipo de dato que necesitemos. En los ejercicios de este apartado vamos a transformar el valor de entrada en un número real de la siguiente forma:

```
float(input("texto"))
```

Mediante *float* lo que hacemos es transformar la cadena de texto leída a un valor real (decimal).

Los ejercicios que vamos a realizar en este apartado siguen el mismo formato:

1. Petición del primer número real.
2. Petición del segundo número real.
3. Operación matemática utilizando los dos números.

En los ejercicios puedes introducir números reales o enteros, en caso de introducir un dato que no sea real o entero obtendrás un error en la consola, hay un capítulo más adelante en el que te enseñaremos a cómo controlar esos errores.

El primer ejercicio consiste en la realización de una resta utilizando los dos números reales leídos, el código fuente es el siguiente:

```
minuendo = float(input("Minuendo: "))
sustraendo = float(input("Sustraendo: "))
print("Resultado: ", minuendo - sustraendo)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Minuendo: 457.12
Sustraendo: 98.2
Resultado:  358.92
```

El segundo ejercicio consiste en la realización de una división utilizando los dos números reales leídos, el código fuente es el siguiente:

```
dividendo = float(input("Dividendo: "))
divisor = float(input("Divisor: "))
print("Resultado: ", dividendo / divisor)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Dividendo: 59.34
Divisor: 22.5
Resultado:  2.6373333333333333
```

El tercer ejercicio consiste en la realización de una multiplicación utilizando los dos números reales leídos, el código fuente es el siguiente:

```
multiplicando = float(input("Multiplicando: "))
multiplicador = float(input("Multiplicador: "))
print("Resultado: ", multiplicando * multiplicador)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Multiplicando: 12.4
Multiplicador: 67.8
Resultado:  840.72
```

10.3.1 Redondeo de números reales

Los números reales tienen un número infinito de decimales, pero Python dispone de una instrucción que permite acortar el número de decimales, se trata de la instrucción *round*.

La instrucción *round* utiliza dos parámetros para ejecutarse:

- Número real a redondear.
- Número de decimales a los que se quiere redondear el número.

El siguiente ejercicio realiza la petición de dos números para realizar una multiplicación y redondear el resultado a un único decimal. El código fuente es el siguiente:

```
multiplicando = float(input("Multiplicando: "))
multiplicador = float(input("Multiplicador: "))
resultado = round(multiplicando * multiplicador,1)
print("Resultado de la multiplicación: ", resultado)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Multiplicando: 98.566
Multiplicador: 455.33
Resultado de la multiplicación:  44880.1
```

10.4 NÚMEROS COMPLEJOS

Los números complejos son aquellos números que están compuestos por una parte real y una parte imaginaría, y en el que cada una de las partes es un número decimal (los vistos en el apartado anterior). Los números complejos se representan en Python como *complex*.

En el capítulo anterior te explicamos que la lectura de la entrada estándar siempre es una cadena de texto, pero que es posible transformarla al tipo de dato que necesitemos. En los ejercicios de este apartado vamos a transformar el valor de entrada en un número complejo de la siguiente forma:

```
complex(float(input("ParteReal")),float(input("ParteImaginaria")))
```

Mediante *complex* lo que hacemos es transformar las dos cadenas de texto leídas en un número complejo. Tal y como puedes observar, ambas cadenas leídas son transformadas en números reales, ya que tanto la parte real como imaginaria de un número complejo está compuesta por un número real (decimal). La primera lectura hace referencia a la parte real del número complejo, y la segunda hace referencia a la parte imaginaria del número complejo.

En los ejercicios puedes introducir números reales o enteros, en caso de introducir un dato que no sea real o entero obtendrás un error en la consola, hay un capítulo más adelante en el que te enseñaremos a cómo controlar esos errores.

El primer ejercicio consiste únicamente en leer un número real y mostrarlo por pantalla, así puedes observar cómo se representan los números complejos en Python. El código fuente es el siguiente:

```
numero = complex(float(input("Parte real: ")),float(input("Parte imaginaria:")))
print(numero)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Parte real: 2.3
Parte imaginaria:2.6
(2.3+2.6j)
```

El segundo ejercicio sigue el formato de ejercicios de los apartados anteriores, se pide al usuario que introduzca dos números complejos y después se realiza una suma de ambos. El código fuente es el siguiente:

```
sumando1 = complex(float(input("Parte real sumando1: ")),float(input("Parte imagi-
naria sumando 1:")))
sumando2 = complex(float(input("Parte real sumando2: ")),float(input("Parte imagi-
naria sumando 2:")))
print("Resultado:", sumando1 + sumando2)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Parte real sumando1: 67.5
Parte imaginaria sumando 1:4
Parte real sumando2: 34.98
Parte imaginaria sumando 2:6.7
Resultado: (102.47999999999999+10.7j)
```

10.5 USO DE PARÉNTESIS

Tal y como hemos explicado al comienzo del capítulo, en Python el orden de ejecución de las operaciones depende de la propia operación que se ejecuta. Para evitar problemas con el orden o confusiones te aconsejamos que utilices paréntesis para establecer el orden de ejecución de las mismas de forma manual.

El siguiente ejercicio muestra un ejemplo de uso de paréntesis en una operación matemática compleja.

```
numero1 = float(input("Primer número: "))
numero2 = float(input("Segundo número: "))
numero3 = float(input("Tercer número: "))
numero4 = float(input("Cuarto número: "))
numero5 = float(input("Quinto número: "))
numero6 = float(input("Sexto número: "))
```

```
resultado = (numero5 + (numero3*(numero1**numero6)))-(numero4//numero2)
print("Resultado: ", resultado)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Primer número: 5
Segundo número: 3
Tercer número: 7
Cuarto número: 5
Quinto número: 8
Sexto número: 2
Resultado:  182.0
```

Veamos cómo ha resuelto el ejemplo.

La operación a ejecutar sustituyendo los valores introducidos en el ejemplo es la siguiente: (8+(7*(5^2)))-(5//3)

1. Primera operación: 5^2
   - *Operación pendiente a resolver:* (8+(7*25))-(5//3)
2. Segunda operación: 7*25
   - *Operación pendiente a resolver:* (8+175)-(5//3)
3. Tercera operación: 8+175
   - *Operación pendiente a resolver:* 183-(5//3)
4. Cuarta operación: 5//3
   - *Operación pendiente a resolver:* 183-1
5. Quinta operación: 183-1
   - *Resultado final:* 182

Puedes comprobar que si eliminas los paréntesis de la operación el resultado es el mismo, pero es cierto que queda más clara la operación a ejecutar si utilizas los paréntesis a si no los utilizas, ganando legibilidad, comprensión y mantenimiento.

11

BOOLEANOS

En este capítulo vamos a explicarte el tipo de datos booleano, junto con los operadores lógicos y relacionales.

Los booleanos son también conocidos como los tipos de datos lógicos y se caracterizan porque únicamente pueden tener dos valores, o *True* o *False*.

Veamos en un ejercicio cómo se asignan valores booleanos a variables y cómo se muestran por pantalla. El código fuente siguiente creará dos variables, una con un valor *True* y otra con un valor *False*, y posteriormente las mostrará por pantalla:

```
verdadero = True
falso = False
print("Valor de la variable verdadero:",verdadero)
print("Valor de la variable falso:",falso)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Valor de la variable verdadero: True
Valor de la variable falso: False
```

11.1 OPERADORES LÓGICOS

Los operadores lógicos son operadores que permites construir expresiones lógicas y que tienen como resultado un valor booleano, *True* o *False*. Los operadores lógicos son operaciones que pueden realizarse sobre variables de tipo booleano, bien sean valores independientes o valores provenientes de otras operaciones.

Los operadores lógicos que puedes utilizar en Python son los siguientes:

- **AND**: operador lógico que realiza la operación lógica 'Y' entre dos elementos. El resultado será *true* si ambos elementos son *true*, en caso contrario será *false*.
- **OR**: operador lógico que realiza la operación lógica 'O' entre dos elementos. El resultado será *true* si uno de los dos elementos es *true*, en caso contrario será *false*.
- **NOT**: operador lógico que realiza la operación lógica 'NO'. El resultado será *true* si el elemento es *false*, y será *false* si es *true*.

Al igual que te explicamos en el capítulo anterior en los operadores aritméticos, es recomendable utilizar paréntesis a la hora de construir operaciones lógicas complejas.

Veamos en detalle cada uno de los operadores lógicos disponibles en Python.

El operador lógico **AND** permite realizar la operación lógica 'Y' entre dos valores booleanos. El formato de la instrucción es el siguiente:

ValorBooleano1 and ValorBooleano2

La operación lógica 'Y' obtendrá el siguiente resultado dependiendo del valor de ambos parámetros:

ValorBooleano1	ValorBooleano2	Resultado
True	True	True
True	False	False
False	True	False
False	False	False

En el siguiente ejercicio puedes comprobar cómo funciona la operación lógica **AND** llevando a código la tabla anterior:

```
print("Resultado de True AND True:", True and True)
print("Resultado de True AND False:", True and False)
print("Resultado de False AND True:", False and True)
print("Resultado de False AND False:", False and False)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Resultado de True AND True: True
Resultado de True AND False: False
Resultado de False AND True: False
Resultado de False AND False: False
```

El operador lógico **OR** permite realizar la operación lógica 'O' entre dos valores booleanos. El formato de la instrucción es el siguiente:

ValorBooleano1 or ValorBooleano2

La operación lógica 'O' obtendrá el siguiente resultado dependiendo del valor de ambos parámetros:

ValorBooleano1	ValorBooleano2	Resultado
True	True	True
True	False	True
False	True	True
False	False	False

En el siguiente ejercicio puedes comprobar cómo funciona la operación lógica **OR** llevando a código la tabla anterior:

```
print("Resultado de True OR True:", True or True)
print("Resultado de True OR False:", True or False)
print("Resultado de False OR True:", False or True)
print("Resultado de False OR False:", False or False)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Resultado de True OR True: True
Resultado de True OR False: True
Resultado de False OR True: True
Resultado de False OR False: False
```

El operador lógico **NOT** permite realizar la operación lógica 'NO' sobre un valor booleano. El formato de la instrucción es el siguiente:

not ValorBooleano

La operación lógica 'NO' obtendrá el siguiente resultado dependiendo del valor de ambos parámetros:

ValorBooleano	Resultado
True	False
False	True

En el siguiente ejercicio puedes comprobar cómo funciona la operación lógica **NOT** llevando a código la tabla anterior:

```
print("Resultado de NOT True:", not True)
print("Resultado de NOT False:", not False)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Resultado de NOT True: False
Resultado de NOT False: True
```

El último ejercicio que vamos a realizar referente a los operadores lógico es uno en el que vamos a combinar diferentes operaciones lógicas en una misma instrucción utilizando paréntesis. A su vez, vamos a utilizar la instrucción *bool* para convertir el valor introducido por el usuario de la aplicación a booleano. La instrucción sigue el siguiente formato:

bool(input("texto"))

El código fuente del ejercicio es el siguiente:

```
booleano1 = bool(input("Primer valor:"))
booleano2 = bool(input("Segundo valor:"))
booleano3 = bool(input("Tercer valor:"))
booleano4 = bool(input("Cuarto valor:"))
booleano5 = bool(input("Quinto valor:"))
print("Resultado:", booleano4 or ((booleano3 and not booleano2) and booleano1)
or booleano5)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior, tal y como puedes comprobar se pueden introducir valores 0 y 1 como

valores booleanos significando *false* y *true* respectivamente, al igual que puedes introducir valores *false* o *true*:

```
Primer valor:true
Segundo valor:false
Tercer valor:0
Cuarto valor:1
Quinto valor:true
Resultado: True
```

Veamos cómo ha resuelto el ejemplo.

La operación a ejecutar sustituyendo los valores introducidos en el ejemplo es la siguiente: true or ((false and not false) and true) or true

1. Primera operación: not false
   - *Operación pendiente a resolver:* true or ((false and true) and true) or true
2. Segunda operación: false and true
   - *Operación pendiente a resolver:* true or (false and true) or true
3. Tercera operación: false and true
   - *Operación pendiente a resolver:* true or false or true
4. Cuarta operación: true or false
   - *Operación pendiente a resolver:* true or true
5. Quinta operación: true or true
   - *Resultado final:* true

11.2 OPERADORES RELACIONALES

Los operadores relacionales son operadores que permiten comparar dos valores entre y cuyo resultado es un valor booleano, *True* o *False*. Los operadores relacionales son operaciones de comparación que pueden realizarse sobre valores que sean valores independientes o valores provenientes de otras operaciones.

El formato de utilización de los operadores relacionales es el siguiente:

Valor1 Operador Valor2

Los operadores relacionales que puedes utilizar en Python son los siguientes:

Operador	Significado
==	Valor1 y Valor2 son iguales
!=	Valor1 y Valor2 son diferentes
>	Valor1 es mayor que Valor2
<	Valor1 es menor que Valor2
>=	Valor1 es mayor o igual que Valor2
<=	Valor1 es menor o igual que Valor2

El resultado de las comparaciones es un booleano que será *True* en caso de que la comparación sea cierta o será *false* en caso de que la comparación no sea cierta.

En el siguiente ejercicio vamos a solicitar una serie de números (reales o enteros) al usuario y vamos a realizar una comparación para cada uno de los operadores relacionales disponibles en Python. El código fuente es el siguiente:

```
numero1 = float(input("Primer número:"))
numero2 = float(input("Segundo número:"))
numero3 = float(input("Tercer número:"))
numero4 = float(input("Cuarto número:"))
print(numero1,"==",numero4,":",numero1==numero4)
print(numero2,"!=",numero3,":",numero2!=numero3)
print(numero3,">",numero2,":",numero3>numero2)
print(numero4,"<",numero1,":",numero4<numero1)
print(numero1,">=",numero3,":",numero1>=numero3)
print(numero2,"<=",numero4,":",numero2<=numero4)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Primer número:23
Segundo número:45
Tercer número:62
Cuarto número:234
23.0 == 234.0 : False
45.0 != 62.0 : True
62.0 > 45.0 : True
234.0 < 23.0 : False
23.0 >= 62.0 : False
45.0 <= 234.0 : True
```

12

CADENAS DE TEXTO

En este capítulo vamos a explicarte qué son las cadenas de texto, cómo se utilizan y qué operaciones puedes hacer con ellas. La manipulación de las cadenas de texto en programación es clave y jugará un papel fundamental dentro de todos los aplicativos que realices.

En Python existen dos formas diferentes de definir las cadenas de texto, utilizando comillas dobles o comillas simples. No existe diferencia entre usar un método u otro.

Veamos cómo se definen cadenas con un pequeño ejercicio. El código fuente es el siguiente:

```
cadena1 = "Hola Python definido con comillas dobles"
cadena2 = 'Hola Python definido con comillas simples'
print(cadena1)
print(cadena2)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hola Python definido con comillas dobles
Hola Python definido con comillas simples
```

Tal y como puedes comprobar, es indiferente definir una cadena de texto con comillas simples o comillas dobles.

Las cadenas de texto están compuestas por una secuencia de caracteres, dichos caracteres son accesibles de forma unitaria dentro de la cadena. Para acceder a un carácter en concreto de la cadena de texto hay que indicar la posición dentro de

la cadena de texto que ocupa y hay que tener en cuenta que el primer carácter de la cadena de texto ocupa la posición 0.

Veamos con un ejercicio como acceder a caracteres de una cadena de texto.

```
cadena = "Python"
print("Carácter posición 0:",cadena[0])
print("Carácter posición 1:",cadena[1])
print("Carácter posición 2:",cadena[2])
print("Carácter posición 3:",cadena[3])
print("Carácter posición 4:",cadena[4])
print("Carácter posición 5:",cadena[5])
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Carácter posición 0: P
Carácter posición 1: y
Carácter posición 2: t
Carácter posición 3: h
Carácter posición 4: o
Carácter posición 5: n
```

12.1 OPERADORES CON CADENAS

Las cadenas de caracteres en Python pueden utilizar los operadores "+" y "*" para concatenar cadenas de texto y para multiplicar una cadena de texto.

La operación de concatenar nos va a permitir unir varias cadenas de texto en una única cadena de texto resultante. La operación de multiplicar nos va a dar como resultado una cadena de texto compuesta por n veces la cadena de texto que se multiplica.

El siguiente ejercicio muestra cómo funcionan la concatenación y la multiplicación de cadenas de texto. El código fuente es el siguiente:

```
cadena1 = input("Introduzca la primera cadena:")
cadena2 = input('Introduzca la segunda cadena:')
cadena3 = input('Introduzca la tercera cadena:')
cadenasuma= cadena1 + ' ' + cadena2 + " " + cadena3
cadenamultiplicacion = (cadena2 + " ") * 5
print("Carácter concatenada:",cadenasuma)
print("Carácter multiplicada:",cadenamultiplicacion)
```

Tal y como puedes comprobar en el código fuente, hemos utilizado comillas dobles y simples de forma aleatoria. En la cadena resultado de la suma hemos añadido un carácter en blanco para separar las cadenas de texto y en la cadena resultado de la multiplicación, lo mismo. A su vez, hemos hecho uso de los paréntesis para indicar que la cadena de texto que tiene que multiplicar es la resultante de añadir el carácter en blanco a la cadena número 2.

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca la primera cadena:Estoy
Introduzca la segunda cadena:aprendiendo
Introduzca la tercera cadena:Python
Cadena concatenada: Estoy aprendiendo Python
Cadena multiplicada: aprendiendo aprendiendo aprendiendo aprendiendo aprendiendo
```

La operación de concatenación puede realizarse de forma simplificada, es posible utilizar el operador "+=" para concatenar la cadena de texto a la cadena que se está asignando. Veámoslo con un ejercicio, el código fuente sería el siguiente:

```
cadena1 = input("Introduzca la primera cadena:")
cadena2 = input('Introduzca la segunda cadena:')
cadena3 = input('Introduzca la tercera cadena:')
cadenasuma = cadena1
cadenasuma += ' '
cadenasuma += cadena2
cadenasuma += ' '
cadenasuma += cadena3
print("Cadena concatenada:",cadenasuma)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca la primera cadena:Estoy
Introduzca la segunda cadena:aprendiendo
Introduzca la tercera cadena:Python
Cadena concatenada: Estoy aprendiendo Python
```

Python ofrece un operador que permite comprobar si una cadena contiene otra cadena o un carácter en concreto. El operador es **in**, y el formato de uso es el siguiente:

Cadena1 in Cadena2

El operador comprueba si la cadena1 está contenida dentro de la cadena2. En el siguiente ejercicio solicitaremos al usuario que introduzca dos cadenas y comprobaremos si la segunda cadena está incluida en la primera. El código fuente es el siguiente:

```
cadena1 = input("Introduzca la primera cadena:")
cadena2 = input('Introduzca la segunda cadena:')
print("¿Está la segunda cadena contenida en la primera?:",cadena2 in cadena1)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca la primera cadena:Python
Introduzca la segunda cadena:th
¿Está la segunda cadena contenida en la primera?: True
```

12.2 CARACTERES ESPECIALES

En el capítulo sobre la entrada y salida estándar te explicamos que existe un conjunto de caracteres que requieren una forma concreta a la hora de ser mostrados por pantalla, bien sea porque aportan funciones dentro de las cadenas de texto (tales como salto de línea o tabulador) o porque pertenecen a la sintaxis de Python (tales como comillas simples o dobles), pues lo explicado en ese capítulo es válido para las cadenas de texto.

El formato de utilización de caracteres especiales dentro de las cadenas de texto es el mismo, utilizando el carácter de barra invertida "\." La siguiente tabla muestra los diferentes caracteres especiales soportados en Python:

Carácter	¿Qué se muestra?
\n	Retorno de línea
\r	Retorno de carro
\t	Tabulador
\v	Tabulador vertical
\\	Diagonal invertida
\"	Comilla doble
\'	Comilla simple
\xNN	Carácter hexadecimal NN en ASCII
\uNN	Carácter hexadecimal NN en Unicode
\oNN	Carácter octal NN

En el siguiente ejercicio vamos a unificar una serie de cadenas de texto en una única cadena de texto pero separada por saltos de líneas, y añadiendo una tabulación para que el mensaje sea más legible. El código fuente es el siguiente:

```
cadena1 = input("Introduzca la primera cadena:")
cadena2 = input('Introduzca la segunda cadena:')
cadena3 = input('Introduzca la tercera cadena:')
cadenaconsaltos = "\n\t" + cadena1 + "\n\t" + cadena2 + '\n\t' + cadena3
print("Cadena con saltos:",cadenaconsaltos)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca la primera cadena:Primera cadena de texto
Introduzca la segunda cadena:Segunda cadena de texto
Introduzca la tercera cadena:Tercera y última cadena de texto
Cadena con saltos:
        Primera cadena de texto
        Segunda cadena de texto
        Tercera y última cadena de texto
```

En Python existe una funcionalidad que permite ignorar los caracteres especiales dentro de una cadena de texto. Para ignorarlos basta con añadir el carácter **r** antes de la definición de la cadena. El siguiente ejercicio modifica el ejercicio anterior añadiendo el ignorar los caracteres especiales. El código fuente es el siguiente:

```
cadena1 = input("Introduzca la primera cadena:")
cadena2 = input('Introduzca la segunda cadena:')
cadena3 = input('Introduzca la tercera cadena:')
cadenaconsaltos = r"\n\t" + cadena1 + r"\n\t" + cadena2 + r'\n\t' + cadena3
print("Cadena con saltos:", cadenaconsaltos)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca la primera cadena:Primera
Introduzca la segunda cadena:Segunda
Introduzca la tercera cadena:Tercera
Cadena con saltos: \n\tPrimera\n\tSegunda\n\tTercera
```

Puedes comprobar en la salida que los saltos de línea y tabulador introducidos antes de las cadenas salen tal y como los hemos añadido en la concatenación, sin ser caracteres especiales. Ésto se debe a que se ha añadido el carácter **r** antes de dichas cadenas con los caracteres especiales.

12.3 FUNCIONES

El tipo de dato cadena de texto en Python posee una serie de funciones que nos permiten manipular las cadenas de texto realizando operaciones complejas de forma sencilla y con una simple instrucción. El formato de uso en la gran mayoría de las funciones es el siguiente:

ValorDevuelto = CadenaTexto.NombreFuncion(Parámetros)

Veamos en detalle cada una de las partes:

- ValorDevuelto: la ejecución de la función tendrá un resultado. Ése resultado puede ser un booleano, un número, otra cadena de texto o una lista de elementos (explicadas en el siguiente capítulo).
- CadenaTexto: cadena sobre la que se ejecutará la función.
- NombreFuncion: nombre de la función que se quiere ejecutar.
- Parámetros: no todas las funciones tienen parámetros para ejecutarse, esta parte es dependiente de la función que se quiere ejecutar.

La funciones de cadenas que pone a nuestra disposición Python son las siguientes:

- **capitalize**: permite poner la primera letra de la cadena de texto en mayúsculas.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.
- **title**: permite poner la primera letra de cada palabra de la cadena de texto en mayúsculas y el resto de letras en minúsculas.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.
- **upper**: permite poner en mayúsculas por completo la cadena de texto.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.
- **lower**: permite poner en minúsculas por completo la cadena de texto.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.

- **len**: permite saber el número de caracteres que componen la cadena de texto.
  - Valor devuelto: entero que indica el número de caracteres que contiene la cadena de texto.
  - Parámetros: no tiene.
- **count**: permite saber el número de veces que aparece una cadena de texto dentro de otra.
  - Valor devuelto: entero que indica el número de apariciones.
  - Parámetros: no tiene.
- **isalnum**: permite comprobar si todos los caracteres que componen la cadena de texto son alfanuméricos o no. Ten en cuenta que los caracteres del punto y del espacio en blanco no son caracteres alfanuméricos.
  - Valor devuelto: booleano que indica si son todos los caracteres alfanuméricos o no.
  - Parámetros: no tiene.
- **isalpha**: permite comprobar si todos los caracteres de la cadena de texto son caracteres alfabéticos. Ten en cuenta que ni los números, ni los puntos, ni los espacios en blanco son caracteres alfabéticos, únicamente las letras componen el conjunto de caracteres alfabéticos.
  - Valor devuelto: booleano que indica si son todos los caracteres alfabéticos o no.
  - Parámetros: no tiene.
- **isdigit**: permite comprobar si todos los caracteres de la cadena de texto son caracteres que representan dígitos.
  - Valor devuelto: booleano que indica si son todos los caracteres dígitos o no.
  - Parámetros: no tiene.
- **isnumeric**: permite comprobar si todos los caracteres de la cadena de texto son caracteres con representación numérica.
  - Valor devuelto: booleano que indica si son todos los caracteres numéricos o no.
  - Parámetros: no tiene.

- **isdecimal**: permite comprobar si todos los caracteres de la cadena de texto son caracteres con representación numérica decimal.
  - Valor devuelto: booleano que indica si son todos los caracteres numéricos decimales o no.
  - Parámetros: no tiene.
- **islower**: permite comprobar si todos los caracteres que componen la cadena están en minúscula.
  - Valor devuelto: booleano que indica si todos los caracteres están en minúscula o no.
  - Parámetros: no tiene.
- **isupper**: permite comprobar si todos los caracteres que componen la cadena de texto están en mayúscula.
  - Valor devuelto: booleano que indica si todos los caracteres están en mayúscula o no.
  - Parámetros: no tiene.
- **istitle**: permite comprobar si el primer carácter de todas las palabras es mayúscula y el resto minúsculas.
  - Valor devuelto: booleano que indica si el primer carácter de todas las palabras es mayúscula y el resto minúsculas o no.
  - Parámetros: no tiene.
- **isprintable**: permite comprobar si todos los caracteres que componen la cadena de texto son imprimibles o no.
  - Valor devuelto: booleano que indica si son todos imprimibles o no.
  - Parámetros: no tiene.
- **isspace**: permite comprobar si la cadena de texto está compuesta exclusivamente por caracteres en blanco.
  - Valor devuelto: booleano que indica si está compuesta únicamente por caracteres en blanco o no.
  - Parámetros: no tiene.
- **lstrip**: permite eliminar los caracteres en blanco al comienzo de la cadena.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.

- **rstrip**: permite eliminar los caracteres en blanco al final de la cadena.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.
- **strip**: permite eliminar los caracteres en blanco al principio y al final de la cadena.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.
- **max**: permite conocer el carácter alfabético mayor de la cadena.
  - Valor devuelto: carácter alfanumérico con mayor valor de la cadena.
  - Parámetros: no tiene.
- **min**: permite conocer el carácter alfabético menor de la cadena.
  - Valor devuelto: carácter alfanumérico con menor valor de la cadena.
  - Parámetros: no tiene.
- **startswith**: permite comprobar si una cadena de texto empieza por una cadena de texto concreta.
  - Valor devuelto: booleano que indica si empieza o no por esa cadena.
  - Parámetros: tiene un parámetro obligatorio que es la cadena que se tiene que comprobar si es por la que empieza la otra cadena.
- **endswith**: permite comprobar si una cadena de texto termina por una cadena de texto concreta.
  - Valor devuelto: booleano que indica si termina o no por esa cadena.
  - Parámetros: tiene un parámetro obligatorio que es la cadena que se tiene que comprobar si es por la que termina la otra cadena.
- **replace**: permite reemplazar una cadena de texto de la cadena de texto por otra cadena.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: tiene dos parámetros, el primero es la cadena que se quiere reemplazar y el segundo es la cadena que se utilizará para hacer el reemplazo del primer parámetro.

- **swapcase**: permite invertir las mayúsculas y minúsculas de la cadena de texto, es decir, las mayúsculas pasarán a ser minúsculas y las minúsculas pasarán a ser mayúsculas.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: no tiene.
- **split**: permite convertir una cadena de texto en una lista de elementos que se encuentran separados por un carácter concreto.
  - Valor devuelto: lista de elementos de tipo cadena de texto.
  - Parámetros: tiene un parámetro opcional que indica el carácter a utilizar para dividir la cadena de texto. En caso de no especificarse se utilizará el espacio en blanco por defecto.
- **splitlines**: permite convertir la cadena de texto en una lista de elementos que se encuentran separados en la cadena de texto por saltos de línea.
  - Valor devuelto: lista de elementos de tipo cadena de texto.
  - Parámetros: no tiene..
- **find**: permite encontrar una cadena de texto dentro de otra buscando de izquierda a derecha.
  - Valor devuelto: en caso de encontrar la cadena devolverá la posición dentro de la cadena en la que empieza la cadena buscada, en caso de no encontrarla devolverá -1.
  - Parámetros: cadena a buscar.
- **rfind**: permite encontrar una cadena de texto dentro de otra buscando de derecha a izquierda.
  - Valor devuelto: en caso de encontrar la cadena devolverá la posición dentro de la cadena en la que empieza la cadena buscada, en caso de no encontrarla devolverá -1.
  - Parámetros: cadena a buscar.
- **center**: permite centrar el texto de la cadena en una cadena de texto de longitud mayor añadiendo espacios en blanco a la izquierda y derecha del mismo.
  - Valor devuelto: cadena de texto modificada.

- Parámetros: tiene un parámetro obligatorio que es la longitud de la cadena de texto que queremos obtener y tiene un parámetro opcional para indicar el carácter con el que rellenar para llegar a esa longitud. Si no se especifica el segundo parámetro lo realiza con espacios en blanco por defecto.

- **ljust**: permite ajustar una cadena de texto a la izquierda con una longitud indicada por parámetros y añadiendo tantos caracteres en blanco a la derecha como sea necesario para llegar a esa longitud.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: tiene un parámetro obligatorio que es la longitud de la cadena de texto que queremos obtener y tiene un parámetro opcional para indicar el carácter con el que rellenar para llegar a esa longitud. Si no se especifica el segundo parámetro lo realiza con espacios en blanco por defecto.

- **rjust**: permite ajustar una cadena de texto a la derecha con una longitud indicada por parámetros y añadiendo tantos caracteres en blanco a la izquierda como sea necesario para llegar a esa longitud.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: tiene un parámetro obligatorio que es la longitud de la cadena de texto que queremos obtener y tiene un parámetro opcional para indicar el carácter con el que rellenar para llegar a esa longitud. Si no se especifica el segundo parámetro lo realiza con espacios en blanco por defecto.

- **expandtabs**: permite sustituir los tabuladores que tiene la cadena de texto por espacios en blanco.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: tiene un parámetro opcional para indicar el número de espacios en blanco por los que sustituye cada tabulador, por defecto lo hace sustituyendo por cuatro espacios en blanco.

- **zfill**: permite crear una cadena de texto de longitud indicada por parámetros compuesta por ceros a la izquierda y a la derecha la cadena de texto.
  - Valor devuelto: cadena de texto modificada.
  - Parámetros: longitud de la cadena resultante.

En los siguientes ejercicios vamos a practicar utilizando las funciones que acabamos de ver.

El primer ejercicio consiste en introducir tres cadenas de texto y realizar operaciones de conversión del texto y comprobaciones del texto. El código fuente es el siguiente:

```
cadena1 = input("Introduzca la primera cadena:")
cadena2 = input('Introduzca la segunda cadena:')
cadena3 = input("Introduzca la tercera cadena:")
print("Longitud de la cadena2 (len):", len(cadena2))
print("Cadena3 toda a mayúsculas (upper):",cadena3.upper())
print("Cadena3 toda a minúsculas (lower):",cadena3.lower())
print("Cadena2 cambia de mayúsculas a minúsculas y viceversa
(swapcase):",cadena2.swapcase())
print("Cadena1 la primera a mayúsculas (capitalize):",cadena1.capitalize())
print("Cadena2 la primera de cada palabra a mayúsculas (title):", cadena2.tit-
le())
print("¿Cadena1 todo minúsculas? (islower):", cadena1.islower())
print("¿Cadena3 todo mayúsculas? (isupper):", cadena3.isupper())
print("¿Cadena2 todo caracteres imprimibles? (isprintable):", cadena2.isprinta-
ble())
print("¿Cadena3 todo caracteres alfanuméricos? (isalnum):", cadena3.isalnum())
print("¿Cadena1 todo caracteres alfabéticos? (isalpha):", cadena1.isalpha())
print("¿Cadena3 la primera de cada palabra en mayúsculas y el resto minúsculas?
(istitle):",cadena3.istitle())
print("¿Cadena2 todo los caracteres son espacios en blanco? (isspace):",cadena2.
isspace())
print("¿Cadena1 todo dígitos? (isdigit):", cadena1.isdigit())
print("¿Cadena2 todos los caracteres con representación numérica?
(isnumeric):",cadena2.isnumeric())
print("¿Cadena3 todos los caracteres son números con representación decimal?
(isdecimal):",cadena3.isdecimal())
print("Carácter más alto de la cadena1 (max):", max(cadena1))
print("Carácter más bajo de la cadena3 (min):", max(cadena3))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca la primera cadena:estoy aprendiendo Python
Introduzca la segunda cadena:me encanta!
Introduzca la tercera cadena:es un lenguaje de programación muy fácil
Longitud de la cadena2 (len): 11
Cadena3 toda a mayúsculas (upper): ES UN LENGUAJE DE PROGRAMACIÓN MUY FÁCIL
Cadena3 toda a minúsculas (lower): es un lenguaje de programación muy fácil
Cadena2 cambia de mayúsculas a minúsculas y viceversa (swapcase): ME ENCANTA!
Cadena1 la primera a mayúsculas (capitalize): Estoy aprendiendo python
Cadena2 la primera de cada palabra a mayúsculas (title): Me Encanta!
¿Cadena1 todo minúsculas? (islower): False
¿Cadena3 todo mayúsculas? (isupper): False
¿Cadena2 todo caracteres imprimibles? (isprintable): True
¿Cadena3 todo caracteres alfanuméricos? (isalnum): False
¿Cadena1 todo caracteres alfabeticos? (isalpha): False
¿Cadena3 la primera de cada palabra en mayúsculas y el resto minúsculas? (istitle): False
¿Cadena2 todo los caracteres son espacios en blanco? (isspace): False
¿Cadena1 todo dígitos? (isdigit): False
¿Cadena2 todos los caracteres con representación numérica? (isnumeric): False
¿Cadena3 todos los caracteres son numeros con representación decimal? (isdecimal): False
Carácter más alto de la cadena1 (max): y
Carácter más bajo de la cadena3 (min): ó
```

El segundo ejercicio consiste en eliminar caracteres de espacios en blanco de la cadena solicitada al usuario. El código fuente es el siguiente:

```
cadena = input("Introduzca una cadena con espacios en blanco al principio y al
final:")
print("Longitud de la cadena:", len(cadena))
cadenalstrip = cadena.lstrip()
print("Cadena (lstrip):",cadenalstrip,end='.')
print("\nLongitud de la cadena (lstrip):", len(cadenalstrip))
cadenarstrip = cadena.rstrip()
print("Cadena (rstrip):",cadenarstrip,end='.')
print("\nLongitud de la cadena (rstrip):", len(cadenarstrip))
cadenastrip = cadena.strip()
print("Cadena (strip):",cadenastrip,end='.')
print("\nLongitud de la cadena (strip):", len(cadenastrip))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca una cadena con espacios en blanco al principio y al final:    Hola Python
Longitud de la cadena: 18
Cadena (lstrip): Hola Python    .
Longitud de la cadena (lstrip): 15
Cadena (rstrip):    Hola Python.
Longitud de la cadena (rstrip): 14
Cadena (strip): Hola Python.
Longitud de la cadena (strip): 11
```

El tercer ejercicio consiste en comprobar si una cadena empieza y termina por otra y en contar cuántas veces aparece dicha cadena en la cadena base. El código fuente es el siguiente:

```
cadena = input("Introduzca una cadena:")
buscar = input("Introduzca una cadena para buscar:")
print("¿Comienza la cadena por la cadena a buscar? (startswith):",cadena.
startswith(buscar))
print("¿Termina la cadena por la cadena a buscar? (endswith):",cadena.
endswith(buscar))
print("¿Cuántas veces aparece la cadena a buscar en la cadena? (count):",cadena.
count(buscar))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca una cadena:me encanta programar en Python
Introduzca una cadena para buscar:m
¿Comienza la cadena por la cadena a buscar? (startswith): True
¿Termina la cadena por la cadena a buscar? (endswith): False
¿Cuántas veces aparece la cadena a buscar en la cadena? (count): 2
```

El cuarto ejercicio consiste en la realización de una búsqueda de izquierda a derecha y de derecha a izquierda de una cadena de texto en otra. El código fuente es el siguiente:

```
cadena = input("Introduzca una cadena:")
buscar = input("Introduzca una cadena para buscar:")
print("La cadena aparece en la posición (find):", cadena.find(buscar))
print("La cadena aparece en la posición (rfind):", cadena.rfind(buscar))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca una cadena:Me encanta aprender Python mientras practico
Introduzca una cadena para buscar:a
La cadena aparece en la posición (find): 6
La cadena aparece en la posición (rfind): 38
```

El quinto ejercicio consiste en el reemplazo de una cadena de texto de una cadena de texto base por otra cadena de texto. El código fuente es el siguiente:

```
cadena = input("Introduzca una cadena:")
reemplazar = input("Introduzca una subcadena de la anterior para reemplazar:")
```

```
reemplazo = input("Introduzca la cadena por la que se reemplazará la anterior:")
print("Cadena original:", cadena)
print("Cadena nueva (replace):", cadena.replace(reemplazar, reemplazo))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca una cadena:La casa es de color blanco
Introduzca una subcadena de la anterior para reemplazar:co
Introduzca la cadena por la que se reemplazará la anterior:TA
Cadena original: La casa es de color blanco
Cadena nueva (replace): La casa es de TAlor blanTA
```

El sexto ejercicio consiste en crear una lista de cadenas de texto a partir de las palabras que componen la cadena de texto. Las listas las veremos en el siguiente capítulo. El código fuente es el siguiente:

```
cadena = input("Introduzca una cadena con varias palabras:")
print("Cadena dividida por espacios en blanco (split):",cadena.split())
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca una cadena con varias palabras:Rosa de Alejandría
Cadena dividida por espacios en blanco (split): ['Rosa', 'de', 'Alejandría']
```

12.4 PORCIONES DE CADENAS

Las cadenas de texto ofrecen la funcionalidad de extraer partes de la cadena de texto, es decir, partiendo de una cadena de texto base nos ofrece la posibilidad de extraer subcadenas para poder trabajar con ellas.

La forma de extraer estas subcadenas es utilizando el operador [n:m], donde n es la posición del carácter inicial y m la posición del carácter final. El operador [n:m] permite omitir uno de los dos índices, en caso de omitir la n la subcadena empezará desde el principio de la cadena y en caso de omitir la m la subcadena llegará hasta el final de la cadena.

En el siguiente ejercicio se extraen diversas subcadenas, primero utilizando ambos índices (*n* y *m*) y posteriormente omitiendo uno de ellos. El código fuente es el siguiente:

```
cadena = 'F.C.Barcelona, Atlético de Madrid, Real Madrid'
print("Primer equipo (cadena[0:13]):", cadena[0:13])
print("Segundo equipo (cadena[15:33]):", cadena[15:33])
print("Tercer equipo (cadena[35:46]):", cadena[35:46])
print("Desde el principio (cadena[:13]):", cadena[:13])
print("Desde el final (cadena[:13]):", cadena[15:])
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Primer equipo (cadena[0:13]): F.C.Barcelona
Segundo equipo (cadena[15:33]): Atlético de Madrid
Tercer equipo (cadena[35:46]): Real Madrid
Desde el principio (cadena[:13]): F.C.Barcelona
Desde el final (cadena[:13]): Atlético de Madrid, Real Madrid
```

12.5 FORMATEO DE CADENAS

Hasta el momento hemos construido cadenas concatenándolas, pero Python nos ofrece dos formas de crear cadenas que están compuestas por varias cadenas. La primera es utilizando un operador concreto para componer la cadena y la segunda es utilizando la función *format*.

12.5.1 Operador %

La primera forma de componer cadenas es utilizando el operador %. El operador indica que dentro de la cadena base se realizará la sustitución de ese operador por la cadena que se especifique. Al operador hay que indicarle el tipo de dato que lleva asociado, la siguiente tabla muestra el valor que tendrá que tener el operador en cada caso de tipo de dato.

Operador	Tipo de dato
%c	Un carácter
%s	Cadena de texto
%d	Número entero
%f	Número real
%o	Número octal
%x	Número hexadecimal

El siguiente ejercicio muestra como construir una cadena sustituyendo los operadores por los valores enteros introducidos por el usuario y también por el resultado de la multiplicación. El código fuente es el siguiente:

```
multiplicando = int(input("Multiplicando:"))
multiplicador = int(input("Multiplicador:"))
print("El resultado de multiplicar %d por %d es %d" % (multiplicando, multipli-
cador, multiplicando*multiplicador))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Multiplicando:5
Multiplicador:7
El resultado de multiplicar 5 por 7 es 35
```

12.5.2 format()

La función *format* devuelve una versión formateada de una cadena de caracteres, usando sustituciones desde argumentos args y kwargs. La forma de especificar dentro de la cadena de texto base la posición que deben ocupar los argumentos se identifican con llaves { } dentro de la cadena y el número del argumento que sustituirá a las llaves.

Vamos a realizar el mismo ejercicio que en el apartado anterior pero utilizando *format* en lugar del operador %. El código fuente es el siguiente:

```
multiplicando = int(input("Multiplicando:"))
multiplicador = int(input("Multiplicador:"))
print("El resultado de multiplicar {0} por {1} es {2}".format(multiplicando,
multiplicador, multiplicando*multiplicador))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Multiplicando:12
Multiplicador:32
El resultado de multiplicar 12 por 32 es 384
```

13

LISTAS, TUPLAS Y DICCIONARIOS

En este capítulo vamos a explicarte una serie de tipo de datos que se caracterizan por estar compuestos por secuencias de elemento. Son las llamadas colecciones. En Python tenemos tres posibles colecciones:

- Listas
- Tuplas
- Diccionarios

A lo largo del capítulo vamos a ver cada uno de ellos en detalle junto con una serie de ejercicios para aprender a utilizarlos.

13.1 LISTAS

En este apartado vamos a hablar sobre los tipos de datos listas, que son un conjunto ordenado de elementos que puede contener datos de cualquier tipo. Una característica muy importante de las listas es que pueden contener elementos de diferentes tipos, por ejemplo una lista puede estar compuesta por cadenas de texto, por números enteros y por supuesto, por otras listas.

De todos los tipos de datos que vamos a ver en este capítulo, las listas son los más flexibles de todos.

