Facultad de Informática

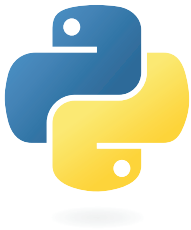
UNIVERSIDAD DE MURCIA

AUTÓMATAS Y LENGUAJES FORMALES

GRADO EN INGENIERÍA INFORMÁTICA

Apuntes de Python

Autores:
Eduardo MARTÍNEZ GRACIÁ
edumart@um.es
Mercedes VALDÉS VELA
mdvaldes@um.es
Santiago PAREDES MORENO
chapu@um.es
José Manuel JUÁREZ HERRERO
jmjuarez@um.es



23 de septiembre de 2021

Agradecimientos

El uso de Python en las prácticas de Autómatas y Lenguajes Formales se podría haber quedado en un simple proyecto si no hubiésemos contado con la ayuda de un alumno interno con suficiente entusiasmo como para iniciar el camino. Adrián Cánovas Rodríguez colaboró con los profesores de la asignatura durante el curso 2017/2018, probando las posibilidades del lenguaje para ser empleado en el desarrollo del tipo de aplicaciones que se plantean en los enunciados de prácticas. Elaboró una primera versión del manual de instalación de Eclipse con Python (que pasó a la historia, porque ahora usamos PyCharm), así como el código para cargar y manejar desde Python los autómatas finitos deterministas generados con las versiones 7 y 8-beta de la herramienta JFLAP.

ÍNDICE GENERAL

1. Introducción **5**
- 1.1. Breve historia de Python . . . 5
- 1.2. Características principales . . . 7
- 1.3. Entorno de desarrollo . . . 8
  - 1.3.1. Instalación de Python . . . 8
  - 1.3.2. PyCharm y Python . . . 10

2. Fundamentos de Python **13**
- 2.1. Un programa básico . . . 13
  - 2.1.1. Comentarios en Python . . . 14
  - 2.1.2. Punto de entrada de un módulo . . . 14
  - 2.1.3. Indentación . . . 15
- 2.2. Tipos básicos . . . 15
  - 2.2.1. Enteros . . . 15
  - 2.2.2. Reales . . . 16
  - 2.2.3. Complejos . . . 16
  - 2.2.4. Cadenas de caracteres . . . 16
  - 2.2.5. Boolean . . . 17
  - 2.2.6. None . . . 17
  - 2.2.7. Conversiones de tipos . . . 17
  - 2.2.8. Comprobaciones de tipos . . . 18
- 2.3. Variables . . . 18
- 2.4. Operadores . . . 19
  - 2.4.1. Operadores aritméticos . . . 20
  - 2.4.2. Operadores relacionales . . . 20
  - 2.4.3. Operadores lógicos . . . 21
  - 2.4.4. Precedencia entre distintos tipos de operador . . . 22
- 2.5. Estructuras de control . . . 23
  - 2.5.1. Sentencia if . . . 23

2.5.2. Sentencia while . . . 23
2.5.3. Sentencia for . . . 24
2.6. Funciones . . . 25
2.6.1. Argumentos con valores por defecto . . . 26
2.6.2. Invocación con parámetros nombrados . . . 27
2.6.3. No hace falta predeclarar funciones . . . 27
2.7. Excepciones . . . 27
2.7.1. Bloque try-except . . . 28
2.7.2. Bloque try-except-else-finally . . . 29
2.7.3. Aserciones . . . 29

3. Módulos y paquetes **31**
3.1. Dos formas de importar . . . 31
3.1.1. Import . . . 32
3.1.2. From - import . . . 32
3.1.3. Cómo funciona la importación . . . 33
3.2. Paquetes . . . 35
3.2.1. Creación de paquetes en PyCharm . . . 36
3.2.2. Gestión de paquetes con `pip` . . . 36

4. Cadenas **39**
4.1. Operaciones básicas . . . 40
4.1.1. Concatenación . . . 40
4.1.2. Repetición . . . 40
4.1.3. Longitud . . . 40
4.1.4. Indexación . . . 40
4.1.5. Troceado . . . 41
4.1.6. Conversión a cadena . . . 41
4.1.7. Conversiones de caracteres . . . 42
4.1.8. Iteración con cadenas . . . 42
4.1.9. Comprobación de subcadena . . . 42
4.2. Métodos de procesado de cadenas . . . 42
4.2.1. Métodos de búsqueda y sustitución . . . 42
4.2.2. Métodos de fragmentación . . . 43
4.3. Formateo de cadenas . . . 43
4.3.1. Expresiones de formateo de cadenas . . . 44
4.3.2. Método de formateo de cadenas . . . 46

5. Listas, diccionarios, conjuntos y tuplas **47**
5.1. Listas . . . 47
5.1.1. Operaciones básicas . . . 47
5.1.2. Modificación de listas . . . 48
5.1.3. Otras operaciones . . . 49
5.2. Diccionarios . . . 50
5.2.1. Operaciones básicas . . . 50
5.3. Conjuntos . . . 52
5.3.1. Operaciones básicas . . . 52
5.3.2. Operaciones de conjuntos . . . 53
5.4. Tuplas . . . 53

6. Entrada y salida **55**
6.1. Argumentos del programa . . . 55
6.2. Entrada, salida y errores . . . 56
6.2.1. Entrada estándar . . . 56
6.2.2. Salida estándar y salida de errores . . . 56
6.3. Ficheros . . . 57
6.3.1. Apertura . . . 57
6.3.2. Lectura . . . 58
6.3.3. Escritura . . . 60
6.3.4. Otros métodos . . . 60

7. Clases **63**
7.1. Introducción . . . 63
7.1.1. Definición . . . 64
7.1.2. Creación y uso de instancias . . . 64
7.1.3. Variables de clase . . . 65
7.1.4. Visibilidad . . . 65
7.2. Herencia . . . 65
7.3. Comprobaciones con objetos . . . 66

8. Autómatas en Python **69**
8.1. Instalación del paquete `jflap` . . . 69
8.2. Instanciación de un Afd . . . 70
8.3. Uso de Afd . . . 70
8.3.1. Mostrar el Afd . . . 71
8.3.2. Obtener el estado inicial . . . 71
8.3.3. Consultar una transición . . . 71
8.3.4. Comprobar si un estado es final . . . 72
8.3.5. Obtener el alfabeto del autómata . . . 72

9. Expresiones regulares **73**
9.1. Paquetes `re` y `regex` . . . 73
9.1.1. Representación de la expresión regular . . . 74
9.1.2. Compilación de la expresión regular . . . 74
9.1.3. Validación de cadenas . . . 74
9.1.4. Búsqueda de cadenas . . . 75
9.1.5. Sustituciones . . . 75
9.2. Grupos . . . 76
9.2.1. Grupos en validación y búsqueda . . . 76
9.2.2. Grupos en sustituciones . . . 77
9.2.3. Otras formas de referenciar grupos . . . 77

10. Documentación de Python **79**
10.1. Formato de la documentación . . . 80
10.1.1. Documentación de una clase . . . 80
10.1.2. Documentación de un método . . . 81
10.1.3. Documentación de un módulo . . . 81
10.2. Uso de la documentación . . . 81

Bibliografía **83**

1

INTRODUCCIÓN

A partir del curso 2018/2019, la asignatura de Autómatas y Lenguajes Formales comienza a emplear Python en las prácticas de programación. Emplear este lenguaje conlleva algunas ventajas con respecto a su predecesor, Java. En primer lugar, el lenguaje Python se concibió para que fuese fácil de aprender. En las primeras semanas del curso se impartirán seminarios sobre este lenguaje que darán las bases para desarrollar aplicaciones de validación de cadenas con autómatas finitos y expresiones regulares, y de traducción entre formatos de ficheros. Además, Python es un lenguaje en continua expansión, cada vez más usado en sectores con una fuerte proyección de futuro, como el análisis de datos, el cálculo científico y la bioinformática. Creemos que el hecho de usar uno de los lenguajes más populares en la actualidad puede dar una mayor motivación para abordar las prácticas de la asignatura.

Es necesario realizar una *advertencia* antes de seguir: estos apuntes no son un manual del lenguaje Python. Únicamente pretenden recoger lo que es necesario para abordar los problemas tratados en la asignatura. Además, están enfocados a alumnos que tienen cierto conocimiento del lenguaje C, ya que en numerosas ocasiones se compara la forma de realizar algo en este lenguaje y en Python. Para un estudio más amplio de Python, se recomienda la consulta de las referencias bibliográficas que aparecen al final de estos apuntes.

1.1. Breve historia de Python

El lenguaje Python fue concebido inicialmente por el informático holandés Guido Van Rossum, cuando trabajaba en el *Centrum Wiskunde & Informatica*[1] (CWI) de Ámsterdam hacia finales de los ochenta. Van Rossum formaba parte del equipo dedicado al desarrollo del sistema operativo distribuido *Amoeba* (proyecto liderado por Andrew Tanenbaum), y su objetivo al crear Python era proporcionar un lenguaje de programación que pudiese cubrir el hueco que existía en Amoeba entre el lenguaje C y el shell del sistema. Hacía falta un lenguaje de alto nivel con mayor po-

[1] Centro para las Matemáticas y la Ciencia de la Computación (`http://www.cwi.nl`).

Guido Van Rossum

Monty Python

Figura 1.1: Personajes relevantes en el origen de Python

tencia expresiva que el shell pero sin las dificultades de C, que en Amoeba tenía un interfaz de programación con el sistema bastante complicado.

Según el propio Van Rossum[2], el nombre del lenguaje es un homenaje a los cómicos ingleses Monty Python. Existen lenguajes cuyos nombres proceden de personajes famosos, como Pascal, Ada o Eiffel. Python sería, salvando las distancias, una continuación de esta tradición.

Una de las fuentes de inspiración de Van Rossum para la concepción de Python fue el lenguaje *ABC*, también desarrollado en el CWI de Ámsterdam, que pretendía ser un sustituto del lenguaje *BASIC* (aparecido hacia los sesenta) para la enseñanza de la programación. Python tiene una sintaxis en la que la sencillez es un principio esencial para atraer a programadores noveles, pero también asume algunas notaciones sintácticas propias de lenguajes con mucha implantación, como C, para que los programadores experimentados puedan usar Python con facilidad.

La primera versión pública de Python se presenta en 1991. El año 1996, Van Rossum continúa desarrollando el lenguaje con un equipo de colaboradores en el *Corporation for National Research Institute* (CNRI) en Reston, Virginia (EE.UU.), una organización sin ánimo de lucro cuyo objetivo es la promoción de las tecnologías de la información. En colaboración con la agencia estatal estadounidense *Defense Advanced Research Projects Agency* (DARPA), el año 1999 se inicia un proyecto para estudiar la aplicación de Python a la enseñanza de la programación. En el año 2001 se creó la *Python Software Foundation* (PSF)[3], siguiendo el modelo de la fundación Apache[4], y se lanzó la versión 2.1 del lenguaje que contaba con una licencia compatible con la *General Public Licence* de GNU. De 2005 a 2012 Van Rossum pasa a trabajar en Google, y de 2013 a la actualidad en Dropbox, potenciando a través de estas compañías la divulgación y el uso del lenguaje Python.