Las listas en Python se representan por una serie de elementos separados por comas y delimitados entre corchetes. Veamos un ejemplo de lista:

[56,"Python","Valeria",345,99]

La lista estaría compuesta por cinco elementos, el número *56*, la cadena de texto *"Python"*, la cadena de texto *"Valeria"*, y los números 345 y 99.

Los elementos en las listas ocupan posiciones concretas, y mediante esa posición que ocupan podemos acceder directamente a los elementos. En la siguiente tabla te mostramos la relación de cada elemento con la posición que ocupa en la lista:

Elemento	Posición
56	0
"Python"	1
"Valeria"	2
345	3
99	4

Los elementos de una lista empiezan por la posición 0, no por la 1 como parecería lo obvio. Tenlo en cuenta cuando utilices listas.

13.1.1 Ejercicios

Los ejercicios con listas los hemos dividido en dos grupos diferentes. El primer grupo de ejercicios consiste en la manipulación de las listas y el segundo grupo consiste en el aprendizaje de los métodos propios de los tipos de datos lista.

13.1.1.1 MANIPULACIÓN

El primer ejercicio que vamos a hacer consiste en la definición de una lista con elementos de diferentes tipos y en mostrarla por pantalla, tanto entera como elemento a elemento accediendo a la posición que ocupan dentro de la misma. El código fuente es el siguiente:

```
lista = ["Python","RA-MA",2019,"Libro",3]
print(lista)
print(lista[0])
print(lista[1])
print(lista[2])
print(lista[3])
print(lista[4])
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
['Python', 'RA-MA', 2019, 'Libro', 3]
Python
RA-MA
2019
Libro
3
```

La primera línea muestra la lista completa mientras que las siguientes líneas muestran cada uno de los elementos presentes en la lista.

El segundo ejercicio consiste en el aprendizaje del operador "+" para realizar uniones de listas. El operador se utiliza de la siguiente manera:

ListaConcatenada = Lista1 + Lista2

Además vamos a aprender a utilizar una función para conocer el número de elementos que componen la lista. Dicha función se llama *len* y devuelve un entero que indica el número de elementos que la componen. Se utiliza de la siguiente manera:

NumeroElementos = len(Lista)

El código fuente del ejercicio es el siguiente:

```
lista1 = ["Camiseta", "Pantalón","Zapatillas"]
lista2 = ["Abrigo", "Jersey", "Sudadera", "Calcetiles"]
print("Número elementos de lista1: ",len(lista1))
print("Lista1: ", lista1)
print("Número elementos de lista2: ",len(lista2))
print("Lista2: ", lista2)
listaconcatenada = lista1 + lista2
print("Número elementos de listaconcatenada: ",len(listaconcatenada))
print("listaconcatenada: ", listaconcatenada)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Número elementos de lista1:  3
Lista1:  ['Camiseta', 'Pantalón', 'Zapatillas']
Número elementos de lista2:  4
Lista2:  ['Abrigo', 'Jersey', 'Sudadera', 'Calcetiles']
Número elementos de listaconcatenada:  7
listaconcatenada:  ['Camiseta', 'Pantalón', 'Zapatillas', 'Abrigo', 'Jersey', 'Sudadera', 'Calcetiles']
```

El tercer ejercicio es del estilo del ejercicio anterior, pero ahora vamos a ir añadiendo elementos a la lista de diferentes formas. Añadir elementos a una lista existente es una operación recurrente dentro de la programación, es por ello que vamos a poner especial interés en esta operación. El código fuente es el siguiente:

```
lista = [“Camiseta”, “Pantalón”,”Zapatillas”]
print(lista)
lista = lista + [“Abrigo”]
print(lista)
lista = lista + [“Jersey”,”Sudadera”]
print(lista)
lista = lista + [“Calcetines”] + [“Bufanda”]
print(lista)
```

La primera inserción de elementos inserta un único elemento, *“Abrigo”*, la segunda inserción inserta un elemento compuesto por dos elementos, *“Jersey”* y *“Sudadera”*, y por último se insertan dos elementos independientes de una misma sentencia, *“Calcetines”* y *“Bufanda”*.

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
['Camiseta', 'Pantalón', 'Zapatillas']
['Camiseta', 'Pantalón', 'Zapatillas', 'Abrigo']
['Camiseta', 'Pantalón', 'Zapatillas', 'Abrigo', 'Jersey', 'Sudadera']
['Camiseta', 'Pantalón', 'Zapatillas', 'Abrigo', 'Jersey', 'Sudadera', 'Calcetines', 'Bufanda']
```

El cuarto ejercicio consiste en aprender a modificar elementos de una lista y en el borrado de elementos de la misma.

La modificación de elementos de una lista se hace de la siguiente forma:

Lista[posición] = NuevoValor

La posición indica el elemento que será modificado dentro de la lista, el valor asignado es el nuevo valor que tendrá dicho elemento.

El borrado de elementos existentes dentro de una lista se hace utilizando la instrucción *del*. Se utiliza de la siguiente forma:

del Lista[posición]

El elemento que se borrará será el indicado por *posición*.

El código fuente del ejercicio es el siguiente:

```
lista = [“Camiseta”, “Pantalón”,”Zapatillas”]
print(lista)
lista[1] = “Cazadora”
print(lista)
del lista[0]
print(lista)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
['Camiseta', 'Pantalón', 'Zapatillas']
['Camiseta', 'Cazadora', 'Zapatillas']
['Cazadora', 'Zapatillas']
```

El quinto ejercicio consiste en el aprendizaje del operador “*” que nos va a permitir concatenar una lista con ella misma un número finito de veces. El operador se utiliza de la siguiente forma:

ListaResultante = Lista * NúmeroEntero

La lista resultante de la multiplicación será una lista compuesta por tantas veces la *lista* como valor tenga el número entero.

El código fuente del ejercicio es el siguiente:

```
lista = [“Camiseta”, “Pantalón”,”Zapatillas”]
print(lista)
listaresultante = lista * 3
print(listaresultante)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
['Camiseta', 'Pantalón', 'Zapatillas']
['Camiseta', 'Pantalón', 'Zapatillas', 'Camiseta', 'Pantalón', 'Zapatillas', 'Camiseta', 'Pantalón', 'Zapatillas']
```

El sexto ejercicio consiste en la creación de listas como elementos de listas y practicar cómo acceder a dichos elementos. El ejercicio creará una lista que tiene una lista en uno de sus elementos y mostrará todos los elementos de la lista principal, incluyendo los elementos de la lista secundaria. El código fuente es el siguiente:

```
lista = [“Camiseta”, [“Calcetines”,”Cazadora”],”Zapatillas”]
print(lista)
print(lista[0])
print(lista[1])
print(lista[2])
print(lista[1][0])
print(lista[1][1])
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
['Camiseta', ['Calcetines', 'Cazadora'], 'Zapatillas']
Camiseta
['Calcetines', 'Cazadora']
Calcetines
Cazadora
Zapatillas
```

Tal y como puedes observar, para acceder a los elementos de la lista secundaria primero tenemos que acceder al elemento de la lista principal y después al elemento concreto de dicha lista. Para acceder al elemento *“Calcetines”* tendremos que acceder al elemento [1] de la lista principal y al elemento [0] de la lista secundaria. Si tuviéramos un elemento dentro de la lista secundaria que fuera otra lista, únicamente tendríamos que añadir el elemento entre corchetes a continuación de la lista secundaria. Por ejemplo, si tuviéramos la lista en la posición 1 de la lista secundaria y quisiéramos acceder al elemento 0 de dicha lista lo haríamos de la siguiente forma:

Lista[1][1][0]

El último ejercicio consiste en extraer una porción de la lista en otra lista nueva. La extracción se realiza utilizando la siguiente instrucción:

Lista[n:m]

La instrucción extraerá una nueva lista que empezará en el índice n y terminará en el m-1. Tienes que tener en cuenta lo siguiente:

- n siempre tiene que ser menor que m.
- Si no se especifica el valor para n se supone que es 0.
- Si no se especifica el valor para m se supone que es el tamaño de la lista menos uno.

Veámoslo en un ejercicio, el código fuente es el siguiente:

```
lista = [1,2,3,4,5,6,7,8,9]
print(lista)
lista1 = lista[3:7]
print(lista1)
lista2 = lista[:5]
print(lista2)
lista3 = lista[6:]
print(lista3)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[4, 5, 6, 7]
[1, 2, 3, 4, 5]
[7, 8, 9]
```

13.1.1.2 MÉTODOS PROPIOS

El tipo de dato lista en Python posee una serie de funciones que nos permiten manipular las listas realizando operaciones complejas de forma sencilla y con una simple instrucción. El formato de uso de la gran mayoría de las funciones es el siguiente:

Lista.NombreFuncion(Parámetros)

Veamos en detalle cada una de las partes:

- Lista: lista que ejecuta la función.
- NombreFuncion: nombre de la función que se quiere ejecutar.
- Parámetros: no todas las funciones tienen parámetros para ejecutarse, esta parte es dependiente de la función que se quiere ejecutar.

La funciones de listas que pone a nuestra disposición Python son las siguientes:

- **append**: añade el elemento pasado como parámetro a la lista.
  - Valor devuelto: lista modificada con el elemento añadido.
  - Parámetros: elemento de cualquier tipo de dato.

- **insert**: añade el elemento pasado como parámetro a la lista en la posición indicada también por parámetro. El elemento que ocupe esa posición y los de posiciones superiores serán desplazados hacia la derecha dentro de la lista.
  - Valor devuelto: lista modificada con el elemento añadido.
  - Parámetros: recibe dos parámetros, el primero indica la posición en la que se insertará el elemento dentro de la lista y el segundo el elemento a insertar.
- **remove**: elimina la primera ocurrencia empezando por la izquierda de la lista del elemento indicado como parámetro. En caso de no encontrar el elemento la función devolverá un error.
  - Valor devuelto: lista modificada con el elemento eliminado o error, dependiendo de si encuentra o no el elemento.
  - Parámetros: .
- **reverse**: invierte el orden de la lista.
  - Valor devuelto: lista modificada.
  - Parámetros: no tiene.
- **sort**: ordena la lista si es posible. Por defecto la ordenación es ascendente, en caso de querer una ordenación descendente hay que indicarlo por parámetro.
  - Valor devuelto: lista modificada.
  - Parámetros: tiene un parámetro opcional (*reverse=true*) para indicar que la ordenación que se quiere realizar es descendente.
- **pop**: elimina un elemento de la lista y lo devuelve como resultado de la operación.
  - Valor devuelto: lista modificada.
  - Parámetros: tiene un parámetro opcional para indicar el elemento sobre el que se quiere ejecutar la función, en caso de no indicar el elemento la operación se realizará sobre el último elemento de la lista.
- **extend**: añade los elementos de una lista a la lista. Es importante diferenciarlo del método *append*, mientras que *append* añadiría la lista como elemento a la lista, la operación *extend* añade los elementos de la

lista a la lista, obteniendo como resultado una lista con los elementos de ambas.

- Valor devuelto: lista modificada.
- Parámetros: lista que se quiere añadir a la lista.

- **count**: cuenta el número de veces que aparece el elemento indicado como parámetro dentro de la lista.
  - Valor devuelto: número de ocurrencias del elemento.
  - Parámetros: elemento del que se quiere saber el número de veces que aparece en la lista.
- **index**: devuelve la posición de la primera ocurrencia de izquierda a derecha en la lista del elemento pasado como parámetro.
  - Valor devuelto: posición que ocupa el elemento dentro de la lista.
  - Parámetros: tiene tres parámetros, el primero es obligatorio y es el elemento a buscar, el segundo es opcional y se utiliza para indicar en qué posición empezar a buscar y por último, el tercer parámetro es opcional y se utiliza para indicar en qué posición finalizar la búsqueda.
- **clear**: elimina todos los elementos de la lista.
  - Valor devuelto: lista vacía.
  - Parámetros: no tiene.

A continuación vamos a realizar un ejercicio para aprender a utilizar todas las funciones que acabamos de explicar. El ejercicio consiste en utilizar todas las funciones sobre un objeto lista que crearemos al principio. El código fuente es el siguiente:

```
lista = [45,32,3,78]
print("Lista original: ", lista)
lista.append(995)
lista.append(7)
print("Lista después de usar append: ", lista)
lista.sort()
print("Lista ordenada: ", lista)
lista.reverse()
print("Lista al revés: ",lista)
listaextend = [1,5,87,45]
lista.extend(listaextend)
```

```
print("Lista después de extend: ",lista)
lista.sort(reverse=True)
print("Lista ordenada al revés: ", lista)
print("Número de elementos 45: ",lista.count(45))
lista.insert(4,111)
print("Lista después de insert: ",lista)
lista.remove(45)
print("Lista después de remove: ", lista)
print("Posición del elemento 111: ",lista.index(111))
lista.pop()
print("Lista después de pop: ", lista)
lista.clear()
print("Lista después de clear: ", lista)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Lista original:  [45, 32, 3, 78]
Lista después de usar append:  [45, 32, 3, 78, 995, 7]
Lista ordenada:  [3, 7, 32, 45, 78, 995]
Lista al revés:  [995, 78, 45, 32, 7, 3]
Lista después de extend:  [995, 78, 45, 32, 7, 3, 1, 5, 87, 45]
Lista ordenada al revés:  [995, 87, 78, 45, 45, 32, 7, 5, 3, 1]
Número de elementos 45:  2
Lista después de insert:  [995, 87, 78, 45, 111, 45, 32, 7, 5, 3, 1]
Lista después de remove:  [995, 87, 78, 111, 45, 32, 7, 5, 3, 1]
Posición del elemento 111:  3
Lista después de pop:  [995, 87, 78, 111, 45, 32, 7, 5, 3]
Lista después de clear:  []
```

13.2 TUPLAS

En este apartado vamos a hablar sobre los tipos de datos tuplas, que son un conjunto ordenado e inmutable de elementos. La diferencia con las listas reside en que en las listas puedes manipular los elementos y en las tuplas no, es decir, no es posible añadir/eliminar elementos, modificarlos, etc. Al igual que las listas, las tuplas pueden contener elementos de diferentes tipos, por ejemplo una tupla puede estar compuesta por cadenas de texto, por números enteros, etc.

Las tuplas en Python se representan por una serie de elementos separados por comas y delimitados entre paréntesis. Veamos un ejemplo de tupla:

("Casa","2",345,"Perro",99)

La tupla estaría compuesta por cinco elementos, las cadenas de texto *"Casa"*, *"2"* y *"Perro"* y los números enteros 345 y 99.

Al igual que en las listas, los elementos de las tuplas ocupan posiciones concretas, y mediante esa posición que ocupan podemos acceder directamente a los elementos. En la siguiente tabla te mostramos la relación de cada elemento con la posición que ocupa en la tupla:

Elemento	Posición
"Casa"	0
"2"	1
345	2
"Perro"	3
99	4

Tanto en las listas como en las tuplas, los elementos empiezan por la posición 0, no por la 1 como parecería lo obvio. Tenlo en cuenta cuando utilices tuplas.

En el siguiente ejercicio vamos a ver cómo definir una tupla y cómo acceder a sus elementos. El código fuente es el siguiente:

```
tupla = ('Casa','2',345,'Perro',99)
print("Elementos de la tupla: ",tupla)
print("Elemento de la posición 0: ",tupla[0])
print("Elemento de la posición 1: ",tupla[1])
print("Elemento de la posición 2: ",tupla[2])
print("Elemento de la posición 3: ",tupla[3])
print("Elemento de la posición 4: ",tupla[4])
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Elementos de la tupla:  ('Casa', '2', 345, 'Perro', 99)
Elemento de la posición 0:  Casa
Elemento de la posición 1:  2
Elemento de la posición 2:  345
Elemento de la posición 3:  Perro
Elemento de la posición 4:  99
```

Al igual que las listas, las tuplas tienen funciones asociadas al tipo de dato, pero el número de funciones disponibles es mucho menor que en las listas. Los métodos disponibles son los siguientes:

- **count**: cuenta el número de veces que aparece el elemento indicado como parámetro dentro de la tupla.

- Valor devuelto: número de ocurrencias del elemento.
- Parámetros: elemento del que se quiere saber el número de veces que aparece en la tupla.

- **index**: devuelve la posición de la primera ocurrencia de izquierda a derecha en la tupla del elemento pasado como parámetro.
  - Valor devuelto: posición que ocupa el elemento dentro de la tupla.
  - Parámetros: tiene tres parámetros, el primero es obligatorio y es el elemento a buscar, el segundo es opcional y se utiliza para indicar en qué posición empezar a buscar y por último, el tercer parámetro es opcional y se utiliza para indicar en qué posición finalizar la búsqueda.

A continuación vamos a hacer realizar un ejercicio para aprender a utilizar ambas funciones de las tuplas. El código fuente es el siguiente:

```
tupla = ('Casa','2',99,345,'Perro',99)
print("Elementos de la tupla: ",tupla)
print("Número de elementos 99: ",tupla.count(99))
print("Posición que ocupa el elemento Perro: ",tupla.index("Perro"))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Elementos de la tupla:  ('Casa', '2', 99, 345, 'Perro', 99)
Número de elementos 99:  2
Posición que ocupa el elemento Perro:  4
```

Las tuplas, al igual que las listas, permiten extraer pociones de ellas en otra tupla nueva. La extracción se realiza utilizando la siguiente instrucción:

Tupla[n:m]

La instrucción extraerá una nueva tupla que empezará en el índice n y terminará en el m-1. Tienes que tener en cuenta lo siguiente:

- n siempre tiene que ser menor que m.
- Si no se especifica el valor para n se supone que es 0.
- Si no se especifica el valor para m se supone que es el tamaño de la lista menos uno.

Veámoslo en un ejercicio, el código fuente es el siguiente:

```
tupla = (1,2,3,4,5,6,7,8,9)
print(tupla)
print(tupla[4:9])
print(tupla[:3])
print(tupla[2:])
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9)
(5, 6, 7, 8, 9)
(1, 2, 3)
(3, 4, 5, 6, 7, 8, 9)
```

La siguiente funcionalidad de las tuplas que vamos a explicar es el uso del operador "+" para realizar uniones de tuplas. El operador se utiliza de la siguiente manera:

TuplaConcatenada = Tupla1 + Tupla2

Además, vamos a aprender a utilizar una función para conocer el número de elementos que componen la tupla. Dicha función se llama *len* y devuelve un entero que indica el número de elementos que la componen. Se utiliza de la siguiente manera:

NumeroElementos = len(Tupla)

El código fuente del ejercicio es el siguiente:

```
tupla1 = (29, "Televisión",8763)
tupla2 = (1,2,3, "Videojuego")
print("Número elementos de tupla1: ",len(tupla1))
print("Tupla1: ", tupla1)
print("Número elementos de tupla2: ",len(tupla2))
print("Tupla2: ", tupla2)
tuplaconcatenada = tupla1 + tupla2
print("Número elementos de tuplaconcatenada: ",len(tuplaconcatenada))
print("tuplaconcatenada: ", tuplaconcatenada)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Número elementos de tupla1:  3
Tupla1:  (29, 'Televisión', 8763)
Número elementos de tupla2:  4
Tupla2:  (1, 2, 3, 'Videojuego')
Número elementos de tuplaconcatenada:  7
tuplaconcatenada:  (29, 'Televisión', 8763, 1, 2, 3, 'Videojuego')
```

La última funcionalidad que vamos a aprender de las tuplas es la utilización del operador "*", el cual nos va a permitir concatenar una tupla con ella misma un número finito de veces. El operador se utiliza de la siguiente forma:

TuplaResultante = Tupla * NúmeroEntero

La tupla resultante de la multiplicación será una tupla compuesta por tantas veces la *Tupla* como valor tenga el número entero.

El código fuente del ejercicio es el siguiente:

```
tupla = (1,2,3,4,5,6,7,8,9,0)
print(tupla)
tuplaresultante = tupla * 4
print(tuplaresultante)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
(1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
```

13.3 DICCIONARIOS

En este apartado vamos a hablar sobre los tipos de datos diccionarios, que son un conjunto ordenado de elementos cuyos índices no son numéricos sino identificadores. Al igual que las listas y las tuplas, los diccionarios puede contener datos de cualquier tipo. En otras palabras, los diccionarios son colecciones de elementos compuestos por una clave y un valor asociado, con la características de que las claves no pueden repetirse.

Los diccionarios en Python se delimitan por corchetes "{ }", con los elementos separados por comas y la clave separada del valor mediante dos puntos. Veamos un ejemplo de diccionario:

{"Clave1" : "Valor1", "Clave2" : "Valor2","Clave3" : "Valor3"}

El diccionario de ejemplo está compuesto por tres elementos, en la siguiente tabla te mostramos la relación entre la clave y el valor asociado:

Clave	Valor
"Clave1"	"Valor1"
"Clave2"	"Valor2"
"Clave3"	"Valor3"

Las claves de los diccionarios pueden ser de diferentes tipos de datos, aunque siempre deberán de ser datos inmutables. Los tipos de datos soportados en Python para ser claves de los diccionarios son:

- Cadenas de texto (str).
- Números (enteros, reales y complejos).
- Booleanos.
- Bytes.
- Tupla.

Aunque puedes utilizar todos ellos como clave, los más comunes son cadenas de texto y enteros.

13.3.1 Ejercicios

Los ejercicios con diccionarios los hemos dividido en dos grupos diferentes. El primer grupo de ejercicios consiste en la manipulación de los diccionarios y el segundo grupo consiste en el aprendizaje de los métodos propios de los tipos de datos diccionario.

13.3.1.1 MANIPULACIÓN

El primer ejercicio que vamos a realizar referente a diccionarios será crear uno y mostrar algunos elementos del mismo. Para acceder a los elementos del diccionario deberás de utilizar la clave del elemento. El código fuente es el siguiente:

```
diassemanaingles = {"Lunes" : "Monday",
                    "Martes" : "Tuesday",
                    "Miercoles" : "Wednesday",
                    "Jueves" : "Thursday",
                    "Viernes" : "Friday"}
```

```
print(diassemanaingles["Lunes"])
print(diassemanaingles["Miercoles"])
print(diassemanaingles["Viernes"])
```

El ejercicio define un diccionario en el que las claves de los elementos son el día de la semana en castellano y el valor es el día de la semana en inglés. El ejercicio muestra la traducción al inglés de los días Lunes, Miércoles y Viernes.

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Monday
Wednesday
Friday
```

El siguiente ejercicio consiste en añadir elementos al diccionario, eliminar elementos del diccionario y en modificar el valor de los elementos del diccionario.

La forma de añadir un elemento al diccionario es la siguiente:

Diccionario[NuevaClave] = NuevoValor

La forma de modificar el valor de un elemento del diccionario es la siguiente:

Diccionario[ClaveQueSeVaAModificar] = NuevoValor

La forma de eliminar un elemento del diccionario es la siguiente:

del Diccionario[ClaveElementoABorrar]

En el ejercicio utilizaremos el diccionario del ejercicio anterior añadiendo los días sábado y domingo, modificando el valor de algún elemento y borrando algún elemento. El código fuente es el siguiente:

```
diassemanaingles = {"Lunes" : "Monday",
                    "Martes" : "Tuesday",
                    "Miercoles" : "Wednesday",
                    "Jueves" : "Thursday",
                    "Viernes" : "Friday"}
print(diassemanaingles)
diassemanaingles["Sabado"] = "Saturday"
print(diassemanaingles)
diassemanaingles["Domingo"] = "Sunday"
print(diassemanaingles)
diassemanaingles["Lunes"] = "MondayBORRAR"
print(diassemanaingles)
```

```
del diassemanaingles["Lunes"]
print(diassemanaingles)
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
{'Lunes': 'Monday', 'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday'}
{'Lunes': 'Monday', 'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday', 'Sabado': 'Saturday'}
{'Lunes': 'Monday', 'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday', 'Sabado': 'Saturday', 'Domingo': 'Sunday'}
{'Lunes': 'Monday@@@@@', 'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday', 'Sabado': 'Saturday', 'Domingo': 'Sunday'}
{'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday', 'Sabado': 'Saturday', 'Domingo': 'Sunday'}
```

Es posible utilizar las funciones *len*, *max* y *min* con los diccionarios. La primera devolverá el número de elementos que contiene el diccionario; la segunda, el elemento con el valor mayor y la tercera, el elemento con el valor menor. El valor mayor y el valor menor serán devueltos siempre que pueda calcularse dependiendo de los elementos que componen el diccionario. Veámoslo en un ejercicio. El código fuente es el siguiente:

```
diassemanaingles = {"Lunes" : "Monday",
                    "Martes" : "Tuesday",
                    "Miercoles" : "Wednesday",
                    "Jueves" : "Thursday",
                    "Viernes" : "Friday"}
print("Número de elementos del diccionario: ",len(diassemanaingles))
print("Elemento mayor del diccionario: ",max(diassemanaingles))
print("Elemento menor del diccionario: ",min(diassemanaingles))
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Número de elementos del diccionario:  5
Elemento mayor del diccionario:  Viernes
Elemento menor del diccionario:  Jueves
```

13.3.1.2 MÉTODOS PROPIOS

El tipo de dato diccionario en Python posee una serie de funciones que nos permiten manipular los diccionarios realizando operaciones complejas de forma sencilla y con una simple instrucción. El formato de uso de la gran mayoría de las funciones es el siguiente:

Diccionario.NombreFuncion(Parámetros)

Veamos en detalle cada una de las partes:

- Diccionario: diccionario que ejecuta la función.
- NombreFuncion: nombre de la función que se quiere ejecutar.
- Parámetros: no todas las funciones tienen parámetros para ejecutarse, esta parte es dependiente de la función que se quiere ejecutar.

La funciones de listas que pone a nuestra disposición Python son las siguientes:

- **copy**: realiza una copia exacta del diccionario en uno nuevo.
  - Valor devuelto: diccionario copiado.
  - Parámetros: no tiene.
- **pop**: obtiene el valor de la clave pasada como parámetro y además elimina el elemento del diccionario.
  - Valor devuelto: valor del elemento o error en caso de no encontrar la clave en el diccionario.
  - Parámetros: clave a buscar en el diccionario.
- **popitem**: obtiene un elemento aleatorio del diccionario y lo elimina del mismo.
  - Valor devuelto: elemento del diccionario y en caso de que no tenga elementos el diccionario da un error.
  - Parámetros: no tiene.
- **get**: obtiene el valor de la clave pasada como parámetro.
  - Valor devuelto: devolverá el valor del elemento en caso de existir dicha clave y en caso de no existir devolverá "None". Existe la posibilidad de especificar mediante un segundo parámetro un valor diferente a "None" como retorno en caso de no existir la clave.
  - Parámetros: tiene dos parámetros, el primero es obligatorio y es la clave del elemento a buscar y el segundo es opcional y es el valor que se quiere retornar en caso de no existir dicha clave en el diccionario.
- **update**: realiza una actualización del diccionario utilizando otro diccionario. Aquellos elementos del diccionario que se utilizan para actualizar el principal sustituirán los existentes en el mismo y aquellos elementos que no existan serán añadidos al diccionario como nuevos elementos.

- Valor devuelto: nuevo diccionario.
- Parámetros: diccionario.

- **setdefault**: intenta insertar un elemento (clave y valor) en el diccionario. En caso de no existir dicho elemento, la función inserta y devuelve el valor del elemento y en caso de existir, lo único que hace es devolver el valor actual.
  - Valor devuelto: diccionario resultante.
  - Parámetros: dos parámetros que son la clave y valor. Es posible no especificar el valor y por defecto se insertará el valor *None* como valor del elemento.
- **clear**: elimina todos los elementos del diccionario.
  - Valor devuelto: diccionario vacío.
  - Parámetros: no tiene.
- **items**: devuelve un objeto iterable (que puede utilizarse en bucles. Lo veremos en próximos capítulos) compuesto por todos los elementos del diccionario.
  - Valor devuelto: objeto iterable compuesto por los elementos del diccionario.
  - Parámetros: no tiene.
- **keys**: devuelve un objeto iterable (que puede utilizarse en bucles. Lo veremos en próximos capítulos) compuesto por todas las claves del diccionario.
  - Valor devuelto: objeto iterable compuesto por las claves del diccionario.
  - Parámetros: no tiene.
- **values**: devuelve un objeto iterable (que puede utilizarse en bucles. Lo veremos en próximos capítulos) compuesto por todos los valores del diccionario.
  - Valor devuelto: objeto iterable compuesto por los valores del diccionario.
  - Parámetros: no tiene.

A continuación vamos a realizar un ejercicio para aprender a utilizar ambas funciones de los diccionarios. El código fuente es el siguiente:

```
diassemanaingles = {"Lunes" : "Monday",
                    "Martes" : "Tuesday",
                    "Miercoles" : "Wednesday",
                    "Jueves" : "Thursday",
                    "Viernes" : "Friday"}
print("Diccionario original: ",diassemanaingles)
diccionariocopia = diassemanaingles.copy()
print("Diccionario copy: ",diccionariocopia)
print("Valor del Lunes (pop): ", diassemanaingles.pop("Lunes"))
print("Diccionario después del pop: ",diassemanaingles)
print("Elemento al azar con popitem: ", diassemanaingles.popitem())
print("Diccionario después del popitem: ",diassemanaingles)
print("Valor del Martes (get): ",diassemanaingles.get("Martes"))
print("Valor del Lunes (get) (no existe): ",diassemanaingles.get("Lunes"))
print("Valor del Lunes (get) (no existe): ",diassemanaingles.get("Lunes","No
existe"))
diassemanaingles.update({"Lunes":"MondayNUEVO","Martes":"TuesdayNUEVO"})
print("Diccionario después del update: ",diassemanaingles)
print("setdefault del Sábado: ",diassemanaingles.setdefault("Sabado","Saturd
ay"))
print("Diccionario después del setdefault (nuevo elemento): ",diassemanaingles)
print("setdefault del Lunes: ",diassemanaingles.setdefault("Lunes","Lunes"))
print("Diccionario después del setdefault (elemento existente): ",diassemanain-
gles)
print("Elemento iterable (items): ",diassemanaingles.items())
print("Elemento iterable (claves): ",diassemanaingles.keys())
print("Elemento iterable (valores): ",diassemanaingles.values())
print("Diccionario después del clear: ",diassemanaingles.clear())
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
Diccionario original:  {'Lunes': 'Monday', 'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday'}
Diccionario copy:  {'Lunes': 'Monday', 'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday'}
Valor del Lunes (pop):  Monday
Diccionario después del pop:  {'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Viernes': 'Friday'}
Elemento al azar con popitem:  ('Viernes', 'Friday')
Diccionario después del popitem:  {'Martes': 'Tuesday', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday'}
Valor del Martes (get):  Tuesday
Valor del Lunes (get) (no existe):  None
Valor del Lunes (get) (no existe):  No existe
Diccionario después del update:  {'Martes': 'TuesdayNUEVO', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Lunes': 'MondayNUEVO'}
setdefault del Sábado:  Saturday
Diccionario después del setdefault (nuevo elemento):  {'Martes': 'TuesdayNUEVO', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Lunes': 'MondayNUEVO', 'Sabado': 'Saturday'}
setdefault del Lunes:  MondayNUEVO
Diccionario después del setdefault (elemento existente):  {'Martes': 'TuesdayNUEVO', 'Miercoles': 'Wednesday', 'Jueves': 'Thursday', 'Lunes': 'MondayNUEVO', 'Sabado': 'Saturday'}
Elemento iterable (items):  dict_items([('Martes', 'TuesdayNUEVO'), ('Miercoles', 'Wednesday'), ('Jueves', 'Thursday'), ('Lunes', 'MondayNUEVO'), ('Sabado', 'Saturday')])
Elemento iterable (claves):  dict_keys(['Martes', 'Miercoles', 'Jueves', 'Lunes', 'Sabado'])
Elemento iterable (valores):  dict_values(['TuesdayNUEVO', 'Wednesday', 'Thursday', 'MondayNUEVO', 'Saturday'])
Diccionario después del clear:  None
```

14

COMENTARIOS DE CÓDIGO

En este apartado vamos a adentrarnos en los comentarios de código fuente, explicando para qué sirven, cómo se utilizan en Python y una serie de recomendaciones a la hora de utilizarlos.

14.1 ¿QUÉ SON?

Los comentarios de código son una utilidad de programación que permiten explicar el código fuente. Los usos más comunes que se le dan a los comentarios son para explicar el funcionamiento de una porción de código fuente, los parámetros de entrada o salida de una función o método, junto con lo que hace dicha función o método. Por tanto, los comentarios son una forma de añadir documentación a los propios ficheros de código fuente.

Podemos decir que los comentarios de código son utilizados para hacer el código fuente más entendible, legible, reutilizable y mantenible.

Hay dos formas diferentes de añadir los comentarios al código fuente:

- **Comentarios de una única línea**: el comentario es escrito únicamente en una única línea y normalmente hace referencia a una única instrucción del código fuente.
- **Bloque de comentarios**: el comentario es escrito en varias líneas y normalmente hacen referencia a un bloque de instrucciones.

Los comentarios son ignoradas por el ordenador a la hora de ejecutar el programa, por tanto, puedes utilizar tantos comentarios como desees o estimes oportuno, aunque como veremos en los siguientes puntos, es recomendable seguir una serie de buenas prácticas a la hora de utilizarlos.

14.2 COMENTARIOS DE CÓDIGO EN PYTHON

En Python podemos utilizar los dos tipos de comentarios de código fuente que hemos visto en el punto anterior, veamos cómo se utilizan:

- **Comentarios de una única línea**: hay que poner el carácter '#' al comienzo de la línea para indicar que esa línea es un comentario. Veamos un ejemplo:

```
# Petición de los números a restar
minuendo = float(input("Introduzca el minuendo: "))
sustraendo = float(input("Introduzca el sustraendo: "))
# Muestra por pantalla la resta de los números solicitados
print("Resultado de la resta: ", minuendo - sustraendo)
```

- **Bloque de comentarios**: hay que poner los caracteres '"""' (triple comilla doble) al comienzo de la primera línea del bloque y al final de la última línea del bloque.

```
""" Pide el minuendo y el sustraendo y
muestra por pantalla el resultado de la resta"""
minuendo = float(input("Introduzca el minuendo: "))
sustraendo = float(input("Introduzca el sustraendo: "))
print("Resultado de la resta: ", minuendo - sustraendo)
```

14.3 RECOMENDACIONES Y BUENAS PRÁCTICAS

Una vez hemos entendido qué son los comentarios, para qué valen y cómo se utilizan, tenemos que tener presente una serie de recomendaciones a la hora de utilizarlos. Son las siguientes:

1. **No llenar todo el código fuente de comentarios**: la documentación dentro del código fuente es importante, pero es necesario seguir una serie de buenas prácticas a la hora de escribir código fuente y una de ellas es que el código fuente debe de ser lo suficientemente claro que

se explique sólo con leerlo, ello implica por ejemplo la utilización de nombres descriptivos para las variables, funciones, clases, etc.

2. **Utilización de comentarios útiles**: siempre que uses un comentario es porque realmente se necesita, si no lo haces así llenarás el código fuente de comentarios que no aportan nada.

3. **Mantenimiento de comentarios**: al igual que el código fuente, los comentarios hay que irlos manteniendo, por tanto, cada vez que cambies el código fuente tienes que revisar que los comentarios siguen siendo válidos para el código fuente que comentan.

4. **Añadir comentarios mientras codificas**: nunca dejes para el final el añadir comentarios al código, ve haciéndolo mientras escribes el código fuente.

5. **No comentar código eliminado**: una práctica muy común en el mundo del desarrollo es meter dentro de comentarios código fuente que ya no se utiliza para no "perderlo". No lo hagas, para eso están los sistemas de control de código fuente.

6. **Mismo estilo**: sigue siempre el mismo estilo a la hora de añadir comentarios al código fuente.

7. **Utiliza etiquetas dentro de los comentarios**: una buena práctica dentro de los comentarios es añadir etiquetas, no sólo para buscar más rápidamente el comentario, sino también para resaltar algo dentro del mismo. Las siguientes etiquetas son un estándar dentro de los comentarios:

   - **TODO**: indica que hay algo pendiente de hacer en esa porción de código fuente.
   - **FIXME**: indica que el código fuente no es correcto y tiene que ser arreglado.
   - **NOTE**: añade un recordatorio o nota al comentario.

15

CONTROL DEL FLUJO

En este capítulo vamos a explicar las estructuras de programación que se utilizarán para realizar el control del flujo de los programas, las bifurcaciones. Para ello vamos a explicarte también los operadores relacionales y la indentación en Python.

En programación, podemos definir bifurcación como la decisión de tomar un camino u otro dentro de la ejecución de un programa.

La decisión de seguir un camino o seguir otro camino viene dada por la evaluación de una condición que, dependiendo del tipo de instrucción de bifurcación que sea, será una comprobación de una comparación con TRUE o FALSE o por el valor propiamente dicho del elemento utilizado en dicha comprobación.

En este apartado vamos a explicarte las diferentes instrucciones de bifurcación que existen en programación. Tienes que tener muy claro qué implica su uso y cómo se utiliza cada una de ellas.

Independientemente del tipo de bifurcación que estés utilizando, tienes que especificar la condición a evaluar y agrupar el conjunto de instrucciones que forman cada camino posible a tomar en el camino correspondiente, indicando el inicio y el fin de cada camino.

15.1 OPERADORES RELACIONALES

Los operadores relacionales son operadores que permiten comparar dos elementos, en caso de que la comparación sea cierta la operación devolverá *True*, en

caso de no ser cierta devolverá *False*. En Python tenemos disponibles los siguientes operadores relacionales:

- **Menor que**: el operador "<" se utiliza para comprobar si el primer elemento es menor que el segundo elemento.
- **Mayor que**: el operador ">" se utiliza para comprobar si el primer elemento es mayor que el segundo elemento.
- **Menor o igual que**: el operador "<=" se utiliza para comprobar si el primer elemento es menor o igual que el segundo elemento.
- **Mayor o igual que**: el operador ">" se utiliza para comprobar si el primer elemento es mayor o igual que el segundo elemento.
- **Igual que**: el operador "==" se utiliza para comprobar si el primer elemento y el segundo son iguales.
- **Distinto que**: el operador "!=" se utiliza para comprobar si el primer elemento y el segundo son distintos.

En la siguiente tabla te mostramos un ejemplo de cada uno de los operadores y el resultado de la comparación:

Operación	Resultado
4<6	True
3>6	False
5<=5	True
7>=4	True
3==3	True
1!=1	False

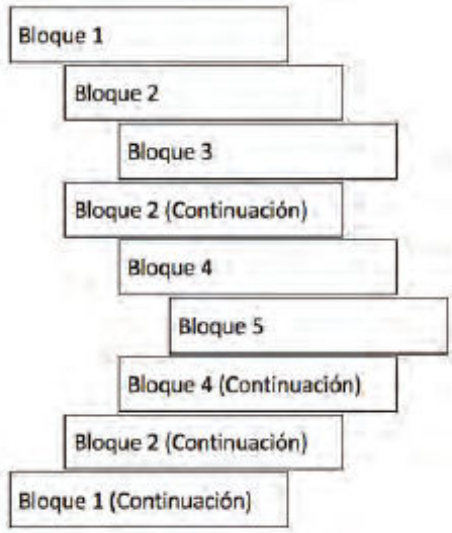
15.2 BLOQUES E INDENTACIÓN

Antes de entrar en detalle de estructuras para el control del flujo de los programas es necesario que entiendas dos conceptos utilizados en desarrollo de software: bloque de código e indentación.

Los bloques de código fuente son un conjunto de sentencias de código fuente que están delimitadas por un inicio y un fin. La forma de delimitar los bloques de código fuente depende del lenguaje de programación.

La indentación o tabulación es una forma de incrementar la legibilidad del código fuente mediante el desplazamiento de las sentencias hacia la derecha insertando espacios en blanco o tabuladores, separándola del margen izquierdo. A diferencia de otros lenguajes, en Python la indentación tiene significado, y no es únicamente una cuestión estética o de legibilidad, es utilizada también como forma de delimitar los bloques de código fuente, es por tanto, parte de la sintaxis del lenguaje.

La siguiente imagen muestra una secuencia de bloques de código indentados:



La imagen muestra un total de cinco bloques con diferentes indentaciones. Existe un bloque principal, bloque 1, que tiene dentro el bloque 2, que a su vez tiene dentro el bloque 3 y el bloque 4, que tiene a su vez dentro el bloque 5. Tal y como puedes observar en la imagen, los bloques pueden aparecer en medio de otros bloques, como puede ser el caso del bloque 3, aparece entre medias de las sentencias del bloque 2, es decir, el bloque 2 tiene sentencias antes y después de las sentencias del bloque 3.

15.3 IF / ELIF /ELSE

La sentencia *if* forma parte de un conjunto de sentencias encargadas de bifurcar la ejecución del código fuente dependiendo de una serie de condiciones. Por tanto, la sentencia *if* te va a permitir tomar decisiones, evaluará una condición de una operación lógica y dependiendo del resultado, ejecutará un código fuente u otro.

Veamos los diferentes comandos relacionados con la sentencia *if* y su significado:

- **if**: te va a permitir generar un bloque de código que se ejecutará si se cumple la condición de entrada que tiene.
- **elif**: te va a permitir generar un camino alternativo con una condición de entrada.
- **else**: te va a permitir generar un camino alternativo que se ejecutará siempre que no se hayan cumplido las condiciones de los posibles caminos de las instrucciones *if* y *elif*.

El uso de las sentencias *if*, *elif* y *else* es el siguiente:

```
If OperaciónLógica1
        BloqueInstrucciones1

elif OperaciónLógica2
        BloqueInstrucciones2

else
        BloqueIntrucciones3
```

Veamos el significado de cada una de ellas:

- **if OperaciónLógica1**: la sentencia *if* evaluará el resultado de la *OperaciónLógica1*, en caso de que el resultado de la operación sea *True* se ejecutará *BloqueInstrucciones1*, en caso contrario se evaluaría la sentencia *elif*.
- **elif OperaciónLógica2**: la sentencia *elif* evaluará el resultado de la *OperaciónLógica2*, en caso de que el resultado de la operación sea *True* se ejecutará *BloqueInstricciones2*, en caso contrario se ejecutaría la sentencia *else*.
- **else**: la sentencia *else* no realiza la evaluación de ninguna operación, simplemente ejecutará *BloqueInstrucciones3*.

A continuación te mostramos un ejemplo con valores reales en las operaciones, vamos a suponer que *GenerarNumero* te da un número al azar entre 0 y 1000:

```
numero1 = GenerarNumero

numero2 = GenerarNumero

if numero1>numero2
        BloqueInstrucciones1

elif numero1!=numero2
        BloqueInstrucciones2

else
        BloqueInstrucciones3
```

Analicemos el flujo del ejemplo:

- **if numero1>numero2**: en caso de que el primer número sea mayor que el segundo se ejecutará el bloque de instrucciones llamado *BloqueInstrucciones1*.
- **elif numero1<numero2**: en caso de que el primer número sea menor que el segundo se ejecutará el bloque de instrucciones llamado *BloqueInstrucciones2*.
- **else**: en caso de que el primer número sea igual que el segundo número se ejecutará el bloque de instrucciones llamado *BloqueInstrucciones3*. Tal y como puedes observar, en este caso no es necesario establecer una condición, ya que a la hora de comparar números únicamente existen tres posibilidades, que el primero sea menor que el segundo, que ambos sean iguales o que el primero sea mayor que el segundo.

Tal y como puedes estar pensando es posible incluir *ifs* dentro de otros *ifs*, teniendo así *ifs* anidados.

Vamos a hacer ahora una serie de ejercicios para afianzar los conocimientos que acabamos de explicar sobre la sentencia *if*.

En el primer ejercicio vas a familiarizarte con el uso de la sentencia *if* únicamente. El programa evaluará una condición y si el resultado es *True* ejecutará el bloque de código fuente que está dentro del mismo y en caso contrario no lo ejecutará. El ejercicio va a pedir la introducción de un número entre el 1 y el 1000 y la sentencia *if* evaluará si el número introducido es mayor que 400, en ese caso lo indicará por pantalla. Para finalizar, el número es escrito por pantalla. El código fuente es el siguiente:

```
numero = int(input("Escriba un numero del 1 al 1000: "))
if numero<400:
    print("¡El numero que has escrito es menor que 400!")
print("Has escrito el numero " + str(numero))
```

El programa tendrá dos posibles ejecuciones, aquella en la que el valor introducido es superior a 400 y aquella en la que el valor introducido es inferior a 400. La siguiente imagen muestra la ejecución en la que el número introducido es superior a 400:

```
Escriba un numero del 1 al 1000: 401
Has escrito el numero 401
```

La siguiente imagen muestra la ejecución en la que el número introducido es inferior a 400:

```
Escriba un numero del 1 al 1000: 399
¡El numero que has escrito es menor que 400!
Has escrito el numero 399
```

El segundo ejercicio consiste en comprobar si una cadena de texto termina o no en vocal. La operación lógica puede ser sustituida por un valor devuelto por una función o método de clase, como por ejemplo puede ser el que vimos en el capítulo sobre las cadenas de texto que comprobaba la terminación de una cadena de texto. Además, con el ejercicio vas a utilizar el operador *OR* para añadir todas las vocales a la operación lógica a evaluar. El código fuente es el siguiente:

```
cadena = input("Introduzca una cadena de texto: ")
if cadena.endswith("a") or cadena.endswith("e") or cadena.endswith("i") or cade-
na.endswith("o") or cadena.endswith("u"):
    print("¡La cadena de texto acaba en vocal!")
print("Has escrito: " + cadena)
```

El programa tendrá dos posibles ejecuciones, aquella en la que la cadena de texto introducida acaba en vocal y aquella en la que el valor introducido no acaba en vocal. La siguiente imagen muestra la ejecución en la que la cadena de texto acaba en vocal:

```
Introduzca una cadena de texto: Valeria
¡La cadena de texto acaba en vocal!
Has escrito: Valeria
```

La siguiente imagen muestra la ejecución en la que la cadena de texto no acaba en vocal:

```
Introduzca una cadena de texto: Python
Has escrito: Python
```

El tercer ejercicio consiste en aprender a utilizar la sentencia *else*. Para ello, vamos a ampliar el primer ejercicio que hemos hecho incluyendo la sentencia *else* y un bloque de código dentro de la misma que se ejecutará siempre que la condición de la sentencia *if* no se cumpla. El código fuente sería el siguiente:

```
numero = int(input("Escriba un numero del 1 al 1000: "))
if numero<400:
    print("¡El numero que has escrito es menor que 400!")
else:
    print("¡El número que has escrito es mayor o igual a 400!")
print("Has escrito el numero " + str(numero))
```

Al igual que el primer ejercicio, éste tendrá dos posibles ejecuciones, aquella en la que el número sea menor que 400 en la que mostrará que es menor y aquella en la que sea mayor o igual a 400 en la que así lo mostrará. La siguiente imagen muestra la ejecución en la que el número es menor que 400:

```
Escriba un numero del 1 al 1000: 123
¡El numero que has escrito es menor que 400!
Has escrito el numero 123
```

La siguiente imagen muestra la ejecución en la que el número es mayor o igual que 400:

```
Escriba un numero del 1 al 1000: 897
¡El número que has escrito es mayor o igual a 400!
Has escrito el numero 897
```

El cuarto ejercicio consiste en aprender a usar sentencias *if* y *else* anidadas. El ejercicio consiste en introducir dos números y realizar una suma de ellos, el resultado se utilizará para evaluar diferentes *ifs* y mostrar mensajes por pantalla. El código fuente es el siguiente:

```
sumando1 = int(input("Escriba el primer sumando: "))
sumando2 = int(input("Escriba el segundo sumando: "))
resultado = sumando1 + sumando2
print("El resultado de la suma es: " + str(resultado))
if resultado%2==0:
    if resultado>=1000:
        print("¡El resultado de la suma es par y mayor o igual que 1000!")
```

```
        else:
            print("¡El resultado de la suma es par y menor que 1000!")
    else:
        if resultado>=1000:
            print("¡El resultado de la suma es impar y mayor o igual que 1000!")
        else:
            print("¡El resultado de la suma es impar y menor que 1000!")
```

Tal y como puedes observar, el programa puede ir por cuatro sitios diferentes dependiendo del resultado de la suma de ambos sumandos. Las cuatro posibilidades son:

1. El resultado de la suma es par y mayor o igual que 1000.
2. El resultado de la suma es par y menor que 1000.
3. El resultado de la suma es impar y mayor o igual que 1000.
4. El resultado de la suma es impar y menor que 1000.

La siguiente imagen muestra una ejecución del caso 1:

```
Escriba el primer sumando: 346
Escriba el segundo sumando: 876
El resultado de la suma es: 1222
¡El resultado de la suma es par y mayor o igual que 1000!
```

La siguiente imagen muestra una ejecución del caso 2:

```
Escriba el primer sumando: 111
Escriba el segundo sumando: 333
El resultado de la suma es: 444
¡El resultado de la suma es par y menor que 1000!
```

La siguiente imagen muestra una ejecución del caso 3:

```
Escriba el primer sumando: 654
Escriba el segundo sumando: 871
El resultado de la suma es: 1525
¡El resultado de la suma es impar y mayor o igual que 1000!
```

La siguiente imagen muestra una ejecución del caso 4:

```
Escriba el primer sumando: 555
Escriba el segundo sumando: 222
El resultado de la suma es: 777
¡El resultado de la suma es impar y menor que 1000!
```

En el quinto ejercicio del capítulo vamos a aprender a manejar la sentencia *elif*, que nos va permitir introducir un camino alternativo al anterior *if* pero con una condición. Esto es lo que le diferencia de la sentencia *else*. Mediante la sentencia *elif* vas a poder crear caminos alternativos infinitos dependiendo del cumplimiento de la condición de entrada y un camino que se tomará en caso de que no se cumpla ninguna de las condiciones de entrada a los caminos previos. En el ejercicio vamos a introducir dos números y vamos a compararlos utilizando las sentencias *if* y *elif* junto con *else*. El código fuente es el siguiente:

```
numero1 = int(input("Escriba el primer número: "))
numero2 = int(input("Escriba el segundo número: "))
if numero1==numero2:
    print("¡Ambos número son iguales!")
elif numero1>numero2:
    print("¡El primer número es mayor que el segundo!")
else:
    print("¡El primer número es menor que el segundo!")
```

Dependiendo de los números introducidos el mensaje por pantalla será diferente. En caso de que ambos números sean iguales, se mostrará el mensaje dentro del bloque de la sentencia *if*. En caso de que el primer número sea mayor que el segundo, se mostrará el mensaje dentro del bloque de la sentencia *elif*. Por último, en caso de que el primer número sea menor que el segundo, se mostrará el mensaje dentro del bloque de la sentencia *else*.

La siguiente imagen muestra una ejecución en la que ambos números son iguales:

```
Escriba el primer número: 67
Escriba el segundo número: 67
¡Ambos número son iguales!
```

La siguiente imagen muestra una ejecución en la que el primer número es mayor que el segundo:

```
Escriba el primer número: 76
Escriba el segundo número: 32
¡El primer número es mayor que el segundo!
```

La siguiente imagen muestra una ejecución en la que el primer número es menor que el segundo:

```
Escriba el primer número: 34
Escriba el segundo número: 78
¡El primer número es menor que el segundo!
```

16

BUCLES

En este capítulo vamos a explicarte todo lo referente a las estructuras de programación conocidas como bucles. En programación podemos definir un bucle como la repetición de la ejecución de un conjunto de instrucciones donde a cada repetición del conjunto de instrucciones se llama iteración.

Existen diferentes tipos de bucles. Tienes que tener muy claro cuándo usar un bucle o cuándo usar otro, ya que cada uno de ellos está recomendado para ser usado en diferentes contextos, que varían dependiendo del tipo de condición de parada de ejecución del bucle que se necesite, en otras palabras, utilizarás un bucle u otro dependiendo de la forma en la que necesites indicarle el número de iteraciones a realizar.

Independientemente de qué tipo de bucle estés utilizando se debe indicar el punto de inicio, el punto final y el número de iteraciones que van a realizarse, aunque para cada tipo de bucle se especifica de una forma distinta todos tienen estos elementos comunes. Resumiendo, los bucles están compuestos por los siguientes componentes:

- Punto de inicio del bucle.
- Punto de fin del bucle.
- Número de iteraciones.
- Bloque de instrucciones a ejecutar.

En el capítulo vamos a aprender a manejar dos tipos de bucles diferentes, los bucles *for* y los bucles *while*. Cada uno se utiliza con un propósito diferente y la forma de especificar el punto de inicio, final y número de iteraciones es diferente.

¡Veámoslos!

16.1 FOR

El tipo de bucle *for* está recomendado para contextos en los que se sabe el número de iteraciones exactas que se van a dar en su ejecución, es decir, es un bucle que busca ejecutar un conjunto de instrucciones de forma repetitiva hasta llegar al número máximo de iteraciones definidas.

En Python, los bucles *for* se ejecutan sobre elementos iterables, como pueden ser listas, tuplas, cadenas de texto o diccionarios. El número de iteraciones que se ejecutarán dependerá del número de elementos de los que está compuesto el elemento iterable.

Los bucles *for* tienen la siguiente sintaxis:

for Variable in ColeccionIterable:
***   BloqueInstrucciones***

Veamos los elementos en detalle:

- **for**: indicador de comienzo del bucle.
- **Variable**: variable que almacena el elemento sobre el que se está iterando de *ColeccionIterable*.
- **in**: indicador que se utiliza para definir el elemento iterable sobre el que se ejecutará el bucle *for*.
- **ColeccionIterable**: elemento sobre el que se ejecuta el bucle.
- **BloqueInstrucciones**: conjunto de instrucciones que se ejecutarán en cada iteración.

El primer ejercicio consiste en recorrer una lista utilizando el bucle *for* y mostrando los elementos por pantalla. En cada iteración del bucle se mostrará por pantalla el elemento de la lista sobre el que se está iterando. El código fuente es el siguiente:

```
lista = ["a","b","c","d","e","f","g","h","i","j","k","l","m","n","ñ","o","p","q"
,"r","s","t","u","v","w","x","y","z"]
for item in lista:
    print(item, end=" ")
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
a b c d e f g h i j k l m n ñ o p q r s t u v w x y z
```

El segundo ejercicio va a consistir en mostrar la lista del ejercicio anterior un número de veces determinadas por otro bucle *for*. Es lo que se conoce como bucles anidados. Los bucles anidados consisten en la utilización de bucles como parte de los bloques de instrucciones de otros bucles. El bucle que se encuentra dentro de otro bucle se suele llama bucle interno o interior, mientras que el bucle que contiene un bucle interior se llama bucle externo o exterior. Puedes tener el nivel de anidamiento que necesites, es decir, un bucle dentro de otro bucle, que a su vez esté dentro de otro bucle que está dentro de otro bucle, etc. El código fuente es el siguiente:

```
listaabecedario = ["a","b","c","d","e","f","g","h","i","j","k","l","m","n","ñ","
o","p","q","r","s","t","u","v","w","x","y","z"]
listaiteraciones = [1,2,3,4,5]
for item in listaiteraciones:
    print("Iteración número: " + str(item))
    for letra in listaabecedario:
        print(letra, end=" ")
    print("\n")
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Iteración número: 1
a b c d e f g h i j k l m n ñ o p q r s t u v w x y z

Iteración número: 2
a b c d e f g h i j k l m n ñ o p q r s t u v w x y z

Iteración número: 3
a b c d e f g h i j k l m n ñ o p q r s t u v w x y z

Iteración número: 4
a b c d e f g h i j k l m n ñ o p q r s t u v w x y z

Iteración número: 5
a b c d e f g h i j k l m n ñ o p q r s t u v w x y z
```

El siguiente ejercicio consiste en la utilización de una lista con elementos de tipos completamente diferentes para iterar con el bucle *for*. En el ejercicio vamos a utilizar números enteros, cadenas de texto y una lista como elementos de la lista iterable. El código fuente es el siguiente:

```
lista = [99,"Casa",["Hola","Adios"],"Perro","Gato", 34]
for item in lista:
    print(item)
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
99
Casa
['Hola', 'Adios']
Perro
Gato
34
```

El último ejercicio que vamos a realizar sobre el bucle *for* consiste en el aprendizaje de una instrucción que permite obtener una lista secuencial de elementos enteros empezando en 0 hasta el valor que se le indique. Dicha sentencia es *range*. El ejercicio consiste en dos bucles *for* anidados que recorrerán dos listas devueltas por la sentencia *range* mostrando el valor de ambas listas. El bucle interno se ejecuta tantas veces como iteraciones tenga el bucle externo, en este caso el bucle interno se ejecutará completamente un total de cinco veces. El código fuente es el siguiente:

```
for item in range(5):
    for item2 in range(3):
        print("Iteración " + str(item) + "," + str(item2))
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
Iteración 0,0
Iteración 0,1
Iteración 0,2
Iteración 1,0
Iteración 1,1
Iteración 1,2
Iteración 2,0
Iteración 2,1
Iteración 2,2
Iteración 3,0
Iteración 3,1
Iteración 3,2
Iteración 4,0
Iteración 4,1
Iteración 4,2
```

El flujo de ejecución del ejercicio es el siguiente:

- Iteración del bucle exterior sobre el primer elemento de la primera lista (0).
  - Iteración de todos los elementos de la lista del bucle interior(0 al 3).
- Iteración del bucle exterior sobre el segundo elemento de la primera lista (1).
  - Iteración de todos los elementos de la lista del bucle interior(0 al 3).

- Iteración del bucle exterior sobre el tercer elemento de la primera lista (2).
  - Iteración de todos los elementos de la lista del bucle interior(0 al 3).
- Iteración del bucle exterior sobre el cuarto elemento de la primera lista (3).
  - Iteración de todos los elementos de la lista del bucle interior(0 al 3).
- Iteración del bucle exterior sobre el quinto elemento de la primera lista (4).
  - Iteración de todos los elementos de la lista del bucle interior(0 al 3).

16.2 WHILE

El tipo de bucle *while* está recomendado para contextos en los que no se sabe exactamente el número de iteraciones que se tienen que ejecutar, pero sí se sabe que hay que ejecutar iteraciones hasta que se deje de cumplir una condición.

La condición que se utiliza para comprobar si se tiene que ejecutar una iteración deberá de ser *true* para que se ejecute, la ejecución del bucle finalizará si la condición tiene el valor *false*. La condición es comprobada en cada iteración del bucle. Las variables que se utilizan en la condición del bucle se llaman variables de control.

Los bucles *while* tienen la siguiente sintaxis:

```
while Condición:
        BloqueInstrucciones
```

Veamos los elementos en detalle:

- **while**: indicador de comienzo del bucle.
- **Condición**: condición que debe de cumplirse para que siga repitiéndose la ejecución del bucle.
- **BloqueInstrucciones**: conjunto de instrucciones que se ejecutarán en cada iteración.

En la utilización de bucles *while* puedes encontrarte con los siguientes problemas:

- **Bucles que no se ejecutan nunca**: pon especial atención a la inicialización de las variables de control del bucle para asegurarte de que la condición es *true*, ya que si la condición es *false* desde el principio, el bucle jamás se ejecutará.
- **Bucles infinitos**: pon especial atención a la modificación de los valores de las variables de control del bucle dentro del bucle, ya que, si dichos valores no se ven alterados jamás, el bucle nunca parará de ejecutarse.

El primer ejercicio de bucle *while* va a consistir en realizar un bucle en el que se muestra el valor de la variable de control que se comprueba en cada iteración. El código fuente es el siguiente:

```
i = 0
while i<10:
    print(i,end=" ")
    i = i + 1
```

Vamos a analizar el código del ejercicio. La primera instrucción inicializa la variable de control del bucle estableciendo el valor a 0. La condición de ejecución del bucle será que si la variable de control es menor que 10 se ejecutará una iteración, en caso contrario el bucle finalizará. Dentro del bucle puedes ver como se actualiza el valor de la variable de control para no entrar en un bucle infinito: en cada iteración se le suma 1 a su valor.

La siguiente imagen muestra la ejecución del código fuente anterior:

```
0 1 2 3 4 5 6 7 8 9
```

En el segundo ejercicio vamos a enseñarte a utilizar una variable de control diferente a un número entero (que son los más comunes). Vamos a utilizar una variable de tipo booleano. El bucle se ejecutará siempre que el valor de la variable sea *True*. En el momento en el que la variable sea *False* el bucle parará. Dentro del bucle vamos a solicitar un número que utilizaremos para comprobar si tiene que finalizar o no la ejecución del bucle. El valor de la variable de control cambiará a *False* en el momento en que el número introducido sea inferior a 10. El código fuente es el siguiente:

```
pedirnumero = True
while pedirnumero:
    valor = int(input("Introduce un entero inferior a 10: "))
```

```
        if valor<10:
            pedirnumero = False
    print("FIN: ¡Ha introducido un valor inferior a 10!")
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
Introduce un entero inferior a 10: 69
Introduce un entero inferior a 10: 54
Introduce un entero inferior a 10: 1112
Introduce un entero inferior a 10: 10
Introduce un entero inferior a 10: 3
FIN: ¡Ha introducido un valor inferior a 10!
```

En el apartado anterior en el que explicamos el bucle *for*, te explicamos lo que eran los bucles anidados e hicimos un ejercicio anidando dos bucles *for*. El siguiente ejercicio consiste en anidar dos bucles *while*. Vamos a realizar el mismo ejercicio que el ejercicio de anidamiento de bucles *for* del apartado anterior pero utilizando bucles *while*. Tienes que tener en cuenta un aspecto muy importante, y es que en cada iteración del bucle exterior tienes que inicializar la variable de control del bucle interior. El código fuente es el siguiente:

```
item1 = 0
while item1<5:
    item2 = 0
    while item2<3:
        print("Iteración " + str(item1) + "," + str(item2))
        item2 = item2 + 1
    item1 = item1 + 1
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
Iteración 0,0
Iteración 0,1
Iteración 0,2
Iteración 1,0
Iteración 1,1
Iteración 1,2
Iteración 2,0
Iteración 2,1
Iteración 2,2
Iteración 3,0
Iteración 3,1
Iteración 3,2
Iteración 4,0
Iteración 4,1
Iteración 4,2
```

El flujo de ejecución del ejercicio es el siguiente:

- Inicialización variable de control bucle exterior.
- Primera iteración del bucle exterior.
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Segunda iteración del bucle exterior.
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Tercera iteración del bucle exterior.
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Cuarta iteración del bucle exterior.
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Quinta iteración del bucle exterior.
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Fin

Por último, vamos a hacer el mismo ejercicio pero anidando un bucle *for* y un bucle *while*. El ejercicio va a ser igual que el de antes en lo que a funcionamiento se refiere. El código fuente es el siguiente:

```
for item1 in range(5):
    item2 = 0
    while item2<3:

        print("Iteración " + str(item1) + "," + str(item2))
        item2 = item2 + 1
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
Iteración 0,0
Iteración 0,1
Iteración 0,2
Iteración 1,0
Iteración 1,1
Iteración 1,2
Iteración 2,0
Iteración 2,1
Iteración 2,2
Iteración 3,0
Iteración 3,1
Iteración 3,2
Iteración 4,0
Iteración 4,1
Iteración 4,2
```

El flujo de ejecución del ejercicio es el siguiente:

- Iteración del bucle exterior sobre el primer elemento de la primera lista (0).
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Iteración del bucle exterior sobre el primer elemento de la primera lista (1).
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Iteración del bucle exterior sobre el primer elemento de la primera lista (2).
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Iteración del bucle exterior sobre el primer elemento de la primera lista (3).
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Iteración del bucle exterior sobre el primer elemento de la primera lista (4).
- Inicialización variable de control bucle interior.
- Todas las iteraciones del bucle interior.
- Fin

17

FUNCIONES

En este capítulo vamos a explicarte qué son las funciones en desarrollo de software, para qué se usan y cuáles son sus bondades.

Una función es un bloque de código fuente que contiene un conjunto de instrucciones que realiza algo concreto y que puede ser utilizada desde el código fuente que escribes tantas veces como necesites.

Las funciones tienen las siguientes características:

- Tienen un nombre para ser identificadas.
- Ejecutan una tarea concreta.
- Puede recibir parámetros para su ejecución.
- Puede devolver un resultado como resultado de su ejecución.

Tanto los parámetros que pueden recibir como el resultado que devuelven son características de las funciones opcionales, es decir, podemos encontrar funciones que no reciben datos y que no devuelven nada, funciones que reciben datos y que no devuelven nada, funciones que no reciben datos y que devuelven datos y por último funciones que reciben datos y que devuelven datos.

Las funciones pertenecen a un tipo de programación que se llama programación estructurada, y tienen las siguientes ventajas:

- **Modularización**: permiten segmentar el código fuente complejo en un conjunto de módulos que hace que el código fuente sea más sencillo de leer, depurar y mantener.
- **Reutilización**: permiten reutilizar las funciones en otros programas o módulos del mismo programa.

Resumiendo, el uso de funciones simplifica el código fuente, provoca una mejor organización del mismo y además permite reutilizar el código fuente que escribes.

La sintaxis de las funciones en Python es la siguiente:

```
def NombreFuncion (parámetros):
        BloqueInstrucciones
        return ValorRetorno
```

Veamos los elementos en detalle:

- **def**: indicador de definición de función.
- **NombreFuncion**: nombre que tendrá la función. Te aconsejamos que utilices nombres de funciones descriptivos que representen lo que la función hace.
- **parámetros**: conjunto de elementos de entrada que tiene la función. Los parámetros son opcionales, es decir, puede haber 0 o más. En caso de ser más de uno los parámetros irán separados por coma.
- **BloqueInstrucciones**: bloque de código que ejecuta la función.
- **return**: retorna datos al código fuente que utilizó la función. Es opcional, ya que el retorno de datos no es obligatorio en las funciones.
- **ValorRetorno**: datos que se retornan.

Hasta aquí te hemos explicado cómo se definen funciones en Python. Para poder utilizar funciones en Python desde el código fuente tienes que hacerlo de la siguiente manera:

```
Variable = NombreFuncion(parámetros)
```

Veamos los elementos en detalle:

- **Variable**: almacenará el valor que devuelve la función. Es opcional, ya que se suprime si la función no devuelve nada.
- **NombreFuncion**: nombre de la función que vamos a utilizar.
- **Parámetros**: parámetros de entrada que tiene la función y que se escriben separados por comas en caso de ser más de uno. Es opcional, por lo que si la función no recibe parámetros se suprimirá.

Cuando usas funciones tienes que tener en cuenta los siguientes puntos:

- La definición de las funciones suele hacerse al principio de los programas.
- La indentación de la función y el bloque de instrucciones que tienen dentro siguen las mismas reglas de indentación que el resto del código.
- Tienes que tener cuidado con el nombre de las variables que utilizas dentro de las funciones ya que si tu programa utiliza variables con el mismo nombre puedes obtener resultados no esperados. Por defecto, las variables que se usan dentro de la función únicamente tienen validez dentro de la función, no vas a poder acceder desde fuera de la función a las variables que tiene definidas dentro.

17.1 EJERCICIOS

El primer ejercicio sobre funciones que vamos a explicarte es el más simple de todos, únicamente vamos a crear una función que muestre un mensaje por pantalla y vamos a usarla un par de veces desde el código fuente del programa principal. El código fuente es el siguiente:

```
def Hola():
    print("¡Hola! ¿Te está gustando Python?")
print("Primera invocación: ", end="")
Hola()
print("Segunda invocación: ", end="")
Hola()
```

Tal y como puedes ver se ha definido una función llamada *Hola* que muestra un mensaje por pantalla y se utiliza dos veces para mostrar el resultado por pantalla.

La siguiente imagen muestra una ejecución del código fuente anterior:

```
Primera invocación: ¡Hola! ¿Te está gustando Python?
Segunda invocación: ¡Hola! ¿Te está gustando Python?
```

El segundo ejercicio consiste en el aprendizaje de la sentencia *return*. *Return* nos va a permitir devolver información como resultado de la ejecución de la función. El código fuente que te mostramos a continuación ha modificado la función *Hola* para que, en lugar de mostrar un mensaje por pantalla, devuelva una cadena de texto con el mismo texto que estaba mostrando:

```
def Hola():
    return "¡Hola! ¿Te está gustando Python?"
print("Primera invocación: " + Hola())
print("Segunda invocación: " + Hola())
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
Primera invocación: ¡Hola! ¿Te está gustando Python?
Segunda invocación: ¡Hola! ¿Te está gustando Python?
```

El tercer ejercicio sobre funciones consiste en el aprendizaje del uso de parámetros en las funciones. Mediante los parámetros vamos a poder pasarle datos a las funciones para que realicen sus operaciones utilizando los datos de entrada que les estamos pasando. El ejercicio consiste en crear una función que comprobará si un número que se pasa por parámetro es par o impar.

```
def EsParOImpar(param):
    if param%2 == 0:
        print("El número es par")
    else:
        print("El número es impar")

numero = int(input("Introduce un numero:"))
EsParOImpar(numero)
```

La ejecución del código fuente anterior tendrá las siguientes salidas dependiendo del número introducido, la siguiente imagen muestra el caso en el que el número introducido es par:

```
Introduce un numero:46
El número es par
```

La siguiente imagen muestra el caso en el que el número introducido es impar:

```
Introduce un numero:3
El número es impar
```

El cuarto ejercicio consiste en crear una función que realice una multiplicación de dos valores pasados como parámetros y devuelva el resultado de dicha multiplicación. El código fuente es el siguiente:

```
def Multiplicar(param1, param2):
    return param1 * param2

multiplicando = int(input("Introduce el multiplicando: "))
multiplicador = int(input("Introduce el multiplicador: "))
resultado = Multiplicar(multiplicando,multiplicador)
print("El resultado de la multiplicación es: ", resultado)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduce el multiplicando: 33
Introduce el multiplicador: 654
El resultado de la multiplicación es:  21582
```

Hasta este punto hemos aprendido de forma básica a utilizar tanto los parámetros de entrada como el retorno de las funciones. En los siguientes ejercicios vamos a explicarte que existen formas un poco más avanzadas de pasar parámetros a las funciones y también en la funcionalidad de retorno de las mismas.

El quinto ejercicio consiste en aprender cómo se le pueden pasar a una función de Python un número variable de parámetros de entrada. En la definición de la función tienes que cambiar cómo se especifican los parámetros. Para ello añadirás un asterisco delante de un parámetro, lo que representará que es un número variable de parámetros. A la hora de utilizar la función podrás poner tantos parámetros como quieras entre paréntesis y separados por comas. El código fuente es el siguiente:

```
def Sumar(*valores):
    resultado = 0
    for item in valores:
        resultado = resultado + item
    return resultado

resultado = Sumar(3,87,45,63,345,3,58,33,22,11,99)
print("El resultado de la suma es: ", resultado)
```

Tal y como puedes ver en el código, el parámetro de entrada *valores* es tratado como un elemento iterable que puedes recorrer con un bucle *for* o incluso acceder mediante la posición a los elementos de la misma.

La siguiente imagen muestra una ejecución del código fuente anterior:

```
El resultado de la suma es:  769
```

El siguiente ejercicio consiste en aprender que con la sentencia *return* es posible devolver más de un valor. La forma de hacerlo es añadiendo tantos elementos a la sentencia como quieras y separarlos todos ellos por comas. A la hora de utilizar la función y asignar el resultado deberás añadir el mismo número de elementos separados por comas antes del operador asignación. El ejercicio amplía el ejercicio anterior realizando la suma de los números pasados como parámetros (se ha modificado el nombre de la función también). El código fuente es el siguiente:

```
def SumarMultiplicar(*valores):
    resultadosuma = 0
    resultadomultiplicacion = 1
    for item in valores:
        resultadosuma = resultadosuma + item
        resultadomultiplicacion = resultadomultiplicacion * item
    return resultadosuma,resultadomultiplicacion

ressuma,resmulti = SumarMultiplicar(3,87,45,63,345,3,58,33,22,11,99)
print("El resultado de la suma es: ", ressuma)
print("El resultado de la multiplicación es: ", resmulti)
```

La siguiente imagen muestra una ejecución del código fuente anterior:

```
El resultado de la suma es:  769
El resultado de la multiplicación es:  35117728294502700
```

El siguiente ejercicio es bastante simple, únicamente vamos a mostrarte que puedes utilizar funciones desde dentro de otras funciones. El ejercicio va a modificar el ejercicio anterior, la función *SumarMultiplicar* va a recibir únicamente dos parámetros que deberá sumar y deberá multiplicar, pero lo hará utilizando una función específica para cada operación. El código fuente es el siguiente:

```
def SumarMultiplicar(param1, param2):
    return Sumar(param1,param2), Multiplicar(param1,param2)

def Sumar(sumando1, sumando2):
    return sumando1 + sumando2

def Multiplicar(multiplicando, multiplicador):
    return multiplicando * multiplicador
```

```
numero1 = int(input("Introduce el primer numero: "))
numero2 = int(input("Introduce el segundo numero: "))
resultadosuma, resultadomultiplicación = SumarMultiplicar(numero1,numero2)
print("El resultado de la suma es: ", resultadosuma)
print("El resultado de la multiplicación es: ", resultadomultiplicación)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduce el primer numero: 77
Introduce el segundo numero: 33
El resultado de la suma es:  110
El resultado de la multiplicación es:  2541
```

17.1.1 Funciones con variables globales

Tal y como te indicábamos al principio, hay que prestar atención a los nombres de las variables dentro de las funciones y a las variables del programa ya que puedes obtener resultados no deseados.

Vamos a verlo con un ejemplo. En el siguiente ejercicio se ha declarado una variable dentro de la función con el mismo nombre que una del programa principal y se muestra tanto en el programa principal como en la función por pantalla. El código fuente es el siguiente:

```
def Variables():
    variable = 3
    print("Valor dentro de la función: " + str(variable))

variable = 5
Variables()
print("Variable en el programa principal: " + str(variable))
```

El programa mostrará un valor diferente a la hora de ejecutarse, ya que por defecto la definición de variables dentro de las funciones tiene únicamente ámbito dentro de la función. Veamos una imagen con un ejemplo de ejecución del código fuente anterior:

```
Valor dentro de la función: 3
Variable en el programa principal: 5
```

Existe una forma para que las variables que se utilizan dentro de la función sean las variables del programa principal. Para ello es necesario añadir una sentencia dentro de la función y antes de definir la variable. Existen dos formas de realizarlo:

- global NombreVariable
- nonlocal NombreVariable

El uso de una u otra sentencia es indiferente, usa la que más te guste.

El siguiente ejercicio amplia el ejercicio anterior definiendo la variable *variable* como global o no local y mostrando su valor en diferentes puntos del programa. El código fuente es el siguiente:

```
def Variables():
    global variable
    print("Valor dentro de la función: " + str(variable))
    variable = 3
    print("Valor dentro de la función: " + str(variable))

variable = 5
print("Variable en el programa principal: " + str(variable))
Variables()
print("Variable en el programa principal: " + str(variable))
```

Analicemos el código, *variable* en un primer momento tendrá el valor 5 asignado y se mostrará por pantalla en el programa principal y posteriormente dentro de la función, obviamente con el mismo valor ya que se ha definido como global la variable dentro de la función. Dentro de la función se actualiza el valor de la variable y se mostrará el valor tanto dentro de la función como en el programa principal con el nuevo valor asignado dentro de la función. La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Variable en el programa principal: 5
Valor dentro de la función: 5
Valor dentro de la función: 3
Variable en el programa principal: 3
```

18

RECURSIVIDAD

En este capítulo vamos a explicarte lo que es la recursividad en desarrollo de software. En el capítulo anterior hemos visto lo que son las funciones, son un bloque de código fuente que contiene un conjunto de instrucciones que realiza algo concreto y que puede ser utilizada desde el código fuente que escribes tantas veces como necesites. En este capítulo vamos a ver funciones recursivas, que no es otra cosa que funciones que se llaman a sí mismas durante su ejecución.

El uso de funciones recursivas se utiliza mayormente para dividir una tarea en subtareas de menor tamaño de forma que sea más fácil abordar el problema y solucionarlo. Nosotros te recomendamos que cuando hables de funciones recursivas pienses en bucles, ya que el código se repite una y otra vez hasta llegar a la solución final.

Las funciones o algoritmos recursivos tienen que cumplir las siguientes tres reglas:

1. Un algoritmo recursivo debe tener un caso base.
2. Un algoritmo recursivo debe cambiar su estado y moverse hacia el caso base.
3. Un algoritmo recursivo debe llamarse a sí mismo, recursivamente.

Básicamente, un algoritmo recursivo tiene dos casos diferentes de ejecución: el caso base y el caso recursivo.

El **caso base** de un algoritmo recursivo es aquel que nos permitirá terminar la función en algún momento. Su objetivo es que se dejen de realizar llamadas recursivas de forma infinita.

El **caso recursivo** de un algoritmo recursivo es aquel en el que se llama de nuevo a sí mismo. Cada llamada recursiva que se haga implicará que se está más cerca de llegar al caso base del algoritmo recursivo.

Es importante tener claro el caso base y saber que el caso recursivo se va acercando a éste y no entra en un estado de llamadas infinitas, es un error común cuando se comienza con la recursividad.

18.1 EJERCICIOS

El primer ejercicio que vamos a hacer referente a la recursividad es el típico ejercicio de calcular el factorial de un número. El cálculo del factorial se realiza realizando una multiplicación de todos los números que van desde el número del que se quiere calcular el factorial hasta el 1. Veamos un ejemplo:

El factorial de 5 sería: 5 * 4 * 3 * 2 * 1 = 120.

Ahora veamos cómo resolvemos el factorial mediante programación. El código fuente sería el siguiente:

```
def Factorial(numero):
    if numero == 1:
        return 1
    else:
        return numero * Factorial(numero-1)

factorial = int(input("Introduzca el número del que quiere calcular el facto-
rial: "))
print("El factorial de " + str(factorial) + " es: " + str(Factorial(factorial)))
```

Analicemos el código fuente. La función *Factorial* incluye tanto el caso base como el caso recursivo, el caso base está identificado cuando el valor del que hay que calcular el factorial es 1 y el caso recursivo es cuando es mayor que uno.

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Introduzca el número del que quiere calcular el factorial: 5
El factorial de 5 es: 120
```

El segundo ejercicio que vamos a realizar es el cálculo de la potencia de un número. Existen diversas opciones para el cálculo de la potencia, pero consideramos que es un ejemplo muy bueno para practicar algoritmos recursivos. El código fuente es el siguiente:

```
def Potencia(base,exponente):
    if exponente <= 0:
        return 1
    else:
        return base * Potencia(base,exponente-1)

base = int(input("Introduzca la base de la potencia: "))
exponente = int(input("Introduzca el exponente de la potencia: "))
print("El valor de " + str(base) + " elevado a " + str(exponente) + " es: " +
str(Potencia(base,exponente)))
```

Analicemos el código fuente. La función *Potencia* incluye tanto el caso base como el caso recursivo, el caso base está identificado cuando el exponente es menor o igual que 0, y el caso recursivo está identificado cuando el exponente es mayor o igual que 1.

La siguiente imagen muestra un ejemplo de ejecución del código fuente:

```
Introduzca la base de la potencia: 2
Introduzca el exponente de la potencia: 8
El valor de 2 elevado a 8 es: 256
```

19

CONTROL DE EXCEPCIONES

En este capítulo vamos a explicarte qué son las excepciones en desarrollo de software y cómo puedes controlarlas para que tus programas sigan funcionando una vez se producen.

19.1 ¿QUÉ SON LAS EXCEPCIONES?

Una excepción es un error lógico que ocurre mientras se ejecuta el programa y que provoca su detención. La detección puede ser controlada y no producirse si se realiza un control de excepciones correcto en el código fuente.

Controlar una excepción no es otra cosa que guardar el estado en el que se encontraba el programa en el momento justo del error e interrumpir el programa para ejecutar un código fuente concreto. En muchos casos, dependiendo del error ocurrido, el control de la excepción implicará que el programa siga ejecutándose después de controlarla, aunque en muchos casos esto no será posible.

El proceso de controlar excepciones es similar en todos los lenguajes de programación. En primer lugar, es necesario incluir el código fuente de ejecución normal dentro de un bloque con la sentencia *try*. Posteriormente, se crea un bloque de código dentro de una sentencia *except* que es la que se ejecutará en caso de error. El bloque *except* permite especificar el tipo de error que se controla con el bloque de código, es por ello por lo que puedes tener tantos bloques *except* como errores quieras controlar, aunque también es posible controlar un error genérico que incluya a todos los errores. En el control de excepciones existe la posibilidad de crear un bloque de código que se ejecute siempre al final, independientemente de si ocurre error o no. Dicho bloque de código se escribe como parte de la sentencia *finally*.

El control de excepciones en Python tiene un aspecto así:

```
try:
        BloqueInstruccionesPrograma

except TipoError1:
        BloqueInstruccionesError1:

except TipoError2:
        BloqueInstruccionesError2

except TipoErrorN:
        BloqueInstruccionesErrorN

finally:
        BloqueCodigoFinally
```

Veamos en detalle cada elemento:

- **try**: indicador de comienzo del bloque de código fuente que se controlará.
- **BloqueInstruccionesPrograma**: conjunto de instrucciones que componen el programa.
- **except**: indicador de comienzo de excepción controlada.
- **TipoError**: indicador del tipo de error que se controla con *except*. El parámetro es opcional, si no se especifica el tipo se controlará la excepción de forma genérica.
- **BloqueInstruccionesError**: conjunto de instrucciones que se ejecuta si se produce el error indicado por *TipoError*.
- **finally**: indicador de comienzo del bloque de código final. La sección es opcional.
- **BloqueCodigoFinally**: conjunto de instrucciones que se ejecutan al acabar cualquiera de los bloques de código anteriores.

19.2 TIPOS DE EXCEPCIONES

En Python existen diferentes tipos de excepciones que pueden controlarse, todas ellas derivan de una serie de excepciones base.

Las excepciones base son las siguientes:

- **Excepción**: tipo de excepción más genérica, de ella derivan todas las excepciones existentes en Python.
- **ArithmeticError**: tipo de excepción genérica para errores aritméticos.
- **BufferError**: tipo de excepción genérica para errores relacionados con buffers.
- **LookupError**: tipo de excepción genérica para errores relacionados con acceso a datos de colecciones.

El grupo de excepciones base anterior da lugar a un grupo de excepciones concretas que te presentamos en la siguiente lista:

- **AssertionError**: excepción que se lanza cuando la instrucción *assert* falla.
- **AttributeError**: excepción que se lanza cuando hay un error a la hora de asignar un valor a un atributo o cuando se intenta acceder a él.
- **EOFError**: excepción que se lanza cuando la instrucción *input* no devuelve datos leídos.
- **FloatingPointError**: excepción que ya no se usa.
- **GeneratorExit**: excepción que se lanza cuando se cierra una función de tipo *generator* o *coroutine*.
- **ImportError**: excepción que se lanza cuando se intenta importar un módulo al programa y falla.
- **ModuleNotFoundError**: excepción que se lanza cuando se intenta importar un módulo y no se encuentra. Deriva de la anterior.
- **IndexError**: excepción que se lanza cuando se intenta acceder a una posición de una secuencia y ésta es superior a la posición mayor.
- **KeyError**: excepción que se lanza cuando se intenta acceder a la clave de un diccionario y no se encuentra.
- **KeyboardInterrupt**: excepción que se lanza cuando el usuario utiliza el comando de interrupción con el teclado (*Control-C* o *delete*).

- **MemoryError**: excepción que se lanza cuando el programa ejecuta una instrucción y ésta supera el máximo de memoria disponible.
- **NameError**: excepción que se lanza cuando el nombre local o global no se encuentra.
- **NotImplementedError**: excepción que se lanza cuando un método de una clase no ha sido implementado todavía y tiene que hacerlo.
- **OSError**: excepción que se lanza cuando el sistema operativo lanza una excepción al ejecutar una instrucción. Existen las siguientes excepciones específicas que puede lanzar el sistema operativo:
  - **BlockingIOError**: excepción que se lanza cuando una operación se bloquea en un objeto que no debería de bloquearse.
  - **ChildProcessError**: excepción que se lanza cuando una operación de un proceso hijo devuelve un error.
  - **ConnectionError**: excepción genérica que se lanza para errores relacionados a conexión.
  - **BrokenPipeError**: excepción que se lanza cuando se intenta escribir en un socket o pipe (tubería) y ya ha sido cerrado.
  - **ConnectionAbortedError**: excepción que se lanza cuando durante un intento de conexión ésta es abortada por el otro extremo.
  - **ConnectionRefusedError**: excepción que se lanza cuando durante un intento de conexión ésta es rechazada por el otro extremo.
  - **ConnectionResetError**: excepción que se lanza cuando la conexión es reseteada por el otro extremo.
  - **FileExistsError**: excepción que se lanza cuando se intenta crear un fichero o directorio y éste ya existe.
  - **FileNotFoundError**: excepción que se lanza cuando se intenta acceder a un fichero o directorio y no existe o no se encuentra.
  - **IsADirectoryError**: excepción que se lanza cuando se intenta ejecutar una operación relacionada con ficheros sobre un directorio.
  - **NotADirectoryError**: excepción que se lanza cuando se intenta ejecutar una operación relacionada con directorios sobre algo que no es un directorio.
  - **PermissionError**: excepción que se lanza cuando se intenta ejecutar una operación y no se tienen los permisos suficientes.