Van Rossum ha sido el *Benevolent Dictator for Life* (BDFL)[5] en la comunidad de desarrolladores de Python, es decir, ha asumido la responsabilidad de tomar las decisiones finales sobre la

[2]En el blog de Guido Van Rossum sobre la historia de Python (http://python-history.blogspot.com), se puede leer lo siguiente: *I picked the first thing that came to mind, which happened to be Monthy Python's Flying Circus, one of my favorite comedy troupes. The reference felt suitably irreverent for what was essentially a "skunkworks project". The word "Python" was also catchy, a bit edgy, and at the same time, it fit in the tradition of naming languages after famous people, like Pascal, Ada, and Eiffel. The Monty Python team may not be famous for their advancement of science or technology, but they are certainly a geek favorite. It also fit in with a tradition in the CWI Amoeba group to name programs after TV shows.*

[3]Web de la fundación: `https://www.python.org/psf-landing/`.

[4]Fundación sin ánimo de lucro dedicada al desarrollo de aplicaciones gratuitas, empezando por el famoso servidor HTTP: `https://www.apache.org/`.

[5]Dictador benevolente vitalicio.

evolución del lenguaje. Su *benevolente dictadura* se ha caracterizado por una actitud conservadora, admitiendo pocos cambios al lenguaje entre versiones sucesivas. Sin embargo, no han sido nada *benevolentes* las fuertes críticas que ha recibido por adoptar esta postura, hasta el punto de que, el 12 de julio de 2018, con un mensaje enviado a la lista de python-committers[6], anunció su completa retirada de la toma de decisiones sobre la evolución del lenguaje sin nombrar a ningún sucesor.

A lo largo de los años, las decisiones sobre el lenguaje Python se han tomado mediante Python Enhancement Proposals (PEPs). Mientras Guido Van Rossum era el BDFL, él tenía la última palabra sobre las características que se iban a integrar en la siguiente versión del lenguaje.

Al abandonar su función de BDFL, Guido Van Rossum pidió a un equipo de desarrolladores del núcleo de Python que pensasen cómo podía ser el modelo de gobierno a partir de ese momento. A finales de 2018 se propusieron varias alternativas, incluyendo la elección de un *Gracious Umpire Influencing Decisions Officer* (GUIDO)[7] o la opción de crear un modelo comunitario basado en el consenso y la votación. En diciembre de 2018 se optó por un modelo basado en un *Steering Council*[8] formado por los desarrolladores principales del lenguaje, elegidos por votación con cada nueva versión principal de Python.

1.2. Características principales

Python es un lenguaje de programación cuyo uso combina un compilador y un intérprete, de forma muy similar a Java. El código de los ficheros fuente de Python, con extensión `.py`, pasa en primer lugar por un compilador, que traduce el programa desde el lenguaje Python a una versión equivalente en un lenguaje de bajo nivel, el llamado *bytecode* de Python (un ensamblador de una máquina virtual). El compilador genera un fichero con extensión `.pyc` con el mismo nombre que el fichero fuente, compilando sólo los ficheros que han sido modificados.

Una vez que están todos los ficheros fuente del programa compilados, se lanza el intérprete de Python para ejecutar el programa. El intérprete tiene una concepción similar a la de la máquina virtual de Java. Carga y ejecuta los bytecodes, permitiendo que cualquier programa Python se ejecute en cualquier sistema para el cual exista un intérprete. Esto facilita la portabilidad del código Python. Tanto el compilador de referencia (`CPython`) como el intérprete de Python están escritos en el lenguaje C, garantizando de esta forma una ejecución rápida de ambos.

Aunque no es habitual en el desarrollo de aplicaciones, se puede usar el intérprete de Python interactivamente desde una consola que permite ejecutar directamente operaciones del lenguaje. Este modo de uso de Python sería equivalente al de un shell del sistema operativo, y podría emplearse para ejecutar secuencias de comandos de un solo uso. También podría permitir hacer pequeñas pruebas de código antes de integrarlo en un fichero `.py`.

Python permite desarrollar aplicaciones usando varios paradigmas de programación. El lenguaje es orientado a objetos, pero se puede usar en gran medida como lenguaje imperativo. Además, es un lenguaje dinámicamente tipado. Esto significa que no es necesario declarar las variables antes de su uso, y que a una misma variable se le pueden asignar valores de tipos de dato distintos a lo

[6] En `https://www.mail-archive.com/python-committers@python.org/msg05628.html` se puede leer el mensaje de retirada de Guido Van Rossum. La revolución en la comunidad de Python se ha desencadenado a raíz de una acalorada discusión en torno a permitir o no el uso de asignaciones en expresiones. Se puede consultar en el Python Enhancement Proposal (PEP) 572: `https://www.python.org/dev/peps/pep-0572/`. Y es que, por lo que parece, la sintaxis de los lenguajes de programación desata pasiones.

[7] Gracioso árbitro que influye en las decisiones oficiales.

[8] Consejo Directivo.

largo de la ejecución del programa. La verificación del uso correcto de los tipos en cada operación que use una variable se realiza en tiempo de ejecución.

Otra característica de Python es el manejo autómatico de la memoria. El programador no debe preocuparse de llamar a funciones que reservan o liberan memoria, evitando el manejo de punteros como en C. Además, Python incluye una librería estándar que permite realizar operaciones de gran utilidad: validación y búsqueda de cadenas con expresiones regulares, consultas sobre ficheros XML, programación de aplicaciones que usan protocolos de red como HTTP, y un largo etcétera. Por esta razón, Python es una opción muy interesante para implementar prototipos de aplicaciones de forma rápida.

1.3. Entorno de desarrollo

En esta sección se indican los pasos necesarios para disponer de una versión actualizada del compilador e intérprete de Python, integrada en la herramienta de desarrollo PyCharm, con el fin de disponer de un entorno de desarrollo similar al que está instalado en los laboratorios de la FIUM.

1.3.1. Instalación de Python

En primer lugar, es necesario descargar e instalar la última versión de Python 3 en nuestro equipo. En el momento de actualizar este manual, está disponible la versión 3.9.6.

En función del sistema que estemos empleando, tenemos dos opciones para hacer la instalación. Si usamos Windows o Mac OS X, podemos descargar el instalador que necesitemos, y si usamos alguna distribución de Linux, será más sencillo emplear el comando adecuado para la instalación de software en el sistema, como `apt-get` o `yum`. Lo vemos en detalle en las siguientes secciones.

1.3.1.1. Windows

Descarga desde `https://www.python.org/downloads` el instalador adecuado a tu Windows (64 o 32 bits). Cuando ejecutes el instalador, es importante que marques la opción que indica *Add Python 3.9 to PATH*, para que se pueda invocar a Python desde línea de comandos (figura 1.2).

Al terminar la instalación, podremos encontrar en nuestro sistema cuatro herramientas:

- Integrated Development and Learning Environment (IDLE). Es un entorno básico de desarrollo de Python. En la sección 1.3.2 se explica cómo emplear PyCharm como herramienta de desarrollo.
- Python 3.9 Module Docs. Documentación de los módulos y paquetes (librerías) de Python instalados en nuestro equipo. En el capítulo 3 se describe el sistema de módulos y paquetes de Python, y se indica el modo de agregar nuevos paquetes a nuestro sistema.
- Python 3.9 Manuals. Documentación muy amplia sobre el lenguaje (tutorial) y sobre otros aspectos relacionados con el funcionamiento de Python en nuestro equipo, como la gestión de módulos/paquetes de nuestra instalación, o la forma de comunicar código desarrollado en C con Python.
- Python 3.9. Intérprete de Python. Al ejecutarlo se abre una ventana de comandos con el prompt de Python >>>, permitiendo un uso interactivo del lenguaje.

Figura 1.2: Instalación en Windows

1.3.1.2. Mac OS X

Ten en cuenta que un sistema Mac OS X ya tiene alguna versión de Python instalada. Lo más normal es que sea alguna versión de Python 2 (una rama en extinción del lenguaje). No pasa nada por tener dos versiones distintas de Python. En la sección 1.3.2 veremos cómo hay que indicarle al entorno de desarrollo cuál es la que nos interesa usar.

Después de la instalación, en el listado de aplicaciones del sistema podremos encontrar una carpeta Python 3.9 que tendrá accesos a:

- Integrated Development and Learning Environment (IDLE).
- Python Documentation (tutorial del lenguaje, manual de módulos y paquetes, e instrucciones sobre la gestión de los mismos).
- Python Launcher. Permite configurar el intérprete que ejecuta los scripts de Python al abrirlos desde Finder.

1.3.1.3. Linux

Para instalar Python en Linux, lo más conveniente es usar el comando que gestiona la instalación de paquetes desde un terminal. Por ejemplo, en Ubuntu se haría así:

```
1 sudo apt-get update
2 sudo apt-get install python3
3 sudo apt-get install python3-pip
```

Es posible que el comando anterior nos indique que ya tenemos instalada la versión más reciente de Python. La última instrucción instala la herramienta de gestión de paquetes de Python. En caso de usar Fedora, la instalación es muy similar salvo que se debe usar `yum` en lugar de `apt-get`.

Con lo anterior, podremos ejecutar desde un terminal el comando `python3` y se iniciará el intérprete de Python.

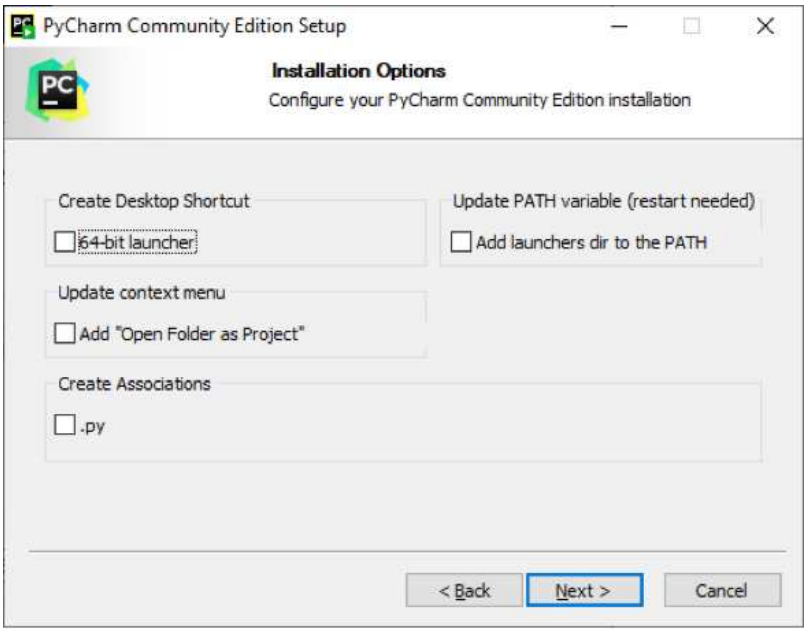


Figura 1.3: Instalación de PyCharm

1.3.2. PyCharm y Python

En Autómatas y Lenguajes Formales vamos a usar PyCharm como entorno de programación de Python. Es uno de los entornos preferidos por los desarrolladores de este lenguaje. En los laboratorios de la Facultad encontrarás la versión 2021.1.3 instalada.

Empezamos con la descarga desde `https://www.jetbrains.com/es-es/pycharm/`. Al lanzar el instalador puedes dejar las opciones de instalación por defecto (figura 1.3).

La primera vez que iniciamos PyCharm después de la instalación, tendremos que aceptar los términos de la licencia de JetBrains y también podremos indicar que si queremos (o no) enviar los datos de uso de la aplicación de forma anónima.

Podemos crear un primer proyecto siguiendo estos pasos:

1. En la ventana de inicio de PyCharm (figura 1.4), selecciona la opción *New Project*.
2. En el cuadro de diálogo para configurar el proyecto (figura 1.5), comprueba que en la opción *Base interpreter* aparece seleccionado Python 3.9. Si tienes varias versiones de Python en tu equipo, podrás ver todas ellas listadas en esta opción.
3. La opción *Virtualenv* permite que toda la ejecución se lleve a cabo en una máquina virtual específica del proyecto. Es decir, podrás instalar librerías dentro del proyecto sin que afecte al resto de tu equipo. Esto es algo conveniente en la fase de desarrollo de la aplicación.
4. Si seleccionas el checkbox *Create a main.py welcome script*, se añadirá automáticamente al proyecto un archivo *main.py* que podrá servir como punto de partida para la implementación de la aplicación.

Cuando pulsemos el botón *Create* pasaremos al ver el entorno de desarrollo de forma similar a la mostrada en la figura 1.6. Esta será la vista principal desde la que podrás editar el código, ejecutar y hacer depuración del programa. Merece que inviertas tiempo en aprender a depurar. Es una auténtica inversión que luego compensará con otro tiempo ahorrado en buscar fallos.

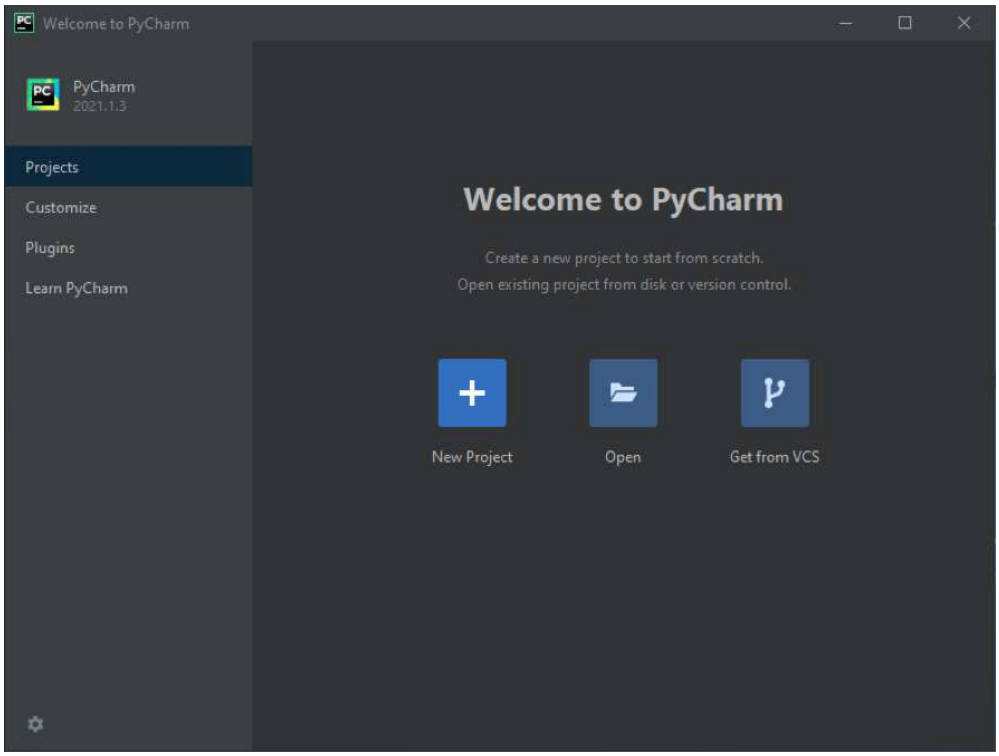


Figura 1.4: Ventana de inicio de PyCharm

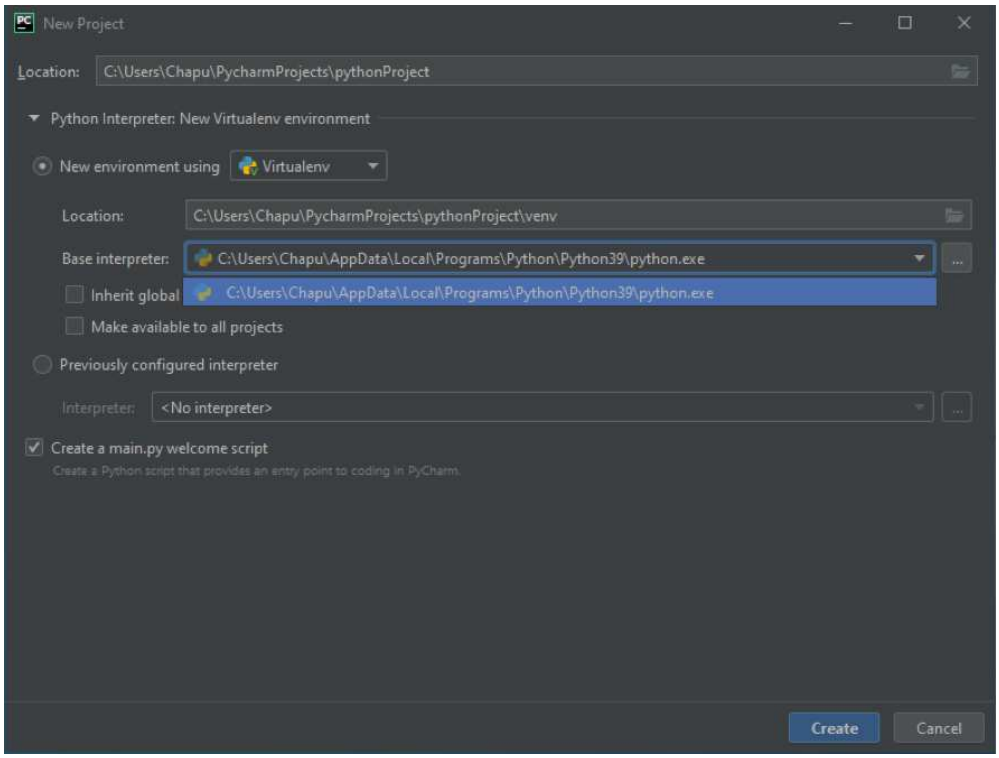


Figura 1.5: Configuración de un proyecto en PyCharm

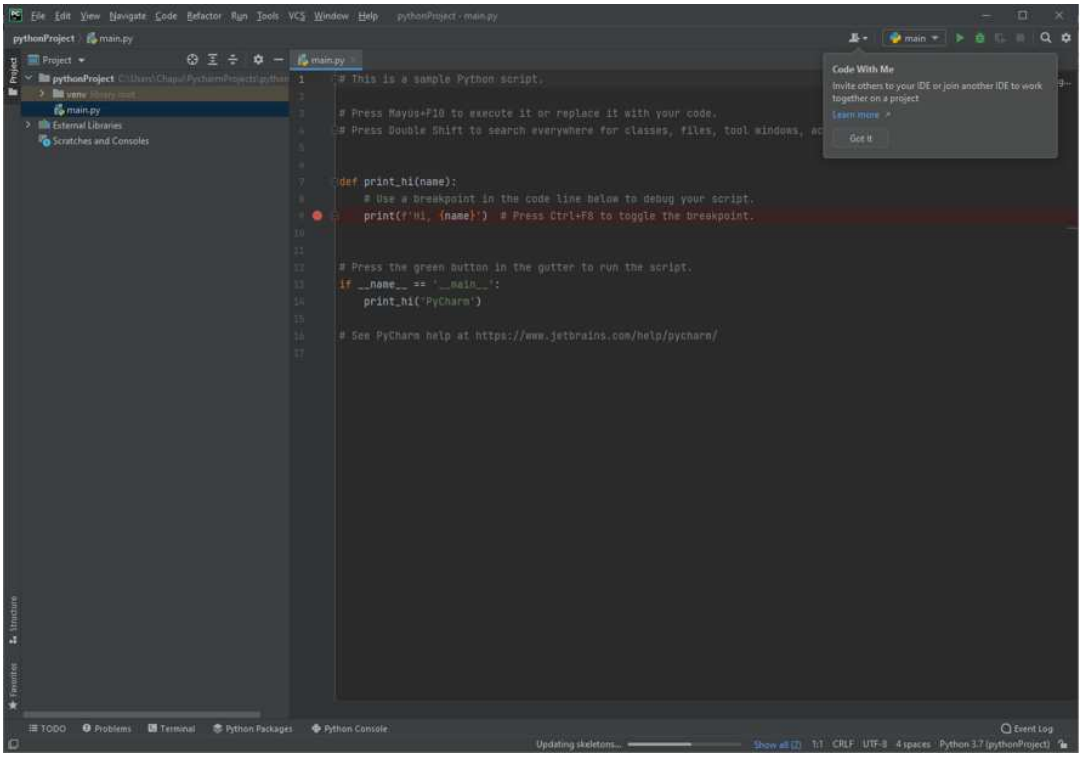


Figura 1.6: Ventana principal de PyCharm

2

FUNDAMENTOS DE PYTHON

Con el entorno de desarrollo ya configurado, y partiendo del proyecto *helloWorld* creado al final del capítulo anterior, podemos empezar a estudiar el funcionamiento del lenguaje. No es nada original el comienzo que proponemos, porque consiste en el típico programa que imprime *¡Hola mundo!* Algunos manuales de Python introducen innumerables características del lenguaje basándose en el uso interactivo del intérprete, antes de dar la pauta sobre cómo escribir un programa ejecutable. No parece buena idea para estudiantes de Informática que necesitarán escribir aplicaciones desde el comienzo. Así que seguiremos el enfoque clásico.

Pero antes de comenzar, es necesaria una pequeña aclaración sobre la terminología de Python: llamaremos *módulo* a un fichero fuente Python con extensión `.py`. Este módulo también tendrá un fichero asociado con extensión `.pyc` una vez que haya sido compilado. Cuando un programa empiece a tener cierto tamaño, será conveniente dividir su código en varios módulos, pero veremos que existe una forma sencilla de reutilizar las funciones de unos módulos en otros. Por otra parte, llamaremos *paquete* a un conjunto de módulos con alguna funcionalidad común. Un paquete es lo mismo que una librería en otros lenguajes de programación y, de hecho, emplearemos indistintamente ambos términos.