- **ProcessLookupError**: excepción que se lanza cuando se ejecuta un proceso que no existe y se ha indicado que sí.
- **TimeoutError**: excepción que se lanza cuando se sobrepasa el tiempo de espera en alguna función del sistema.

- **OverflowError**: excepción que se lanza cuando el resultado de una operación matemática es demasiado grande para ser representado.
- **RecursionError**: excepción que se lanza cuando el número de recursividades supera el máximo permitido.
- **ReferenceError**: excepción que se lanza al intentar acceder a ciertos atributos por parte de la clase *proxy* y que ya se encuentran en el recolector de basura.
- **RuntimeError**: excepción que se lanza cuando el error que ocurre no puede ser categorizado en ninguno de los tipos existentes.
- **StopIteration**: excepción que se lanza cuando se intenta acceder al siguiente elemento de un iterador y ya no tiene más elementos sobre los que iterar.
- **StopAsyncIteration**: excepción que se lanza cuando se intenta acceder al siguiente elemento de un iterador asíncrono y ya no tiene más elementos sobre los que iterar.
- **SyntaxError**: excepción que se lanza cuando el analizador sintáctico encuentra un error de sintaxis.
- **IndentationError**: excepción genérica que se lanza cuando se encuentran errores de indentación del código fuente.
- **TabError**: excepción que se lanza cuando se encuentran errores de uso en las tabulaciones y espaciados del código fuente. La excepción deriva de la anterior.
- **SystemError**: excepción que se lanza cuando el intérprete de Python encuentra un error interno mientras ejecuta el programa.
- **SystemExit**: excepción que se lanza al ejecutar la instrucción *sys.exit()* y que provoca que se pare la ejecución del programa.
- **TypeError**: excepción que se lanza cuando una operación o función se usa con un tipo de dato incorrecto.

- **UnboundLocalError**: excepción que se lanza cuando se utiliza una variable local en una función o método y no ha sido asignado ningún valor previamente.
- **UnicodeError**: excepción que se lanza cuando se produce un error a la hora de codificar o decodificar Unicode.
- **UnicodeEncodeError**: excepción que se lanza cuando se produce un error a la hora de realizar una codificación a Unicode.
- **UnicodeDecodeError**: excepción que se lanza cuando se produce un error a la hora de realizar una decodificación de Unicode.
- **UnicodeTranslateError**: excepción que se lanza cuando se produce un error a la hora de traducir Unicode.
- **ValueError**: excepción que se lanza cuando una operación o función recibe un parámetro del tipo correcto, pero con un valor incorrecto.
- **ZeroDivisionError**: excepción que se lanza cuando se realiza una división por cero.

19.3 EJERCICIOS

A continuación vamos a realizar una serie de ejercicios para afianzar los conocimientos teóricos adquiridos sobre las excepciones y su control.

El primer ejercicio que vamos a hacer es simplemente lanzar una excepción no controlada en un programa. Una de las excepciones más típicas que ocurren es la división por cero. El código fuente simplemente es mostrar por pantalla el resultado de una división por cero:

```
print(str(17/0))
```

La siguiente imagen muestra la ejecución del código fuente anterior, puedes observar el mensaje de "*Zero DivisionError: division by zero*":

```
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/Excepciones/Ej1
.py", line 1, in <module>
    print(str(17/0))
ZeroDivisionError: division by zero
```

En el siguiente ejercicio vamos a controlar la excepción lanzada por la división por cero del ejercicio anterior. El objetivo es que no se muestre el error de antes por pantalla y mostremos un error personalizado. El código fuente es el siguiente:

```
try:
    print(str(17/0))
except:
    print("ERROR: Division por cero")
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
ERROR: Division por cero
```

Hasta ahora hemos aprendido a incluir código fuente dentro de la sección *try* y a controlar excepciones mediante una sección *except*, pues el siguiente ejercicio consiste en aprender a utilizar la sección *finally*, que añadirá un bloque de código que se ejecutará siempre independientemente se produzca o no una excepción. El código fuente es el siguiente:

```
print("¡Iniciando programa!")
print("\n¡Comenzando primera parte del programa!")
try:
    print(str(17/0))
except:
    print("ERROR: Division por cero")
finally:
    print("¡Primera parte de programa acabada!")

print("\n¡Comenzando segunda parte del programa!")
try:
    print(str(17/1))
except:
    print("ERROR: Division por cero")
finally:
    print("¡Segunda parte de programa acabada!")
```

Analicemos el código fuente. Hemos separado el programa en dos partes, una primera en la que se realizará una división por cero y se mostrará el mensaje de la sección *except* y el de la sección *finally*, y otra parte en la que lo que se ha hecho es

que la división no sea por cero y se mostrará el mensaje del resultado de la división y el de la sección *except*.

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
¡Iniciando programa!

¡Comenzando primera parte del programa!
ERROR: Division por cero
¡Primera parte de programa acabada!

¡Comenzando segunda parte del programa!
17.0
¡Segunda parte de programa acabada!
```

En el control de excepciones es posible añadir bloques *else* (explicados en el capítulo de Control de Flujo) para ejecutar un bloque de código concreto que no se quiere ejecutar como parte del bloque *finally*. Para entender cómo se usa y cuál es su funcionamiento vamos a ampliar el ejercicio anterior añadiendo un bloque de código *else* a cada una de las partes en las que dividimos el programa anterior. El código fuente quedaría de la siguiente forma:

```
print("¡Iniciando programa!")
print("\n¡Comenzando primera parte del programa!")
try:
    print(str(17/0))
except:
    print("ERROR: Division por cero")
else:
    print("INFO: no se han producido errores")
finally:
    print("¡Primera parte de programa acabada!")

print("\n¡Comenzando segunda parte del programa!")
try:
    print(str(17/1))
except:
    print("ERROR: Division por cero")
else:
    print("INFO: no se han producido errores")
finally:
    print("¡Segunda parte de programa acabada!")
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
¡Iniciando programa!

¡Comenzando primera parte del programa!
ERROR: Division por cero
¡Primera parte de programa acabada!

¡Comenzando segunda parte del programa!
17.0
INFO: no se han producido errores
¡Segunda parte de programa acabada!
```

En la ejecución del programa puedes comprobar que el bloque de código dentro de la sección *else* únicamente se ejecutará cuando no se produzcan excepciones.

El siguiente ejercicio consiste en controlar excepciones de un tipo específico en secciones *except*. Vamos a añadir un control para el tipo de excepción *ZeroDivisionError* y vamos a mantener la captura de excepciones genérica, por si pudiera producirse otra excepción que no fuera la específica, de esta forma, capturaríamos de forma específica la excepción de la división y de forma genérica el resto de las excepciones. El código fuente es el siguiente:

```
print("¡Iniciando programa!")
try:
    print(str(17/0))
except ZeroDivisionError:
    print("ERROR: Division por cero")
except:
    print("ERROR: General")
else:
    print("¡No se han producido errores!")
finally:
    print("¡Programa acabado!")
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
¡Iniciando programa!
ERROR: Division por cero
¡Programa acabado!
```

Tal y como puedes comprobar, el mensaje que se ha mostrado al realizar la división por cero es el que se encuentra dentro de la sección *except* con el tipo de excepción *ZeroDivisionError*, no ha sido el de la sección *except* general.

20

EJERCICIO INTERMEDIO

En este capítulo no vamos a explicar nuevos conceptos, vamos a realizar un ejercicio para afianzar los conocimientos que te hemos explicado hasta ahora.

El ejercicio consiste en la implementación de una calculadora que permita realizar las siguientes operaciones:

- Sumar dos números.
- Restar dos números.
- Multiplicar dos números.
- Dividir dos números.
- Calcular el factorial de un número.
- Calcular la potencia de un número.

Para la implementación del ejercicio vamos a utilizar las siguientes funciones:

- **Función *Sumar***: se encargará de realizar todo el proceso de suma, incluyendo la solicitud de los dos números a sumar. El código fuente es el siguiente:

```
def Sumar():
    sum1 = int(input("Sumando uno:"))
    sum2 = int(input("Sumando dos:"))
    print ("La Suma es:", sum1+sum2)
```

- **Función *Restar***: se encargará de realizar todo el proceso de restar, incluyendo la solicitud del minuendo y el sustraendo. El código fuente es el siguiente:

```
def Restar():
    minuendo = int(input("Minuendo:"))
    sustraendo = int(input("Sustraendo:"))
    print ("La Resta es:", minuendo-sustraendo)
```

- **Función *Multiplicar***: se encargará de realizar todo el proceso de multiplicar, incluyendo la solicitud de los dos números a multiplicar. El código fuente es el siguiente:

```
def Multiplicar():
    multiplicando = int(input("Multiplicando:"))
    multiplicador = int(input("Multiplicador:"))
    print ("La Multiplicacion es:", multiplicando*multiplicador)
```

- **Función *Dividir***: se encargará de realizar todo el proceso de dividir, incluyendo la solicitud del dividendo y el divisor. El código fuente es el siguiente:

```
def Dividir():
    try:
        dividendo = int(input("Dividendo:"))
        divisor = int(input("Divisor:"))
        print ("La Division es:", dividendo/divisor)
    except ZeroDivisionError:
        print ("ERROR: No se puede dividir por cero")
```

- **Función *Factorial*:** se encargará de solicitar el número del que se quiere calcular el factorial. Una vez tenga el número invocará a la función de calcular el factorial. El código fuente es el siguiente:

```
def Factorial():
    factorial = int(input("Introduzca el número del que quiere calcu-
lar el factorial: "))
    print("El factorial de " + str(factorial) + " es: " + str(Factoria
lCalculo(factorial)))
```

- **Función *FactorialCalculo***: función recursiva que realiza el cálculo del factorial del número que recibe por parámetros. El código fuente es el siguiente:

```
def FactorialCalculo(numero):
    if numero <=1 1:
```

```
            return 1
        else:
            return numero * FactorialCalculo(numero-1)
```

- **Función *Potencia***: se encargará de solicitar la base y el exponente de la potencia e invocará a la función de calcular la potencia. El código fuente es el siguiente:

```
def Potencia():
    base = int(input("Introduzca la base de la potencia: "))
    exponente = int(input("Introduzca el exponente de la potencia: "))
    print("El valor de " + str(base) + " elevado a " + str(exponente)
+ " es: " + str(PotenciaCalculo(base,exponente)))
```

- **Función *PotenciaCalculo***: función recursiva que realiza el cálculo de la potencia de los números pasados como parámetro. El código fuente es el siguiente:

```
def PotenciaCalculo(base,exponente):
    if exponente <= 0:
        return 1
    else:
        return base * PotenciaCalculo(base,exponente-1)
```

- **Función *Calculadora***: se encargará de ejecutar el bucle y pedir la opción a ejecutar al usuario.

Una vez hemos definido las funciones que necesitamos es el momento de unirlas todas ellas dentro de un mismo programa. El código fuente es el siguiente:

```
def Sumar():
    sum1 = int(input("Sumando uno:"))
    sum2 = int(input("Sumando dos:"))
    print ("La Suma es:", sum1+sum2)

def Restar():
    minuendo = int(input("Minuendo:"))
    sustraendo = int(input("Sustraendo:"))
    print ("La Resta es:", minuendo-sustraendo)

def Multiplicar():
    multiplicando = int(input("Multiplicando:"))
    multiplicador = int(input("Multiplicador:"))
```

```
    print ("La Multiplicacion es:", multiplicando*multiplicador)

def Dividir():
    try:
        dividendo = int(input("Dividendo:"))
        divisor = int(input("Divisor:"))
        print ("La Division es:", dividendo/divisor)
    except ZeroDivisionError:
        print ("ERROR: No se puede dividir por cero")

def Factorial():
    factorial = int(input("Introduzca el número del que quiere calcular el fac-
torial: "))
    print("El factorial de " + str(factorial) + " es: " + str(FactorialCalculo(f
actorial)))

def FactorialCalculo(numero):
    if numero <=1:
        return 1
    else:
        return numero * FactorialCalculo(numero-1)

def Potencia():
    base = int(input("Introduzca la base de la potencia: "))
    exponente = int(input("Introduzca el exponente de la potencia: "))
    print("El valor de " + str(base) + " elevado a " + str(exponente) + " es: "
+ str(PotenciaCalculo(base,exponente)))

def PotenciaCalculo(base,exponente):
    if exponente <= 0:
        return 1
    else:
        return base * PotenciaCalculo(base,exponente-1)

def Calculadora():
    fin = False
    while not(fin):
        opc = int(input("Opcion:"))
        if (opc==1):
            Sumar()
        elif(opc==2):
            Restar()
        elif(opc==3):
            Multiplicar()
        elif(opc==4):
            Dividir()
        elif(opc==5):
            Factorial()
        elif(opc==6):
```

```
            Potencia()
        elif(opc==7):
            fin = 1

print ("""**************
Calculadora
************
Menu
1) Suma
2) Resta
3) Multiplicacion
4) Division
5) Factorial
6) Potencia
7) Salir""")
Calculadora()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior utilizando cada una de las funciones de la calculadora:

```
************
Calculadora
************
Menu
1) Suma
2) Resta
3) Multiplicacion
4) Division
5) Factorial
6) Potencia
7) Salir
Opcion:1
Sumando uno:7
Sumando dos:8
La Suma es: 15
Opcion:2
Minuendo:34
Sustraendo:11
La Resta es: 23
Opcion:3
Multiplicando:3
Multiplicador:9
La Multiplicacion es: 27
Opcion:4
Dividendo:9
Divisor:3
La Division es: 3.0
Opcion:5
Introduzca el número del que quiere calcular el factorial: 7
El factorial de 7 es: 5040
Opcion:6
Introduzca la base de la potencia: 3
Introduzca el exponente de la potencia: 3
El valor de 3 elevado a 3 es: 27
Opcion:7
```

21

MANEJO DE FICHEROS

En este capítulo vamos a explicarte cómo interactuar con ficheros de texto. En el capítulo vamos a cubrir todos los aspectos relacionados con la creación de ficheros, lectura de ficheros y actualización del contenido de los mismos.

El proceso de manipulación de ficheros en Python sigue los siguientes pasos:

1. Apertura del fichero a manipular.
2. Manipulación del fichero (lectura/escritura).
3. Cierre del fichero.

Básicamente todos los lenguajes tienen los mismos pasos a la hora de manipular ficheros.

Vamos a ver cada uno de los pasos que hemos listado antes.

21.1 APERTURA Y CIERRE DE FICHEROS

En Python para manipular ficheros es necesario antes abrirlos, para ello Python proporciona una función llamada ***open*** que devuelve un objeto que te permitirá realizar operaciones con el fichero que has abierto.

La función *open* tiene dos parámetros de entrada:

- Ruta del fichero que se desea abrir.
- Modo de apertura del fichero.

El modo de apertura del fichero indica y limita las operaciones que puedes realizar con el mismo. Veamos los diferentes modos de apertura disponibles:

Parámetro	Significado
"r"	Abre el fichero para lectura. Es el modo de apertura por defecto en el caso de que no se especifique uno.
"w"	Abre el fichero para escritura truncándolo, es decir, borrando todo el contenido que tiene para empezar a escribir de nuevo desde cero.
"x"	Crea un fichero para escribir en él. En caso de que ya exista devuelve un error.
"a"	Abre el fichero para escritura situando el cursor de escritura al final del fichero.
"b"	Abre el fichero en modo binario. Un fichero binario es un tipo de fichero con información representada en ceros y unos en lugar de texto plano, por ejemplo, fotografías, archivos ejecutables, ficheros de Microsoft Word, etc.
"t"	Abre el fichero en modo fichero de texto. Es el modo de apertura por defecto en el caso de que no se especifique que sea binario o de texto.
"+"	Abre el fichero para lectura y escritura.

Una vez has abierto el fichero es cuando puedes realizar una serie de operaciones de manipulación del contenido del mismo. Una vez has acabado de trabajar con el fichero de texto es necesario que cierres el fichero. Para ello está la función ***close***, que te permitirá terminar de trabajar con el fichero que abriste previamente.

21.2 MANIPULACIÓN: LECTURA

La lectura de los ficheros es una operación que utilizarás muy a menudo. En el apartado vamos a presentarte las diferentes sentencias y formas que puedes utilizar para realizar la operación.

Antes de empezar con los ejercicios relativos a la lectura de ficheros te aconsejamos que crees un fichero de texto en la misma ubicación en la que vas a guardar los programas de los ejercicios del capítulo. En el fichero introduce el texto que quieras, además, te aconsejamos que escribas más de una línea. Si por algún casual el fichero lo creas en otra ruta que no es la misma que el programa, deberás de modificar el parámetro de la ruta del fichero de la función *open*.

En los ejercicios que vamos a realizar vamos a utilizar un fichero llamado *"fichero.txt"* con el siguiente contenido:

```
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genial
```

Una vez tengas creado el fichero vamos a proceder con los ejercicios relacionados con la lectura de ficheros de texto con Python. El primer ejercicio que vamos a realizar consiste en la lectura de un fichero de texto y en mostrar su contenido por pantalla. La realización de la lectura se hace con el comando *read*, que lee todo el contenido del fichero y lo almacena como una cadena de texto. El código fuente es el siguiente:

```
f = open("fichero.txt","r")
texto = f.read()
print(texto)
f.close()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genial
```

El siguiente ejercicio consiste en utilizar un bucle *for* para realizar la lectura línea a línea del fichero de texto. En la propia definición del bucle se realiza la apertura del fichero y cada iteración del bucle lee una línea de éste. El ejercicio irá leyendo línea a línea el fichero de texto y mostrándolo por pantalla. El código fuente es el siguiente:

```
for linea in open("fichero.txt","r"):
    print(linea)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Hola, estoy aprendiendo Python.

En un lugar de la mancha

de cuyo nombre no quiero acordarme

etc etc etc

Python es genial
```

El tercer ejercicio relativo a lectura de ficheros de texto consiste en leer el fichero línea a línea utilizando el comando *readline*, que devuelve el contenido de

una línea, dejando el cursor de lectura en la siguiente línea para la siguiente lectura. El ejercicio mostrará por pantalla todas las líneas que lee. En el ejercicio hemos incluido únicamente la lectura de las cinco primeras líneas. En caso de que tu fichero tenga más líneas, deberás de incluir la sentencia "*print(f.readline())*" tantas veces como líneas tenga tu fichero. El código fuente es el siguiente:

```
f = open("fichero.txt","r")
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
f.close()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genial
```

El siguiente ejercicio consiste en la lectura de todas las líneas del fichero de texto con el comando *readlines*, que devuelve todo el contenido del fichero en una lista de elementos donde cada elemento es una línea del fichero. El ejercicio mostrará por pantalla todas las líneas leídas. En el ejercicio hemos mostrado únicamente la lectura de las cinco primeras líneas. En caso de que tu fichero tenga más líneas, deberás de incluir la sentencia "*print(lineas[X])*" tantas veces como líneas tenga tu fichero e indicando en la X el número de la línea que quieres mostrar, o bien utilizar un bucle *for* para iterar la lista e ir mostrando elemento a elemento de la lista. El código fuente es el siguiente:

```
f = open("fichero.txt","r")
lineas = f.readlines()
f.close()
print(lineas[0])
print(lineas[1])
print(lineas[2])
```

```
print(lineas[3])
print(lineas[4])
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genial
```

El último ejercicio referente a la lectura de ficheros de texto consiste en realizar lo mismo que has realizado en el ejercicio anterior pero haciéndolo de otra foma. En este ejercicio vamos a utilizar la sentencia *list* para obtener una lista donde cada elemento de la misma es una línea del fichero de texto. El fichero será mostrado utilizando un bucle *for*. El código fuente es el siguiente:

```
f = open("fichero.txt","r")
lineas = list(f)
f.close()
for item in lineas:
    print(item)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genial
```

21.3 MANIPULACIÓN: ESCRITURA

Al igual que la lectura de ficheros, la escritura de ficheros es una operación que vas a utilizar en multitud de ocasiones en desarrollo de software.

El primer ejercicio que vamos a realizar referente a la escritura de ficheros consiste en añadir una línea nueva a un fichero ya existente, concretamente al fichero que hemos utilizado durante el apartado anterior referente a la lectura de ficheros. El primer paso que se tiene que dar para escribir en un fichero de texto es realizar la apertura en modo "*a*", apertura para escritura al final del fichero. El ejercicio mostrará el fichero de texto antes de añadir la línea y después de añadirla, de este modo comprobaremos que se ha añadido correctamente. El código fuente es el siguiente:

```
print("FICHERO INICIAL")
flectura = open("fichero.txt","r")
texto = flectura.read()
flectura.close()
print(texto)
print("INSERTANDO LÍNEA...\n")
fescritura = open("fichero.txt","a")
fescritura.write("Nueva linea en el fichero\n")
fescritura.close()
print("FICHERO INICIAL")
flectura = open("fichero.txt","r")
texto = flectura.read()
flectura.close()
print(texto)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
FICHERO INICIAL
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genialNueva linea en el fichero

INSERTANDO LÍNEA...

FICHERO INICIAL
Hola, estoy aprendiendo Python.
En un lugar de la mancha
de cuyo nombre no quiero acordarme
etc etc etc
Python es genialNueva linea en el fichero
Nueva linea en el fichero
```

El segundo ejercicio consiste en crear un fichero nuevo para escribir en él. El modo de apertura para crear un fichero nuevo y escribir en él es el modo "*x*", dicho modo devuelve un error si el fichero ya existe. En el ejercicio se va a crear un fichero

y se va a escribir información en él, para posteriormente mostrar el contenido del fichero por pantalla. El código fuente es el siguiente:

```
fcrear = open("ficheronuevo.txt","x")
fcrear.write("Estoy aprendiendo Python...\n")
fcrear.write("...y me encanta.\n")
fcrear.write("Me parece un lenguaje de programación\n")
fcrear.write("muy facil de aprender.\n")

fcrear.close()

print("FICHERO CREADO")

flectura = open("ficheronuevo.txt","r")
texto = flectura.read()
flectura.close()
print(texto)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
FICHERO CREADO
Estoy aprendiendo Python...
...y me encanta.
Me parece un lenguaje de programación
muy facil de aprender.
```

Tal y como te comentamos anteriormente, el modo de apertura "x" devuelve error en el caso de que el fichero exista. La siguiente imagen muestra el error que aparecería si volvieras a ejecutar el programa.

```
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/ManejoFicheros/
Ej7.py", line 1, in <module>
    fcrear = open("ficheronuevo.txt","x")
FileExistsError: [Errno 17] File exists: 'ficheronuevo.txt'
```

El último ejercicio referente a escritura de ficheros consiste en aprender a escribir en ficheros de texto truncándolos en su apertura, lo que significa que eliminará el contenido del mismo y empezará a escribir el nuevo. El modo de apertura para abrir con truncado es el modo "*w*". En el ejercicio se va a escribir en un fichero de texto ya existente que se truncará utilizando el modo de apertura "w", posteriormente el contenido se muestra por pantalla. El fichero que utiliza el ejercicio es el del ejercicio anterior. El código fuente es el siguiente:

```
fcrear = open("ficheronuevo.txt","w")
fcrear.write("Fichero creado desde cero\n")
fcrear.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit,")
fcrear.write("sed eiusmod tempor incidunt ut labore et dolore magna aliqua.")
fcrear.write("Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquid ex ea commodi consequat.")
fcrear.write("Quis aute iure reprehenderit in voluptate velit esse cillum dolore
eu fugiat nulla pariatur.")
fcrear.write("Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.\n")
fcrear.close()

print("### Fichero creado ###")

flectura = open("ficheronuevo.txt","r")
texto = flectura.read()
flectura.close()
print(texto)
```

La ejecución del código fuente anterior tendrá la siguiente salida:

```
### Fichero creado ###
Fichero creado desde cero
Lorem ipsum dolor sit amet, consectetur adipiscing elit,sed eiusmod tempor incid
unt ut labore et dolore magna aliqua.Ut enim ad minim veniam, quis nostrud exerc
itation ullamco laboris nisi ut aliquid ex ea commodi consequat.Quis aute iure r
eprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.Exce
pteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt m
ollit anim id est laborum.
```

21.4 RESUMEN DE FUNCIONES DE FICHEROS

El manejo de ficheros en Python incluye una serie de funciones útiles que te van a permitir utilizar ficheros de una forma más fácil y sencilla. Algunos de ellos ya los hemos explicado y utilizado en el capítulo, otros todavía no. La siguiente lista muestra todas las funciones incluidas en Python para el manejo de los ficheros:

- **open**: lo hemos visto en el primer apartado del capítulo, se utiliza para abrir un fichero especificando la ruta y el modo de apertura. La función devuelve un objeto de tipo *File* que es el que utilizarás para interactuar con el fichero.
- **close**: lo hemos visto en el primer apartado del capítulo, se utiliza para cerrar un fichero que hemos abierto previamente con la función *open*.

- **writable**: función que comprueba si el fichero ha sido abierto en modo escritura. Devolverá *True* en caso afirmativo y *False* en caso negativo.
- **readable**: función que comprueba si el fichero ha sido abierto en modo lectura. Devolverá *True* en caso afirmativo y *False* en caso negativo.
- **seekable**: función que comprueba si es posible desplazarse dentro del archivo abierto. Devolverá *True* en caso afirmativo y *False* en caso negativo.
- **read**: lo hemos visto en el segundo apartado del capítulo, se utiliza para realizar la lectura del fichero completo.
- **readline**: lo hemos visto en el segundo apartado del capítulo, se utiliza para realizar la lectura de una línea del fichero abierto.
- **readlines**: lo hemos visto en el segundo apartado del capítulo, se utiliza para leer todas las líneas de un fichero de texto devolviéndolas como una lista.
- **write**: lo hemos visto en el tercer apartado del capítulo, se utiliza para escribir contenido en el fichero de texto en la posición en la que se encuentre el cursor, sobrescribiendo el texto en caso de que haya en esa posición.
- **tell**: función que devuelve la posición en la que se encuentra el cursor dentro del fichero.
- **seek**: función que mueve el puntero a la posición indicada por parámetro.

22

PROGRAMACIÓN ORIENTADA A OBJETOS

En este capítulo vamos a explicarte la programación orientada a objetos para que empieces a utilizarla como paradigma de desarrollo en todos los programas que realices.

En el capítulo vamos a explicarte el cambio de paradigma que supuso la programación orientada a objetos en el mundo del desarrollo de software. Vamos a explicarte también lo qué es una clase y un objeto. Entraremos de lleno a explicarte lo que es la composición, la encapsulación y la herencia, técnicas imprescindibles para conseguir el objetivo principal de la programación orientada a objetos: la reutilización y la extensibilidad.

¡Empecemos!

22.1 CAMBIO DE PARADIGMA

El mundo del desarrollo de software es un mundo en constante evolución y cambio, y allá por los años 60 se empezó a hablar de un nuevo paradigma de desarrollo que era la programación orientada a objetos. La programación orientada a objetos no tenía otro objetivo que no fuera intentar paliar las deficiencias existentes en la programación en ese momento, que eran las siguientes:

- **Distinta abstracción del mundo**: la programación en ese momento se centraba en comportamientos representado por verbos normalmente, mientras que la programación orientada a objetos se centra en seres, representados por sustantivos normalmente. Se pasa de utilizar funciones que representan verbos, a utilizar clases, que representan sustantivos.

- **Dificultad de modificación y actualización**: los datos suelen ser compartidos por los programas, por lo que cualquier ligera modificación de los datos podía provocar que otro programa dejara de funcionar de forma indirecta.
- **Dificultad de mantenimiento**: la corrección de errores que existía en ese momento era bastante costosa y difícil de realizar.
- **Dificultad de reutilización**: las funciones/rutinas suelen ser muy dependientes del contexto en el que se crearon y eso dificulta reaprovecharlas en nuevos programas.

La programación orientada a objetos, básicamente, apareció para aportar lo siguiente:

- Nueva abstracción del mundo centrándolo en seres y no en verbos mediante nuevos conceptos como clase y objeto, que veremos en el siguiente apartado.
- Control de acceso a los datos mediante encapsulación de éstos en las clases.
- Nuevas funcionalidades de desarrollo para clases, como por ejemplo herencia y composición, que permiten simplificar el desarrollo.

22.2 CLASE Y OBJETO

Antes de empezar vamos a hacer un símil de la programación orientada a objetos con el mundo real. Mira a tu alrededor, ¿qué ves? La respuesta es: objetos. Estamos rodeados de objetos, como pueden ser coches, lámparas, teléfonos, mesas… El nuevo paradigma de programación orientada a objetos está basado en una abstracción del mundo real que nos va a permitir desarrollar programas de forma más cercana a cómo vemos el mundo, pensando en objetos que tenemos delante y acciones que podemos hacer con ellos.

Una clase es un tipo de dato cuyas variables se llaman objetos o instancias. Es decir, la clase es la definición del concepto del mundo real y los objetos o instancias son el propio "objeto" del mundo real. Piensa por un segundo en un coche. Antes de ser fabricado, un coche tiene que ser definido, tiene que tener una plantilla que especifique sus componentes y lo que puede hacer. Esa plantilla es lo que se conoce como Clase. Una vez el coche es construido, ese coche sería un objeto o instancia de la clase Coche, que es quien define qué es un coche y qué se puede hacer con un coche.

Las clases están compuestas por dos elementos:

- **Atributos**: información que almacena la clase.
- **Métodos**: operaciones que pueden realizarse con la clase.

Piensa ahora en el coche de antes, la clase coche podría tener atributos tales como número de marchas, número de asientos, cilindrada... y podría realizar las operaciones tales como subir marcha, bajar marcha, acelerar, frenar, encender el intermitente… Un objeto es un modelo de coche concreto.

La definición más básica de las clases en Python se hace de la siguiente forma:

```
class NombreClase:

    def __init__ (self, variable1, variable2):
        self.Atributo1 = valor1
        self.Atributo2 = valor2

    def NombreMetodo(self):
        BloqueCodigo
```

Veamos en detalle cada uno de los componentes:

- **class**: palabra reservada en Python para definir una clase.
- **NombreClase**: nombre de la clase que quieres crear.
- **def**: palabra reservada en Python que se utiliza para definir tanto el constructor de la clase (método que se ejecuta la primera vez que usas una clase) como los diferentes métodos que tiene.
- **__init__**: palabra reservada en Python para definir el método constructor de la clase. El método *__init__* es lo primero que se ejecuta cuando creas un objeto de una clase. Un error muy común que se suele cometer cuando estás aprendiendo programación orientada a objetos en Python es escribir mal __init__, son dos rayas bajas, normalmente la gente que está aprendiendo pone únicamente una.
- **(self,variableX)**: parámetro del constructor de la clase. El parámetro *self* es obligatorio y después puedes tener tanto parámetros como quieras. La forma de añadir parámetros es la misma que en las funciones.
- **self.AtributoX**: forma de utilización y acceso a los atributos de la clase.

- **NombreMetodo**: nombre del método de la clase.
- **(self)**: parámetros del método. El parámetro *self* es obligatorio y después puedes tener tantos parámetros como quieras. La forma de añadir parámetros es la misma que en las funciones.
- **BloqueCodigo**: instrucciones que ejecutará el método.

Cuando defines una clase tienes que tener los siguientes puntos en cuenta:

- Puedes definir tantos atributos como necesites.
- Puedes definir tantos métodos como necesites.
- Puedes definir tantos parámetros en el constructor y en los métodos como necesites.

El primer ejercicio que vamos a realizar en el capítulo consiste en la creación de una clase que represente a una persona. Los atributos que vamos a crear van a ser el nombre, los apellidos y la edad. En la clase vamos a crear un método para mostrar la información de la persona. La ejecución del programa consistirá en crear un objeto del tipo *Persona* y mostrar los datos almacenados que tiene utilizando el método que crearemos para ello. El código fuente es el siguiente:

```
class Persona:
    def __init__ (self, nombre, apellidos, edad):
        self.Nombre = nombre
        self.Apellidos = apellidos
        self.Edad = edad
    def MostrarPersona(self):
        print("Nombre: " + self.Nombre)
        print("Apellidos: " + self.Apellidos)
        print("Edad: " + str(self.Edad))

p1 = Persona("Alfredo","Moreno Muñoz", 35)
p1.MostrarPersona()
```

Tal y como has podido observar en el código fuente, la forma de crear un objeto de una clase es parecido a usar una función. El resultado de utilizar *Persona(…)* tienes que asignarlo a una variable (*p1*) para poder utilizar posteriormente dicho objeto. La forma de utilizar los métodos es *NombreObjeto.NombreMetodo*, en el ejercicio *p1.MostrarPersona()*.

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Nombre: Alfredo
Apellidos: Moreno Muñoz
Edad: 35
```

El siguiente ejercicio consiste en crear dos objetos diferentes de la clase con el objetivo de que compruebes que cada uno tiene su propia información. También vamos a enseñarte como modificar el valor de los atributos del objeto. La forma de acceder a los atributos de un objeto es *NombreObjeto.NombreAtributo*. El código fuente es el siguiente:

```
class Persona:
    def __init__ (self, nombre, apellidos, edad):
        self.Nombre = nombre
        self.Apellidos = apellidos
        self.Edad = edad
    def MostrarPersona(self):
        print("Nombre: " + self.Nombre)
        print("Apellidos: " + self.Apellidos)
        print("Edad: " + str(self.Edad))

print("OBJETOS ORIGINALES")
p1 = Persona("Alfredo","Moreno Muñoz", 35)
p1.MostrarPersona()
p2 = Persona("Valeria","Moreno", 1)
p2.MostrarPersona()
p1.Edad = 36
p2.Apellidos = "Moreno Córcoles"
print("OBJETOS MODIFICADOS")
p1.MostrarPersona()
p2.MostrarPersona()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
OBJETOS ORIGINALES
Nombre: Alfredo
Apellidos: Moreno Muñoz
Edad: 35
Nombre: Valeria
Apellidos: Moreno
Edad: 1
OBJETOS MODIFICADOS
Nombre: Alfredo
Apellidos: Moreno Muñoz
Edad: 36
Nombre: Valeria
Apellidos: Moreno Córcoles
Edad: 1
```

El último ejercicio que vamos a realizar de este apartado es la asignación de objeto. Podrás comprobar que asignando objetos se asignan los valores que tienen sus atributos. En el ejercicio vamos a modificar el código fuente anterior asignando el objeto *p2* al objeto *p1*, mostraremos ambos objetos antes y después de la asignación. El código fuente es el siguiente:

```
class Persona:
    def __init__ (self, nombre, apellidos, edad):
        self.Nombre = nombre
        self.Apellidos = apellidos
        self.Edad = edad
    def MostrarPersona(self):
        print("Nombre: " + self.Nombre)
        print("Apellidos: " + self.Apellidos)
        print("Edad: " + str(self.Edad))

print("OBJETOS ORIGINALES")
p1 = Persona("Alfredo","Moreno Muñoz", 35)
p1.MostrarPersona()
p2 = Persona("Valeria","Moreno", 1)
p2.MostrarPersona()
p1=p2
print("OBJETOS TRAS ASIGNACIÓN")
p1.MostrarPersona()
p2.MostrarPersona()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
OBJETOS ORIGINALES
Nombre: Alfredo
Apellidos: Moreno Muñoz
Edad: 35
Nombre: Valeria
Apellidos: Moreno
Edad: 1
OBJETOS TRAS ASIGNACIÓN
Nombre: Valeria
Apellidos: Moreno
Edad: 1
Nombre: Valeria
Apellidos: Moreno
Edad: 1
```

22.3 COMPOSICIÓN

La primera técnica que vamos a explicarte referente a la reutilización de código fuente en la programación orientada a objetos es la **composición**, que consiste en la creación de nuevas clases a partir de otras clases ya existentes que actúan como elementos compositores de la nueva. Las clases existentes serán atributos de la nueva clase.

En programación orientada a objetos la composición significa que entre las dos clases existe una relación del tipo "tiene un".

Pongamos un ejemplo del mundo real para luego hacerlo en código: una coordenada en dos dimensiones está compuesta por dos valores, el valor en el eje de las X y el valor en el eje de las Y, ésto podría ser una clase. Un cuadrado está compuesto por cuatro coordenadas que son los cuatro vértices, ésto podría ser una clase que está compuesta por cuatro clases del objeto coordenada. Vamos a ver cada una de las clases:

- Clase Coordenada:
  - Valor eje de las X.
  - Valor eje de las Y.
- Clase Cuadrado:
  - Coordenada vértice 1.
  - Coordenada vértice 2.
  - Coordenada vértice 3.
  - Coordenada vértice 4.

Vamos a ver cómo hacemos esto en Python. El código fuente sería el siguiente:

```
class Coordenada:
    def __init__ (self, x, y):
        self.X = x
        self.Y = y
    def MostrarCoordenada(self):
        print("(",self.X,",",self.Y,")")

class Cuadrado:
    def __init__ (self, v1,v2,v3,v4):
        self.V1 = v1
        self.V2 = v2
        self.V3 = v3
        self.V4 = v4
```

```
    def MostrarVertices(self):
        print("El cuadrado está compuesto por los siguiente vértices:")
        self.V1.MostrarCoordenada()
        self.V2.MostrarCoordenada()
        self.V3.MostrarCoordenada()
        self.V4.MostrarCoordenada()

v1 = Coordenada(1,1)
v2 = Coordenada(4,1)
v3 = Coordenada(4,4)
v4 = Coordenada(1,4)
cuadrado = Cuadrado(v1,v2,v3,v4)
cuadrado.MostrarVertices()
```

Para ambas clases hemos definido un método que muestra la información por pantalla de los atributos de la misma. Puedes observar que el método de mostrar por pantalla los vértices del cuadrado no accede a los atributos de las coordenadas y utiliza el método *MostrarCoordenada* de la clase *Coordenada*. Esto se debe a la técnica que veremos en el siguiente apartado, la encapsulación.

La siguiente imagen muestra la ejecución del código fuente anterior:

```
El cuadrado está compuesto por los siguiente vértices:
( 1 , 1 )
( 4 , 1 )
( 4 , 4 )
( 1 , 4 )
```

22.4 ENCAPSULACIÓN

Uno de los objetivos que tiene la programación orientada a objetos es proteger los datos de acceso o usos no controlados, y ésto es lo que se conoce como **encapsulación**.

Los datos (atributos) que componen una clase pueden ser de dos tipos:

- **Públicos**: los datos son accesibles sin control, es decir, los datos pueden ser usados sin ningún tipo de mecanismo que proteja ante usos no autorizados o indebidos.
- **Privados**: los datos no pueden ser accedidos sin control y para acceder a ellos se deberá implementar un método que acceda a ellos. De ésta forma, los datos únicamente serán accedidos directamente por la propia clase.

La encapsulación no solo puede realizarse sobre los atributos de la clase, también es posible realizarla sobre los métodos, es decir, aquellos métodos que indiquemos que son privados no podrán ser utilizados por elementos externos al propio objeto.

Veamos cómo se encapsulan los datos de una clase.

El primer ejercicio consiste en la creación de dos clases que contienen la misma información pero que se diferencia en que una tiene sus atributos declarados como públicos (*PersonaPublica*) y la otra los tiene como privado (*PersonaPrivada*).

La clase privada va a tener dos métodos por cada atributo de clase que tiene, uno para realizar la lectura del atributo (*Get*) y otro para realizar la escritura del mismo (*Set*). Ésto es algo común que se hace en programación orientada a objetos, por lo que es necesario que te vayas acostumbrando a crear estos métodos de clase para cada atributo.

En el ejercicio vas a ver las diferencias de definición, uso y acceso a los atributos públicos y privados. La definición de atributos privados se realiza incluyendo los caracteres "__" (dos rayas bajas) entre la palabra "*self.*" y el nombre del atributo. El código fuente es el siguiente:

```
class PersonaPublica:
    def __init__ (self, nombre, apellidos, edad):
        self.Nombre = nombre
        self.Apellidos = apellidos
        self.Edad = edad

class PersonaPrivada:
    def __init__ (self, nombre, apellidos, edad):
        self.__Nombre = nombre
        self.__Apellidos = apellidos
        self.__Edad = edad
    def GetNombre(self):
        return self.__Nombre
    def GetApellidos(self):
        return self.__Apellidos
    def GetEdad(self):
        return self.__Edad
    def SetNombre(self, nombre):
        self.__Nombre = nombre
    def SetApellidos(self, apellidos):
        self.__Apellidos = apellidos
    def SetEdad(self, edad):
        self.__Edad = edad
```

```
publico = PersonaPublica("Alfredo","Moreno",35)
privado = PersonaPrivada("Valeria","Moreno",1)
print("PERSONA PÚBLICA")
print("Nombre: " + publico.Nombre)
print("Apellidos: " + publico.Apellidos)
print("Edad: " + str(publico.Edad))
print("PERSONA PRIVADA")
print("Nombre: " + privado.GetNombre())
print("Apellidos: " + privado.GetApellidos())
print("Edad: " + str(privado.GetEdad()))
print("\nModificación de valores en ambos objetos...")
publico.Apellidos = "Moreno Córcoles"
privado.SetApellidos("Moreno Muñoz")
print("PERSONA PÚBLICA")
print("Nombre: " + publico.Nombre)
print("Apellidos: " + publico.Apellidos)
print("Edad: " + str(publico.Edad))
print("PERSONA PRIVADA")
print("Nombre: " + privado.GetNombre())
print("Apellidos: " + privado.GetApellidos())
print("Edad: " + str(privado.GetEdad()))
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
PERSONA PÚBLICA
Nombre: Alfredo
Apellidos: Moreno
Edad: 35
PERSONA PRIVADA
Nombre: Valeria
Apellidos: Moreno
Edad: 1

Modificación de valores en ambos objetos...
PERSONA PÚBLICA
Nombre: Alfredo
Apellidos: Moreno Córcoles
Edad: 35
PERSONA PRIVADA
Nombre: Valeria
Apellidos: Moreno Muñoz
Edad: 1
```

El siguiente ejercicio consiste únicamente en ver el error que recibes si intentas acceder a un atributo privado de una clase. El código fuente es el siguiente:

```
class PersonaPrivada:
    def __init__ (self, nombre, apellidos, edad):
```

```
            self.__Nombre = nombre
            self.__Apellidos = apellidos
            self.__Edad = edad

    privado = PersonaPrivada("Valeria","Moreno",1)
    print("Nombre: " + privado.__Nombre)
    print("Apellidos: " + privado.GetApellidos())
    print("Edad: " + str(privado.GetEdad()))
```

El error que recibes es el siguiente:

```
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/POO/Ej6.py", li
ne 8, in <module>
    print("Nombre: " + privado.__Nombre)
AttributeError: 'PersonaPrivada' object has no attribute '__Nombre'
```

El siguiente ejercicio del apartado consiste en aprender a aplicar encapsulación a los métodos de las clases. Al igual que en los atributos de la clase, en los métodos hay que poner "__" (dos rayas bajas) delante del nombre del método para definirlos como privado.

En el ejercicio vamos a crear dos clases, la primera de ellas será la clase *Coordenada* que almacenará la coordenada en el eje de las X y de las Y de un punto. Los atributos de la clase serán privados y crearemos un método para cada uno de ellos para realizar la lectura y la escritura en dicho atributo. La segunda de las clases será *Cuadrado* y estará compuesta por cuatro atributos privados de la clase *Coordenada*. En la clase *Cuadrado* vamos a crear un método privado por cada uno de los atributos que muestre su valor por pantalla. También crearemos un método público que mostrará los cuatro atributos que componen la clase utilizando los métodos privados. El código fuente es el siguiente:

```
class Coordenada:
    def __init__ (self, x, y):
        self.__X = x
        self.__Y = y
    def GetX(self):
        return self.__X
    def GetY(self):
        return self.__Y
    def SetX(self,x):
        self.__X = x
    def SetY(self,y):
        self.__Y = y
```

```
class Cuadrado:
    def __init__ (self, v1,v2,v3,v4):
        self.__V1 = v1
        self.__V2 = v2
        self.__V3 = v3
        self.__V4 = v4
    def __MostrarCoordenadaV1(self):
        print("(",self.__V1.GetX(),",",self.__V1.GetY(),")")
    def __MostrarCoordenadaV2(self):
        print("(",self.__V2.GetX(),",",self.__V2.GetY(),")")
    def __MostrarCoordenadaV3(self):
        print("(",self.__V3.GetX(),",",self.__V3.GetY(),")")
    def __MostrarCoordenadaV4(self):
        print("(",self.__V4.GetX(),",",self.__V4.GetY(),")")
    def MostrarVertices(self):
        print("El cuadrado está compuesto por los siguiente vértices:")
        self.__MostrarCoordenadaV1()
        self.__MostrarCoordenadaV2()
        self.__MostrarCoordenadaV3()
        self.__MostrarCoordenadaV4()

v1 = Coordenada(1,1)
v2 = Coordenada(4,1)
v3 = Coordenada(4,4)
v4 = Coordenada(1,4)
cuadrado = Cuadrado(v1,v2,v3,v4)
cuadrado.MostrarVertices()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
El cuadrado está compuesto por los siguiente vértices:
( 1 , 1 )
( 4 , 1 )
( 4 , 4 )
( 1 , 4 )
```

El último ejercicio referente a la encapsulación es para que veas el mensaje que te saldría si intentases utilizar un método privado de una clase. El código fuente es el siguiente:

```
class Coordenada:
    def __init__ (self, x, y):
        self.__X = x
        self.__Y = y
```

```
        def __GetX(self):
            return self.__X
        def GetY(self):
            return self.__Y
        def SetX(self,x):
            self.__X = x
        def SetY(self,y):
            self.__Y = y
    coordenada = Coordenada(3,4)
    print("(",coordenada.__GetX(),",",coordenada.GetY(),")")
```

Hemos reutilizado la clase *Coordenada* del ejercicio anterior y hemos cambiado uno de los métodos a privado.

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/POO/Ej8.py", li
ne 14, in <module>
    print("(",coordenada.__GetX(),",",coordenada.GetY(),")")
AttributeError: 'Coordenada' object has no attribute '__GetX'
```

22.5 HERENCIA

Tal y como te hemos comentado durante el capítulo, la programación orientada a objetos tiene como uno de sus objetivos principales la reutilización de código. La herencia es un ejemplo de ello. La herencia consiste en la definición de una clase utilizando como base una clase ya existente. La nueva clase derivada tendrá todas las características de la clase base y ampliará el concepto de ésta, es decir, tendrá todos los atributos y métodos de la clase base.