2.1. Un programa básico

Un programa básico de Python necesita, al menos, un módulo. Esto es de una lógica aplastante. Se has creado un proyecto marcando la opción *Create a main.py*, ya tienes a tu disposición un módulo. Si no, los pasos que necesitamos dar desde PyCharm para crear un nuevo módulo son los siguientes:

1. Selecciona el proyecto en el que quieres crear el nuevo módulo. Supongamos que es *helloWorld*.

2. En el menú *File*, usa la opción *New* y la subopción *Python File*.

3. Indica el nombre del módulo, que puede ser *main*.

4. PyCharm crea un archivo *main.py* dentro del proyecto.

Supongamos que alguien nos dicta este contenido para el módulo principal:

```
1 '''
2 Created on 31 ago. 2021
3
4 @author: jmjuarez
5 '''
6
7 if __name__ == '__main__':
8     print('¡Hola Mundo!')
```

2.1.1. Comentarios en Python

Hay unas cuantas cosas raras en el fichero. Para empezar, que un tal *jmjuarez* esté el 31 de agosto escribiendo un manual de Python. Pero vayamos a lo importante. En primer lugar, entre las líneas 1 y 5 aparece un comentario limitado por las dos marcas `'''`. Es un comentario que puede ocupar varias líneas y sirve para documentar el código. Alternativamente se podría haber usado como limitador del comentario `"""`. Trataremos la documentación de código Python más adelante.

Otra forma de poner comentarios en un programa Python es usando la marca `#`. Permite generar comentarios de una línea, o más concretamente, desde la marca hasta el final de la línea.

2.1.2. Punto de entrada de un módulo

Lo siguiente que debe resultar extraño es la línea 7 del código anterior. Para ir aclarando cosas, este código no tiene ninguna función. En Python puedes escribir código fuera de funciones. Esta es una característica de muchos lenguajes interpretados. Esto significa que, al ejecutar el módulo anterior, se alcanza directamente la línea 7, que contiene un `if`. Vamos posponiendo cosas sin parar, pero una vez más, queda para un apartado posterior la explicación de las sentencias de control.

Hay que volver a la idea de módulo para explicar bien la funcionalidad de esa línea. En Python, cualquier módulo puede ser un punto de entrada al programa, es decir, cualquier módulo puede tener el código *principal* del programa. Esto no sucede con C o C++. De hecho, si pusiésemos la función *main()* en todos los ficheros de un proyecto en C, el compilador se quejaría diciendo que la función está repetida por todos sitios.

La explicación de por qué en Python podemos tener múltiples puntos de entrada es sencilla: el lenguaje se pensó para desarrollar *scripts*, es decir, pequeños programas autocontenidos en un módulo. Pero tranquilo, que se puede reutilizar el código de un módulo en otro. Una ventaja de esta característica del lenguaje es que facilita el desarrollo de aplicaciones basado en pruebas. En cada módulo, puedo añadir un código de prueba de las funciones de ese módulo, y podré verificar si todo va bien seleccionando el módulo y ejecutándolo en PyCharm. Cuando haya hecho todas las verificaciones oportunas, podré pasar a usar el módulo importando sus funciones en otro módulo, y así sucesivamente. Siempre habrá, claro está, un módulo final con el programa principal general que servirá para lanzar toda la aplicación.

Y por fin, la explicación de la misteriosa línea 7: este doble uso de los módulos de Python se puede distinguir en tiempo de ejecución, usando la variable predefinida `__name__`. Esta variable toma el valor `__main__` cuando un módulo se ejecuta como punto de entrada principal. En PyCharm lo puedes hacer seleccionando el módulo con el botón derecho y pulsando la opción *Run*; cuando un módulo se importa desde otro módulo para hacer llamadas a sus funciones, esta línea 7 también se ejecuta, pero la variable `__name__` toma como valor el nombre del módulo importado, de forma que no se ejecuta nada en este caso.

Para terminar esta explicación del primer programa, en la línea 8 tenemos una sentencia para imprimir `¡Hola Mundo!` con una llamada a la función `print`. Observa que es `print` y no `printf`.

2.1.3. Indentación

Queda una cosa más, de las raras, pendiente de comentar, que en un programa tan pequeño como `hello.py` no llama la atención, pero en uno grande sí lo haría: el hecho de tener que poner la llamada a la función `print` donde está, después de un *tabulador*[1] al comienzo de la línea 8.

Python no usa palabras clave como `begin` y `end` de Pascal, o signos como `{` y `}` de C y Java, para delimitar bloques de sentencias. ¡Python te obliga a que los delimites con tabuladores! No es válido un programa Python que no respete esa forma de escritura. Esto tiene una ventaja: el código resulta mucho más legible, y eso es algo que puede simplificar mucho la vida de los profesores que tienen que revisar las prácticas.

Por cierto, la palabra *indentación* que introduce esta sección no existe en español, la RAE te lo puede certificar. En su lugar se usa *sangrado*, pero dada la extensión del uso de *indentación*, y teniendo en cuenta que no queremos hacer un manual de Python excesivamente cruento[2], nos permitimos la licencia de desobedecer a los reales académicos.

En fin, nadie dijo que los comienzos fuesen fáciles. Pero ahora podemos seguir un camino algo más sistemático para hablar del lenguaje. El resto del capítulo lo dedicamos a esas cosas que tanto nos gusta estudiar cuando aprendemos un lenguaje nuevo: tipos de dato básicos, uso de variables, operadores y sus precedencias, estructuras de control, etcétera. Puedes ir probando todos los ejemplos de los siguientes apartados en el módulo `hello.py`, pero ¡acuérdate de los tabuladores!

2.2. Tipos básicos

2.2.1. Enteros

Existen dos tipos de enteros en Python: `int` y `long`. Los primeros tienen una precisión de 32 bits (si el sistema es de 32 bits) o 64 bits (si es de 64). Los segundos, tienen precisión ilimitada, es decir, el único límite es la memoria de tu equipo. Por ejemplo, no hay problema en escribir:

```
1 print(9879879879879879879879879879879879879+1)
```

Así que ya puedes programar esos cálculos astronómicos para los que no habías encontrado un lenguaje adecuado. No será necesario indicar si tu entero es `int` o `long`. Dependiendo de su

[1] En realidad, el tabulador se convierte en 4 espacios en blanco en un editor como PyCharm.

[2] Ahora se entiende bien aquello de que *la letra con sangre entra*, que en estos apuntes podríamos reescribir como *Python con sangre entra*.

tamaño, Python lo tratará de una forma u otra de manera implícita.

Puedes especificar un signo al comienzo del entero, ¡incluido el positivo! También puedes usar una notación distinta de la base 10, añadiendo delante del entero el prefijo que denota la base, como `0b` para notación binaria, `0o` para octal y `0x` para hexadecimal. Las letras b, o y x pueden estar en mayúsculas.

```
print(0b101)
print(0o101)
print(0x101)
```

2.2.2. Reales

Puedes usar valores reales (flotantes) usando la notación clásica (parte entera, separador . y parte decimal), o bien la notación científica con el exponente indicado tras el caracter `e`:

```
print(-3.1416)
print(44.10e+3)
print(7.5e-4)
```

La notación científica es la misma que usa Java, y está definida en el estándar *IEEE 754*.

2.2.3. Complejos

Python es un lenguaje muy apreciado por los programadores que hacen cálculo científico. Y es perfectamente comprensible, porque además de poder hacer cálculos con enteros largos y reales con alta precisión, el lenguaje incluye como tipo básico a los números complejos, usando la notación *<parte real>+<parte imaginaria>j*. En Python ganaron los físicos a los matemáticos en la notación de los complejos. Por ejemplo:

```
print((1+3j)+(2+5j))
```

Podemos recuperar la parte real y la parte imaginaria del número complejo:

```
z = 1+3j
print(z.real) # 1
print(z.imag) # 3
```

2.2.4. Cadenas de caracteres

Python tiene múltiples formas de representar las cadenas de caracteres. Habrás observado que, en el ejemplo de `hello.py`, la cadena usada es `'¡Hola Mundo!'`. Igualmente se podría haber empleado `"¡Hola Mundo!"`. Lo importante es que, si empiezas la cadena con una comilla simple, debes terminarla con una comilla simple, y de forma similar con las comillas dobles.

En realidad, hay una diferencia curiosa entre usar una comilla simple o doble para delimitar una cadena: si quieres que tu cadena incluya caracteres de comillas simples en su contenido, no hay problema, usas comillas dobles al comienzo y final, y viceversa:

```
print("Esto es 'curioso', ¿no?")
print('Vaya "rarezas" tiene Python')
```

Siempre es posible usar caracteres de escape dentro de las cadenas de caracteres, antecedidos con la contrabarra, como:

```
print("Esto es una cadena al \"estilo\" de C\n")
print("---\t---")
```

Observa que la función `print`, tal y como la usamos por ahora, introduce un salto de línea al final de la cadena aunque no lo indiquemos con `\n`. También es posible hacer que una cadena ocupe varias líneas usando tres comillas simples (o dobles) en los extremos:

```
print('''Ahora empezamos por aquí y
continuamos hasta aquí''')
```

Sí, eso mismo es lo que se usa en los comentarios multilínea para documentar el código, pero aquí el intérprete de Python comprenderá que estamos especificando la cadena que debe imprimir.

Para terminar este apartado, hay una notación adicional para representar cadenas de caracteres que va a ser interesante al definir expresiones regulares[3]: las cadenas *raw*. Podríamos traducirlo como cadenas *en bruto* o *crudas*. Una cadena de este tipo tiene que ir precedida de *r* o *R*, y especifica que los caracteres de escape no se procesan, es decir, que el carácter de contrabarra \ se deja intacto en la cadena. Por ejemplo:

```
print(r'\d\t\d')
```

2.2.5. Boolean

Python ofrece un tipo básico *boolean*[4] para representar los valores de verdad: `True` y `False`. Hay otros valores, como los enteros 1 y 0, que Python permite que se usen con el mismo sentido que `True` y `False`.

2.2.6. None

Y para completar los tipos básicos, Python incluye la posibilidad de representar la *ausencia de valor*. Puede ser útil cuando, en una función, no se devuelve nada porque no se cumplen las condiciones para hacer la operación que se requiera. El modo de representar esa ausencia de valor es `None`. Las comparaciones son odiosas, y entre lenguajes de programación más todavía, porque dan lugar a discusiones apasionadas pero, salvando las distancias, `None` vendría a ser algo parecido al `NULL` de C.

2.2.7. Conversiones de tipos

Muy frecuentemente nos encontraremos con la necesidad de convertir una cadena de caracteres a otro tipo de dato con el cual podamos hacer operaciones aritméticas. Python nos permite hacerlo con una serie de funciones integradas en el propio lenguaje (no hay que importar ninguna librería):

```
x = int("123") # Convierte una cadena en entero corto
y = float("1.2e-3") # Convierte una cadena en real
z = complex("1+2j") # Convierte una cadena en complejo
```

[3] Gran parte del trabajo de prácticas consistirá en el uso de una representación de patrones de cadenas de caracteres, expresados con una notación formal que se denomina *expresión regular*.

[4] Otra vez nos saltamos la RAE. En el ámbito informático, estamos acostumbrados.

Siempre podemos verificar el tipo de cualquier variable o valor con la función `type()`:

```
z = 1+2j
print(type(z)) # Indica <class 'complex'>
```

Las funciones de conversión de tipos que acabamos de ver (usando `int()` y similares) pueden emplearse pasando como argumento otros valores que no sean cadenas de caracteres. Hay que llevar cuidado con la posible pérdida de información. Por ejemplo:

```
y = 1.2e-3
x = int(y)
print(x) # Imprimirá 0
```

Los anteriores son ejemplos de conversiones *explícitas*. Sin que haga falta indicar nada, Python puede realizar conversiones *implícitas* de tipos. Por ejemplo:

```
x = 1.2e-3
y = 3
z = x + y # Se convierte el entero de y en un real antes de sumar
```

2.2.8. Comprobaciones de tipos

Al no tener declaración de variables con tipos, es posible que a una parte de nuestro código Python llege un valor que no tengamos ni idea del tipo que tiene. Si nuestro código tuviese que hacer algo en función del tipo del valor, necesitaríamos una forma de comprobar cuál es. Afortunadamente Python nos ofrece la función integrada en el lenguaje `isinstance()`. Tiene dos argumentos: un valor y un tipo; y devuelve `True` o `False`:

```
print(isinstance(34,int))
```

Con los tipos que hemos visto hasta ahora, el segundo argumento – que coincidiría con lo devuelto por la función `type()` – puede ser alguno de los siguientes:

Tipo	Descripción
`int`	Entero
`float`	Reales
`complex`	Complejos
`str`	Cadenas de caracteres
`bool`	Valores lógicos

2.3. Variables

En Python, las variables no tienen que ser declaradas antes de su uso, como sucede en muchos otros lenguajes de programación como Pascal, Java y C. Algunos opinan que esto puede dar lugar a errores en la programación difíciles de detectar. Todo depende del cuidado que se lleve al programar.

Las variables pueden tener nombres arbitrariamente largos (lo que la memoria de tu equipo y tu capacidad de comprensión te dejen). Pueden contener letras, números y el carácter _, con la restricción de que comiencen con letra o _. Las letras pueden ser mayúsculas y minúsculas, y Python distingue un uso u otro. Además, Python 3 permite que uses cualquier letra Unicode en

los nombres de las variables, incluyendo vocales acentuadas y ñ. ¡Se acabaron las excusas por las faltas de ortografía al usar términos del español como variables![5]

Python, como otros lenguajes, tiene *palabras reservadas* para especificar estructuras de control, operadores, funciones básicas y algunas constantes. En la versión 3 de Python se definen las siguientes 33 palabras reservadas, ordenadas alfabéticamente, siempre en minúscula, salvo tres que contienen una mayúscula:

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

Para empezar a usar una variable, basta con darle un valor. De hecho, la asignación de un valor a una variable por primera vez corresponde a su *definición*. Si no se hace esto primero, Python nos va a indicar que es erróneo el uso de la variable. La asignación en Python se indica con = entre la variable y el valor:

```
1 n = 10
2 print(n)
```

Python permite asignaciones múltiples de dos formas. Para dar el mismo valor a varias variables, podemos encadenar las asignaciones:

```
1 a = b = c = 10
```

Otra posibilidad es dar distinto valor a las variables. En este caso, se separan con comas a ambos lados de la asignación las variables y los valores:

```
1 a, b, c = 2, 4, 16
```

Los lenguajes que obligan a declarar las variables antes de su uso, imponen que las variables deban tener el mismo tipo de dato siempre[6]. Estos lenguajes se denominan *estáticamente tipados*. Python, por el contrario, no tiene esta restricción:

```
1 a = 23.5
2 print(a)
3 a = "Y ahora soy una 'cadena de caracteres'"
4 print(a)
```

2.4. Operadores

Una vez que hemos visto los valores literales y las variables, realizamos ahora un recorrido por los operadores principales del lenguaje que permiten la manipulación de esos elementos en

[5] No habría que abusar de los caracteres españoles si vas a compartir tu código a nivel mundial.

[6] Dejamos al margen el tipado dinámico que se logra con la herencia en los lenguajes orientados a objetos.

expresiones, agrupándolos en tres bloques: operadores aritméticos, operadores relacionales (comparaciones) y operadores lógicos[7].

2.4.1. Operadores aritméticos

La siguiente tabla muestra un resumen de los operadores aritméticos de Python, ordenados de menor a mayor precedencia. Cada sección horizontal representa la misma precedencia. Por ejemplo, la suma y la resta tienen igual precedencia.

Operador aritmético	Descripción
$x + y$	suma
$x - y$	resta
$x * y$	multiplicación
$x \,/\, y$	división
$x \,//\, y$	división entera
$x \,\%\, y$	módulo
$+x$	más unario
$-x$	menos unario
$x ** y$	potencia

Atención: el resultado de la división de enteros es siempre un número real, ¡incluso si el dividendo es divisible por el divisor! En cuanto a la división entera, si el resultado es positivo, se trunca la porción decimal, dejando sólo la entera. Pero si el resultado es negativo, el resultado se redondea al siguiente entero menor:

```
1 print(10/2) # El resultado es 5.0
2 print(10/4) # El resultado es 2.5
3 print(10//4) # El resultado es 2
4 print(-10//4) # El resultado es -3
```

La precedencia entre los operadores determina el orden en el que Python los ejecuta en caso de encontrar una expresión en la que hay varios. Por ejemplo:

```
1 print(10+2**3) # Primero se ejecuta 2**3 y el resultado se suma a 10
```

Todos los operadores aritméticos pueden combinarse con la asignación para acortar la notación cuando se usa la misma variable a los dos lados de la asignación:

```
1 x = 1
2 x += 1 # Equivalente a x = x + 1
3 y = 2
4 y /= 5 # Equivalente a y = y / 5
```

2.4.2. Operadores relacionales

Los operadores relacionales, también llamados *operadores de comparación*, nos permiten especificar condiciones booleanas que nos harán falta en sentencias de control de flujo. Todos estos operadores dan como resultado `True` o `False`. La siguiente tabla muestra un resumen de estos operadores en Python, que comparten la misma precedencia:

[7] Como *quien mucho abarca poco aprieta*, tenemos que dejarnos muchas cosas en el tintero. Por ejemplo, los operadores a nivel de bit.

Operador relacional	Descripción
$a == b$	igualdad
$a != b$	desigualdad
$a < b$	menor que
$a <= b$	menor o igual que
$a > b$	mayor que
$a >= b$	mayor o igual que

Atención: las comparaciones de números reales pueden dar lugar a resultados inesperados:

```
x = 1.1 + 2.2
print(x == 3.3) # Imprime False porque x contiene 3.3000000000000003
```

Esto tiene que ver con la representación interna en Python de los números reales. En este caso, lo más recomendable es hacer una comprobación de lo próximos que se encuentran los dos números comparados, mediante la función `abs()`, que devuelve el valor absoluto:

```
tolerancia = 1e-10
x = 1.1 + 2.2
print(abs(x - 3.3) < tolerancia) # Imprime True
```

Python permite comparaciones encadenadas, que implícitamente están usando una conjunción, evaluando las comparaciones de izquierda a derecha:

```
a = 1
b = 2
c = 2
print(a < b <= c) # True porque a<b y b<=c
```

2.4.3. Operadores lógicos

Llega el turno de tratar los operadores que nos permiten construir condiciones lógicas complejas: conjunción, disyunción y negación. Primero, una pequeña tabla para verlos de un vistazo, incluyendo su precedencia de menor a mayor:

Operador lógico	Descripción
x or y	Disyunción
x and y	Conjunción
not x	Negación

La interpretación de los operadores es inmediata cuando los operandos son de tipo booleano. Pero Python permite que usemos operandos que no son booleanos. En general, en el contexto de una expresión booleana, se considera que es falso:

- El valor booleano `False`.
- Cualquier valor de un tipo numérico básico que sea cero: 0, 0.0, 0+0j.
- Cualquier cadena vacía, como `''`, `""`, y con dos o tres comillas simples o dobles.
- El valor `None`.

Cualquier otro valor se considera verdadero en el contexto de una expresión booleana. Usando la jerga de programadores de Python o Javascript, usamos el término *falsy* para referirnos a un valor

falso distinto de `False`, mientras que el término *truthy* lo empleamos para los valores verdaderos distintos de `True`.

Y visto lo esencial sobre los operadores lógicos, añadimos algunas consideraciones peculiares de Python. En primer lugar, si se usan los operadores `and` y `or` con operandos que no son booleanos, el resultado no es `True` o `False`, sino:

x	x or y	x and y
si x es *truthy*	x	y
si x es *falsy*	y	x

Aquí tenemos un ejemplo de lo anterior:

```
a = 0
b = 1
c = 2
print(b or c) # 1
print(a or c) # 2
print(b and c) # 2
print(a and c) # 0
```

¿Qué utilidad podría tener esto? En el caso del operador de disyunción, puede servir para implementar una forma de asignar valores algo distinta a la habitual. Si en una asignación `y = x` el valor de `x` es *falsy*, podemos asignar a `y` un valor por defecto:

```
x = ""
y = x or "valor por defecto"
print(y)
```

Esto puede ser interesante para evitar errores en ciertos casos en los que no se puede manejar un valor *falsy*.

Por otro lado, como en la mayoría de lenguajes de programación, las condiciones lógicas complejas que usan conjunciones o disyunciones múltiples se implementan con una evaluación de *cortocircuito*. En el momento en que se verifique que la condición es *True* o *False*, se detiene la evaluación. Esto habrá que tenerlo en cuenta si se usan llamadas a funciones como operandos de la conjunción o disyunción múltiple, ya que la llamada podría no realizarse.

2.4.4. Precedencia entre distintos tipos de operador

¿Qué orden de evaluación emplea Python cuando combino operadores aritméticos, relacionales y lógicos? Al igual que dentro de un mismo tipo de operador existe una precedencia (salvo los relacionales, que tienen todos la misma), entre los distintos tipos también hay un orden de evaluación por defecto:

Tipo de operadores	Orden de evaluación
Operadores lógicos	menor precedencia
Operadores relacionales	↓
Operadores aritméticos	mayor precedencia

En caso de duda, siempre podemos recurrir a los paréntesis para ordenar la evaluación.

```
x = 1
y = False
```

```
z = True
print(x+2*4 < 0 or y and -z%1)
# Equivale a (((x+(2*4)) < 0) or (y and ((-z)%1)))
```

2.5. Estructuras de control

Y ahora que sabemos lo que Python puede aceptar como expresión lógica, podemos pasar a estudiar la forma de las sentencias que permiten representar estructuras de control: sentencias `if`, `while` y `for`.

2.5.1. Sentencia if

Ya hemos visto en la sección 2.1 el uso de un `if`. Esta estructura de control tiene la forma:

```
if <condición>:
    <sentencias_condición_verdadera>
```

donde `<condición>` es una expresión que se puede interpretar como *falsy* o *truthy*, y las sentencias que se ejecutan cuando se cumple la condición tienen todas ellas un nivel de tabulación adicional al del `if`.

Una variante de esta construcción, con un bloque `else`, tendría esta forma:

```
if <condición>:
    <sentencias_condición_verdadera>
else:
    <sentencias_condición_falsa>
```

Si has estudiado la ambigüedad típica de los lenguajes de programación tipo C con las sentencias `if-else`, comprenderás que en Python no hay ambigüedad posible porque la indentación marca explícitamente cuál es el `if` con el que está asociado cada `else`.