En programación orientada a objetos la herencia significa que entre las dos clases existe una relación del tipo "es un".

Veamos un ejemplo de la vida real, pensemos en una persona como una clase, la persona tendría una serie de atributos como pueden ser el nombre, los apellidos, la edad, etc. Esas características de una persona serían compartidas por todas aquellas clases hijas como pueden ser alumno y profesor. Es decir, alumno y profesor heredarían las propiedades de la clase persona y tendrían sus propias propiedades, diferentes entre ellas, como por ejemplo el curso en el que está el alumno y el horario de tutorías del profesor.

Veámoslo en detalle:

- **Clase base**: Persona.
  - Atributos:
    - Nombre.
    - Apellidos.
    - Edad.

- **Clase derivada**: Alumno.
  - Atributos:
    - Curso.
    - Asignaturas.

- **Clase derivada**: Profesor.
  - Atributos:
    - Antigüedad.
    - Tutorías.
    - Teléfono.

A la hora de programar, el objeto que te crees de la clase que corresponda tendría los siguientes atributos:

- **Persona**:
  - Nombre.
  - Apellidos.
  - Edad.

- **Alumno**:
  - Nombre.
  - Apellidos.
  - Edad.
  - Curso.
  - Asignaturas.

- **Profesor**:
  - Nombre.
  - Apellidos.
  - Edad.
  - Antigüedad.
  - Tutorías.
  - Teléfono.

Todo lo explicado para los atributos es igual de válido para los métodos de las clases, es decir, una clase derivada heredará también los métodos de la clase base de la que herede.

La herencia en Python se especifica de la siguiente manera:

class NombreClase(ListaDeClasesBase)

En *ListaDeClasesBase* especificaremos todas aquellas clases de la que hereda, separadas por comas.

Hagamos un par de ejercicios para afianzar estos conocimientos.

El primer ejercicio va a consistir en crear las clases *Persona* y *Alumno* que hemos visto anteriormente, añadiendo un método a la clase *Alumno* que muestre toda la información por pantalla. El código fuente es el siguiente:

```
class Persona:
    def __init__ (self):
        self.__Nombre = ""
        self.__Apellidos = ""
        self.__Edad = 0
    def GetNombre(self):
        return self.__Nombre
    def SetNombre(self,nombre):
        self.__Nombre = nombre
    def GetApellidos(self):
        return self.__Apellidos
    def SetApellidos(self,apellidos):
        self.__Apellidos = apellidos
    def GetEdad(self):
        return self.__Edad
    def SetEdad(self,edad):
        self.__Edad = edad


class Alumno(Persona):
    def __init__ (self):
        self.__Curso = ""
        self.__Asignaturas = ""
    def GetCurso(self):
        return self.__Curso
    def SetCurso(self,curso):
        self.__Curso = curso
    def GetAsignaturas(self):
```

```
            return self.__Asignaturas
        def SetAsignaturas(self,asignaturas):
            self.__Asignaturas = asignaturas
        def MostrarAlumno(self):
            print("Alumno:")
            print("\tNombre:",self.GetNombre())
            print("\tApellidos:",self.GetApellidos())
            print("\tEdad:",self.GetEdad())
            print("\tCurso:",self.__Curso)
            print("\tMatriculas:",self.__Asignaturas)

    alumno = Alumno()
    alumno.SetNombre("Alfredo")
    alumno.SetApellidos("Moreno Muñoz")
    alumno.SetEdad(35)
    alumno.SetCurso("Bachillerato")
    alumno.SetAsignaturas(["Matemáticas","Tecnología","Inglés"])
    alumno.MostrarAlumno()
```

Puedes observar como al crear un objeto de la clase *Alumno* estamos interactuando con los métodos de la clase *Persona* como si fueran de la propia clase *Alumno*.

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Alumno:
        Nombre: Alfredo
        Apellidos: Moreno Muñoz
        Edad: 35
        Curso: Bachillerato
        Matriculas: ['Matemáticas', 'Tecnología', 'Inglés']
```

El segundo ejercicio va a consistir en añadir al ejercicio anterior la clase *Profesor*, a la que añadiremos un método también para mostrar la información por pantalla. El código fuente es el siguiente:

```
class Persona:
    def __init__ (self):
        self.__Nombre = ""
        self.__Apellidos = ""
        self.__Edad = 0
    def GetNombre(self):
        return self.__Nombre
    def SetNombre(self,nombre):
        self.__Nombre = nombre
```

```
    def GetApellidos(self):
        return self.__Apellidos
    def SetApellidos(self,apellidos):
        self.__Apellidos = apellidos
    def GetEdad(self):
        return self.__Edad
    def SetEdad(self,edad):
        self.__Edad = edad


class Alumno(Persona):
    def __init__ (self):
        self.__Curso = ""
        self.__Asignaturas = ""
    def GetCurso(self):
        return self.__Curso
    def SetCurso(self,curso):
        self.__Curso = curso
    def GetAsignaturas(self):
        return self.__Asignaturas
    def SetAsignaturas(self,asignaturas):
        self.__Asignaturas = asignaturas
    def MostrarAlumno(self):
        print("Alumno:")
        print("\tNombre:",self.GetNombre())
        print("\tApellidos:",self.GetApellidos())
        print("\tEdad:",self.GetEdad())
        print("\tCurso:",self.__Curso)
        print("\tMatriculas:",self.__Asignaturas)

class Profesor(Persona):
    def __init__ (self):
        self.__Antigüedad = ""
        self.__Tutorias = ""
        self.__Telefono = ""
    def GetAntigüedad(self):
        return self.__Antigüedad
    def SetAntigüedad(self,antigüedad):
        self.__ Antigüedad = antigüedad
    def GetTutorias(self):
        return self.__Tutorias
    def SetTutorias(self,tutorias):
        self.__Tutorias = tutorias
    def GetTelefono(self):
        return self.__Telefono
```

```
        def SetTelefono(self,telefono):
            self.__Telefono = telefono
        def MostrarProfesor(self):
            print("Profesor:")
            print("\tNombre:",self.GetNombre())
            print("\tApellidos:",self.GetApellidos())
            print("\tEdad:",self.GetEdad())
            print("\tAntigüedad:",self.__Antigüedad)
            print("\tTutorias:",self.__Tutorias)
            print("\tTelefono:",self.__Telefono)

    alumno = Alumno()
    alumno.SetNombre("Alfredo")
    alumno.SetApellidos("Moreno Muñoz")
    alumno.SetEdad(35)
    alumno.SetCurso("Bachillerato")
    alumno.SetAsignaturas(["Matemáticas","Tecnología","Inglés"])
    alumno.MostrarAlumno()

    profesor = Profesor()
    profesor.SetNombre("Profesor")
    profesor.SetApellidos("Casa Papel")
    profesor.SetEdad(50)
    profesor.SetAntigüedad(15)
    profesor.SetTutorias([["Lunes","16-18"],["Jueves","12-14"],["Viernes","11-13"]])
    profesor.SetTelefono("654321098")
    profesor.MostrarProfesor()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Alumno:
        Nombre: Alfredo
        Apellidos: Moreno Muñoz
        Edad: 35
        Curso: Bachillerato
        Matrículas: ['Matemáticas', 'Tecnología', 'Inglés']
Profesor:
        Nombre: Profesor
        Apellidos: Casa Papel
        Edad: 50
        Antigüedad:
        Tutorias: [['Lunes', '16-18'], ['Jueves', '12-14'], ['Viernes', '11-13']]
        Telefono: 654321098
```

El último ejercicio del capítulo va a consistir en aplicar la técnica de herencia múltiple, con la que vamos a poder tener un objeto que herede de más de una clase.

En el ejercicio vamos a utilizar una parte de los ejercicios anteriores, las clases *Persona* y *Profesor*, y vamos a añadir dos nuevos, la clase *Investigador* que heredará de la clase persona y la clase *ProfesorUniversitario* que heredará de la clase *Profesor* y la clase *Investigador*. Veamos el detalle de cada una de las clases:

- **Clase**: Persona.
  - Atributos:
    - Nombre.
    - Apellidos.
    - Edad.
- **Clase**: Profesor.
  - **Hereda de**: Persona
  - Atributos:
    - Antigüedad.
    - Tutorías.
    - Teléfono.
- **Clase**: Investigador.
  - **Hereda de**: Persona
  - Atributos:
    - Especialidad.
    - Años.
- **Clase**: ProfesorUniversitario.
  - **Hereda de**: Persona e Investigador
  - Atributos:
    - Universidad.
    - Departamento.

Viendo en detalle la clase *ProfesorUniversitario* podemos ver que estará compuesta por los siguientes atributos:

- Nombre.
- Apellidos.
- Edad.
- Antigüedad.
- Tutorías.
- Teléfono.

- Especialidad.
- Años.
- Universidad.
- Departamento.

Veamos el código fuente:

```
class Persona:
    def __init__ (self):
        self.__Nombre = ""
        self.__Apellidos = ""
        self.__Edad = 0
    def GetNombre(self):
        return self.__Nombre
    def SetNombre(self,nombre):
        self.__Nombre = nombre
    def GetApellidos(self):
        return self.__Apellidos
    def SetApellidos(self,apellidos):
        self.__Apellidos = apellidos
    def GetEdad(self):
        return self.__Edad
    def SetEdad(self,edad):
        self.__Edad = edad

class Profesor(Persona):
    def __init__ (self):
        self.__Antigüedad = ""
        self.__Tutorias = ""
        self.__Telefono = ""
    def GetAntiguedad(self):
        return self.__Antigüedad
    def SetAntiguedad(self,antigüedad):
        self.__Antigüedad = antigüedad
    def GetTutorias(self):
        return self.__Tutorias
    def SetTutorias(self,tutorias):
        self.__Tutorias = tutorias
    def GetTelefono(self):
        return self.__Telefono
    def SetTelefono(self,telefono):
        self.__Telefono = telefono
```

```
class Investigador(Persona):
    def __init__ (self):
        self.__Especialidad = ""
        self.__Años = ""
    def GetEspecialidad(self):
        return self.__Especialidad
    def SetEspecialidad(self,especialidad):
        self.__Especialidad = especialidad
    def GetAños(self):
        return self.__Años
    def SetAños(self,años):
        self.__Años = años

class ProfesorUniversitario(Profesor,Investigador):
    def __init__ (self):
        self.__Universidad = ""
        self.__Departamento = ""
    def GetUniversidad(self):
        return self.__Universidad
    def SetUniversidad(self,universidad):
        self.__Universidad = universidad
    def GetDepartamento(self):
        return self.__Departamento
    def SetDepartamento(self,departamento):
        self.__Departamento = departamento
    def MostrarProfesorUniversitario(self):
        print("Profesor Universitario:")
        print("\tNombre:",self.GetNombre())
        print("\tApellidos:",self.GetApellidos())
        print("\tEdad:",self.GetEdad())
        print("\tAntigüedad:",self.GetAntiguedad())
        print("\tTutorias:",self.GetTutorias())
        print("\tTelefono:",self.GetTelefono())
        print("\tEspecialidad:",self.GetEspecialidad())
        print("\tAños::",self.GetAños())
        print("\tUniversidad:",self.GetUniversidad())
        print("\tDepartamento:",self.GetDepartamento())

profesor = ProfesorUniversitario()
profesor.SetNombre("Alfredo")
profesor.SetApellidos("Moreno Muñoz")
profesor.SetEdad(35)
profesor.SetAntiguedad(15)
profesor.SetTutorias([["Lunes","16-18"],["Jueves","12-14"],["Viernes","11-13"]])
profesor.SetTelefono("654321098")
```

```
profesor.SetEspecialidad("Desarrollo de Software")
profesor.SetAños(15)
profesor.SetUniversidad("Universidad de Extremadura")
profesor.SetDepartamento("Lenguajes y Sistemas informáticos")
profesor.MostrarProfesorUniversitario()
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Profesor Universitario:
        Nombre: Alfredo
        Apellidos: Moreno Muñoz
        Edad: 35
        Antigüedad: 15
        Tutorias: [['Lunes', '16-18'], ['Jueves', '12-14'], ['Viernes', '11-13']]
        Telefono: 654321098
        Especialidad: Desarrollo de Software
        Años:: 15
        Universidad: Universidad de Extremadura
        Departamento: Lenguajes y Sistemas informáticos
```

23

PILAS Y COLAS

En este capítulo vamos a dar un paso en lo que a estructuras de datos se refiere. Partiendo del capítulo en el que te explicamos las listas, vamos a explicarte una serie de estructuras que utilizan listas para implementar estructuras de datos lineales, pero más avanzadas. Vamos a explicarte y a enseñarte a implementar pilas y colas, estructuras muy útiles dentro del desarrollo de software.

En este capítulo además vas a afianzar más todavía los conceptos sobre programación orientada a objetos aprendidos en el capítulo anterior.

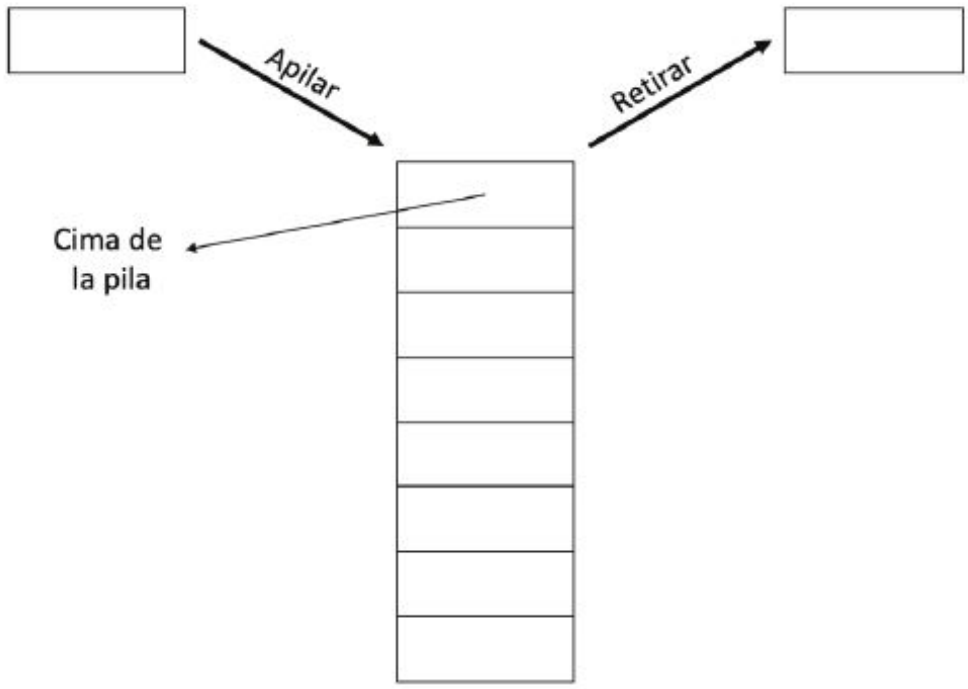
23.1 PILAS

Las pilas son estructuras de datos que podemos asemejarlas a una pila de platos. La pila está compuesta por elementos en los que el primer elemento es el elemento más nuevo en la misma. A la hora de coger un elemento cogeremos el elemento más arriba de la pila (el más nuevo) y a la hora de dejar un elemento lo dejaremos en lo más alto de la pila, de ahí su semejanza con una pila de platos.

Las pilas tienen dos operaciones principales:

- **Apilar**: coloca un elemento en la pila.
- **Retirar**: retira el último elemento apilado.

Veamos una pila gráficamente:



Las pilas tienen un modo de acceso a los elementos **LIFO**, que proviene del inglés "*Last In First Out*", que traducido significa "*Último en entrar primero en salir*".

23.1.1 Implementación

En este apartado vamos a ver cómo implementar la pila y vamos a realizar un ejercicio que simulará el funcionamiento de una pila.

Para la implementación de la pila vamos a definir una clase que se llamará *Pila* y tendrá los siguientes métodos:

- **Constructor**: inicializará la lista de elementos que componen la pila.
- **Apilar**: insertará un elemento en la cima de la pila.
- **Retirar**: retirará el elemento que se encuentra en la cima de la pila.

- **LeerCima**: devuelve el elemento que se encuentra encima de la pila sin retirarlo.
- **EstaVacia**: devuelve *True* en caso de que la pila esté vacía y *False* en caso contrario.
- **NumeroElementos**: devuelve el número de elementos que tiene la pila.
- **MostrarPila**: muestra la pila por pantalla.

En el simulador de una pila vamos a añadir un menú interactivo para que el usuario de la aplicación pueda manejar la pila, apilando elementos, retirando, mostrando la pila, etc. La operación *Retirar* deberá comprobar que la pila no está vacía antes de retirar e indicar que no se puede retirar el elemento por estar vacía en caso de estarlo. La operación para leer la cima de la pila deberá comprobar que la pila no está vacía, en caso de estarlo lo indicará por pantalla.

El código fuente es el siguiente:

```
class Pila:
    def __init__(self):
        self.__items = []

    def EstaVacia(self):
        if len(self.__items) == 0:
            return True
        else:
            return False

    def Apilar(self, item):
        self.__items.append(item)

    def Retirar(self):
        return self.__items.pop()

    def LeerCima(self):
        return self.__items[len(self.__items)-1]

    def NumeroElementos(self):
        return len(self.__items)

    def MostrarPila(self):
        print("Pila: ", self.__items,"<-- CIMA")
```

```
def SimuladorPila():
    fin = False
    pila = Pila()
    while not(fin):
        opc = input("Opcion:")
        if (opc=='1'):
            item = input("Introduzca elemento a apilar: ")
            pila.Apilar(item)
            print("Elemento apilado: ",item)
        elif(opc=='2'):
            if pila.EstaVacia():
                print("La pila está vacía, no puede retirarse ningún elemento")
            else:
                item = pila.LeerCima()
                pila.Retirar()
                print("Elemento retirado: ",item)
        elif(opc=='3'):
            if pila.EstaVacia():
                print("La pila está vacía, no puede leerse la cima")
            else:
                print("La cima es: ", pila.LeerCima())
        elif(opc=='4'):
            print("La pila tiene ",pila.NumeroElementos(), " elementos")
        elif(opc=='5'):
            if pila.EstaVacia():
                print("La pila está vacía")
            else:
                print("La pila no está vacía")
        elif(opc=='6'):
            pila.MostrarPila()
        elif(opc=='7'):
            fin = 1

print ("""******************
Simulador de Pila
*****************
Menu
1) Apilar
2) Retirar
3) Leer cima
4) Número de elementos
5) ¿Está vacía?
6) Mostrar pila
7) Salir""")
SimuladorPila()
```

A continuación vamos a ver diferentes capturas con un ejemplo de ejecución del código fuente anterior. La primera imagen muestra el menú en el que se selecciona lo que se quiere hacer con la aplicación:

```
****************
Simulador de Pila
****************
Menu
1) Apilar
2) Retirar
3) Leer cima
4) Número de elementos
5) ¿Está vacía?
6) Mostrar pila
7) Salir
```

La siguiente imagen muestra un proceso de añadir tres elementos a la pila y después mostrar la pila. Puedes ver como el último elemento añadido a la pila es el que se encuentra como cima a la hora de mostrar la pila por pantalla:

```
Opcion:1
Introduzca elemento a apilar: Perro
Elemento apilado:  Perro
Opcion:1
Introduzca elemento a apilar: Gato
Elemento apilado:  Gato
Opcion:1
Introduzca elemento a apilar: Caballo
Elemento apilado:  Caballo
Opcion:6
Pila:  ['Perro', 'Gato', 'Caballo'] <-- CIMA
```

La siguiente imagen muestra las opciones de mostrar el número de elementos que tiene la pila, la comprobación de si la pila está vacía o no, la cima de la pila y por último retira un elemento mostrando como ha quedado la pila después de retirar la cima. Puedes comprobar que ahora la cima de la pila la ocupa el elemento que antes estaba en segunda posición:

```
Opcion:4
La pila tiene  3  elementos
Opcion:3
La cima es:  Caballo
Opcion:5
La pila no está vacía
Opcion:2
Elemento retirado:  Caballo
Opcion:6
Pila:  ['Perro', 'Gato'] <-- CIMA
```

La siguiente imagen muestra la opción de mostrar el número de elementos que tiene la pila y posteriormente va retirando los elementos de la pila y mostrando

como va quedando la pila hasta quedar vacía, momento en el que se comprueba cuántos elementos tiene la pila y si está vacía:

```
Opcion:4
La pila tiene  2  elementos
Opcion:2
Elemento retirado:  Gato
Opcion:6
Pila:  ['Perro'] <-- CIMA
Opcion:2
Elemento retirado:  Perro
Opcion:6
Pila:  [] <-- CIMA
Opcion:5
La pila está vacía
Opcion:4
La pila tiene  0  elementos
```

23.2 COLAS

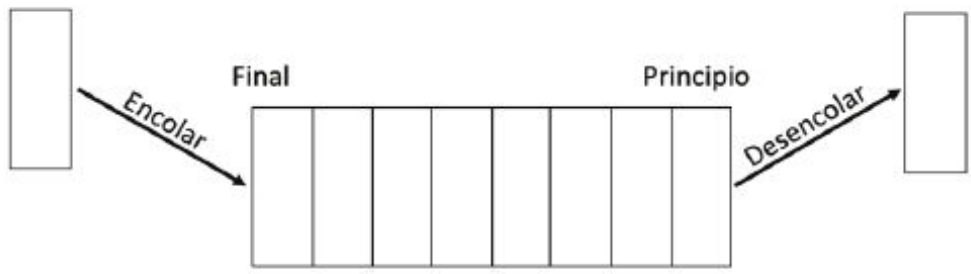
Las colas son estructuras de datos que funcionalmente son iguales a las colas que puedes encontrar en el mundo real: aeropuertos, supermercados, etc.

Funcionalmente hablando, en las colas la primera persona o el primer elemento que entra será el primero en salir, son estructuras tipo **FIFO**, del inglés "*First In First Out*", traducido al castellano "*Primero en entrar primero en salir*".

Las colas tienen dos operaciones principales:

- **Encolar**: introduce un elemento en la última posición de la cola.
- **Desencolar**: retira el primer elemento de la cola.

Veamos una cola gráficamente:



23.2.1 Implementación

En este apartado vamos a ver cómo implementar la cola y vamos a realizar un ejercicio que simulará el funcionamiento de una cola.

Para la implementación de la cola vamos a definir una clase que se llamará *Cola* y tendrá los siguientes métodos:

- **Constructor**: inicializará la lista de elementos que componen la cola.
- **Encolar**: insertará un elemento al final de la cola.
- **Desencolar**: retirará el elemento que se encuentra en la primera posición de la cola.
- **LeerPrimero**: devuelve el elemento que se encuentra en la primera posición de la cola sin desencolarlo.
- **EstaVacia**: devuelve *True* en caso de que la cola esté vacía y *False* en caso contrario.
- **NumeroElementos**: devuelve el número de elementos que tiene la cola.
- **MostrarCola**: muestra la cola por pantalla.

En el simulador de una cola vamos a añadir un menú interactivo para que el usuario de la aplicación pueda manejar la cola, encolando elementos, desencolando, mostrando el primer elemento de la cola, etc. La operación *Desencolar* deberá comprobar que la cola no está vacía antes de desencolar e indicar que no se puede retirar el elemento por estar vacía en caso de estarlo. La operación para leer el primer elemento de la cola deberá comprobar que la cola no está vacía, en caso de estarlo lo indicará por pantalla.

El código fuente es el siguiente:

```
class Cola:
    def __init__(self):
        self.__items = []

    def EstaVacia(self):
        if len(self.__items) == 0:
            return True
        else:
            return False
```

```
    def Encolar(self, item):
        self.__items.insert(0,item)

    def Desencolar(self):
        return self.__items.pop()

    def LeerPrimerElemento(self):
        return self.__items[len(self.__items)-1]

    def NumeroElementos(self):
        return len(self.__items)

    def MostrarCola(self):
        print("Cola: ", self.__items,"<-- Primer elemento")

def SimuladorCola():
    fin = False
    cola = Cola()
    while not(fin):
        opc = input("Opcion:")
        if (opc=='1'):
            item = input("Introduzca elemento a encolar: ")
            cola.Encolar(item)
            print("Elemento encolado: ",item)
        elif(opc=='2'):
            if cola.EstaVacia():
                print("La cola está vacía, no puede desencolarse ningún elemen-
to")
            else:
                item = cola.LeerPrimerElemento()
                cola.Desencolar()
                print("Elemento desencolado: ",item)
        elif(opc=='3'):
            if cola.EstaVacia():
                print("La cola está vacía, no puede leerse ningún elemento")
            else:
                print("El primer elemento es: ", cola.LeerPrimerElemento())
        elif(opc=='4'):
            print("La cola tiene ",cola.NumeroElementos(), " elementos")
        elif(opc=='5'):
            if cola.EstaVacia():
                print("La cola está vacía")
            else:
                print("La cola no está vacía")
        elif(opc=='6'):
            cola.MostrarCola()
```

```
        elif(opc=='7'):
            fin = 1

print ("""********************
Simulador de Cola
*****************
Menu
1) Encolar
2) Desencolar
3) Leer primer elemento
4) Número de elementos
5) ¿Está vacía?
6) Mostrar cola
7) Salir""")
SimuladorCola()
```

A continuación vamos a ver diferentes capturas con un ejemplo de ejecución del código fuente anterior. La primera imagen muestra el menú en el que se selecciona lo que se quiere hacer con la aplicación:

```
*****************
Simulador de Cola
*****************
Menu
1) Encolar
2) Desencolar
3) Leer primer elemento
4) Número de elementos
5) ¿Está vacía?
6) Mostrar cola
7) Salir
Opcion:
```

La siguiente imagen muestra un proceso de añadir tres elementos a la cola y después mostrar la cola. Puedes ver como el primer elemento añadido a la cola es el que se encuentra como primer elemento a la hora de mostrar la cola por pantalla:

```
Opcion:1
Introduzca elemento a encolar: Alfredo
Elemento encolado:  Alfredo
Opcion:1
Introduzca elemento a encolar: Python
Elemento encolado:  Python
Opcion:1
Introduzca elemento a encolar: Editorial Ra-Ma
Elemento encolado:  Editorial Ra-Ma
Opcion:6
Cola:  ['Editorial Ra-Ma', 'Python', 'Alfredo'] <-- Primer elemento
```

La siguiente imagen muestra las opciones de mostrar el número de elementos que tiene la cola, la comprobación de si la cola está vacía o no, el primer elemento de la cola y por último desencola un elemento mostrando como ha quedado la cola después de desencolar el primer elemento. Puedes comprobar que ahora la primera posición de la cola la ocupa el elemento que antes estaba en segunda posición:

```
Opcion:4
La cola tiene  3  elementos
Opcion:5
La cola no está vacía
Opcion:3
El primer elemento es:  Alfredo
Opcion:2
Elemento desencolado:  Alfredo
Opcion:6
Cola:  ['Editorial Ra-Ma', 'Python'] <-- Primer elemento
```

La siguiente imagen muestra la opción de mostrar el número de elementos que tiene la cola y posteriormente va desencolando los elementos de la cola y mostrando como va quedando la cola hasta quedar vacía, momento en el que se comprueba cuántos elementos tiene la cola y si está vacía:

```
Opcion:4
La cola tiene  2  elementos
Opcion:2
Elemento desencolado:  Python
Opcion:6
Cola:  ['Editorial Ra-Ma'] <-- Primer elemento
Opcion:2
Elemento desencolado:  Editorial Ra-Ma
Opcion:6
Cola:  [] <-- Primer elemento
Opcion:4
La cola tiene  0  elementos
Opcion:5
La cola está vacía
```

24

LIBRERÍA ESTÁNDAR

En este capítulo vamos a hablarte sobre la librería estándar de Python, que es un conjunto de módulos y paquetes que son distribuidos junto a Python. Hasta ahora hemos utilizado una serie de funciones básicas que provee Python. En este capítulo vamos a utilizar un conjunto extra de funciones, variables, clases…

En el capítulo vamos a aprender los módulos más importantes, cómo importarlos a nuestros programas y cómo utilizarlos. Uno de los módulos más importantes, el módulo *unittest*, no lo vamos a abordar en este capítulo, lo trataremos en un capítulo aparte más adelante. Veremos los siguientes módulos:

- **random**: módulo que permite realizar elecciones al azar.
- **sys:** módulo para añadir argumentos en el arranque de los programas.
- **os**: módulo para interactuar con el sistema operativo.
- **shutil**: módulo para administrar ficheros y grupos de ficheros.
- **math**: módulo que añade funciones matemáticas.
- **statistics**: módulo que permite utilizar funciones estadísticas básicas.
- **datetime**: módulo que permite manejar fechas y tiempos de forma sencilla.

¡Vamos a verlas en detalle!

24.1 MÓDULO RANDOM

El módulo *random* va a permitirte hacer elecciones de forma aleatoria, tanto elegir al azar un valor de una lista como generar números aleatorios u obtener un número al azar dentro de un rango.

Las funciones más importantes son:

- **randrange**: devuelve un número aleatorio en un rango especificado por parámetro.
  - Retorno: número aleatorio.
  - Parámetros: rango utilizado para obtener el número aleatorio.
- **sample**: devuelve una lista de números aleatorios dentro de un rango establecido por parámetro.
  - Retorno: lista con números aleatorios.
  - Parámetros: tiene dos parámetros, el primero es el rango sobre el que se seleccionarán los números aleatorios y el segundo es el número de elementos que tendrá la lista que se devuelve.
- **random**: devuelve un número en coma flotante.
  - Retorno: número en coma flotante.
  - Parámetros: no tiene.
- **choice**: selecciona un elemento de la lista que recibe por parámetro al azar.
  - Retorno: un elemento de la lista.
  - Parámetros: lista de elementos.

En el siguiente ejercicio vamos a hacer un ejemplo de uso del módulo, para ello tienes que importarlo en tu código fuente utilizando la sentencia *import random*. El código fuente es el siguiente:

```
import random
print("Número aleatorio del 1 al 1000: ", random.randrange(1000))
print("Lista aleatoria de números entre el 1 y el 1000: ",random.
sample(range(1000), 3))
print("Número aleatorio en coma flotante: ",random.random())
print("Elección aleatoria [Python, Java, C#, Ruby, Go]: ", random.
choice(['Python', 'Java', 'C#', 'Ruby', 'Go']))
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Número aleatorio del 1 al 1000:  813
Lista aleatoria de números entre el 1 y el 1000:  [682, 840, 462]
Número aleatorio en coma flotante:  0.9870719321759741
Elección aleatoria [Python, Java, C#, Ruby, Go]:  Python
```

24.2 MÓDULO SYS

El módulo *sys* va a permitirte utilizar los argumentos que se le pasan a los programas a la hora de ejecutarlos.

En el siguiente ejercicio vamos a hacer un ejemplo de uso del módulo, para ello tienes que importarlo en tu código fuente utilizando la sentencia *import sys*. El código fuente es el siguiente:

```
import sys
for item in sys.argv:
    print(item)
```

El ejercicio lo que va a hacer es mostrar por pantalla los parámetros recibidos en *sys.argv*. Para ejecutar el ejercicio te recomendamos que lo hagas desde el terminal del sistema operativo ya que es más sencillo.

La siguiente imagen muestra un ejemplo de varias ejecuciones del código fuente anterior:

```
Alfre:Desktop alfre$ python sys.py
sys.py
Alfre:Desktop alfre$ python sys.py param1 param2
sys.py
param1
param2
Alfre:Desktop alfre$ python sys.py 29 11 17 file.log
sys.py
29
11
17
file.log
Alfre:Desktop alfre$
```

24.3 MÓDULO OS Y SHUTIL

El módulo *os* nos va a permitir interactuar con el sistema operativo y manejar todas aquellas funcionalidades que ofrece.

El módulo *os* es un módulo muy amplio y algunas de sus funcionalidades requieren conocimientos profundos de sistemas operativos, por lo que en este punto vamos a hacer un pequeño ejercicio simplemente para mostrarte cómo se utiliza con una serie de funciones muy básicas. Si quieres adentrarte más en el módulo te recomendamos ir a la documentación que tienes disponible en la página web de Python.

En el ejercicio vamos a utilizar las siguientes funciones:

- **getcwd**: devuelve el directorio actual de trabajo de la aplicación.
- **chdir**: cambia el directorio de trabajo de la aplicación al pasado por parámetro.
- **getpid**: devuelve el identificador del proceso del aplicativo.
- **getuid**: devuelve el identificador del usuario del proceso del aplicativo.

El código fuente es el siguiente:

```
import os
print("Directorio de trabajo actual: ",os.getcwd())
os.chdir("/Users/alfre/Desktop/")
print("Nuevo directorio de trabajo: ",os.getcwd())
print("ID del proceso: ", os.getpid())
print("ID del usuario del proceso: ", os.getuid())
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Directorio de trabajo actual:  /Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/LibreriaStandard
Nuevo directorio de trabajo:  /Users/alfre/Desktop
ID del proceso:  1600
ID del usuario del proceso:  501
```

En el segundo ejercicio vamos a aprender a utilizar el módulo *shutil* que nos va a permitir administrar ficheros de forma sencilla y vamos a aprender a manejar otras funciones del módulo *os*.

Del módulo *os* vamos a aprender a manejar las siguientes funciones:

- **listdir**: lista el contenido del directorio de trabajo actual.
- **mkdir**: crea un nuevo directorio dentro del directorio de trabajo actual.
- **rename**: renombra un fichero.

Del módulo *shutil* vamos a aprender a manejar las siguientes funciones:

- **copy**: realizar la copia del fichero parametrizado en uno nuevo con el nombre especificado por parámetro.
- **move**: mueve el fichero especificado por parámetro a la ruta especificada por parámetro.

El código fuente del ejercicio es el siguiente:

```
import os
import shutil
print("¡Cambiando directorio de trabajo!")
os.chdir("/Users/alfre/Desktop/Ejercicio/")
print("Nuevo directorio de trabajo: ",os.getcwd())
print("Contenido del directorio: ", os.listdir())
print("¡Copiando el fichero prueba.txt!")
shutil.copy("prueba.txt","NuevaPrueba.txt")
print("Contenido del directorio: ", os.listdir())
print("¡Renombrando el fichero NuevaPrueba.txt!")
os.rename("NuevaPrueba.txt","Ejercicio.txt")
print("Contenido del directorio: ", os.listdir())
print("¡Creando el nuevo directorio!")
os.mkdir("NuevoDirectorio")
print("Contenido del directorio: ", os.listdir())
print("¡Moviendo el fichero al nuevo directorio!")
shutil.move("Ejercicio.txt","NuevoDirectorio")
print("Contenido del directorio: ", os.listdir())
print("¡Cambiando directorio de trabajo!")
os.chdir("/Users/alfre/Desktop/Ejercicio/NuevoDirectorio")
print("Nuevo directorio de trabajo: ",os.getcwd())
print("Contenido del directorio: ", os.listdir())
```

El ejercicio consiste en cambiar el directorio de trabajo de la aplicación, copiar un fichero y posteriormente renombrarlo, después de eso se creará una carpeta nueva dentro del directorio de trabajo actual y moveremos el fichero nuevo que hemos copiado a dicha carpeta. Por último se muestra el contenido de ambos directorio, el inicial en el que teníamos el fichero y el final que hemos creado.

La siguiente imagen muestra un ejemplo de ejecución del ejercicio anterior:

```
¡Cambiando directorio de trabajo!
Nuevo directorio de trabajo:  /Users/alfre/Desktop/Ejercicio
Contenido del directorio:  ['prueba.txt']
¡Copiando el fichero prueba.txt!
Contenido del directorio:  ['NuevaPrueba.txt', 'prueba.txt']
¡Renombrando el fichero NuevaPrueba.txt!
Contenido del directorio:  ['Ejercicio.txt', 'prueba.txt']
¡Creando el nuevo directorio!
Contenido del directorio:  ['NuevoDirectorio', 'Ejercicio.txt', 'prueba.txt']
¡Moviendo el fichero al nuevo directorio!
Contenido del directorio:  ['NuevoDirectorio', 'prueba.txt']
¡Cambiando directorio de trabajo!
Nuevo directorio de trabajo:  /Users/alfre/Desktop/Ejercicio/NuevoDirectorio
Contenido del directorio:  ['Ejercicio.txt']
```

El último ejercicio que vamos a realizar consiste en borrar tanto los ficheros como el directorio que hemos utilizado en el ejercicio anterior. Para ello vamos a utilizar dos funciones, una del módulo *shutil* que elimina un directorio y todo su contenido (*rmtree*) y otra del módulo *os* que elimina un fichero concreto (*remove*).

El código fuente es el siguiente:

```
import os
import shutil
print("¡Cambiando directorio de trabajo!")
os.chdir("/Users/alfre/Desktop/Ejercicio/")
print("Nuevo directorio de trabajo: ",os.getcwd())
print("Contenido del directorio: ", os.listdir())
print("¡Eliminando el directorio NuevoDirectorio!")
shutil.rmtree("NuevoDirectorio")
print("Contenido del directorio: ", os.listdir())
print("¡Borrando el fichero prueba.txt!")
os.remove("prueba.txt")
print("Contenido del directorio: ", os.listdir())
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
¡Cambiando directorio de trabajo!
Nuevo directorio de trabajo:  /Users/alfre/Desktop/Ejercicio
Contenido del directorio:  ['NuevoDirectorio', 'prueba.txt']
¡Eliminando el directorio NuevoDirectorio!
Contenido del directorio:  ['prueba.txt']
¡Borrando el fichero prueba.txt!
Contenido del directorio:  []
```

24.4 MÓDULO MATH

El módulo *math* nos va a dar acceso a una serie de funciones matemáticas y constantes matemáticas que te serán útiles en algún programa que desarrolles.

El módulo *math* es muy amplio, aquí vamos a ponerte aquellas funciones y constantes más importantes y relevantes, si quieres ver todas las funciones y constantes incluidas te recomendamos que las consultes en la documentación oficial disponible en la página web de Python.

Las funciones que vamos a ver son las siguientes:

- **fabs**: función que retorna el valor absoluto de un número.

- Retorno: valor absoluto.
- Parámetros: número del que se quiere calcular el valor absoluto.

◤ **factorial**: función que calcula el factorial de un número.

- Retorno: valor del factorial del número.
- Parámetros: número del que se quiere calcular el factorial.

◤ **gcd**: función que calcula el máximo común divisor de dos números.

- Retorno: el valor del máximo común divisor de los dos parámetros.
- Parámetros: los dos números sobre los que se quiere calcular el máximo común divisor.

◤ **isnan**: función que comprueba si el parámetro no es un número.

- Retorno: retornará *False* en caso de que sea un número y *True* en caso contrario.
- Parámetros: el número sobre el que se quiere comprobar que no sea un número.

◤ **log**: función que calcula el logaritmo en base X de un número.

- Retorno: valor del logaritmo.
- Parámetros: tiene dos parámetros, el primero es el número del que se quiere calcular el logaritmo y el segundo la base del logaritmo.

◤ **pow**: función que calcula la potencia de un número.

- Retorno: valor de la potencia calculada.
- Parámetros: tiene dos parámetros, el primero de ellos es la base y el segundo el exponente.

◤ **sqrt**: función que calcula la raíz cuadrada de un número.

- Retorno: valor de la raíz cuadrada.
- Parámetros: el número del que se quiere calcular el valor de la raíz cuadrada.

◤ **acos**: función que calcula el arcocoseno en radianes.

- Retorno: valor del arcocoseno.
- Parámetros: ángulo del que se quiere calcular el arcocoseno.

◤ **asin**: función que calcula el arcoseno en radianes.

- Retorno: valor del arcoseno.
- Parámetros: ángulo del que se quiere calcular el arcoseno.

- **atan**: función que calcula la arcotangente en radianes.
  - Retorno: valor de la arcotangente.
  - Parámetros: ángulo del que se quiere calcular la arcotangente.
- **cos**: función que calcula el coseno en radianes.
  - Retorno: valor del coseno.
  - Parámetros: ángulo del que se quiere calcular el coseno.
- **sin**: función que calcula el seno en radianes.
  - Retorno: valor del seno.
  - Parámetros: ángulo del que se quiere calcular el seno.
- **tan**: función que calcula la tangente en radianes.
  - Retorno: valor de la tangente.
  - Parámetros: ángulo del que se quiere calcular la tangente.
- **degrees**: función que convierte un ángulo en radianes en grados.
  - Retorno: ángulo en grados.
  - Parámetros: ángulo en radianes.
- **radians**: función que convierte un ángulo en grados en radianes.
  - Retorno: ángulo en radianes.
  - Parámetros: ángulo en grados.

Las constantes que vamos a ver son las siguientes:

- **pi**: tiene el valor de la constante π.
- **e**: tiene el valor de la constante e.
- **tau**: tiene el valor de la constante τ.
- **inf**: representación del valor infinito.
- **nan**: representación de NaN ("*Not a Number*").

El primer ejercicio te va a enseñar a utilizar todas las funciones descritas anteriormente. El código fuente es el siguiente:

```
import math
print("El valor absoluto de -123 es: ", math.fabs(-123))
print("El factorial de 7 es: ", math.factorial(7))
print("El máximo común divisor de 9 y 15 es: ", math.gcd(9,15))
print("isnan(3): ",math.isnan(3))
```

```
print("El logaritmo en base 10 de 100 es: ", math.log(89,10))
print("El valor de 3 elevado a 4 es: ", math.pow(3,4))
print("La raiz cuadrada de 49 es: ", math.sqrt(49))
print("El ángulo 90 en radianes es: ", math.radians(90))
print("El ángulo 1.23 en grados es: ", math.degrees(1.23))
print("El seno de un ángulo de 90 es: ", math.sin(math.radians(90)))
print("El coseno de un ángulo de 90 es: ", math.cos(math.radians(90)))
print("La tangente de un ángulo de 90 es: ", math.tan(math.radians(90)))
print("El arcoseno de 1 es: ", math.degrees(math.asin(1)))
print("El arcocoseno de 1 es: ", math.degrees(math.acos(1)))
print("La arcotangente de 1 es: ", math.degrees(math.atan(1)))
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
El valor absoluto de -123 es:  123.0
El factorial de 7 es:  5040
El máximo común divisor de 9 y 15 es:  3
isnan(3):  False
El logaritmo en base 10 de 100 es:  1.9493900066449126
El valor de 3 elevado a 4 es:  81.0
La raiz cuadrada de 49 es:  7.0
El ángulo 90 en radianes es:  1.5707963267948966
El ángulo 1.23 en grados es:  70.47380880109125
El seno de un ángulo de 90 es:  1.0
El coseno de un ángulo de 90 es:  6.123233995736766e-17
La tangente de un ángulo de 90 es:  1.6331239353195370e+16
El arcoseno de 1 es:  90.0
El arcocoseno de 1 es:  0.0
La arcotangente de 1 es:  45.0
```

El segundo ejercicio te va a enseñar a cómo utilizar las constantes descritas anteriormente. El código fuente es el siguiente:

```
import math
print("El valor de pi es: ", math.pi)
print("El valor de e es: ", math.e)
print("El valor de tau es: ", math.tau)
print("El valor infinito es: ", math.inf)
print("El valor NaN es: ", math.nan)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
El valor de pi es:  3.141592653589793
El valor de e es:  2.718281828459045
El valor de tau es:  6.283185307179586
El valor infinito es:  inf
El valor NaN es:  nan
```

24.5 MÓDULO STATISTICS

El módulo *statistics* te va a permitir realizar cálculos estadísticos sobre conjuntos de datos.

Las funciones más importantes del módulo son las siguientes:

- **mean**: calcula la media de un listado de números pasados como parámetro.
- **median**: calcula la mediana de un listado de números pasados como parámetro.
- **median_low**: calcula la mediana inferior de un listado de números pasados como parámetro.
- **median_high**: calcula la mediana superior de un listado de números pasados como parámetro.
- **mode**: calcula la moda de un listado de números pasados como parámetro.
- **variance**: calcula la varianza de un listado de números pasados como parámetro.

El ejercicio que vamos a realizar consiste en aprender a aplicar las funciones que acabamos de explicar. El código fuente es el siguiente:

```
import statistics
print("La media de 5,3,7,9,4,2,2 es: ", statistics.mean([5,3,7,9,4,2,2]))
print("La mediana de 5,3,7,9,4,2 es: ", statistics.median([5,3,7,9,4,2]))
print("La mediana inferior de 5,3,7,9,4,2 es: ", statistics.median_
low([5,3,7,9,4,2]))
print("La mediana superior de 5,3,7,9,4,2 es: ", statistics.median_
high([5,3,7,9,4,2]))
print("La moda de 5,3,7,9,4,2,2 es: ", statistics.mode([5,3,7,9,4,2,2]))
print("La varianza de 5,3,7,9,4,2,2 es: ", statistics.variance([5,3,7,9,4,2,2]))
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
La media de 5,3,7,9,4,2,2 es:  4.571428571428571
La mediana de 5,3,7,9,4,2 es:  4.5
La mediana inferior de 5,3,7,9,4,2 es:  4
La mediana superior de 5,3,7,9,4,2 es:  5
La moda de 5,3,7,9,4,2,2 es:  2
La varianza de 5,3,7,9,4,2,2 es:  6.9523809523809526
```

24.6 MÓDULO DATETIME

El módulo *datetime* es probablemente uno de los mas grande incluidos dentro de la librería estándar de Python y va a permitirte manipular fechas de forma sencilla.

En este apartado vamos a explicarte una serie de funciones que son básicas para trabajar con fechas. Existen funciones mucho más complejas pero que no se utilizan mucho. Las funciones que vamos a ver son las siguientes:

- **datetime.now**: función que retorna la fecha y la hora de ahora mismo.
- **date.today**: función que retorna el día de hoy.
- **date**: función que permite crear un objeto de tipo fecha pasándole como parámetros el año, el mes y el día.
- **datetime**: función que permite crear un objeto de tipo *Datetime* pasándole como parámetros el año, el mes, el día, la hora, los minutos, los segundos y los microsegundos.

El primer ejercicio consiste en aprender a utilizar estas funciones. El código fuente es el siguiente:

```
import datetime
print("El dia y la hora de hoy es: ", datetime.datetime.now())
print("El dia de hoy es: ", datetime.date.today())
fecha = datetime.date(1984,5,10)
print(fecha)
fechahora = datetime.datetime(2017,11,29,23,19,00,123)
print(fechahora)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
El dia y la hora de hoy es:  2019-06-23 12:19:52.056720
El dia de hoy es:  2019-06-23
1984-05-10
2017-11-29 23:19:00.000123
```

El segundo ejercicio consiste en aprender a mostrar por pantalla un objeto de tipo *Datetime* accediendo a los diferentes elementos que lo componen (año, mes, etc). El código fuente es el siguiente:

```
import datetime
fecha = datetime.datetime(2017,11,29,23,19,00,123)
print(fecha)
print("Año: ",fecha.year)
print("Mes: ",fecha.month)
print("Día: ",fecha.day)
print("Hora: ",fecha.hour)
print("Minutos: ",fecha.minute)
print("Segundos: ",fecha.second)
print("Microsegundos: ",fecha.microsecond)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
2017-11-29 23:19:00.000123
Año:  2017
Mes:  11
Día:  29
Hora:  23
Minutos:  19
Segundos:  0
Microsegundos:  123
```

El último ejercicio consiste en aprender cómo calcular la diferencia entre dos *datetime*. La forma de realizar el cálculo es realizando una resta de ambos *datetime*. El código fuente es el siguiente:

```
import datetime
ahora = datetime.datetime.now()
fecha = datetime.datetime(2017,11,29,23,19,00,123)
print("Se va a realizar la resta de las siguientes fechas:")
print("1.- ", ahora)
print("2.- ", fecha)
resultado = ahora - fecha
print("Resultado: ",resultado)
```

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Se va a realizar la resta de las siguientes fechas:
1.-  2019-06-23 12:30:24.356922
2.-  2017-11-29 23:19:00.000123
Resultado:  570 days, 13:11:24.356799
```

Los dos últimos ejercicios que acabamos de hacer han sido utilizando el tipo de dato *datetime*. La forma de acceso a los elementos y el cálculo de la diferencia entre dos objetos se realizan de la misma forma con objetos de tipo *date*, pero teniendo en cuenta que únicamente están compuestos por año, mes y día.

25

PROGRAMACIÓN PARALELA

En este capítulo se va a explicar la programación paralela, tipos de programación paralela, sus ventajas y desventajas, hilos, procesos, programación concurrente y el **G**lobal **I**nterpreter **L**ock (GIL), junto con una serie de ejercicios para consolidar dichos conceptos.

25.1 INTRODUCCIÓN A LA PROGRAMACIÓN PARALELA

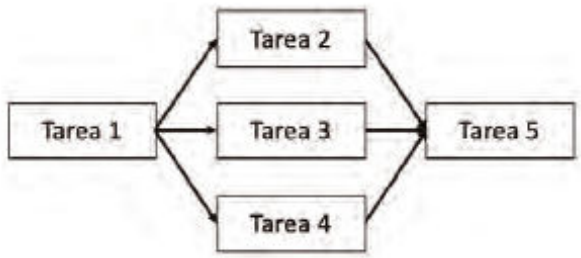
Aunque actualmente en todas las ramas de la informática el término de paralelismo es ampliamente conocido, no siempre ha sido así, ya que su aparición es posterior a las primeras computadoras.

La programación paralela tiene su origen en la necesidad de resolver problemas con tiempo de cómputo elevado. A finales de los años 50 se empezó a trabajar en ordenadores que permitieran trabajar de forma paralela, y fue durante los años 60 y 70 cuando aparecieron las primeras supercomputadoras, con multitud de procesadores y que trabajaban con datos compartidos. En la década de los 80 se fabricó el primer supercomputador para aplicaciones científicas que utilizaba procesadores que se podían adquirir en tiendas a pie de calle.

Pero… ¿Qué es la computación paralela? Es la utilización de múltiples recursos computacionales mediante los cuales se pueden resolver diferentes problemas de forma simultánea.

La siguiente imagen muestra la ejecución de tareas de forma secuencial, primero se ejecuta la tarea número 1 y una vez acaba, empieza la ejecución de la tarea 2, y así sucesivamente:

En la siguiente imagen se muestra una ejecución de tareas que combina ejecución secuencial con ejecución paralela. La primera tarea que se ejecuta es la número 1, posteriormente se ejecutan a la misma vez las tareas número 2, 3 y 4, la tarea 5 se ejecutará una vez acaben su ejecución paralela las anteriores:



Los pasos básicos para resolver un problema de forma paralela son los siguientes:

1. División del problema en subproblemas independientes que pueden ser resueltos de forma simultánea.
2. Descomponer cada subproblema en instrucciones.
3. Ejecución de cada subproblema en un procesador.
4. Mediante un mecanismo global de control y coordinación se resuelve el problema general con la unión de las resoluciones de los subproblemas.

25.2 TIPOS DE PARALELISMO

Una vez entendido qué es el paralelismo, es la hora de entender qué tipos de paralelismos existen. Hemos dividido la clasificación del paralelismo en dos grupos, el primer grupo en función del método utilizado para conseguir el paralelismo durante la ejecución de nuestros programas y el segundo grupo en función de cómo se indica qué código es paralelizable y cómo.

La primera clasificación es la siguiente, y va en función del método utilizado para paralelizar la ejecución de un programa:

- **A nivel de bit**: utiliza el tamaño de la cadena de bits que representan las instrucciones que tiene que realizar el procesador para realizar más de una instrucción a la vez. Por ejemplo, supongamos que queremos realizar una suma y nuestro procesador es de 8 bits y la suma son 16 bits, en ese caso necesitaríamos dos instrucciones de 8 bits; si tuviéramos un procesador de 16 bits se podría realizar únicamente con una instrucción. El método de paralelismo a nivel de bits no ha sido evolucionado desde que aparecieron las arquitecturas de procesador de 32 y 64 bits.

- **A nivel de instrucción**: consiste en la alteración de las instrucciones del programa agrupándolas para que puedan ser ejecutadas en paralelo sin alterar el resultado final. La idea surge de las tuberías (pipeline), se puede introducir más agua en las tuberías sin necesidad de esperar a que salga toda el agua que tiene. Los procesadores modernos tienen una pipeline que separa las instrucciones en varias etapas, en las que la etapa X necesitará la salida de la etapa X-1 y la salida de la etapa X será necesaria para la etapa X+1. La siguiente imagen muestra una comparativa del procesamiento de instrucciones sin utilizar pipeline y utilizando pipeline, tal y como puedes observar el tiempo de ejecución utilizando pipelines es muy inferior:

El tiempo en ejecutar los dos paquetes de instrucciones sin utilizar pipeline sería de "8" (vamos a suponer que cada cajita es 1) mientras que si utilizamos pipeline sería de "5". En la primera imagen se ejecutan primero todas las instrucciones del primer grupo y después todas las del segundo, mientras que en la segunda imagen se ejecuta primero a instrucción número 1 del primer grupo, posteriormente la número 2 del primer grupo junto con la 1 del segundo, la 3 del primero junto a la 2 del segundo, la 4 del primero junto a la 3 del segundo y para acabar la 4 del segundo.

- **A nivel de datos**: consiste en dividir los datos con los que trabajar entre varios procesos. Un ejemplo puede ser procesar miles de ficheros, pues dividiéndolos en dos procesos cada uno haría una mitad.
- **A nivel de tareas**: consiste en la división de tareas independientes unas de las otras entre los diferentes procesos.

La segunda clasificación se corresponde con cómo se indica qué código es paralelizable y cómo:

- **Paralelismo automático**: en este tipo de paralelismo el compilador de código fuente es quien identifica posible código que puede ejecutarse de forma paralela.
- **Paralelismo manual**: en este tipo de paralelismo es la persona que escribe el código fuente, quién utilizando directivas del compilador, indica cómo quiere paralelizar el código fuente.

Lo más normal a la hora de paralelizar es utilizar un método que utilice paralelismo automático y paralelismo manual.

25.3 VENTAJAS Y DESVENTAJAS

La programación paralela presenta una serie de ventajas respecto a la programación secuencial, son las siguientes:

- Resolución de problemas que no se pueden resolver mediante programación secuencial.
- Resolución de problemas que no se pueden resolver en un tiempo razonable de forma secuencial.
- Los problemas a resolver pueden ordenarse para optimizar el rendimiento y/o tiempo de resolución.
- Permite ejecutar código más rápidamente.

- Permite la ejecución simultánea de más de una instrucción.
- Permite dividir una tarea compleja en varias tareas independiente, lo que permite que las tareas a resolver sean mayores que en el paradigma secuencial ya que son divididas en otras más pequeñas.

No todo son ventajas en la programación paralela, también nos encontramos una serie de desventajas, que son las siguientes:

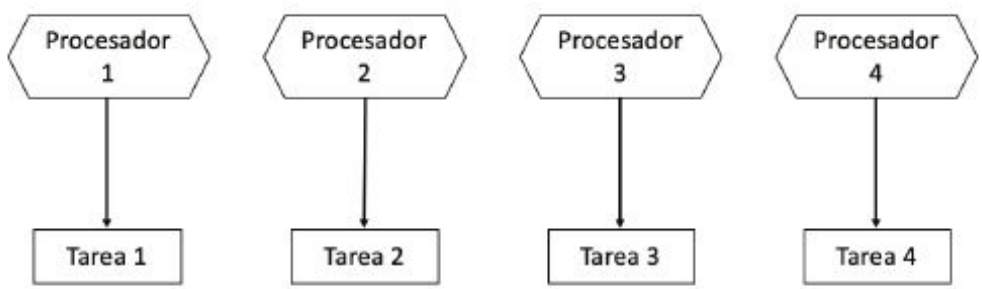
- Mayor dificultad a la hora de escribir programas.
- Mayor consumo de energía.
- Dificultad a la hora de sincronizar las diferentes tareas, por lo que pueden existir retardos a la hora de sincronizar procesos.
- Pueden existir corrupciones de datos debido a una mala sincronización y comunicación entre tareas.

25.4 PARALELO VS CONCURRENTE

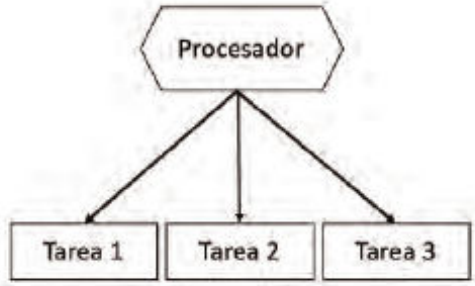
Paralelo no es lo mismo que concurrente. El objetivo de este apartado es que aprendas a diferenciar ambos términos.

Paralelo vas a entenderlo como un conjunto de actividades que tienen lugar al mismo tiempo, es decir, cada actividad es ejecutada por un procesador de forma independiente. Concurrente vas a entenderlo como la capacidad de operar un conjunto de actividades al mismo tiempo, todas ejecutadas desde el mismo procesador.

La siguiente imagen representa gráficamente el significado de paralelo. Cada procesador ejecuta una única tarea:



La siguiente imagen representa gráficamente el significado de concurrente, un mismo procesador ejecutar a la vez diferentes tareas.



Portanto, podemos concluir que paralelo está relacionado con tareas, procesadores y simultaneidad y concurrente con tareas, un único procesador y simultaneidad.

25.5 PROCESOS VS HILOS

En programación paralela existen dos términos muy importantes y que debes diferenciar de forma correcta, son *Proceso* e *Hilo*.

Un proceso es un programa en ejecución en un ordenador, es decir, cada programa que tienes abierto ejecutándose en tú ordenador es un proceso. Entendido lo que es un proceso, ahora introduciremos el término hilo, que no es otra cosa que un conjunto de instrucciones dentro de un proceso que se ejecutan de forma independiente. Un proceso puede estar compuesto por 1 o más hilos, si el número de hilos es mayor a 1 hablaremos del término multihilo.

Veamos en más detalle una comparativa entre hilo y proceso:

Parámetro	Proceso	Hilo
Independencia	Los procesos son independientes entre sí.	Los hilos no son independientes entre sí.
Creación	La creación de nuevos procesos es costosa, es decir, requiere más recursos de la máquina.	La creación de hilos es barata, es decir, no requiere de muchos recursos de la máquina.
Creación hilos	Pueden crear hilos.	Los hilos pueden crear hilos hijos.
Estados	Pueden encontrarse en los estados listo, bloqueado, en ejecución y terminado.	Pueden encontrarse en los estados listo, bloqueado, en ejecución y terminado.
Memoria	Los procesos no comparten memoria entre ellos.	Los hilos comparten memoria ya que están dentro de un mismo proceso.
Sincronización memoria	No es necesario sincronizar la memoria entre los diferentes procesos ya que es independiente.	Requiere sincronizar memoria ya que es compartida por todos los hilos.
Procesador	Comparten procesador y únicamente puede haber uno activo.	Comparten procesador y únicamente puede haber uno activo.
Tiempo de conmutación en el procesador	Alto.	Bajo.

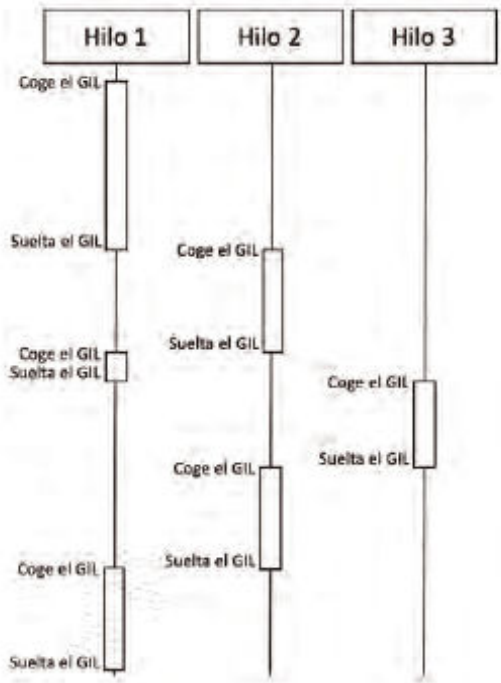
25.6 GLOBAL INTERPRETER LOCK

En este apartado vamos a hacer una pequeña introducción al **G**lobal **I**nterpreter **L**ock, conocido como **GIL**.

El GIL es un mecanismo utilizado en Python cuyo objetivo es que únicamente un hilo tenga el control del intérprete de Python, lo que implica que únicamente habrá un hilo activo en el intérprete en cada momento.

En programas que no utilizan hilos, el GIL es completamente invisible para los programadores y el programa, pero en aquellos programas desarrollados con hilos es un elemento que se ha ganado una reputación no muy buena dentro de la comunidad de Python, ya que a modo preventivo únicamente permite la ejecución de un único hilo dentro del intérprete.

En la siguiente imagen te mostramos un ejemplo con 3 hilos en ejecución que se van alternando a la hora de coger el GIL:



En la imagen anterior, el primero en coger el GIL es el hilo número 1, que se ejecuta hasta que lo suelta, acto seguido lo coge el hilo número 2 que se ejecuta hasta que lo suelta y vuelve a cogerlo el hilo número 1. El hilo número 3 coge el GIL después de soltarlo en número uno, y tras el hilo 3 lo coge el 2 y por último de nuevo el hilo número 1.

25.7 HILOS EN PYTHON

La utilización de hilos en Python es muy sencilla, lo primero que hay que hacer es importar la librería *threading* que contiene todo lo necesario para utilizar hilos. La sentencia para importar dicha librería es la siguiente:

```
import threading
```

La forma de crear un hilo en Python es a través de dicha librería e indicando la función que ejecutará dicho hilo. La creación sigue el siguiente formato:

```
Variable = threading.Thread(target=NombreFuncion)
```

Esta sentencia de código creará única y exclusivamente el hilo, no lo inicializará, para ello hay que invocar al método *start* de la clase de la siguiente manera:

```
Variable.start()
```

La sentencia iniciará la ejecución del hilo.

El último método que veremos de la clase *Thread* es el método *join*, que nos va a permitir indicarle al programa que se quede detenido en dicha sentencia hasta que el hilo acabe. Se utiliza de la siguiente manera:

```
Variable.join()
```

Por último, vamos a ver una funcionalidad muy útil en el mundo de la programación con hilos que nos vale para saber qué hilo se está ejecutando y cuál es su identificador. Se utilizan de la siguiente manera:

- **threading.current_thread().getName()**: retorna el nombre del hilo en ejecución.
- **threading.current_thread().ident**: retorna el identificador del hilo en ejecución.

A continuación vamos a realizar una serie de ejercicios para afianzar los conocimientos que acabamos de explicar referentes a la utilización de hilos en Python.

El primer ejercicio consiste en la ejecución de una función llamada *worker* desde el hilo principal del programa y desde dos hilos creados dentro del mismo. Además, se ha añadido una pequeña parada utilizando la clase *time* para que puedas observar con claridad los diferentes threads e identificadores. El código fuente es el siguiente:

```
import time
import threading

def worker():
    print('Hilo:',threading.current_thread().getName(),'con identificador:',
threading.current_thread().ident)
    time.sleep(1)

worker()
worker()

hilo1 = threading.Thread(target=worker)
hilo2 = threading.Thread(target=worker)
hilo1.start()
time.sleep(1)
hilo2.start()
hilo1.join()
hilo2.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo: MainThread con identificador: 4464670144
Hilo: MainThread con identificador: 4464670144
Hilo: Thread-1 con identificador: 123145496563712
Hilo: Thread-2 con identificador: 123145501818880
```

Tal y como puedes observar, al utilizar la función *worker* con el hilo principal del programa, los métodos de obtener el hilo en ejecución y su identificador devuelven los valores del hilo principal (MainThread). La utilización de la función con los hilos devuelve los valores Thread-1 para el primer hilo y Thread-2 para el segundo hilo.

El segundo ejercicio consiste en la eliminación de las pequeñas paradas que se habían introducido en el ejercicio anterior con la clase *time*. El objetivo es que aprecies que los threads se ejecutan a la vez ya que los mensajes se muestran por pantalla mezclados. El código fuente es el siguiente:

```
import threading

def worker():
    print('Hilo:',threading.current_thread().getName(),'con identificador:',
threading.current_thread().ident)

worker()
```

```
    worker()

    hilo1 = threading.Thread(target=worker)
    hilo2 = threading.Thread(target=worker)
    hilo1.start()
    hilo2.start()
    hilo1.join()
    hilo2.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo: MainThread con identificador: 4526335424
Hilo: MainThread con identificador: 4526335424
Hilo:Hilo:  Thread-1Thread-2  con identificador:con identificador:  123145414250496123145419505664
```

El siguiente ejercicio consiste en la creación de hilos dentro de un bucle, estableciendo mediante la sentencia de creación del hilo el nombre del mismo. La sentencia es la siguiente:

threading.Thread(name = 'NombreHilo',target=NombreFuncion)

En el ejercicio se han añadido pequeñas paradas utilizando la clase *time* para que puedas observar de forma sencilla los diferentes nombres de hilos e identificadores. El código fuente es el siguiente:

```
    import time
    import threading

    def worker():
        print('Hilo:',threading.current_thread().getName(),'con identificador:',
    threading.current_thread().ident)
        time.sleep(5)

    HILOS = 5

    for num_hilo in range(HILOS):
        hilo = threading.Thread(name = 'hilo %s' %num_hilo,target=worker)
        hilo.start()
        time.sleep(1)
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo: hilo 0 con identificador: 123145389924352
Hilo: hilo 1 con identificador: 123145395179520
Hilo: hilo 2 con identificador: 123145400434688
Hilo: hilo 3 con identificador: 123145405689856
Hilo: hilo 4 con identificador: 123145410945024
```

El cuarto ejercicio consiste en la eliminación de las paradas introducidas mediante la clase time en el ejercicio anterior. El código fuente quedaría de la siguiente forma:

```
import time
import threading

def worker():
    print('Hilo:',threading.current_thread().getName(),'con identificador:',
threading.current_thread().ident)

HILOS = 5

for num_hilo in range(HILOS):
    hilo = threading.Thread(name = 'hilo %s' %num_hilo,target=worker)
    hilo.start()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo:Hilo:Hilo:Hilo:Hilo:     hilo 0hilo 1hilo 2hilo 3hilo 4     con identificador:co
n identificador:con identificador:con identificador:con identificador:     1231453566
7302412314536192819212314536718336012314537243852812314537769369 6
```

Tal y como puedes observar en la ejecución, los hilos se ejecutan a la vez, ya que escriben por pantalla toda la información mezclada.

El quinto ejercicio se basa en el anterior y consiste en utilizar el método *join* de la clase *Thread* para no crear un nuevo hilo hasta que se haya terminado la ejecución del anterior. El código fuente quedaría de la siguiente forma:

```
import time
import threading

def worker():
    print('Hilo:',threading.current_thread().getName(),'con identificador:',
threading.current_thread().ident)
```

```
HILOS = 5

for num_hilo in range(HILOS):
    hilo = threading.Thread(name = 'hilo %s' %num_hilo,target=worker)
    hilo.start()
    hilo.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo: hilo 0 con identificador: 123145432559616
Hilo: hilo 1 con identificador: 123145432559616
Hilo: hilo 2 con identificador: 123145432559616
Hilo: hilo 3 con identificador: 123145432559616
Hilo: hilo 4 con identificador: 123145432559616
```

Tal y como puedes observar el nombre de cada hilo es diferente, pero el identificador es el mismo ya que los identificadores son reutilizables dentro del mismo programa.

El sexto ejercicio consiste en aprender a pasar parámetros a la función que ejecuta el hilo. La función *worker* ha sido ampliada con dos parámetros de entrada, el primero de ellos es un entero que representa el número de hilo y el segundo es una secuencia variable de parámetros. La forma de indicar al hilo los parámetros es en la sentencia de creación del hilo y sigue el siguiente formato:

threading.Thread(target=NombreFuncion,args= (Parametros),kwargs={Lista ParámetrosVariables})

La funcionalidad del ejercicio consiste en crear diferentes hilos y dichos hilos mostrarán por pantalla la secuencia indicada por parámetros a la función. El código fuente es el siguiente:

```
import threading

def worker(num_hilo, **secuencia):
    print('Hilo:',threading.current_thread().getName(),'con identificador:',
threading.current_thread().ident)
    for valor in range(secuencia['comienzo'],secuencia['fin'],1):
        print(valor);

HILOS = 5

for num_hilo in range(HILOS):
```

```
        comienzo = num_hilo*10
        fin = 10 + num_hilo*10
        hilo = threading.Thread(target=worker,args=(num_hilo,),kwargs={'comienzo':co
    mienzo, 'fin':fin})
        hilo.start()
        hilo.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo: Thread-1 con identificador: 123145424080896
0
1
2
3
4
5
6
7
8
9
Hilo: Thread-2 con identificador: 123145424080896
10
11
12
13
14
15
16
17
18
19
Hilo: Thread-3 con identificador: 123145424080896
20
21
22
23
```

Tal y como puedes observar, al añadir el método *join* al código fuente, la creación y ejecución de los hilos se realiza de forma secuencial y todos tienen el mismo identificador.

En el último ejercicio referente a hilos vamos a realizar lo mimos que en el anterior eliminando el *join* que se realiza para que no se creen y se ejecuten de forma secuencial. El código fuente es el siguiente:

```
    import threading

    def worker(num_hilo, **secuencia):
        print('Hilo:',threading.current_thread().getName(),'con identificador:',
```

```
    threading.current_thread().ident)
        for valor in range(secuencia['comienzo'],secuencia['fin'],1):
            print(valor);

    HILOS = 5

    for num_hilo in range(HILOS):
        comienzo = num_hilo*10
        fin = 10 + num_hilo*10
        hilo = threading.Thread(target=worker,args=(num_hilo,),kwargs={'comienzo':co
    mienzo, 'fin':fin})
        hilo.start()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
Hilo:Hilo:Hilo:Hilo:Hilo:     Thread-1Thread-2Thread-3Thread-4Thread-5     con identifi
cador:con identificador:con identificador:con identificador:con identificador:     1231
455573893121231455626444801231455678996481231455731548161231455784099 84



010203040



111213141



212223242



313233343



414243444
```

25.8 PROCESOS EN PYTHON

La utilización de procesos en Python es muy sencilla, únicamente hay que añadir la librería *multiprocessing* y tendremos todas las funcionalidades relacionadas con los procesos. La sentencia para añadir la librería es la siguiente:

import multiprocessing

La forma de crear un proceso en Python es a través de dicha librería e indicando la función que ejecutará el proceso. La creación sigue el siguiente formato:

Variable = multiprocessing.Process(target=NombreFuncion())

Esta sentencia de código creará única y exclusivamente el proceso, no lo inicializará, para ello hay que invocar al método *start* de la clase de la siguiente manera:

Variable.start()

La sentencia iniciará la ejecución del proceso.

El último método que veremos es el método *join*, que nos va a permitir indicarle al programa que se quede detenido en dicha sentencia hasta que el proceso acabe. Se utiliza de la siguiente manera:

Variable.join()

Al igual que vimos en el apartado anterior referente a hilos, en lo referente a procesos también es posible obtener los identificadores y nombres de todos los procesos. Veamos cómo obtener cada uno de ellos:

- **Identificador del proceso**: el identificador se obtiene utilizando la librería *os* de Python, concretamente con el método *getpid*. La sentencia sería la siguiente:

  os.getpid()

- **Nombre del proceso**: el nombre del proceso se obtiene mediante la librería *multiprocessing* a través de la siguiente sentencia:

  multiprocessing.current_process().name

A continuación vamos a realizar una serie de ejercicios para afianzar los conocimientos sobre procesos. En todo el libro estamos utilizando IDLE para ejecutar los programas que estamos escribiendo, te recomendamos ejecutarlos desde el terminal ya que la gestión de los procesos desde IDLE no funciona tan bien como funciona desde el propio terminal.

El primer ejercicio consiste en la ejecución de una función llamada *worker* desde el propio programa y después desde dos procesos nuevos que se crean. La salida por pantalla mostrará el identificador de cada proceso (PID) junto con su nombre y el hilo que se está ejecutando.

```
import os
import time
import threading
import multiprocessing

def worker():
    print("PID: %s, Nombre proceso: %s, Nombre hilo: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name))

worker()

proceso1 = multiprocessing.Process(target=worker)
proceso2 = multiprocessing.Process(target=worker)
proceso1.start()
proceso2.start()
proceso1.join()
proceso2.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
PID: 10766, Nombre proceso: MainProcess, Nombre hilo: MainThread
PID: 10767, Nombre proceso: Process-1, Nombre hilo: MainThread
PID: 10768, Nombre proceso: Process-2, Nombre hilo: MainThread
```

En la salida puedes observar lo siguiente:

- La primera línea se corresponde con el proceso del programa principal, de ahí que en el nombre del proceso se muestre *MainProcess*.
- La segunda línea se corresponde con el primer proceso que se crea, de ahí que el nombre sea *Process-1*. Puedes comprobar también que el identificador es diferente respecto al principal, ya que son procesos independientes.
- La tercera línea se corresponde con el segundo proceso, de ahí que el nombre sea Process-2. Puedes comprobar también que el identificador es diferente al del programa principal y al del proceso número 1.

El segundo ejercicio consiste en añadir hilos al ejercicio primero. El objetivo es mostrar por pantalla la información de todos los procesos e hilos que se ejecutan. El código fuente es el siguiente:

```
import os
import time
import threading
import multiprocessing


def workerhilos():
    print("PID: %s, Nombre proceso: %s, Nombre hilo: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name))
    time.sleep(1)

def workerprocesos():
    workerhilos()
    hilo1 = threading.Thread(target=workerhilos)
    hilo2 = threading.Thread(target=workerhilos)
    hilo1.start()
    time.sleep(1)
    hilo2.start()
    hilo1.join()
    hilo2.join()

workerprocesos()

proceso1 = multiprocessing.Process(target=workerprocesos)
proceso2 = multiprocessing.Process(target=workerprocesos)
proceso1.start()
proceso2.start()
proceso1.join()
proceso2.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
PID: 10888, Nombre proceso: MainProcess, Nombre hilo: MainThread
PID: 10888, Nombre proceso: MainProcess, Nombre hilo: Thread-1
PID: 10888, Nombre proceso: MainProcess, Nombre hilo: Thread-2
PID: 10889, Nombre proceso: Process-1, Nombre hilo: MainThread
PID: 10890, Nombre proceso: Process-2, Nombre hilo: MainThread
PID: 10889, Nombre proceso: Process-1, Nombre hilo: Thread-3
PID: 10890, Nombre proceso: Process-2, Nombre hilo: Thread-3
PID: 10890, Nombre proceso: Process-2, Nombre hilo: Thread-4
PID: 10889, Nombre proceso: Process-1, Nombre hilo: Thread-4
```

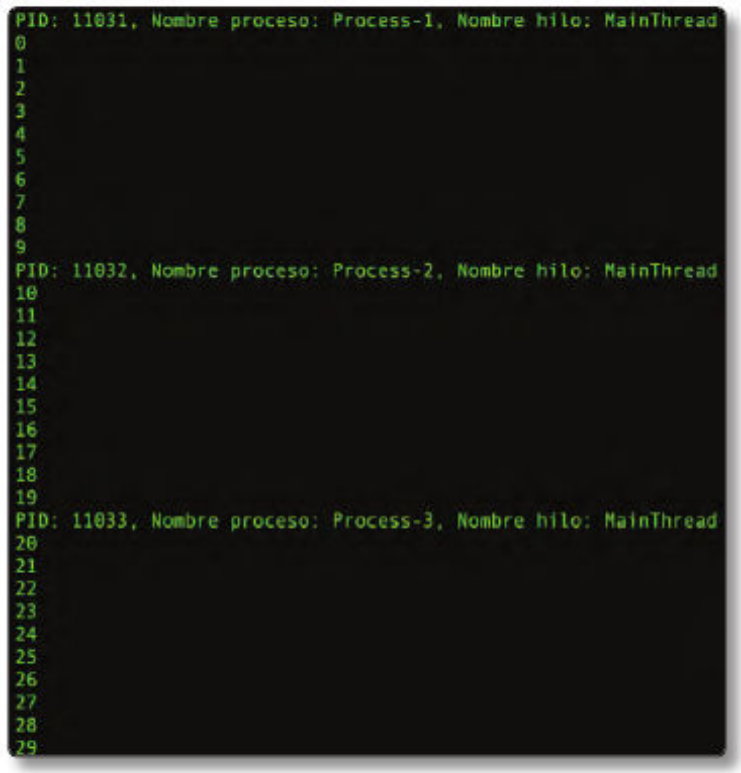
En la imagen puedes observar como los hilos que se han creado dentro del mismo proceso comparten identificador y nombre del proceso, PID igual a 10889 y nombre Process-1 para los hilos creados por el proceso uno y PID igual a 10890 y nombre Process-2 para los hilos creados por el proceso dos. Puedes comprobar también que cuando se crea un proceso los identificadores que utilizan para los hilos son los consecutivos a partir del último hilo creado por el proceso que los crea, es por ello que los hilos de los procesos uno y dos tienen los mismos nombres (Thread-3 y Thread-4).

El último ejercicio consiste en realizar uno de los ejercicios que hemos realizado para los hilos pero esta vez con procesos, de forma que aprendas cómo pasar parámetros a las funciones que utilizan los procesos. La forma de pasar parámetros es exactamente igual que a los hilos, mediante *args* y *kwargs*. En el ejercicio vamos a pasar dos parámetros fijos a la función que indicarán el comienzo y el fin del conteo que realiza. El código fuente es el siguiente:

```
import os
import time
import threading
import multiprocessing

def worker(comienzo, fin):
    print("PID: %s, Nombre proceso: %s, Nombre hilo: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name))
    for valor in range(comienzo,fin,1):
        print(valor);


NUM_PROCESOS = 3

for num_proceso in range(NUM_PROCESOS):
    comienzo = num_proceso*10
    fin = 10 + num_proceso*10
```

```
    proceso = multiprocessing.Process(target=worker,args=(comienzo, fin,))
    proceso.start()
    proceso.join()
```

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:



Tal y como puedes observar, cada proceso tiene un identificador y nombre diferente y cada uno realiza únicamente el conteo que se le ha pasado como parámetro a la función.

26

PYTHON Y LAS BASES DE DATOS

En este capítulo se va a explicar la utilización de bases de datos con Python, junto a los conceptos básicos de bases de datos y SQL.

26.1 INTRODUCCIÓN A LAS BASES DE DATOS

En este apartado se van a explicar los conceptos básicos para entender qué es una base de datos, por qué es beneficioso utilizarlas, tipos de bases de datos existentes, el modelo relacional y una pequeña introducción al lenguaje SQL.

26.1.1 ¿Qué es una base de datos?

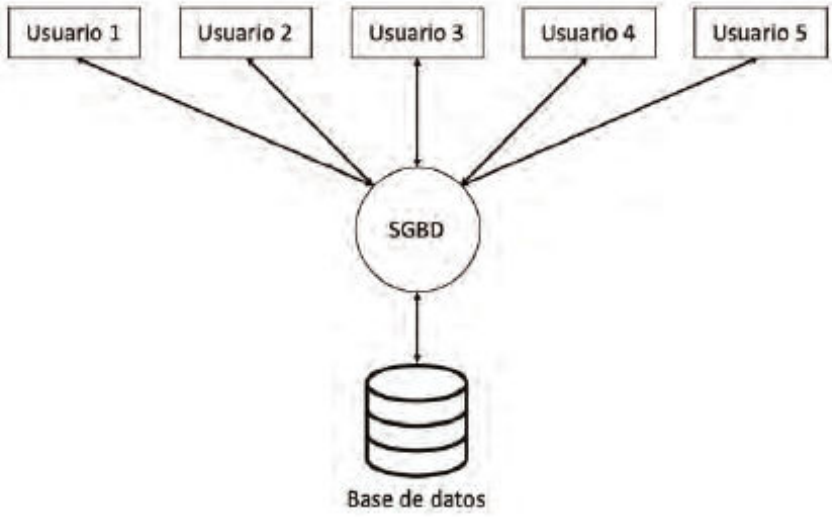
Las bases de datos son uno de los elementos informáticos más importantes que existen en la actualidad y juegan un papel fundamental en la sociedad moderna e informatizada. En nuestro día a día, interactuamos con un sin fin de bases de datos, desde actividades tan cotidianas como hacer la compra o ir al médico a realizar operaciones bancarias con el móvil. Podemos decir que casi todas las actividades que hacemos en nuestras vidas están ligadas al uso de una base de datos, aunque normalmente no somos conscientes de ello.