Y una tercera variante del `if` permitiría introducir condiciones alternativas con `elif`:

```
if <condición1>:
    <sentencias_condición1_verdadera>
elif <condición2>:
    <sentencias_condición1_falsa_y_condición2_verdadera>
        ...
else:
    <sentencias_todos_condiciones_falsas>
```

Esta tercera variante puede ser útil para implementar algo equivalente a una sentencia *switch-case* de C, porque lo creas o no, ¡en Python no existe! Una de las filosofías del lenguaje es no ofrecer varias formas de hacer lo mismo.

2.5.2. Sentencia while

Veamos ahora un primer tipo de sentencia de Python para implementar bucles, introducido por la palabra clave `while`:

```
while <condición>:
    <sentencias_mientras_condicion_verdadera>
```

No hay ninguna diferencia notable con respecto al modo de hacer algo similar en C. ¿Y si quiero implementar un *do-while*? Pues nos encontramos con el mismo problema que con las sentencias *switch-case*: no existe en Python. En este caso, la solución podría consistir en usar un esquema como el siguiente (atención, contiene material algo fuerte para puristas de la programación):

```
while True:
    <sentencias_mientras_condicion_verdadera>
    if not <condición>:
        break
```

Es decir, la condición del `while` es siempre cierta, de modo que el bloque de sentencias se ejecuta una vez al menos. La condición de salida *real* del bucle la ponemos al final del bloque de sentencias, con un `if` que, en caso de cumplirse, ejecuta una sentencia `break`. La condición de salida es la negación de la condición para continuar en el bucle, obviamente.

La sentencia `break`, al igual que en C, finaliza el bucle inmediatamente anterior que la contiene. Y para ser completos, indicamos también que Python incluye una sentencia `continue` para iniciar el siguiente ciclo del bucle que la contiene. Después de esta revisión obligada de aspectos polémicos, exigida por la visión distante que debe acompañar a la enseñanza universitaria, volvemos a un terreno más llevadero.

2.5.3. Sentencia for

Ahora sí que vamos a encontrar alguna diferencia con respecto a C. Estrictamente hablando, cualquier *for* de C se puede implementar con un `while`. Por esta razón, aplicando a rajatabla el criterio de no ofrecer dos formas de hacer lo mismo, las sentencias `for` de Python son otra cosa distinta. En realidad deberíamos llamarlas sentencias `for-in`. Esta construcción de Python nos permite hacer un recorrido por los elementos de un conjunto ordenados en una secuencia.

Supongamos que tengo una lista concreta de valores que quiero manipular por igual. Si la lista es corta, puedo expresarla entre corchetes, separando los valores con comas: $[v_1, v_2, \ldots, v_i]$. La sentencia `for-in` me permite definir una variable, con el nombre que prefiera, que va a tomar uno a uno el valor de los elementos de esta lista, en la secuencia en que han sido expresados en su definición. Dentro del bloque de sentencias del `for-in` puedo emplear la variable que actúa como índice:

```
for x in [5,'7',11,'13']:
    print(x,'es un número primo')
```

Vemos en el ejemplo que la lista puede contener valores con distinto tipo de dato. También podemos ver que la función `print` puede llamarse con una lista de argumentos separados por comas que, al mostrarlos por consola, se separan con un espacio en blanco.

Hay un tipo de dato básico que es una secuencia de valores por sí mismo: una cadena de caracteres. Así que, en Python resulta muy fácil recorrer uno por uno los caracteres de una cadena:

```
a = 0
for c in 'supercalifragilisticoespialidoso':
```

```
3     print(a,c)
4     a += 1
```

Una función muy útil para hacer recorridos por secuencias de enteros es `range()`:

- `range(stop)`: genera una lista de enteros desde 0 hasta `stop-1`.
- `range(start,stop,step)`: genera una lista desde `start` hasta `stop-1` saltando con intervalos dados por `step`. Si `step` no se indica, se entiende que el intervalo es 1. Podemos usar un valor negativo en `step` para generar listas de enteros decrecientes.

Un ejemplo de uso de `range()`:

```
1 for c in range(0,-10,-1):
2     print('c=',c)
```

2.6. Funciones

Las funciones de Python se introducen con la palabra clave `def` seguida del nombre de la función[8], los argumentos[9] entre paréntesis separados por comas, y seguidamente dos puntos. Tras la cabecera de la función se inicia su bloque de sentencias, usando la indentación correspondiente. Por ejemplo:

```
1 def f(a,b,c):
2     print('suma=',a+b+c)
```

Los argumentos de la función actúan como variables locales del bloque de la función. Igualmente, cualquier variable definida dentro de la función pertenece a un ámbito propio a la función. Por esta razón, se pueden ocultar variables con igual nombre definidas fuera de la función:

```
1 a = 0
2
3 def f():
4     a = 5
5     print('a',a)
6
7 if __name__ == '__main__':
8     f()
9     print('a',a)
```

Sí, estamos de acuerdo, la característica habitual de unos apuntes o un manual sobre un lenguaje es que los ejemplos de código son bastante absurdos. Pero al menos los de estos apuntes sirven para dejar clara la cuestión que se trata en cada momento. En el pequeño trozo anterior tenemos varias cuestiones interesantes. En primer lugar, podemos definir variables *globales* en un módulo, simplemente definiéndolas fuera de una función. En segundo lugar, una función tiene que estar definida antes de ser usada, de forma similar a las variables. Y en tercer lugar, y de esto era de lo que trataba el ejemplo fundamentalmente, la variable global `a` queda oculta dentro de la función `f()` por una variable local con el mismo nombre.

[8] Se suele indicar en minúsculas y con guiones bajos separando las palabras: `destruir_facultad_ahora()`.
[9] Indistintamente usamos los términos argumento y parámetro.

Entonces, ¿no hay forma de modificar una variable global desde dentro de una función en Python? La respuesta es que sí, pero requiere que le echemos una mano al compilador de Python para que pueda distinguir este caso. La forma de hacerlo es sencilla: en el bloque de la función hay que introducir una sentencia con la palabra clave `global` y la lista de variables globales a las que se quiere acceder dentro de la función, separadas por comas:

```
a = 0

def f():
    global a
    a = 5

if __name__ == '__main__':
    f()
    print('a',a)
```

El valor devuelto por una función como `f()` en el ejemplo anterior es `None`. Para poder devolver un resultado, es necesario emplear la palabra clave `return`, *nihil sub sole novum*, como dirían los clásicos. Como Python es muy liberal en cuestión de tipos, tienes manga ancha para devolver valores de tipos distintos en la misma función, aunque se recomienda un uso moderado:

```
def f(a):
    if a == 1:
        return 42
    elif a == 2:
        return 1-1j
    elif a == 3:
        return +1e-12
    else:
        return 'And now for something completely different!'

if __name__ == '__main__':
    print(f(1),f(2),f(3),f(4))
```

2.6.1. Argumentos con valores por defecto

Una función en cuya definición se indican los parámetros con sus nombres únicamente, requiere que las llamadas tengan exactamente el número de parámetros indicados en la definición. En otro caso, se produciría un error. Pero es muy habitual tener funciones con algún argumento que puede tomar un valor por defecto, de modo que el programador puede evitar pasar siempre ese parámetro a la función. Por ejemplo:

```
def f(a,b,c=0):
    return a+b+c

if __name__ == '__main__':
    print(f(1,2)) # El resultado es 3
    print(f(1,2,3)) # El resultado es 6
```

En Python, los argumentos con valores por defecto tienen que estar situados en la parte final de la lista de argumentos.

2.6.2. Invocación con parámetros nombrados

Python permite que los argumentos de una función no tengan que ser indicados en la llamada en el orden de su definición. Para ello basta con poner en la llamada el nombre del argumento correspondiente seguido de un igual y el valor:

```
1 def f(a,b,c):
2     return a*b+c
3
4 if __name__ == '__main__':
5     print(f(c=2,a=1,b=3)) # El resultado es 5
6     print(f(2,c=0,b=5)) # El resultado es 10
```

En el ejemplo anterior vemos que se puede llamar a una función pasando al comienzo argumentos sin nombre seguidos de argumentos con nombre. Los argumentos sin nombre se asignan a los parámetros siguiendo el orden de la definición. Obviamente, no se admiten llamadas que asignen más de un valor a un mismo argumento.

2.6.3. No hace falta predeclarar funciones

Supongamos un código así en C:

```
1 /* Predeclaro la función a() */
2 void a();
3 void b() {
4     // Hace algo y entonces llama a a()
5     a();
6 }
7 void a() {
8     // Otra cosa, quizás llamar a b()
9 }
```

El compilador de C necesita la línea 2 con la predeclaración de la función `a()`, porque en caso contrario fallaría la compilación en la línea 5. Sin embargo, Python no requiere la predeclaración:

```
1 def b():
2     // Hace algo y entonces llama a a()
3     a()
4 }
5 def a():
6     // Otra cosa, quizás llamar a b()
7 }
```

La razón es sencilla: Python no *resuelve* la localización de una función hasta el momento de hacer la invocación. Al llegar a la línea 3, el compilador de Python genera bytecodes que, en el momento de ejecutarse, tendrán que localizar dónde está la función `a()`, para saltar a su código.

2.7. Excepciones

Cuesta creerlo, pero es necesario aceptar que no somos perfectos. Solemos cometer un montón de errores mientras programamos, que dan lugar a problemas durante la ejecución. Además, la

ejecución también puede estar condicionada por los valores que proporciona el usuario, que siempre encontrará la forma de llevar al límite a nuestro programa. Por eso, no viene nada mal que el lenguaje nos ofrezca alguna herramienta para evitar que, cuando hay problemas, todo salte por los aires sin control. En Python esta herramienta son las *excepciones*.

Vamos a explicarlo con un tipo de error muy básico, pero conforme vayas programando código en Python te darás cuenta de que hay mucha variedad de errores posibles, y para todos ellos hay una forma de representarlos. El caso básico que nos servirá de ejemplo es una división por cero. Supongamos que tenemos un módulo `c.py` con el siguiente código:

```
1 def operación(x,y,z):
2     return x/y + z
3
4 if __name__ == '__main__':
5     res = operación(5,0,3)
6     print('Resultado =',res)
7     print('Finalizando')
```

Obviamente no deberíamos escribir un código que va a producir una división por cero tan evidente. Pero imagina que la llamada a `operación` se realiza con tres variables que proceden de una lectura de teclado. El caso es que, al ejecutar el programa anterior, tendremos un bonito mensaje en la consola:

```
1 Traceback (most recent call last):
2   File "/workspace/helloWord/c.py", line 5, in <module>
3     res = operación(5,0,3)
4   File "/workspace/helloWord/c.py", line 2, in operación
5     return x/y + z
6 ZeroDivisionError: division by zero
```

Nuestro programa finaliza descontroladamente, pero al menos hay información muy útil en el mensaje de la consola. Empezando por el final, está claro lo que ha pasado: `ZeroDivisionError: division by zero`. De esa línea nos interesa especialmente el comienzo: `ZeroDivisionError` es el nombre que Python le da a ese tipo de error, o excepción. Lo usaremos en el código que veremos enseguida para evitar que todo se descontrole. Será importante si estamos implementando el controlador de un reactor nuclear.

Las restantes líneas indican la secuencia de llamadas que ha provocado el problema. En la línea 5 de `c.py` del código de primer nivel se invoca `operación` con los argumentos correspondientes; dentro de la función `operación`, en la línea 2 del mismo módulo, se encuentra la división que lanza la excepción.

2.7.1. Bloque try-except

Para tener un control sobre los posibles errores en tiempo de ejecución de un programa, y evitar que provoquen la finalización abrupta del mismo, Python nos ofrece una sentencia que funciona, básicamente, con dos palabras clave, `try` y `except`, que introducen dos bloques de código. El bloque de `try` contiene el código en el que, potencialmente, puede producirse la excepción, mientras que el bloque de `except` incluye el código que *maneja* la excepción. Por ejemplo:

```
1 if __name__ == '__main__':
2     try:
```

```
3         res = operación(5,0,3)
4         print('Resultado =',res)
5     except ZeroDivisionError:
6         print('Problemas con una división por cero')
7     print('Finalizando')
```

Detrás de la palabra clave `except` podríamos haber dejado los dos puntos directamente, sin indicar el nombre de la excepción que queremos capturar. Eso nos permitiría escribir un manejador de errores genérico. Sin embargo, conviene indicar expresamente cuál es la excepción que queremos capturar. En este caso, el tratamiento del error no es ninguna maravilla: un mensaje de error al usuario. Algo más conveniente sería indicarle que reintroduzca los datos para hacer otro intento. Si ejecutas el código anterior, podrás observar que no se llega al `print` de la línea 4. Es normal: si se produce la excepción en una sentencia del bloque del `try`, deja de ejecutarse ese bloque y se pasa a ejecutar el bloque del `except` que especifica la excepción que se ha producido.

2.7.2. Bloque try-except-else-finally

El bloque de tratamiento de excepciones que hemos visto en la sección anterior es un caso de la estructura más completa que podemos usar en Python, y que seguiría la siguiente plantilla:

```
1  try:
2      # Bloque que puede lanzar excepciones
3  except Excepción1:
4      # Tratamiento de las excepciones tipo Excepción1
5  except (Excepción2,Excepción3):
6      # Tratamiento de las excepciones tipo Excepción2 y Excepción3
7  ...
8  except:
9      # Tratamiento de otras excepciones
10 else:
11     # Bloque que se ejecuta si no ha habido excepciones
12 finally:
13     # Bloque que se ejecuta en cualquier caso al final
```

Como ves, si el tratamiento de varias excepciones va a ser igual, podemos indicarlo listándolas entre paréntesis en un único `except`.

2.7.3. Aserciones

Una forma de depurar programas Python es usar la sentencia `assert`, que permite generar una excepción cuando no se cumple alguna condición necesaria para la correcta ejecución de una porción de código. Por ejemplo:

```
1  def operación(x,y,z):
2      assert y!=0,'Parámetro y es 0'
3      return x/y + z
4
5  if __name__ == '__main__':
6      try:
7          res = operación(5,0,3)
8          print('Resultado =',res)
9      except AssertionError as error:
```

```
10          print('No se cumple una aserción:', error)
11      print('Finalizando')
```

La sentencia `assert` requiere una condición, que se evalúa para comprobar si es `False`. En este caso, se genera una excepción de tipo `AssertionError` que contiene el dato que se especifica a continuación de la coma. Esa excepción puede ser capturada y manipulada convenientemente dentro de una estructura `try-except` del programa. Observa la línea 9 del código anterior. La línea `except` termina con `as error`. La palabra clave `as` permite darle un nombre a la instancia de la excepción `AssertionError` que se captura en esta línea, y más concretamente el nombre es `error`. Dentro del bloque que introduce este `except` se puede usar esta variable para hacer referencia a la excepción. Muy frecuentemente se imprimirá mediante una sentencia `print()`.

3

MÓDULOS Y PAQUETES

UNA de las claves del éxito de Python es la gran facilidad que ofrece el propio lenguaje para la organización del código. Esta organización se construye sobre el concepto de *módulo* que ya hemos introducido, y se extiende con el de *paquete*. La ventaja principal de esta estructura es la reusabilidad de código, que ha dado lugar a la disponibilidad de una gran cantidad de librerías de una comunidad de programadores de Python creciente. En este capítulo analizamos cómo se organiza el código de un proyecto de Python. Aunque las prácticas de la asignatura de Autómatas y Lenguajes Formales no requieran grandes proyectos, no deja de resultar útil conocer el funcionamiento del sistema de módulos y paquetes de Python dadas las ventajas que supone, incluso en proyectos pequeños, gracias a la utilización de librerías.

3.1. Dos formas de importar

Un programa Python consiste en un conjunto de ficheros de código (módulos) de entre los cuales hay uno que contiene el código principal. Para integrar los distintos ficheros entre sí, es necesario usar un mecanismo que permita *importar* las definiciones (funciones o variables) de un módulo en otro módulo. Empezamos considerando el caso más sencillo posible: tenemos dos módulos `a.py` y `b.py` que se encuentran en el mismo directorio, y queremos importar definiciones del segundo en el primero. Veamos un ejemplo de un módulo `b.py` que se va a importar.

```
1 # Modulo b.py. Define x y spam
2 x = 10
3 def spam(text):
4     print(text,'is not')
5     i = 0
6     while i<x:
7         print('spam')
8         i += 1
```

3.1.1. Import

En Python, podemos realizar la importación de definiciones de un módulo empleando las sentencias del lenguaje `import` y `from`. La primera opción, `import`, permite que *todas* las definiciones de un módulo se incorporen a un segundo módulo, que es precisamente el que contiene la sentencia `import`. Por ejemplo, el módulo `a.py` que realiza esta importación puede hacerlo así:[1]:

```
# Modulo a.py
import b # Importamos todas las definiciones de b.py

if __name__ == '__main__':
    b.x = 5
    b.spam('zarangollo')
```

Observa que las definiciones importadas desde `b.py` con `import` se tienen que usar en `a.py` precediendo el identificador con el prefijo `b.`, es decir, el nombre del módulo importado sin la extensión `py`. A esto se le llama *espacio de nombres*, y te resultará familiar de otros lenguajes de programación. Por ejemplo, si en C declaramos una variable cuyo tipo es una estructura, para acceder a los campos de la estructura tenemos que usar el nombre de la variable como espacio de nombres. Con los módulos de Python es similar.

El hecho de que el nombre de un módulo se use como parte del identificador de una definición al usar este tipo de importación, tiene una implicación: los nombres de los ficheros Python deben cumplir las mismas restricciones que los nombres de las variables.

Si no te gusta el nombre del módulo, por alguna razón, puedes indicar un nombre distinto en el momento de hacer la importación, ampliando la llamada a `import` con `as nombre`:

```
import dijkstra as d # d.x para acceder al identificador x de dijsktra
```

3.1.2. From - import

La segunda opción de importación es `from`. En este caso, podemos especificar qué definiciones queremos importar. Por ejemplo, si quisiéramos usar sólo la función `spam`[2] de `b.py`:

```
# Modulo a.py
from b import spam

if __name__ == '__main__':
    spam('zarangollo')
```

Si fuesen varias las definiciones importadas, se indicarían separadas por comas. Si se quieren importar todas, se usa un asterisco `*`. Observa que, en esta segunda forma de importar, ya no es necesario el uso del nombre del módulo `b` como espacio de nombres de las definiciones importadas. Estas quedan fusionadas en el espacio de nombres por defecto del módulo que recibe la importación. Esto tiene una primera implicación: si importamos desde varios módulos una definición que tiene el mismo nombre, no podremos usar `from`, sino que tendremos que emplear `import` para

[1] Como ves, no es necesaria la condición `if __name__` en el módulo `b.py`. Estrictamente hablando, tampoco lo es la de `a.py`, pero conviene acostumbrarse a usarla, ya que podría darse el caso de que `a.py` pasara a ser importado desde otro fichero `c.py` que tuviese el código principal.

[2] En los libros y webs sobre Python verás que es frecuente usar ejemplos con *spam*. Tiene su explicación: la palabra adquirió su significado actual por un famoso sketch de los Monty Python que puedes encontrar en Youtube.

poder distinguir una de otra. Además, al usar `from`, si tuviésemos una definición con el mismo nombre en el módulo que importa y en el módulo importado, podrían ocultarse una y otra:

```
# Modulo a.py
x = 5
from b import *

if __name__ == '__main__':
    print('x =',x) # Imprime 10 en lugar de 5
```

3.1.3. Cómo funciona la importación

Un error frecuente de los programadores de C al llegar a Python es considerar que `import` y `from` funcionan de forma similar a `#include`. Pero las diferencias son notables: `#include` es una directiva de *precompilación*, que sirve para pegar el contenido de un fichero (normalmente un fichero de cabecera `.h`) en otro fichero. Sin embargo, `import` y `from` son *sentencias* de Python, es decir, operaciones básicas incluidas en el propio lenguaje. Esto supone que pueden aparecer en cualquier sitio en el que sea aceptable una sentencia de Python, y se ejecutan en el momento en que la ejecución llega a ellas.

Además, las dos sentencias de importación de Python realizan tres pasos:

1. Encontrar el módulo que se quiere importar.
2. Compilar a bytecodes el módulo importado, si fuese necesario.
3. Ejecutar el código del módulo importado.

Vamos a analizar brevemente cada uno de estos tres pasos.

3.1.3.1. Encontrar el módulo

El ejemplo sencillo de la sección anterior es un caso muy frecuente: el módulo importado está en el mismo directorio que el módulo que realiza la importación. No obstante, podríamos tener una panoplia de situaciones muy variada, en función de la organización del código de nuestro proyecto. Por esta razón, Python define unas reglas de búsqueda para la importación de módulos. El módulo importado se busca sucesivamente en las siguientes localizaciones, por orden:

1. Primero se intenta localizar en el mismo directorio del módulo que realiza la importación. Al usar PyCharm, el directorio es el que lleva el nombre del proyecto y se localiza dentro de la carpeta de `workspace`.
2. Si lo anterior falla, se consulta el contenido de la variable `PYTHONPATH`.
3. Si todavía no se ha localizado el módulo, se intenta con los directorios que contienen las librerías estándares[3] de Python en el sistema.
4. Por último, Python intenta localizar el módulo en el directorio de librerías adicionales, instaladas con herramientas como `pip` que luego describimos. El subdirectorio último de esta ruta suele denominarse *site-packages*.

[3]Las librerías estándares son las que están en cualquier instalación de Python, sin necesidad de añadir nada adicional aparte del propio sistema de Python. Permiten hacer muchas operaciones que en otros lenguajes requerirían la instalación de librerías adicionales. Puedes ver un listado completo para Python 3 aquí: `https://docs.python.org/3/library/`.

Existe alguna posibilidad adicional después de las anteriores para hacer otro intento de localización, pero no se van a tratar aquí. ¿Y si no se encuentra el módulo? Pues se produce una excepción de tipo `ImportError`.

3.1.3.2. Compilación a bytecodes

Después de localizar el fichero de código fuente (con extensión `.py`) del módulo que hay que importar, Python compila el código a bytecodes si es necesario. Para determinar si es necesario, se hace una comparación del instante de última modificación del fichero fuente con el del fichero compilado. Si el primero es posterior al segundo, se realiza la compilación del módulo importado. El resultado es un fichero con extensión `.pyc` que queda almacenado en el subdirectorio `__pycache__` dentro del directorio del proyecto.