Una **base de datos** es un conjunto de datos de cualquier tipo que pertenecen a un mismo contexto y que son almacenados para ser usados posteriormente. Una base de datos debe permitir almacenar grandes cantidades de información y su uso por aplicaciones informáticas.

Algunos ejemplos de uso de bases de datos pueden ser:

- Universidades: almacenar información de los alumnos, matrículas, asignaturas y notas.
- Salud: almacenar información de los pacientes, historial médico, información de los médicos, información de los medicamentos y citas.
- Banca: almacenar información de los clientes, información de las cuentas bancarias, transacciones bancarias, información de las hipotecas y préstamos.
- Tienda: almacenar información de los productos, información del stock y precios.
- Compañía telefónica: almacenar información de los clientes, información de las líneas, registros de llamadas y mensajes.
- Departamento de Recursos Humanos: almacenar información de los empleados, salarios y nóminas.

Las bases de datos son sistemas complejos que son manejados y gestionados por un software específico llamado **S**istema **G**estor de **B**ases de **D**atos (**SGBD**). Los SGBD permiten a los usuarios de las bases de datos abstraerse del almacenamiento físico de los datos, consistencia en los datos almacenados, seguridad de la información, simplicidad de operaciones, etc. En otras palabras, los SGBD facilitan la vida a los usuarios de las bases de datos, brindando una capa intermedia entre la base de datos y la aplicación. En la siguiente imagen puedes ver un diagrama de lo explicado:



Los SGBD son sistemas complejos que deben proporcionar una serie de funcionalidades para la utilización de la base de datos. Los componentes más importantes por los que están compuestos son los siguientes:

- **Gestor de archivos**: encargado de la gestión de los archivos que componen la base de datos.
- **Gestor de la base de datos**: componente que se encarga de comunicar las aplicaciones y la base de datos. El componente evalúa la petición y realiza la consulta al gestor de archivos para realizarla.
- **Controlador de autorización**: controla los permisos de acceso a la base de datos y a la información, es decir, es el encargado de asegurar que quien quiere acceder a la base de datos tiene permisos para acceder y que únicamente accede a la información sobre la que tiene permiso para hacerlo.
- **Procesador de consultas**: encargado de realizar las órdenes a la base de datos.
- **Controlador de integridad**: controla que las órdenes ejecutadas por el procesador de consultas no dejen a la base de datos en un estado corrupto de la información.
- **Optimizador de consultas**: encargado de determinar la estrategia óptima para la ejecución de las consultas.
- **Gestor de transacciones**: encargado de gestionar las transacciones con la base de datos. Una transacción es un conjunto de consultas que se realizan de forma atómica sobre la base de datos, es decir, todas juntas en una misma operación.
- **Planificador**: encargado de que las operaciones contra la base de datos se realicen de forma concurrente y sin conflictos.
- **Gestor de recuperación**: encargado de recuperar la base de datos en caso de que haya algún fallo y se quede inconsistente.
- **Gestor del buffer**: encargado de transferir la información de memoria a almacenamiento físico y viceversa.

26.1.2 Beneficios de uso

Los beneficios más relevantes de utilizar bases de datos para el almacenamiento de la información son los siguientes:

- **Mayor independencia**: los datos son una entidad independiente de los aplicativos y de los usuarios que los utilizan.
- **Mayor uso**: almacenando los datos en una base de datos se garantiza que puedan ser utilizados por más de un aplicativo y más de un usuario.
- **Mayor seguridad**: teniendo los datos en una base de datos se pueden utilizar mecanismos de seguridad propios de las bases de datos, tal y como puede ser la replicación de la base de datos o control de acceso a la base de datos.
- **Mayor sincronización**: al estar almacenados los datos en un lugar centralizado, todas las aplicaciones tendrán y usarán siempre los datos actualizados, algo que no se puede garantizar si los datos están almacenados en diferentes orígenes.
- **Menor redundancia**: ligado al punto anterior, al estar en un único sitio, los datos no se encuentran en diferentes orígenes que se tienen que mantener actualizados.
- **Menor memoria requerida**: el espacio necesario para almacenar la información será menor, ya que únicamente se tienen almacenados los datos una única vez.
- **Mayor coherencia**: la calidad de los datos aumenta al estar almacenados en un único punto.
- **Mayor eficiencia**: al almacenar los datos en una base de datos se está garantizando que los datos se encuentran en un único punto, por tanto, a la hora de operar con ellos se tiene que ir a un único sitio a por ellos, en lugar de tener que ir a diferentes puntos o tener que estar sincronizando datos entre diferentes fuentes de datos.
- **Mayor focalización en el uso de datos**: el usuario únicamente tiene que preocuparse de utilizar los datos, ya que del resto de operaciones se encarga el SGBD.

26.1.3 Tipos de bases de datos

El concepto de base de datos está claro, un conjunto de datos relacionados con un contexto concreto y que son almacenados para su uso posterior, pero, existen

diferentes tipos de bases de datos que organizan y manejan la información de forma diferente. Veamos cuáles son los tipos más comunes:

- **Bases de datos jerárquicas**: almacenan la información en una estructura jerárquica, es decir, almacenan la información en forma de árbol en el que un nodo padre (raíz) tiene uno o más nodos hijos (hojas). Son útiles en aplicaciones que presentan volúmenes de datos grandes y compartidos, aunque no tienen problemas con la redundancia de datos.

- **Bases de datos en red**: es una evolución de las bases de datos jerárquicas, pero permitiendo que un nodo pueda tener 1 o más padres. De esta forma, se eliminaba el problema de la redundancia pero generaba un problema de manejabilidad y uso por usuario, siendo únicamente utilizada por programadores.

- **Bases de datos relacionales**: es el modelo más utilizado en la actualidad. La idea fundamental de este tipo de bases de datos es la utilización del concepto de relaciones para definir dependencias entre los datos almacenados. Lo más importante de estos tipos de bases de datos son las relaciones, no el cómo se almacena y en dónde.

- **Bases de datos orientadas a objeto**: es un tipo de base de datos de reciente creación y que surge a partir de la aparición de la programación orientada a objetos con el objetivo de almacenar objetos directamente en la base de datos. La base de datos incorpora de serie todas las características de la programación orientada a objetos que vimos en el apartado referente a dicho paradigma de programación.

- **Bases de datos documentales**: son bases de datos especializadas en el almacenamiento de textos completos, incorporando sistemas muy potentes de tratamiento de cadenas de caracteres.

- **Bases de datos transaccionales**: es un tipo de base de datos optimizada para el envío y recepción continuo de grandes cantidades de datos. La gestión del almacenamiento y redundancia caen fuera de las funcionalidades de éste tipo de base de datos.

- **Bases de datos deductivas/lógicas**: son bases de datos capaces de deducir información mediante el uso de inferencias contra los datos. Su funcionamiento se basa en la creación de reglas y hechos.

- **Bases de datos distribuidas**: es una base de datos que parece ser única pero que está compuesta por diferentes bases de datos que se encuentra distribuidas en diferentes orígenes de red.

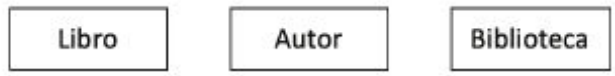
No existe el tipo de base de datos perfecto, dependiendo del uso que se le quiera dar será conveniente utilizar una u otra.

En este libro vamos a utilizar las bases de datos relacionales, ya que son las bases de datos más utilizadas y más comunes en todos los proyectos.

26.1.4 Modelo entidad-relación

El modelo entidad-relación es el modelo de representación de la estructura de las bases de datos relacionales. El modelo representa la base de datos con una serie de diagramas y está compuesto por diversos elementos. Veamos los elementos del mismo:

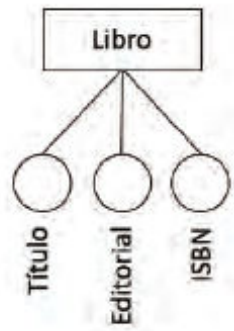
- **Entidad**: las entidades representan cosas u objetos independientemente de que sean reales o abstractos y que están claramente diferenciados entre sí. En el ejemplo siguiente puedes ver cómo se representas las entidades en el diagrama:



  El diagrama estará compuesto por tres entidades: Libro, Autor y Biblioteca.

  Las entidades se corresponden con **tablas** en la base de datos, es decir, en base de datos se crearán, normalmente, tantas tablas como entidades se definan. En las tablas se almacena la información de las entidades.

- **Atributos**: los atributos son las características/propiedades de cada entidad que proporcionan información sobre la misma. Un ejemplo de atributos de la entidad Libro que hemos visto antes podría ser:



La entidad libro estaría compuesta por los atributos Título, Editorial e ISBN, aunque podría tener muchos más.

Los atributos se corresponden con **campos o columnas** de las tablas en base de datos, es decir, cada tabla estará compuesta por tantas columnas como atributos tenga una entidad. Cada elemento de la tabla se llama **registro**, y está compuesto por la información de cada una de las columnas que lo componen.

- **Claves**: es un atributo de la entidad que se utiliza para identificar cada registro de la misma de forma única.

- **Relación**: representan vínculos entre entidades. En el ejemplo que estamos viendo, un libro está escrito por un autor. La forma de representarlo sería así:

Las relaciones son los conceptos más difíciles de crear en una base de datos, ya que tienen cardinalidad, como veremos más adelante. Existen diferentes formas de implementarlas en una base de datos, en algunos casos se utilizará una tabla para relacionar dos entidades mediante sus claves, en otros se utilizará un campo extra en una de las entidades para referenciar a la otra entidad, etc.

Las relaciones pueden tener cardinalidad, en el ejemplo que estamos viendo un libro puede estar escrito por 1 o más autores, por lo que es necesario indicarlo en el esquema. Veamos qué tipos de cardinalidades hay:

- Uno a Uno: los registros de las entidades únicamente se relacionan con un registro de la misma. Por ejemplo, si considerásemos el ISBN como una entidad en lugar de como un atributo la relación sería uno a uno y sería representada de la siguiente manera:

- Uno a Varias o Varios a Uno: un registro de la entidad puede estar relacionado varias veces con otro registro de la otra entidad pero cada

registro de la misma únicamente existir una sola vez. En el ejemplo que estamos viendo, la relación entre libro y autor sería un ejemplo de este tipo de cardinalidad ya que un libro puede estar escrito por uno o varios autores y se representaría así:

Existe la posibilidad de que en lugar de que la relación sea uno a varios sea cero a varios, ésto indica que la entidad puede o no existir.

- Varios a Varios: indica que un registro de una entidad puede relacionarse con ninguno o varios registros de la otra entidad y viceversa. En el ejemplo este tipo de relación podría ser la de libros y biblioteca, un libro puede estar en ninguna, una o varias bibliotecas y una biblioteca puede tener ningún ejemplar del libro, uno o varios, La forma de representarlo sería la siguiente:

26.1.5 SQL

SQL (**S**tructured **Q**uery **L**anguage) es el lenguaje que se utiliza para definir, gestionar y manipular la información de las bases de datos relacionales. Una de las características principales es que es un lenguaje sencillo y potente, lo que facilita que cualquier usuario de la base de datos pueda utilizarlo, bien sea un programador, un usuario o un administrador.

La forma de utilizar el lenguaje es muy simple, una vez construida la consulta, el usuario la tiene que ejecutar contra la base de datos, y es el SGBD el que decide cómo ejecutarla para obtener los resultados solicitados.

SQL es un lenguaje estándar por ANSI (Instituto Nacional Americano de Estándares) desde 1986 e ISO (Organización internacional de Normalización) desde 1987.

26.1.5.1 TIPOS DE DATOS

Las bases de datos pueden tener diferentes tipos de datos almacenados. Al ser tanta la variedad vamos a centrarnos en probablemente los dos más importantes y que más se usan en todas las bases de datos:

- **Integer**: es un tipo de dato compuesto por los números naturales tanto positivos como negativos junto al 0. No poseen parte decimal.
- **Text**: es un tipo de dato compuesto por una secuencia de caracteres ordenados. Las principales características es que la longitud es variable pero finita y que puede estar compuesta por cualquier tipo de carácter.

En los ejercicios que realizaremos utilizaremos datos de éstos tipos.

26.1.5.2 SENTENCIAS

Las operaciones que se pueden utilizar contra una base de datos relacional mediante SQL se llaman sentencias, y no son otra cosa que una orden escrita en una forma concreta y que es interpretada y ejecutada por el SGBD.

Las sentencias se encuentran divididas en tres tipos diferentes:

- **Sentencias para la definición de datos**: son sentencias para manipular la estructura de la base de datos. Tales como crear una base de datos, modificar una base de datos, crear una tabla en base de datos, modificar una tabla de bases de datos, borrar una tabla en base de datos, etc.
- **Sentencias para la manipulación de datos**: son las sentencias que nos permiten interactuar con los datos que tiene la base de datos. Básicamente, son las operaciones **CRUD** (**C**reate, **R**ead, **U**pdate, **D**elete). En Castellano son: Crear, Leer, Modificar y Borrar.
- **Sentencias de control de datos**: son las sentencias que permitirán ejecutar órdenes relacionadas con la administración de la base de datos.

En los ejercicios que haremos en el capítulo se van a trabajar las siguientes sentencias:

- Crear base de datos.
- Crear tabla.
- Todas las sentencias para la manipulación de datos: Crear, Leer, Modificar y Borrar.

La parte teórica asociada a dichas sentencias será explicada en los propios ejercicios.

26.2 SQLITE

SQLite es una biblioteca escrita en el lenguaje de programación C y que implementa un SGBD completo. El código fuente de SQLite es de dominio público y puede ser utilizado en cualquier tipo de proyecto, además, se trata de una librería multiplataforma, por lo que puede ser utilizada en cualquier tipo de sistema operativo (Microsoft Windows, Mac, Linux...). La documentación que posee el proyecto es muy buena y el código fuente está estructurado de forma sencilla para permitir su compresión por el mayor número de programadores posible.

Una de las características más importantes que tiene SQLite es que no necesita un proceso independiente funcionando como servidor, ésto es posible gracias a que lee y escribe directamente en los ficheros almacenados en el disco duro.

La razón de elegir SQLite para enseñarte a trabajar con bases de datos y Python es porque es una base de datos sencilla, que no requiere instalación ni configuración, es gratuita, multiplataforma, multilenguaje y además implementa el estándar SQL.

26.3 EJERCICIOS

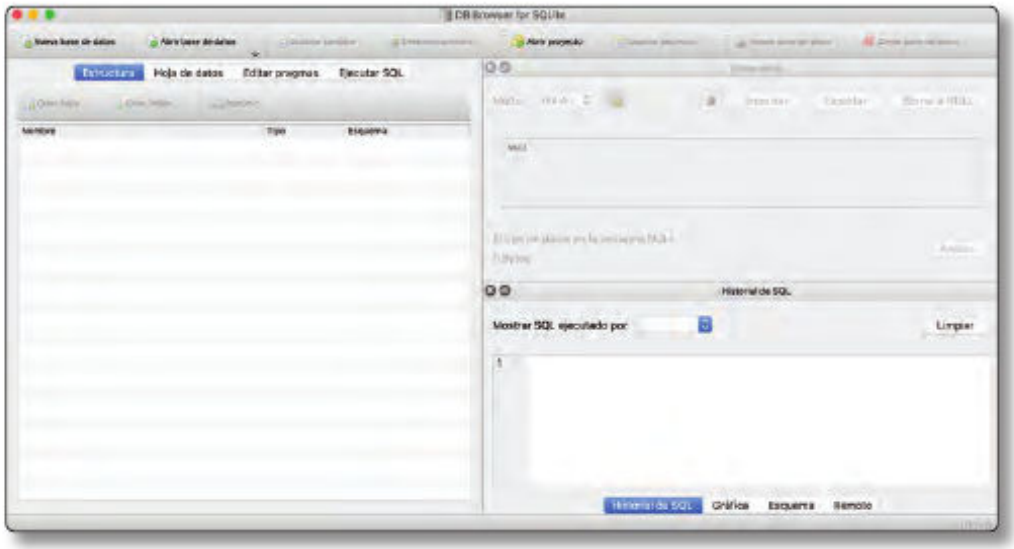
En este apartado vamos a realizar los ejercicios correspondientes al capítulo. Empezaremos con un primer ejercicio en el que crearemos la base de datos que utilizaremos en el resto de ejercicios, en los que haremos inserciones, modificaciones, lectura y borrados de registros de la base de datos.

En los ejercicios vamos a utilizar una aproximación a coches y fabricante, con el siguiente modelo:

26.3.1 Creación de la base de datos

En este ejercicio vamos a crear la base de datos utilizando el programa SQLite Browser, puedes descargarlo de: *https://sqlitebrowser.org*. La descarga tienes que hacerla desde la pestaña de *Download*, una vez dentro elige la versión a descargar dependiendo del sistema operativo que tengas instalado en tu ordenador.

Una vez instalado el programa, ábrelo y verás la siguiente pantalla:



Para crear la base de datos tienes que presionar con el ratón en la opción "Nueva base de datos" que se encuentra en la esquina superior izquierda. Después de hacerlo, tienes que elegir la ruta donde quieres guardar la base de datos y elegir el nombre para la misma, te recomendamos hacerlo en la misma carpeta en la que vas a guardar los ejercicios y ponerle de nombre "*Coches.db*".

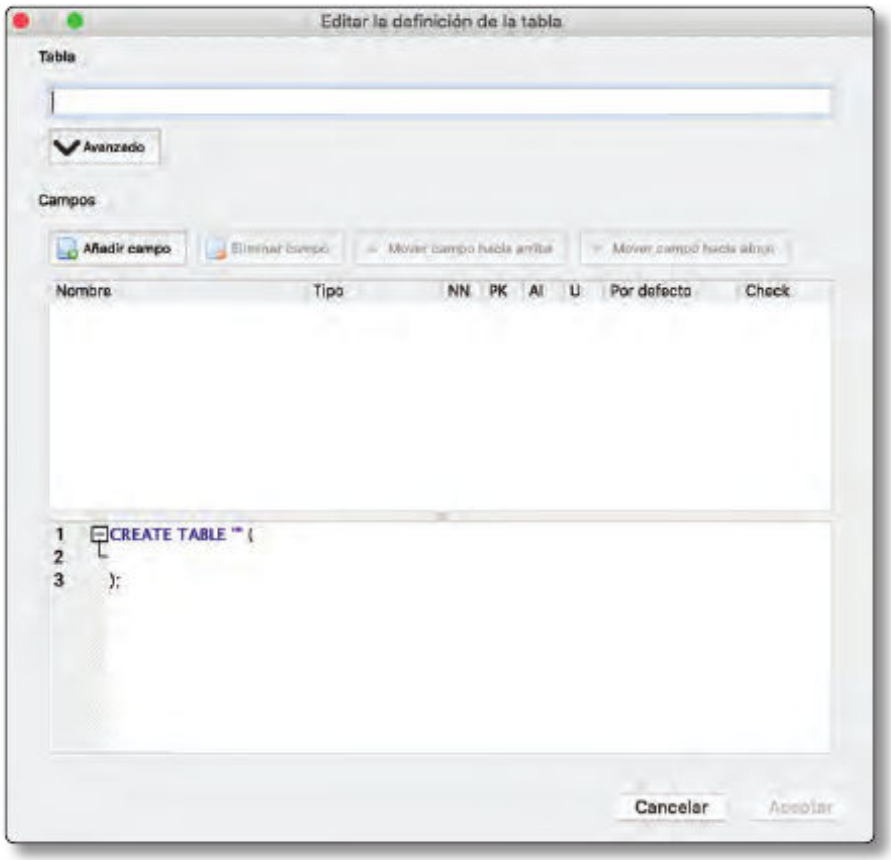
La siguiente pantalla que se abre después de elegir ruta y nombre es la pantalla para crear una tabla en base de datos. En la pantalla verás 3 paneles diferentes:

- Tabla: en este panel se establece el nombre de la tabla.
- Campos: en este panel se establecen los campos que tendrán la base de datos. Para cada campo hay que establecer la siguiente información:
  - Nombre: nombre del campo de la tabla.
  - Tipo: tipo de datos del campo.
  - NN: indica si es campo puede ser nulo o no, es decir, si es obligatorio de rellenar.
  - PK: indica si el campo es clave primaria de la tabla.
  - AI: indica si el campo es un auto incremental (válido para enteros o claves).
  - U: indica si el campo es único, es decir, no pueden existir valores repetidos para ese campo en toda la tabla.

- Por defecto: indica un valor por defecto para el campo.
- Check: se usa para verificar si el valor de la columna está en el dominio especificado. Esto impide que se asignen valores no válidos a la columna.

◤ SQL: en este panel se puede ver el código SQL de la tabla, se va construyendo a medida que se rellena la información de los paneles anteriores. El código fuente no se puede modificar si no es mediante los paneles superiores.

La pantalla es la siguiente:

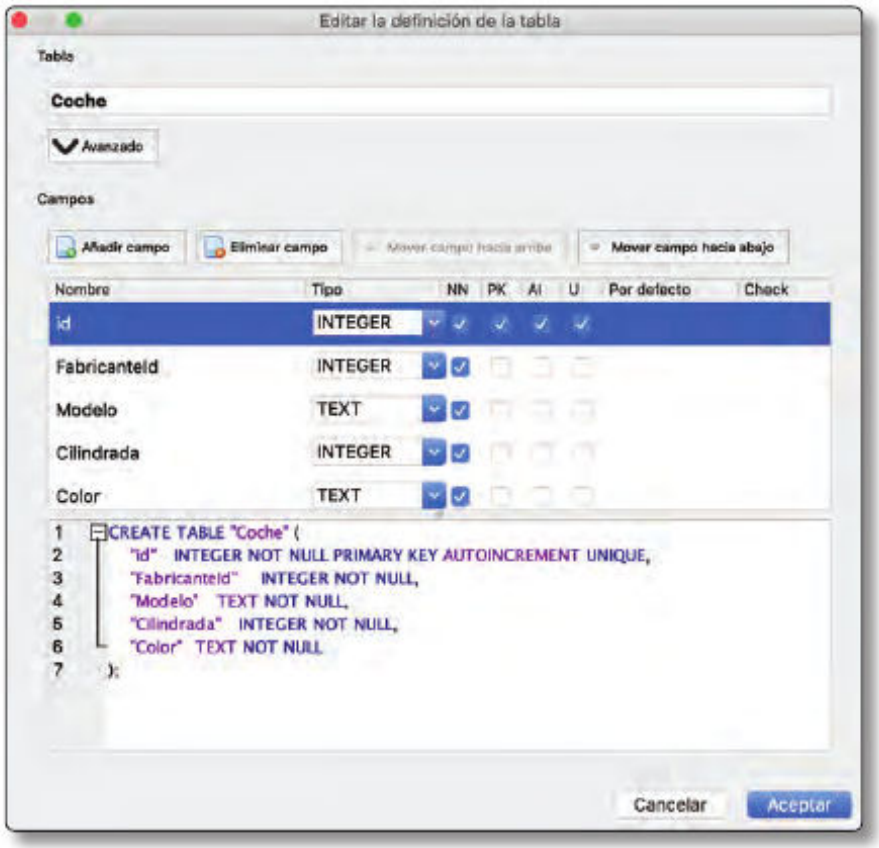


La primera tabla que vamos a crear es la tabla ***Coches***, que almacenará la información de los coches. Estará compuesta por los siguientes campos:

◤ id: campo utilizado para identificar al coche de forma única. El campo será la clave primaria de la tabla, por lo tanto será no nula y única. El campo será auto incremental.

- FabricanteId: identificador del fabricante. Se utilizará para relacionar el coche con su fabricante, para los que crearemos una tabla posteriormente. El campo es no nulo.
- Modelo: indica el modelo del coche. El campo es no nulo.
- Cilindrada: indica la cilindrada del coche. El campo es no nulo.
- Color: indica el color del coche. El campo es no nulo.

En la siguiente imagen puedes ver la definición de la tabla:

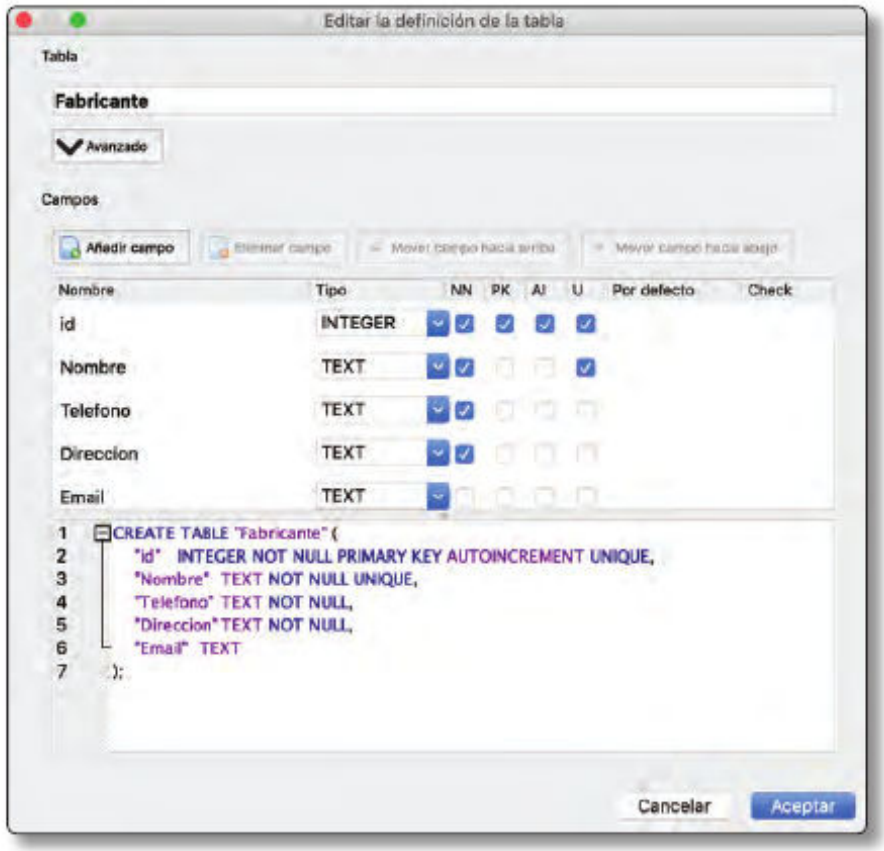


La siguiente tabla a crear es ***Fabricante***, que almacenará la información de todos los fabricantes de coches. Estará compuesta por los siguientes campos:

- id: campo utilizado para identificar al fabricante de forma única. El campo será la clave primaria de la tabla, por lo tanto será no nula y única. El campo será auto incremental. El identificador es utilizado en la tabla *Coches* para relacionar las dos entidades.

- Nombre: indica el nombre del fabricante. El nombre es un campo único, ya que no pueden existir más de un fabricante con el mismo nombre. El campo es no nulo.
- Telefono: indica el teléfono de contacto con el fabricante. El campo es no nulo.
- Direccion: indica la dirección en la que se encuentra el fabricante. El campo es no nulo.
- Email: indica el email de contacto del fabricante. El campo admite nulos, ya que no todos los fabricantes disponen de email.

En la siguiente imagen puedes ver la definición de la tabla:



Una vez creadas las dos tablas, tienes la base de datos lista para continuar con los siguientes ejercicios. La base de datos también se encuentra disponible en el material descargable del libro dentro de los ejercicios de base de datos.

26.3.2 Insertando datos

El primer ejercicio con código fuente que vamos a realizar es la inserción de datos en la base de datos. El comando que se utiliza para hacer inserciones en bases de datos es el siguiente:

INSERT INTO *NombreTabla* VALUES(*ListaDeValores*)

A destacar del comando:

- NombreTabla: indica el nombre de la tabla en la que se desea insertar información.
- ListaDeValores: son los valores del registro a insertar, se ponen separados por coma.

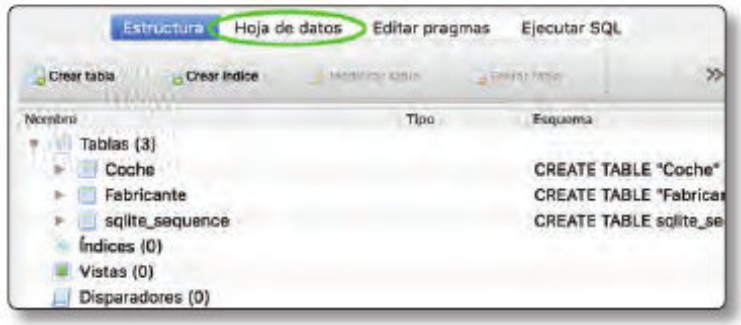
El código fuente sigue los siguientes pasos:

1. Apertura de la conexión a la base de datos.
2. Generación de los registros de fabricantes a insertar en base de datos.
3. Creación de un cursor para realizar operaciones contra la base de datos e inserción de los registros.
4. Consolidar la información en la base de datos mediante el comando *commit*. En caso de no ejecutar el comando la información no se guardará en base de datos ya que no estará consolidada.
5. Generación de los registros de los coches a insertar en base de datos.
6. Inserción de los registros.
7. Consolidar la información en la base de datos.
8. Cerrar la conexión a la base de datos.

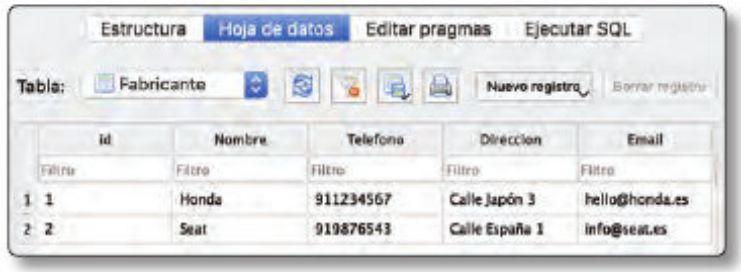
El código fuente del ejercicio es el siguiente:

```
# Importa la librería para utilizar SQLite
import sqlite3

# Conecta a la base de datos
database = sqlite3.connect('coches.db')

# Crea los registros de fabricantes
register1 = (1, 'Honda', '911234567', 'Calle Japón 3','hello@honda.es')
register2 = (2, 'Seat', '919876543', 'Calle España 1','info@seat.es')

# Apertuda de un cursor e inserción de los fabricantes
cursor = database.cursor()
cursor.execute("INSERT INTO Fabricante VALUES(?,?,?,?,?)", register1)
cursor.execute("INSERT INTO Fabricante VALUES(?,?,?,?,?)", register2)

# Commit de las operaciones
database.commit()

# Crea los registros de los coches
register1 = (1, 1, 'Civic', 1600,'Azul')
register2 = (2, 1, 'HR-V', 2400,'Negro')
register3 = (3, 2, 'Ibiza', 1200,'Rojo')
register4 = (4, 2, 'Ateca', 1800,'Blanco')

# Inserción de los modelos de coches
cursor.execute("INSERT INTO Coche VALUES(?,?,?,?,?)", register1)
cursor.execute("INSERT INTO Coche VALUES(?,?,?,?,?)", register2)
cursor.execute("INSERT INTO Coche VALUES(?,?,?,?,?)", register3)
cursor.execute("INSERT INTO Coche VALUES(?,?,?,?,?)", register4)

# Commit de las operaciones
database.commit()

# Cierra la conexión a la base de datos
database.close()
```

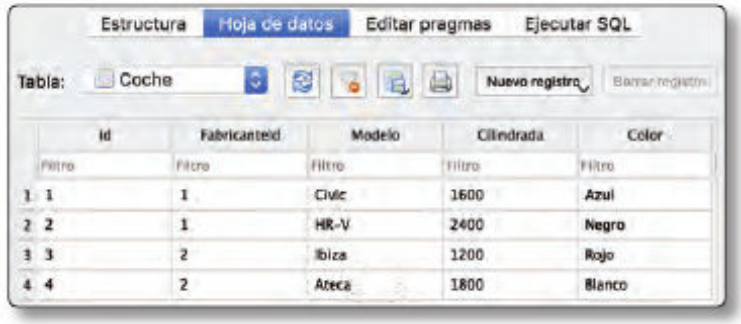
Después de ejecutar el programa, abre la aplicación SQLite Browser y vete al menú "Hoja de datos":



En el desplegable *Tabla* puedes seleccionar la tabla de la que quieres consultar los datos. La siguiente imagen muestra lo que verás en la tabla *Fabricante*:



La siguiente imagen muestra lo que verás en la tabla *Coche*:



26.3.3 Leyendo datos

El segundo ejercicio consiste en la realización de diferentes lecturas de información previamente insertada en el ejercicio uno. El comando para realizar consultas a la base de datos es el siguiente:

SELECT * FROM NombreTabla WHERE Condición

A destacar del comando:

- *: información que se lee de base de datos. El asterisco indica que se leerá toda la información de la tabla, aunque puede ser sustituido por el nombre de las columnas que se quieren leer.
- NombreTabla: nombre de la tabla sobre la que se ejecutará el comando.
- Condición: el *WHERE* es opcional en el comando, por tanto la condición es opcional. En la condición se introduce una comparación, por ejemplo, que el nombre del fabricante sea *Honda*.

El código fuente sigue los siguientes pasos:

1. Apertura de la conexión a la base de datos.
2. Creación de un cursor para realizar operaciones contra la base de datos.
3. Lectura y visualización de todos los datos de la tabla *Fabricante*.
4. Lectura y visualización de todos los datos de la tabla *Coche*.
5. Lectura de los datos de la tabla *Fabricante* para realizar posteriormente la lectura de los coches usando el identificador del fabricante.
6. Creación de un segundo cursor para realizar operaciones contra la base de datos.
7. Lectura y visualización de la información de la tabla Coche utilizando el identificador leído previamente del fabricante. La consulta de lectura tiene un parámetro que es el identificador, presta atención a cómo se define el parámetro y a cómo es utilizado posteriormente como parámetro de la consulta (sección de código comentada con *#Ejecución de la query*).
8. Cerrar la conexión a la base de datos.

El código fuente del ejercicio es el siguiente:

```
# Importa la librería para utilizar SQLite
import sqlite3

# Conecta a la base de datos
database = sqlite3.connect('coches.db')

# Apertuda de un cursor
cursor = database.cursor()
```

```
# Lectura de todos los fabricantes
cursor.execute("SELECT * FROM Fabricante")
print("Mostrando todos los fabricantes:")
for registro in cursor:
    print(registro)

# Lectura de todos los coches
cursor.execute("SELECT * FROM Coche")
print("Mostrando todos los coches:")
for registro in cursor:
    print(registro)

# Lectura de todos los coches a partir del fabricantes
cursor.execute("SELECT * FROM Fabricante")
for registro in cursor:
    print("Mostrando todos los coches del fabricante: ", registro[1])
    # Nuevo cursor para ejecutar la consulta para los coches
    cursorcoches = database.cursor()
    # Ejecución de la query
    parametro = (registro[0],)
    cursorcoches.execute("SELECT Modelo, Cilindrada, Color FROM Coche where Fa-
bricanteId = ?",parametro)
    # Muestra la información de cada coche asociada al fabricante
    for registrococche in cursorcoches:
        print(registro[1]," ",registrococche[0],", ",registrococche[1], "cc, ",
registrococche[2])

# Cierra la conexión a la base de datos
database.close()
```

La información de todo lo leído es mostrada por pantalla, la siguiente imagen muestra lo que verás en la consola una vez ejecutes el código fuente:

```
Mostrando todos los fabricantes:
(1, 'Honda', '911234567', 'Calle Japón 3', 'hello@honda.es')
(2, 'Seat', '919876543', 'Calle España 1', 'info@seat.es')
Mostrando todos los coches:
(1, 1, 'Civic', 1600, 'Azul')
(2, 1, 'HR-V', 2400, 'Negro')
(3, 2, 'Ibiza', 1200, 'Rojo')
(4, 2, 'Ateca', 1800, 'Blanco')
Mostrando todos los coches del fabricante:  Honda
Honda   Civic ,  1600 cc,  Azul
Honda   HR-V ,  2400 cc,  Negro
Mostrando todos los coches del fabricante:  Seat
Seat   Ibiza ,  1200 cc,  Rojo
Seat   Ateca ,  1800 cc,  Blanco
```

26.3.4 Modificando datos

El tercer ejercicio consiste en la modificación de algunos registros de la base de datos. El comando para hacer modificaciones en la información de base de datos es el siguiente:

```
UPDATE NombreTabla SET NombreColumna = Valor WHERE Condición
```

A destacar del comando:

- NombreTabla: nombre de la tabla sobre la que se ejecutará el comando.
- NombreColumna = Valor: indica el valor que se le asignará a la columna que se desea cambiar. Es posible introducir más de una asignación, en el caso de hacerlo irán separadas por comas.
- Condición: el *WHERE* es opcional en el comando, por tanto la condición es opcional. En la condición se introduce una comparación, por ejemplo, que el nombre del fabricante sea *Honda*.

El código fuente sigue los siguientes pasos:

1. Apertura de la conexión a la base de datos.
2. Creación de un cursor para realizar operaciones contra la base de datos.
3. Lectura y visualización de todos los datos de la tabla *Fabricante*.
4. Actualización del email del fabricante *Honda* por uno nuevo. En el comando se utilizada la condición para actualizar que es que el *id* sea igual a 1, valor que es conocido por los ejercicios anteriores.
5. Lectura y visualización de todos los datos de la tabla *Fabricante*.
6. Lectura y visualización de todos los datos de la tabla *Coche*.
7. Actualización del color y cilindrada del coche con *id* igual a 1, *Honda Civic*.
8. Actualización de la cilindrada del coche con *id* igual a 3, *Seat Ibiza*.
9. Lectura y visualización de todos los datos de la tabla *Coche*.
10. Cerrar la conexión a la base de datos.

El código fuente del ejercicio es el siguiente:

```
# Importa la librería para utilizar SQLite
import sqlite3

# Conecta a la base de datos
database = sqlite3.connect('coches.db')

# Apertuda de un cursor
cursor = database.cursor()

# Lectura de todos los fabricantes
cursor.execute("SELECT * FROM Fabricante")
print("Mostrando todos los fabricantes:")
for registro in cursor:
    print(registro)

# Actualización de un fabricante
query = "UPDATE Fabricante SET Email = 'nuevoemail@honda.es' WHERE id = 1"
cursor.execute(query)
database.commit()

# Lectura de todos los fabricantes
cursor.execute("SELECT * FROM Fabricante")
print("Mostrando todos los fabricantes:")
for registro in cursor:
    print(registro)

# Lectura de todos los coches
cursor.execute("SELECT * FROM Coche")
print("Mostrando todos los coches:")
for registro in cursor:
    print(registro)

# Actualización de un coche
query = "UPDATE Coche SET Color = 'Verde', Cilindrada = 2000 WHERE id = 1"
cursor.execute(query)
query = "UPDATE Coche SET Cilindrada = 1600 WHERE id = 3"
cursor.execute(query)
database.commit()

# Lectura de todos los coches
cursor.execute("SELECT * FROM Coche")
print("Mostrando todos los coches:")
```

```
for registro in cursor:
    print(registro)

# Cierra la conexión a la base de datos
database.close()
```

La siguiente imagen muestra la información que es mostrada en la consola durante la ejecución del programa:

```
Mostrando todos los fabricantes:
(1, 'Honda', '911234567', 'Calle Japón 3', 'hello@honda.es')
(2, 'Seat', '919876543', 'Calle España 1', 'info@seat.es')
Mostrando todos los fabricantes:
(1, 'Honda', '911234567', 'Calle Japón 3', 'nuevoemail@honda.es')
(2, 'Seat', '919876543', 'Calle España 1', 'info@seat.es')
Mostrando todos los coches:
(1, 1, 'Civic', 1600, 'Azul')
(2, 1, 'HR-V', 2400, 'Negro')
(3, 2, 'Ibiza', 1200, 'Rojo')
(4, 2, 'Ateca', 1800, 'Blanco')
Mostrando todos los coches:
(1, 1, 'Civic', 2000, 'Verde')
(2, 1, 'HR-V', 2400, 'Negro')
(3, 2, 'Ibiza', 1600, 'Rojo')
(4, 2, 'Ateca', 1800, 'Blanco')
```

Si lo deseas, puedes comprobar la información utilizando SQLite Browser, tal y como hiciste en el ejercicio sobre inserción de datos.

26.3.5 Borrando datos

El cuarto ejercicio consiste en el borrado de información de la base de datos. El comando para realizar borrados es el siguiente:

DELETE FROM NombreTable WHERE Condición

A destacar del comando:

- NombreTabla: nombre de la tabla sobre la que se ejecutará el comando.
- Condición: el *WHERE* es opcional en el comando, por tanto la condición es opcional. En la condición se introduce una comparación, por ejemplo, que el nombre del fabricante sea *Honda*.

El código fuente sigue los siguientes pasos:

1. Apertura de la conexión a la base de datos.
2. Creación de un cursor para realizar operaciones contra la base de datos.
3. Lectura y visualización de todos los datos de la tabla *Coche*.
4. Eliminación del registro con *id* igual a 3, *Seat Ibiza*.
5. Lectura y visualización de todos los datos de la tabla *Coche*.
6. Cerrar la conexión a la base de datos.

El código fuente del ejercicio es el siguiente:

```
# Importa la librería para utilizar SQLite
import sqlite3

# Conecta a la base de datos
database = sqlite3.connect('coches.db')

# Apertuda de un cursor
cursor = database.cursor()

# Lectura de todos los coches
cursor.execute("SELECT * FROM Coche")
print("Mostrando todos los coches:")
for registro in cursor:
    print(registro)

# Eliminación de un coche
query = "DELETE FROM Coche WHERE id = 3"
cursor.execute(query)
database.commit()

# Lectura de todos los coches
cursor.execute("SELECT * FROM Coche")
print("Mostrando todos los coches:")
for registro in cursor:
    print(registro)

# Cierra la conexión a la base de datos
database.close()
```

La siguiente imagen muestra la información que es mostrada en la consola durante la ejecución del programa:

```
Mostrando todos los coches:
(1, 1, 'Civic', 2000, 'Verde')
(2, 1, 'HR-V', 2400, 'Negro')
(3, 2, 'Ibiza', 1600, 'Rojo')
(4, 2, 'Ateca', 1800, 'Blanco')
Mostrando todos los coches:
(1, 1, 'Civic', 2000, 'Verde')
(2, 1, 'HR-V', 2400, 'Negro')
(4, 2, 'Ateca', 1800, 'Blanco')
```

Si lo deseas, puedes comprobar la información utilizando SQLite Browser, tal y como hiciste en el ejercicio sobre inserción de datos.

27

MÓDULOS

En este capítulo vamos a explicarte cómo puedes crear tus propios módulos y como puedes usarlos en los programas que escribes para así reutilizarlos en futuros programas que desarrolles

En un capítulo anterior te hemos explicado la librería estándar, pues lo que vamos a aprender a hacer en este capítulo es a crear nuestras propias librerías y poder utilizarlas de la misma manera que utilizamos la librería estándar.

Un módulo es un archivo de código fuente de Python que incorporamos a nuestro programa mediante la sentencia *import* y que nos permite utilizar todas las funcionalidades que dicho módulo tiene.

A continuación vamos a hacer una serie de ejercicios en los que crearemos una serie de módulos que podrás reutilizar en otros programas.

27.1 EJERCICIOS

El primer ejercicio consiste en evolucionar el ejercicio intermedio que hemos realizado previamente y crear un módulo independiente que contenga todas las operaciones matemáticas que realizaba en el ejercicio.

El nuevo módulo tendrá las siguientes operaciones:

- **Sumar**: la operación recibirá como parámetros los dos sumandos y devolverá el resultado de la suma.
- **Restar**: la operación recibirá el minuendo y el sustraendo como parámetros y devolverá el resultado de la resta.

- **Multiplicar**: la operación recibirá los dos multiplicando como parámetros y devolverá el resultado de la multiplicación.
- **Dividir**: la operación recibirá el dividendo y el divisor como parámetros y devolverá el resultado de la división. En caso de producirse una división por cero se devolverá -1.
- **Factorial**: la operación recibirá como parámetro el número del que se quiere calcular el factorial y devolverá el resultado de dicho cálculo.
- **Potencia**: la operación recibirá la base y el exponente como parámetros y devolverá el resultado del cálculo de la potencia.

El código fuente del módulo es el siguiente:

```
def Sumar(sum1,sum2):
    return sum1+sum2

def Restar(minuendo,sustraendo):
    return minuendo-sustraendo

def Multiplicar(multiplicando,multiplicador):
    return multiplicando*multiplicador

def Dividir(dividendo,divisor):
    try:
        resultado = dividendo/divisor
        return resultado
    except ZeroDivisionError:
        return -1

def Factorial(numero):
    if numero <=1:
        return 1
    else:
        return numero * Factorial(numero-1)

def Potencia(base,exponente):
    if exponente <= 0:
        return 1
    else:
        return base * Potencia(base,exponente-1)
```

Guarda el fichero con el nombre "*Operaciones.py*".

Una vez tienes el módulo desarrollado es el momento de implementar la calculadora. Sigue los siguientes pasos:

1. Crear un fichero dentro de la misma carpeta donde has guardado el fichero del módulo y ponle como nombre "*Calculadora.py*".
2. Copia el código fuente del ejercicio intermedio en el fichero que acabas de crear.
3. Modificación del fichero para utilizar el módulo. Hay que añadir en la primera línea del fichero la sentencia "*import Operaciones*".
4. Adaptar el código fuente del fichero para utilizar las operaciones del módulo.

El código fuente quedaría de la siguiente manera:

```
import Operaciones

def Sumar():
    sum1 = int(input("Sumando uno:"))
    sum2 = int(input("Sumando dos:"))
    print ("La Suma es:", Operaciones.Sumar(sum1,sum2))

def Restar():
    minuendo = int(input("Minuendo:"))
    sustraendo = int(input("Sustraendo:"))
    print ("La Resta es:", Operaciones.Restar(minuendo,sustraendo))

def Multiplicar():
    multiplicando = int(input("Multiplicando:"))
    multiplicador = int(input("Multiplicador:"))
    print ("La Multiplicacion es:", Operaciones.Multiplicar(multiplicando,multip
licador))

def Dividir():
    dividendo = int(input("Dividendo:"))
    divisor = int(input("Divisor:"))
    print ("La Division es:", Operaciones.Dividir(dividendo,divisor))

def Factorial():
    factorial = int(input("Introduzca el número del que quiere calcular el fac-
torial: "))
    print("El factorial de " + str(factorial) + " es: " + str(Operaciones.
```

```
Factorial(factorial)))

def Potencia():
    base = int(input("Introduzca la base de la potencia: "))
    exponente = int(input("Introduzca el exponente de la potencia: "))
    print("El valor de " + str(base) + " elevado a " + str(exponente) + " es: "
+ str(Operaciones.Potencia(base,exponente)))

def Calculadora():
    fin = False
    while not(fin):
        opc = int(input("Opcion:"))
        if (opc==1):
            Sumar()
        elif(opc==2):
            Restar()
        elif(opc==3):
            Multiplicar()
        elif(opc==4):
            Dividir()
        elif(opc==5):
            Factorial()
        elif(opc==6):
            Potencia()
        elif(opc==7):
            fin = 1

print ("""**************
Calculadora
************
Menu
1) Suma
2) Resta
3) Multiplicacion
4) Division
5) Factorial
6) Potencia
7) Salir""")
Calculadora()
```

Fíjate en cómo se utilizan las operaciones del módulo que hemos importado: *NombreMódulo.Operación*.

El fichero que tienes que ejecutar desde IDLE es *Ej1.py*.

La siguiente imagen muestra un ejemplo de ejecución del código fuente anterior:

```
************
Calculadora
************
Menu
1) Suma
2) Resta
3) Multiplicacion
4) Division
5) Factorial
6) Potencia
7) Salir
Opcion:1
Sumando uno:22
Sumando dos:45
La Suma es: 67
Opcion:2
Minuendo:87
Sustraendo:44
La Resta es: 43
Opcion:3
Multiplicando:3
Multiplicador:6
La Multiplicacion es: 18
Opcion:4
Dividendo:56
Divisor:12
La Division es: 4.666666666666667
Opcion:5
Introduzca el número del que quiere calcular el factorial: 5
El factorial de 5 es: 120
Opcion:6
Introduzca la base de la potencia: 4
Introduzca el exponente de la potencia: 4
El valor de 4 elevado a 4 es: 256
Opcion:7
```

El segundo ejercicio consiste en modificar uno de los ejercicios que hemos hecho en el capítulo de programación orientada a objetos. El ejercicio que vamos a modificar es el ejercicio que hicimos referente a heredar varias clases de una clase. La modificación consiste en crear un módulo aparte para cada una de las clases, es decir, cada clase en un fichero y un fichero con el código fuente principal.

Las clases que vamos a crear son las siguientes:

- **Persona**:
  - **Atributos**:
    - Nombre.
    - Apellidos.
    - Edad.

- **Métodos**:
  - GetNombre.
  - SetNombre.
  - GetApellidos.
  - SetApellidos.
  - GetEdad.
  - SetEdad.

- **Alumno**: heredará de la clase *Persona*.
  - **Atributos**:
    - Curso.
    - Asignaturas.
  - **Métodos**:
    - GetCurso.
    - SetCurso.
    - GetAsignaturas.
    - SetAsignaturas.
    - MostrarAlumno.

- **Profesor**: heredará de la clase *Persona*.
  - **Atributos**:
    - Antigüedad.
    - Tutorías.
    - Teléfono.
  - **Métodos**:
    - GetAntigüedad.
    - SetAntigüedad.
    - GetTutorias.
    - SetTutorias.
    - GetTelefono.
    - SetTelefono.
    - MostrarProfesor.

Cada clase la vas a crear en un fichero aparte:

- **Persona**: la crearás en el fichero "*persona.py*".
- **Alumno**: la crearás en el fichero "*alumno.py*".
- **Profesor**: la crearás en el fichero "*profesor.py*".

A la hora de realizar las importaciones de los módulos tendrás que hacerlo de la siguiente manera:

- Persona: no realizará ninguna importación.
- Alumno: realizará la importación del módulo *Persona*.
- Profesor: realizará la importación del módulo *Persona*.
- Código principal: realizará la importación de los módulos *Alumno* y *Profesor*.

Veamos cómo quedaría cada módulo o fichero:

Persona.py

```
class Persona:
    def __init__ (self):
        self.__Nombre = ""
        self.__Apellidos = ""
        self.__Edad = 0
    def GetNombre(self):
        return self.__Nombre
    def SetNombre(self,nombre):
        self.__Nombre = nombre
    def GetApellidos(self):
        return self.__Apellidos
    def SetApellidos(self,apellidos):
        self.__Apellidos = apellidos
    def GetEdad(self):
        return self.__Edad
    def SetEdad(self,edad):
        self.__Edad = edad
```

Alumno.py

```
import Persona

class Alumno(Persona.Persona):
    def __init__ (self):
        self.__Curso = ""
        self.__Asignaturas = ""
    def GetCurso(self):
```

```
        return self.__Curso
    def SetCurso(self,curso):
        self.__Curso = curso
    def GetAsignaturas(self):
        return self.__Asignaturas
    def SetAsignaturas(self,asignaturas):
        self.__Asignaturas = asignaturas
    def MostrarAlumno(self):
        print("Alumno:")
        print("\tNombre:",self.GetNombre())
        print("\tApellidos:",self.GetApellidos())
        print("\tEdad:",self.GetEdad())
        print("\tCurso:",self.__Curso)
        print("\tMatriculas:",self.__Asignaturas)
```

Profesor.py

```
import Persona

class Profesor(Persona.Persona):
    def __init__ (self):
        self.__Antigüedad = ""
        self.__Tutorias = ""
        self.__Telefono = ""
    def GetAntigüedad(self):
        return self.__Antigüedad
    def SetAntigüedad(self,antigüedad):
        self.__Curso = antigüedad
    def GetTutorias(self):
        return self.__Tutorias
    def SetTutorias(self,tutorias):
        self.__Tutorias = tutorias
    def GetTelefono(self):
        return self.__Telefono
    def SetTelefono(self,telefono):
        self.__Telefono = telefono
    def MostrarProfesor(self):
        print("Profesor:")
        print("\tNombre:",self.GetNombre())
        print("\tApellidos:",self.GetApellidos())
        print("\tEdad:",self.GetEdad())
        print("\tAntigüedad:",self.__Antigüedad)
```

```
        print("\tTutorias:",self.__Tutorias)
        print("\tTelefono:",self.__Telefono)
```

Ej2.py

```
import Alumno
import Profesor

alumno = Alumno.Alumno()
alumno.SetNombre("Alfredo")
alumno.SetApellidos("Moreno Muñoz")
alumno.SetEdad(35)
alumno.SetCurso("Bachillerato")
alumno.SetAsignaturas(["Matemáticas","Tecnología","Inglés"])
alumno.MostrarAlumno()

profesor = Profesor.Profesor()
profesor.SetNombre("Profesor")
profesor.SetApellidos("Casa Papel")
profesor.SetEdad(50)
profesor.SetAntigüedad(15)
profesor.SetTutorias([["Lunes","16-18"],["Jueves","12-14"],["Viernes","11-13"]])
profesor.SetTelefono("654321098")
profesor.MostrarProfesor()
```

El fichero que tienes que ejecutar desde IDLE es el fichero *Ej2.py*.

La siguiente imagen muestra la ejecución del código fuente anterior:

```
Alumno:
        Nombre: Alfredo
        Apellidos: Moreno Muñoz
        Edad: 35
        Curso: Bachillerato
        Matrículas: ['Matemáticas', 'Tecnología', 'Inglés']
Profesor:
        Nombre: Profesor
        Apellidos: Casa Papel
        Edad: 50
        Antigüedad:
        Tutorias: [['Lunes', '16-18'], ['Jueves', '12-14'], ['Viernes', '11-13']]
        Telefono: 654321098
```

28

PRUEBAS UNITARIAS

En este capítulo vamos a explicarte qué son las pruebas unitarias, qué importancia tienen dentro del desarrollo de software, qué beneficios tiene su utilización y por último vamos a realizar una serie de ejercicios utilizando a librería del módulo estándar *unittest*.

Una de las cosas más importantes que existen en el desarrollo de software es probar el código fuente que se escribe. Probar el código fuente no significa ejecutar la aplicación y probar empíricamente que funciona y que hace lo que debería de hacer, probar el software significa escribir pruebas en el código fuente que verifiquen de forma automática que el código fuente hace lo que tiene que hacer y que funciona cómo debería.

28.1 ¿QUÉ SON LOS TESTS UNITARIOS?

Las pruebas unitarias son una herramienta de programación utilizada para garantizar que el código fuente que se ha desarrollado y que se prueba con la prueba hace lo que debería. Las pruebas unitarias son una herramienta de carácter general en programación, lo que significa que no están ligados a un lenguaje de programación.

Las pruebas unitarias son la clave para poder garantizar la calidad del cualquier código, por tanto, al escribirlas se automatiza el control de calidad del código fuente.

La pregunta clave en todo ésto es: ¿Cuándo se escriben las pruebas unitarias? La respuesta es sencilla, a la par que el código fuente. Es aconsejable que escribas pruebas unitarias para cada función, método o módulo que desarrolles, de tal forma que asegurarás el correcto funcionamiento de los mismos.

¿Por qué deberíamos de utilizar pruebas unitarias? Pues realmente existen muchos motivos, pero de todos ellos nos gustaría destacar los siguientes:

- **Mejoran la calidad**: garantizan que el código fuente funciona correctamente.
- **Mejoran los tiempos de desarrollo**: mediante las pruebas unitarias podremos garantizar que una vez se introduzcan modificaciones en el código fuente éste sigue funcionando de forma correcta.
- **Documentación**: las pruebas unitarias son la documentación del código fuente ya que describen las funcionalidades del mismo. Si se modifican funcionalidades del software se tendrán que actualizar las pruebas unitarias.

Escribir pruebas unitarias no es diferente a escribir código fuente, básicamente las pruebas es un conjunto de líneas de código que prueban que las líneas de código principales funcionan como deberían. Por tanto, las pruebas deben de ser tratadas como si fuera código fuente, lo que implica que:

- Las pruebas pueden contener errores de código que te tocará corregir para asegurar que la prueba es correcta.
- El código fuente de las pruebas debe seguir el mismo estándar que el código fuente normal en términos de: uso correcto de nombres, sintaxis clara, añadir comentarios que aclaren las pruebas, código bien organizado, etc.
- Las pruebas tienen que poder ser cambiadas fácilmente si cambia el código fuente que prueban.
- Las pruebas, al igual que el código, deben de permitir ser modificadas para un mejor entendimiento y legibilidad.

28.1.1 La realidad

Utilizar pruebas unitarias tiene muchísimos beneficios que veremos más adelante, pero, aunque no te lo creas, no son una herramienta que tenga un uso extendido dentro de la comunidad de desarrollo de software.

Existe una razón principal por la que no está muy extendido y es que existe mucho desconocimiento sobre la materia, poca tradición y diversos falsos mitos.

El mito más claro referente a las pruebas unitarias es que se piensa que escribir pruebas unitarias implica escribir el doble de código fuente (el código normal y el código de la prueba), algo que es totalmente falso, es simplemente escribir un poco más de código.

En lo referente a tradición tenemos que hablar sobre que, normalmente, en los proyectos de desarrollo de software se suele dejar la escritura de las pruebas para el final, lo que provoca dos posibles escenarios:

- Las pruebas nunca llegan a implementarse dado que el proyecto desarrollado se considera "acabado".
- Las pruebas que se implementan son de baja calidad porque las personas encargadas de escribirlas consideran que su trabajo ya estaba "acabado" antes de realizar las pruebas.

En nuestros proyectos las pruebas siempre las tenemos que ir escribiendo a medida que se desarrolla el software. A medida que desarrollamos vamos probando nuestro código, lo que nos permite asegurar que la función, método o módulo queda terminado correctamente, libre de errores.

La realización de pruebas unitarias debe ser un proceso obligatorio en nuestros desarrollos y que no queden a la voluntad del desarrollador. Si el desarrollador no está habituado a su uso diario es muy fácil que tienda a evitar realizar este tipo de pruebas.

28.2 CARACTERÍSTICAS DE UNA BUENA PRUEBA UNITARIA

Llegados a este punto, tenemos que hablar sobre qué características debe de tener una prueba unitaria para considerarla que es una buena prueba unitaria. A continuación te listamos las características que debe de tener:

- Tiene que poder ejecutarse de forma automática para poder automatizarla.
- Tiene que poder repetirse tantas veces como se quiera.
- Tiene que poder ser ejecutadas en cualquier entorno y equipo de desarrollo e independientemente del estado en el que se encuentren dichos entornos o equipos.
- Tiene que dejar el entorno o equipo en el que se ejecutan en el mismo estado en el que se lo han encontrado al comenzar su ejecución.
- Tiene que ser totalmente independientes con el resto de pruebas, es decir, no puede afectar a otras pruebas ni verse afectada por otras.

- Tiene que tener un objetivo bien definido y entendible por todos los desarrolladores.
- Tiene que simular relaciones con otras funciones, métodos o módulo para así evitar dependencias con ellos.
- El conjunto de pruebas debe poder cubrir casi la totalidad del código fuente de la aplicación, por tanto, cuanto más código fuente tengamos cubierto con las pruebas así de buenas serán.

Tienes que tener en cuenta que aunque la lista de características que acabamos de exponerte definen si una prueba es buena o no, no siempre será posible ni necesario cumplir con todas estas reglas y será la experiencia la que nos guiará en la realización de las mismas.

28.3 BENEFICIOS DE LAS PRUEBAS UNITARIAS

El uso de pruebas unitarias en nuestros desarrollos nos aporta una serie de beneficios que son los siguientes:

- Aceleran el desarrollo de software ya que facilitan los cambios en la aplicación ya que las pruebas nos asegurarán que los nuevos cambios no han introducido errores.
- Los tiempos de depuración y corrección de incidencias se ven reducidos. Al tener preparadas pruebas para cada función, método o módulo los errores están más acotados y son más fáciles de localizar ya que las pruebas los encontrarían de forma rápida.
- Permiten probar funciones, módulos y métodos sin tener todo el programa desarrollado.
- Facilitan otro tipo de pruebas, como por ejemplo las pruebas de integración, ya que permiten llegar a la fase de integración asegurando que el código fuente está funcionando correctamente.
- Las pruebas ayudan a entender mejor el código fuente.
- Las pruebas documentan el código, son un libro abierto sobre el funcionamiento de la función y los resultados esperados.
- Ayuda a tener un código desacoplado gracias a que cada una de nuestras funciones está pensada para devolver un resultado que podrá ser testeado.

Resumiendo, las pruebas nos van a permitir tener a los desarrolladores más contentos ya que todo el código estará totalmente documentado por las pruebas, también tendrán menos estrés a la hora de hacer modificaciones ya que mediante las pruebas podrán comprobar que "no se han cargado nada". El cliente que utiliza el software también estará más contento, ya que con las pruebas aseguraremos que el producto hace lo que él espera que haga. Además, le daremos tiempos de respuesta a incidencias menores y tiempos de desarrollo de nuevas funcionalidades menores también.

28.4 PRUEBAS UNITARIAS EN PYTHON

En el capítulo en el que te explicamos la librería estándar de Python te dijimos que dicha librería contiene una librería para realizar pruebas unitarias, la librería se llama *unittest* y es la que vamos a utilizar en este capítulo.

La librería *unittest* nos va a permitir incluir pruebas unitarias en nuestro código implementando una clase heredada de la clase *unittest.TestCase* que contiene el módulo. En la clase que creemos podremos incluir tantas pruebas como queramos, pero cada prueba devolverá su propio resultado de ejecución. Los resultados que una prueba puede devolver son los siguientes:

- **OK**: la prueba se ha ejecutado correctamente.
- **FAIL**: la prueba no se ha ejecutado correctamente pero el error es producido de forma controlada.
- **ERROR**: la prueba no se ejecutado correctamente pero el error es una excepción no controlada dentro de la prueba.

El primer ejercicio consiste en crear una prueba que devuelva un OK, otra que devuelva FAIL y otra que devuelva ERROR. En el ejercicio no probaremos nada, simplemente queremos que te familiarices con la ejecución de las mismas y seas capaz de leer lo que la consola de Python muestra por pantalla. El ejercicio lo hemos dividido en tres partes, una para cada posible resultado de la prueba.

El código fuente del ejercicio que se devuelve un OK es el siguiente:

```
import unittest

class Pruebas(unittest.TestCase):
    def test(self):
        pass

unittest.main()
```

El método de prueba que se ejecutará es *test*, que de forma inmediata ejecutará la sentencia *pass* indicando de ésta forma que el test ha pasado.

La siguiente imagen muestra lo que se ve en la consola de Python al ejecutar la prueba:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.019s

OK
```

La consola de Python nos está indicando que ha ejecutado 1 test en 0.019 segundos y que el resultado ha sido: OK.

Veamos ahora la segunda parte del primer ejercicio. El código fuente del ejercicio que se devuelve un FAIL es el siguiente:

```
import unittest

class Pruebas(unittest.TestCase):
    def test(self):
        raise AssertionError()

unittest.main()
```

El cambio respecto al ejercicio anterior es que en lugar de ejecutar la sentencia *pass* el método de pruebas lanza un error de forma controlada mediante la sentencia *raise AssertionError()*.

La siguiente imagen muestra lo que se ve en la consola de Python al ejecutar la prueba:

```
F
======================================================================
FAIL: test (__main__.Pruebas)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/PruebasUnitarias/Ej1
_2.py", line 5, in test
    raise AssertionError()
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.020s

FAILED (failures=1)
```

La consola de Python nos está indicando el número de tests que ha ejecutado (uno en este caso) y el resultado del mismo: FAILED.

Veamos ahora la tercera parte del primer ejercicio. El código fuente del ejercicio que se devuelve un ERROR es el siguiente:

```
import unittest

class Pruebas(unittest.TestCase):
    def test(self):
        3/0

unittest.main()
```

Para provocar que el resultado sea un error no controlado hemos añadido una división por cero, lo que provocará que se lance una excepción no controlada.

La siguiente imagen muestra lo que se ve en la consola de Python al ejecutar la prueba:

```
E
======================================================================
ERROR: test (__main__.Pruebas)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/PruebasUnitarias/Ej1
_3.py", line 5, in test
    3/0
ZeroDivisionError: division by zero

----------------------------------------------------------------------
Ran 1 test in 0.032s

FAILED (errors=1)
```

Compara la imagen de este ejercicio con la del anterior, puedes comprobar que en la anterior el resultado era FAIL y en esta ERROR, aunque en el resumen inferior ponga FAILED en ambos el texto entre paréntesis indica si ha sido un fallo o un error.

Llegados a este punto vamos a adentrarnos de lleno en cómo se prueba el código fuente con la librería *unittest*. Cuando pruebas métodos, funciones o módulos la forma de hacerlo es añadir pruebas en las que sabes el resultado que debería de devolver lo que estás probando y comprobar si lo devuelve o no.

La librería *unittest* pone a nuestra disposición un conjunto de operaciones que nos van a facilitar la vida a la hora de realizar pruebas, dichas operaciones

devolverán OK en caso de que lo que se está probando y el resultado esperado sea el mismo y devolverá FAIL en caso contrario.

Dichas operaciones son las siguientes (vamos a suponer que a y x son los métodos que se están probando y b es el resultado esperado:

- **assertEqual(a, b)**: devolverá OK en caso de que a y b sean iguales y FAIL en caso de que no lo sean.
- **assertNotEqual(a, b)**: devolverá OK en caso de que a y b sean diferentes y FAIL en caso contrario.
- **assertTrue(x)**: devolverá OK en caso de que el valor sea *True* y FAIL en caso de que sea *False*.
- **assertFalse(x)**: devolverá OK en caso de que el valor sea *False* y FAIL en caso de que el valor sea *True*.
- **assertIs(a, b)**: devolverá OK en caso de que a sea b y FAIL en caso contrario.
- **assertIsNot(a, b)**: devolverá OK en caso de que a no sea b y FAIL en caso contrario.
- **assertIsNone(x)**: devolverá OK en caso de que x sea None y FAIL en caso contrario.
- **assertIsNotNone(x)**: devolverá OK en caso de que x no sea None y FAIL en caso contrario.
- **assertIn(a, b)**: devolverá OK en caso de que a esté en b y FAIL en caso contrario.
- **assertNotIn(a, b)**: devolverá OK en caso de que a no esté en b y FAIL en caso contrario.
- **assertIsInstance(a, b)**: devolverá OK en caso de que a sea una instancia de b y FAIL en caso contrario.
- **assertNotIsInstance(a, b)**: devolverá OK en caso de que a no sea una instancia de b y FAIL en caso contrario.

Tal y como has podido observar mientras leías, el conjunto de operaciones *assert* que nos ofrece la librería *unittest* se corresponden con una serie de operaciones

de comparación de elementos. La siguiente tabla muestra la relación entre la operación *assert* y su operación equivalente:

Función	Operación equivalente
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Vamos a hacer un ejercicio en el que probaremos un método que devuelve un valor booleano, en caso de que el valor devuelto sea *True* el resultado de la prueba será correcto y en caso contrario no. El código fuente es el siguiente:

```
import unittest

def SonIguales(num1,num2):
    if num1==num2:
        return True
    else:
        return False

class Pruebas(unittest.TestCase):
    def test(self):
        self.assertTrue(SonIguales(3,3))

unittest.main()
```

Fíjate la forma de utilizar las funciones *assert* que acabamos de ver, dicha función está probando la función *SonIguales* pasándole como argumentos dos números 3.

La siguiente imagen muestra la salida de la ejecución en la consola:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.031s

OK
```

Ahora prueba a cambiar el valor de uno de los 3 para que no sean iguales y el resultado de la prueba sea errónea. La función *assert* estará esperando que el resultado sea *True* y el método devolverá *False*. Cámbialo y ejecútalo, en la consola de Python verás lo siguiente:

```
F
======================================================================
FAIL: test (__main__.Pruebas)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/PruebasUnitarias/Ej2
_1.py", line 11, in test
    self.assertTrue(SonIguales(3,4))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 1 test in 0.029s

FAILED (failures=1)
```

El tercer ejercicio consiste en probar el fichero *Operaciones.py* que creamos en el capítulo en el que te explicamos los módulos. El fichero *Operaciones.py* tiene el siguiente código fuente:

```
def Sumar(sum1,sum2):
    return sum1+sum2

def Restar(minuendo,sustraendo):
    return minuendo-sustraendo

def Multiplicar(multiplicando,multiplicador):
    return multiplicando*multiplicador

def Dividir(dividendo,divisor):
    try:
        resultado = dividendo/divisor
        return resultado
    except ZeroDivisionError:
        return -1
```

```
def Factorial(numero):
    if numero <=1:
        return 1
    else:
        return numero * Factorial(numero-1)

def Potencia(base,exponente):
    if exponente <= 0:
        return 1
    else:
        return base * Potencia(base,exponente-1)
```

El ejercicio consiste en probar cada una de las seis operaciones que tiene el fichero. La prueba la vamos a realizar desde un fichero diferente pero que se encuentra dentro de la misma carpeta en la que se encuentra el fichero *Operaciones.py*, lo llamaremos *OperacionesPrueba.py* y tendrá el siguiente código fuente:

```
import unittest
import Operaciones

class Pruebas(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(Operaciones.Sumar(3,4),7)
    def test_resta(self):
        self.assertEqual(Operaciones.Restar(33,17),16)
    def test_multiplicar(self):
        self.assertEqual(Operaciones.Multiplicar(12,4),48)
    def test_dividir(self):
        self.assertEqual(Operaciones.Dividir(33,3),11)
    def test_potencia(self):
        self.assertEqual(Operaciones.Potencia(3,3),27)
    def test_factorial(self):
        self.assertEqual(Operaciones.Factorial(5),120)

unittest.main()
```

En el fichero de pruebas vamos a importar el módulo de operaciones y vamos a realizar la prueba de cada uno de los métodos utilizando la función *assertEqual* mediante la que comprobamos que el resultado devuelto por cada una de las funciones es el esperado.

La siguiente imagen muestra el resultado de ejecutar las pruebas:

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.109s

OK
```

Ahora prueba a cambiar una de las pruebas para que no devuelvan todas OK. Tendrás una salida parecida a la siguiente imagen en la consola:

```
.....F
======================================================================
FAIL: test_suma (__main__.Pruebas)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/alfre/Dropbox/Libro Python RA-MA/Libro/Ejercicios/PruebasUnitarias/Ope
raciones/OperacionesPruebas.py", line 6, in test_suma
    self.assertEqual(Operaciones.Sumar(3,2),7)
AssertionError: 5 != 7

----------------------------------------------------------------------
Ran 6 tests in 0.112s

FAILED (failures=1)
```

La librería *unittest* nos va a permitir preparar un pequeño entorno dentro de nuestras pruebas, creando atributos de la clase de pruebas por ejemplo, y también nos va a permitir liberar dicho entorno una vez acabe la prueba. La librería pone a nuestra disposición los siguientes métodos de clase que podremos utilizar en nuestras clases de prueba:

- **setUp**: método que preparar el contexto de las pruebas.
- **tearDown**: método que se ejecuta al acabar las pruebas y eliminará el contexto de las mismas.

Vamos a ver cómo se utilizan con un ejercicio. El ejercicio consiste en crear una lista de números enteros dentro del método *setUp*, ejecutar las pruebas de una función que devuelve un número aleatorio del 0 al 9 y después eliminar dicha lista de elementos con el método *tearDown*. El código fuente es el siguiente:

```
import unittest
import random

def Aleatorio10():
    return random.randrange(10)

class Prueba(unittest.TestCase):
```

```
        def setUp(self):
            self.numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        def test(self):
            self.assertIn(Aleatorio10(), self.numeros)

        def tearDown(self):
            del(self.numeros)

    unittest.main()
```

La siguiente imagen muestra la salida en la consola de Python de la ejecución de la prueba:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.033s

OK
```

Anexo 1

GLOSARIO

- **AND**: operador lógico que realiza la operación lógica 'Y' entre dos elementos. El resultado será *true* si ambos elementos son *true*, en caso contrario será *false*.
- **Atributo**: componente que tienen las clases para almacenar información.
- **Base de datos**: conjunto de datos pertenecientes a un mismo contexto que se almacenan para su uso posterior.
- **Bloque de código**: conjunto de sentencias de código fuente que están delimitadas por un inicio y un fin.
- **Booleano**: tipo de datos lógico que se caracteriza porque únicamente puede tener dos valores, o *True* o *False*.
- **Bucle**: sentencia específica que se repite durante un tiempo. El número de repeticiones puede ir en función de diversos factores, pero están indicados en la propia sentencia en la que se define el bucle.
- **Cadena de texto**: tipo de dato compuesto por una secuencia de caracteres.
- **Caso base**: caso de ejecución de una función recursiva que no llama de nuevo a la función recursiva ya que es el caso que no puede descomponerse en casos más pequeños.
- **Caso recursivo**: caso de ejecución de una función recursiva que llama de nuevo a la función recursiva ya que puede descomponerse en casos más pequeños.
- **Clase**: tipo de datos que define un conjunto de atributos y métodos.

- **Código fuente**: es el conjunto de líneas de texto que forman un programa. Las líneas de texto indican cómo se debe ejecutar dicho programa y lo qué tiene que hacer.
- **Cola**: tipo de dato que funciona exactamente igual que una cola del mundo real, el primero en entrar en la cola es el primero en salir de la misma.
- **Comentario de código**: utilidad de programación que permite añadir documentación dentro del código fuente.
- **Compilador**: traductor que traduce el programa en el lenguaje de programación en el que está escrito a otro lenguaje, normalmente el lenguaje máquina.
- **Composición**: técnica de la programación orientada a objetos que consiste en la creación de nuevas clases a partir de otras clases ya existentes que actúan como elementos compositores de la nueva. Las clases existentes serán atributos de la nueva clase.
- **Complejo**: número compuesto por parte real y parte imaginaria.
- **Concurrente**: capacidad de operar un conjunto de actividades al mismo tiempo, todas ejecutadas desde el mismo procesador.
- **Consola**: aplicación que se utiliza para interactuar con el sistema operativo o con el intérprete de Python.
- **Depurador**: es un programa usado para probar y eliminar los errores de otros programas.
- **Diccionario**: conjunto ordenado de elementos cuyos índices no son numéricos sino identificadores.
- **Encapsulación**: técnica utilizada en la programación orientada a objetos mediante la cual podemos ocultar atributos y métodos de la clase para que no puedan ser accedidos por elementos externos a la clase.
- **Entero**: número entero con límite de valor.
- **Entorno de desarrollo**: programa informático que contiene integradas todas las herramientas, utilidades y funcionalidades necesarias para facilitar la tarea de desarrollo de software.
- **Entrada estándar**: forma de entrada de información por defecto al programa.
- **Excepción**: error lógico que ocurre mientras se ejecuta el programa y que provoca su detención.

- **For**: tipo de bucle en el que se sabe el número de iteraciones exactas que se van a dar en su ejecución, es decir, es un bucle que busca ejecutar un conjunto de instrucciones de forma repetitiva hasta llegar al número máximo de iteraciones definidas.
- **Framework**: conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.
- **Función**: bloque de código fuente que contiene un conjunto de instrucciones que realiza algo concreto y que puede ser utilizada desde el código fuente que escribes tantas veces como necesites.
- **GIL**: el **G**lobal **I**nterpreter **L**ock, conocido como GIL, es un mecanismo utilizado en Python cuyo objetivo es que únicamente un hilo tenga el control del intérprete de Python, lo que implica que únicamente habrá un hilo activo en el intérprete en cada momento.
- **Herencia**: técnica utilizada en la programación orientada a objetos para crear nuevas clases a partir de otras.
- **Hilo**: conjunto de instrucciones dentro de un proceso que se ejecutan de forma independiente.
- **IDE**: programa informático que contiene integradas todas las herramientas, utilidades y funcionalidades necesarias para facilitar la tarea de desarrollo de software.
- **IDLE**: entorno de desarrollo de Python.
- **Lenguaje de programación**: lenguaje formal utilizado por los ingenieros de software para escribir programas. Mediante el lenguaje de programación se indican todas las sentencias que debe de ejecutar el programa.
- **Lenguaje de alto nivel**: lenguaje de programación caracterizado por expresar las sentencias de forma entendible al programador y no en lenguaje máquina.
- **Lenguaje interpretado**: tipo de lenguaje de programación que no es compilado si no ejecutado por un intérprete.
- **Lista**: conjunto ordenado de elementos que puede contener datos de cualquier tipo.
- **Método**: componente de una clase que aporta funcionalidad a la misma.

- **Multiplataforma**: característica del software que indica si el programa se puede ejecutar en los diferentes sistemas operativos sin ninguna restricción que afecte a su funcionamiento.
- **NOT**: operador lógico que realiza la operación lógica 'NO'. El resultado será *true* si el elemento es *false*, y será *false* si es *true*.
- **Objeto**: instancia de una clase.
- **Operador aritmético**: operadores que nos van a permitir realizar operaciones aritméticas con los datos.
- **Operador lógico**: operadores que permiten construir expresiones lógicas y que tienen como resultado un valor booleano, *True* o *False*.
- **Operador relacional**: operadores que permiten comparar dos valores entre y cuyo resultado es un valor booleano, *True* o *False*.
- **OR**: operador lógico que realiza la operación lógica 'O' entre dos elementos. El resultado será *true* si uno de los dos elementos es *true*, en caso contrario será *false*.
- **Paralelo**: ejecución de tareas de forma simultánea.
- **Pila**: tipo de dato que funciona exactamente igual que una pila del mundo real, el último en entrar es el primero en salir.
- **Proceso**: programa en ejecución.
- **Programa**: conjunto de instrucciones que se le dan a un ordenador para que realice una tarea específica.
- **Programar**: acción de escribir el conjunto de instrucciones que componen un programa.
- **Prueba unitaria**: herramienta de programación utilizada para garantizar que el código fuente que se ha desarrollado y que se prueba con la prueba hace lo que debería.
- **Python**: lenguaje de programación de propósito general muy sencillo y fácil de aprender, a la vez que poderoso y flexible.
- **Real**: número compuesto por parte entera y parte decimal.
- **Recursividad**: técnica de programación que se basa en utilizar funciones que se llaman a sí mismas durante su ejecución.

- **Salida estándar**: forma de salida de información por defecto del programa.
- **Serie**: ejecución de tareas de forma secuencial, una detrás de otra.
- **SGBD**: sistemas complejos que deben proporcionar una serie de funcionalidades para la utilización de bases de datos.
- **Shell**: aplicación que se utiliza para interactuar con el sistema operativo o con el intérprete de Python.
- **Sintaxis**: conjunto de reglas y orden que los lenguajes de programación utilizan para escribir las sentencias de los programas.
- **Software libre**: conjunto de programas, instrucciones y reglas informáticas que permiten ejecutar distintas tareas en una computadora y que los usuarios tienen la libertad de ejecutar, copiar, distribuir, estudiar, modificar y mejorar con total libertad.
- **Tabla**: componente de una base de datos que almacena los datos.
- **Terminal**: aplicación que se utiliza para interactuar con el sistema operativo o con el intérprete de Python.
- **Tupla**: conjunto ordenado e inmutable de elementos. Las tuplas pueden contener elementos de diferentes tipos.
- **Variable**: espacio de memoria que utilizan los programas para almacenar datos y utilizarlos durante su ejecución.
- **While**: tipo de bucle en el que no se sabe exactamente el número de iteraciones que se tienen que ejecutar, pero sí se sabe que hay que ejecutar iteraciones hasta que se deje de cumplir una condición.

Anexo 2

PALABRAS RESERVADAS

El objetivo de este anexo es presentar una serie de palabras reservadas que posee Python y que no puedes utilizar como nombres a la hora de escribir código fuente. Las palabras reservadas no puedes usarlas como nombres de:

- Variables.
- Funciones.
- Clases.
- Atributos.
- Métodos.

La siguiente tabla muestra las palabras reservadas en Python:

Palabra	Significado
and	Representación lógica de Y.
as	Tiene dos funciones, la primera de ellas es para asignar una excepción a un determinado objeto y la segunda es para renombrar un módulo importado al código.
assert	Se emplea para la depuración de código para lanzar errores si se cumplen ciertas condiciones.
async	Se emplea para definir una función como asíncrona.
await	Se emplea para realizar una pausa en una función asíncrona.
break	Sirve para finalizar un bucle.
class	Utilizada para definir una clase.
continue	Suspende la iteración de un bucle y salta a la siguiente iteración de éste.
def	Se emplea para definir funciones.

del	Tiene dos funciones, la primera de ellas es eliminar la referencia de un objeto concreto y la segunda es para eliminar elementos de una lista.
elif	Definición de una bifurcación alternativa con condición.
else	Definición del camino sin condición en una bifurcación.
except	Utilizada para capturar excepciones ocurridas durante la ejecución del código fuente.
False	Se emplea para representar el valor booleano 0 / falso.
finally	Utilizada para definir un bloque de código fuente que se ejecutará al final del procesamiento de las excepciones.
for	Utilizada para definir bucles for.
from	Se emplea para importar elementos de módulos externos a nuestro código.
global	Se emplea para modificar objetos en un ámbito inferior, creando un objeto nuevo y sin alterar el valor del objeto del ámbito superior.
if	Definición de una bifurcación con condición.
import	Importa un módulo externo a nuestro código. Se puede utilizar junto a *from*, pero en ese caso importará elementos en vez del módulo completo.
in	Determina la existencia de un elemento en una lista, tupla, diccionario o cualquier objeto iterable.
is	Determina si dos objetos son iguales. No es lo mismo dos objetos con los mismos valores que dos objetos iguales.
lambda	Utilizada para definir funciones lambda.
None	Se emplea para representar la ausencia de valor.
nonlocal	Permite modificar el valor de un objeto definido en un ámbito anterior.
not	Representación lógica de NO.
or	Representación lógica de O.
pass	Únicamente tiene funciones estéticas para rellenar huecos en el código fuente.
print	Utilizada para imprimir por pantalla una cadena de texto.
raise	Utilizada para lanzar excepciones.
return	Se emplea para devolver un elemento al finalizar la ejecución de una función.
True	Se emplea para representar el valor booleano 1 / verdadero.
try	Utilizada para capturar excepciones ocurridas durante la ejecución del código fuente.
while	Utilizada para definir bucles while.
with	Se emplea para encapsular la ejecución de un bloque de código fuente.
yield	Utilizada para devolver más de un elemento al finalizar la ejecución de una función.

MATERIAL ADICIONAL

El material adicional de este libro puede descargarlo en nuestro portal web: *http://www.ra-ma.es.*

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un ichero ZIP con una contraseña de este tipo: 978-84-9964-849-1

Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el ichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

Python práctico

Herramientas, conceptos y técnicas

El gran secreto para aprender algo es practicar, una y otra vez, hasta que consigues el objetivo que te habías propuesto.

En este libro vas a encontrar todo el contenido necesario para que aprendas a programar y aprendas a hacerlo con Python. Vamos a explicarte todos los conocimientos que tienes que saber para poder empezar, divididos en tres grandes bloques y acompañado de más de 150 ejercicios y ejemplos prácticos.

- Conceptos teóricos.
- Puesta en marcha.
- Aprendizaje práctico.

En el primer bloque vamos a explicarte los conceptos básicos de programación para que te vayas familiarizando con la programación. Además te explicaremos conceptos teóricos del lenguaje de programación Python y te contaremos su historia.

En el segundo bloque vamos a explicarte cómo poner en marcha en tu ordenador todo lo que necesitas para aprender a programar y hacerlo con Python. ¡Da igual el sistema operativo que tengas!

En el tercer bloque vamos a explicarte todos los conceptos de programación junto con ejercicios para que practiques lo aprendido. Los capítulos están organizados en orden de aprendizaje progresivo, y ordenados de tal forma que facilitan el aprendizaje afianzando los conocimientos aprendidos en capítulos anteriores con los nuevos conocimientos que tienes que aprender en cada uno de los capítulos.

El libro está pensado para utilizarse como:

- Apoyo para la docencia, ya que cuenta con toda la teoría necesaria para aprender a programar y aprender Python junto con una serie de ejercicios que permitirán que los alumnos adquieran los conocimientos teóricos aplicándolos a los ejercicios.
- Material para aprender de forma autónoma, ya que guía paso a paso al lector para aprender todos lo que necesitar saber para dominar la programación y Python.
- Obra de consulta para profesionales ligados a la programación, ya que contiene todos los fundamentos teóricos y prácticos, tanto generales como relacionados con Python.

El libro contiene material adicional que podrá descargar accediendo a la ficha del libro en **www.ra-ma.es**. Este material incluye la construcción y código propuestos en esta obra.

ISBN: 978-958-792-168-7

ra-ma.es

ediU

Ra-Ma®