Ten en cuenta que la compilación sólo se realiza cuando se importa un módulo. Por esta razón, normalmente no verás un fichero `.pyc` para el módulo principal, a menos que sea importado en otro sitio. ¿Significa esto que el módulo principal no se compila? No, también se compila, pero se hace cada vez que se ejecuta y sus bytecodes no se guardan en ningún sitio, sino que son desechados al finalizar el programa. Conviene que el módulo principal sea lo más pequeño posible para acelerar la ejecución del programa.

3.1.3.3. Ejecución del módulo importado

Hay un último paso en la importación de un módulo: su ejecución. Aquí tienes una buena razón para entender que la importación de módulos de Python no tiene mucho que ver con la directiva de precompilación `#include` de C.

Podemos distinguir dos tipos de código dentro de un módulo: definiciones de funciones y código de *primer nivel*. Las funciones se declaran, como ya hemos visto, con la palabra clave `def`. En cierta forma podemos considerar que estas declaraciones se *ejecutan* al cargar un módulo en el sentido de que las funciones quedan disponibles para su uso en el módulo que realiza la importación. En cuanto al código de *primer nivel*, se trata de código Python del módulo importado que no forma parte de la definición de funciones. Podríamos considerar que esto de *primer nivel* tiene que ver con el nivel de indentación inicial. Por ejemplo, la inicialización de variables o la típica sentencia `if __name__ == __main__:` etc., es un caso de código de *primer nivel* de un módulo. Este código se ejecuta al importar el módulo, indistintamente de si se usa `import` o `from`.

3.1.3.4. Algunas consideraciones más

Como `import` y `from` son sentencias, pueden estar dentro de otras sentencias condicionales. Esto implica que los módulos se puede importar en función de las condiciones que se produzcan durante la ejecución del programa. Por otra parte, es posible realizar varias llamadas de importación de un módulo en el mismo programa. Sólo la primera tendrá como consecuencia la ejecución de los tres pasos que acabamos de ver. Las restantes llamadas de importación reutilizarán el módulo que ya está en memoria.

Cuidado con los nombres de las variables y funciones definidas en un módulo. Si son demasiado generales, como `x` o `f`, es posible que tengas montado un buen lío al importar varios módulos en otro si hay coincidencia de nombres y usas `from`.

3.2. Paquetes

Hasta ahora, sólo hemos visto cómo importar un módulo en otro. Posiblemente, tus primeros desarrollos en Python no necesiten mucho más. Pero, ¿para qué ponerse límites? En el momento en que tus programas Python vayan creciendo, posiblemente necesites una estructura de módulos ordenados en distintos directorios. Aquí es donde viene al rescate el concepto de *paquete* (también lo hemos llamado *librería* en varias ocasiones): un paquete Python no es más que un directorio con una serie de módulos Python, y opcionalmente con subdirectorios (subpaquetes) con más módulos. Sólo hace falta un pequeño detalle para que Python sepa que un directorio es un paquete: debe contener un fichero con el divertido nombre de `__init__.py`. Ese fichero puede estar vacío, aunque lo normal es que contenga lo siguiente:

- Código de inicialización del paquete. Por ejemplo, la definición de valores de variables por defecto o la apertura de algún recurso del sistema.
- Un listado de los módulos que se importan si se usa `*` en la importación del paquete en una sentencia `from`. La idea es que quizás no todos los módulos del paquete contienen funciones para ser usadas desde fuera, y sólo queremos que se ejecute la importación de algunos de ellos. El listado se especifica en una variable con el nombre `__all__`.

Supongamos que tenemos un paquete en un directorio llamado `ordenar`, dentro del cual hay un módulo por cada algoritmo de ordenación: `bubblesort.py`, `quicksort.py`, etc. Supongamos que todos estos módulos tienen un método `sort()` con un argumento que es una lista de enteros, y que el método imprime el resultado de la ordenación.

Dentro del fichero `__init__.py` del directorio `ordenar` podemos dar valor a `__all__` con la lista de módulos que se importan desde el paquete si se usa `from` con `*`:

```
1 # __init__.py del paquete ordenar
2 __all__ = ["bubblesort","quicksort","mergesort","cocktailsort"]
```

A nivel básico, la importación de definiciones de paquetes funciona como con los módulos invididuales, pero ahora usando el nombre del paquete (ruta de directorios) delante:

```
1 import ordenar.bubblesort
2 # Uso el método sort del módulo bubblesort
3 ordenar.bubblesort.sort([5,2,9,34,12])
```

El mecanismo de importación realiza la búsqueda del paquete siguiendo el mismo orden de localizaciones posibles que en el caso de los módulos individuales[4]. Como puedes ver en el código anterior, para invocar al método `sort` es necesario indicar delante el nombre completo del módulo, incluyendo el nombre del paquete delante. Por esta razón, puede ser conveniente usar `as` en las sentencias `import`:

```
1 import ordenar.bubblesort as bubble
2 # Uso el método sort del módulo bubblesort renombrado como bubble
3 bubble.sort([5,2,9,34,12])
```

También podemos realizar la importación de un paquete usando la sentencia `from`:

[4]Si se importa un paquete contenido dentro de otro paquete, lo importante es localizar al primero, el paquete de más alto nivel, ya que los subpaquetes son subdirectorios del primero.

```
from ordenar import bubblesort
# Uso el método sort del módulo bubblesort
bubblesort.sort([5,2,9,34,12])
```

Y si queremos afinar mucho, podemos indicar específicamente el método que queremos usar:

```
from ordenar.bubblesort import sort
# Uso el método sort del módulo bubblesort
sort([5,2,9,34,12])
```

Por el contrario, si no tenemos claro qué queremos usar de un paquete, siempre se puede usar una importación a lo salvaje:

```
from ordenar import *
# Uso el método sort del módulo mergesort
mergesort.sort([5,2,9,34,12])
```

Observa que, en este último caso, tienes que indicar el módulo delante del método `sort` que quieres usar, ya que se han importado todos los módulos indicados en la lista `__all__` o, en caso de que no esté definida, todos los módulos del paquete.

3.2.1. Creación de paquetes en PyCharm

En la sección 2.1 te indicamos cómo se puede crear un módulo dentro de PyCharm. Otra de las alternativas posibles que aparecen al seleccionar el proyecto y la opción *File* > *New* es crear un nuevo *Python Package*. Si elegimos esta opción, veremos que se crea un subdirectorio con el nombre del paquete. Ese subdirectorio se incluye en el directorio del proyecto y dentro del paquete se incluye automáticamente un archivo `__init__.py` vacío.

3.2.2. Gestión de paquetes con `pip`

Si todo fue correctamente durante la instalación de Python (sección 1.3.1), desde la línea de comandos de tu sistema tendrás la posibilidad de ejecutar el comando `pip` (podrías necesitar tener permisos de administrador para hacerlo). Es la herramienta básica para acceder al *Python Package Index* (PyPI), el repositorio de paquetes para el lenguaje Python que actualmente cuenta con casi 325000 proyectos[5].

Lo primero que puede resultar conveniente es actualizar la propia herramienta `pip` para estar seguros de que tenemos la última versión. Necesitamos usar un comando que es todo un homenaje a la recursión:

```
pip install --upgrade pip
```

Supongamos que, tras buscar un rato en Google, hemos localizado un paquete para reconocimiento de voz que nos interesa usar. El nombre del paquete es `SpeechRecognition`. Sabiendo el nombre del paquete, la forma de instalarlo es realmente sencilla:

```
pip install SpeechRecognition
```

[5]Puedes acceder a un buscador de paquetes del repositorio en `https://pypi.org/`.

Aparecerá una barra que muestra el progreso de la descarga, y en poco tiempo tendremos instalado el paquete en el equipo. Si eres un poco maniático y quieres tener un control exhaustivo de qué ficheros contiene el paquete que se ha instalado, y exactamente dónde se encuentran, no hay problema, se lo preguntamos a `pip`:

```
pip show --files SpeechRecognition
```

Para saber cuáles son los paquetes que tenemos instalados mediante `pip`, podemos solicitar un listado de la siguiente forma:

```
pip list
```

Junto a cada paquete aparece indicada la versión que tenemos instalada en el sistema. Si quisiéramos ver los paquetes que están desactualizados, podemos hacerlo así:

```
pip list --outdated
```

Y si encontramos un paquete desactualizado, podemos actualizarlo a su última versión así:

```
pip install --upgrade paquete
```

Desinstalar un paquete es igual de sencillo que instalarlo:

```
pip uninstall paquete
```

El comando `pip` tiene la deferencia de consultarnos si realmente queremos hacer la desinstalación. Si tienes curiosidad, puedes ver que `pip` se puede hacer el harakiri a sí mismo.

4

CADENAS

LAS cadenas (strings) en Python se usan para representar cualquier cosa que pueda ser codificada como texto o bytes. En la parte relativa al texto, las cadenas incluyen símbolos de cualquier alfabeto conocido, empleando para ello el sistema Unicode que permite la representación de todo tipo de caracteres de lenguajes humanos. Y por otro lado, una cadena puede almacenar bytes en crudo para representar contenidos binarios de cualquier tipo, como mensajes de red o ficheros multimedia.

Al comenzar a tratar las cadenas en profundidad, es necesario realizar una aclaración sobre la división que Python realiza entre los tipos de dato que se manejan desde el propio lenguaje: tipos de dato inmutables y mutables.

En Python se denomina *inmutable* a un tipo de dato cuyas instancias no se pueden modificar. Son inmutables los tipos básicos que se describieron en la sección 2.2, incluyendo las cadenas. Cualquier operación realizada sobre una instancia de estos tipos no altera la instancia, sino que genera una instancia nueva. En concreto, en el caso de las cadenas, son secuencias de caracteres o bytes con un orden en su posición de izquierda a derecha que no pueden ser modificadas. Por ejemplo, si se quiere concatenar una cadena con otra, se genera una tercera cadena con el resultado de la concatenación, pero no se alteran ninguna de las dos cadenas concatenadas. Sucede de forma similar con todas las operaciones realizables sobre las cadenas, que veremos en este capítulo. En el siguiente capítulo se describen los tipos mutables de listas y diccionarios.

Además de la creación de cadenas de caracteres que vimos en la sección 2.2, es posible crear cadenas binarias anteponiendo el prefijo `b` a la cadena. Por ejemplo, el siguiente código genera una cadena binaria de dos bytes, el primero con valor 0 y el segundo con valor 255:

```
1 a = b'\x00\xFF'
```

Cada byte se especifica usando el código de escape `\x`*hh*, donde *hh* son dos dígitos hexadecimales.

4.1. Operaciones básicas

Las siguientes operaciones se pueden aplicar sobre cadenas por el hecho de que son secuencias ordenadas. Más adelante veremos que también es posible hacerlo con otros tipos de dato que cumplen esta misma condición.

4.1.1. Concatenación

La concatenación de cadenas se realiza con el operador +. Por ejemplo:

```
1 s = 'abc' + 'def' # forma la cadena 'abcdef'
2 print(s)
```

Es importante tener en cuenta que Python no sobrecarga el operador + para concatenar cadenas con otros tipos de datos, como números por ejemplo. Si intentamos algo como `'abc'+5` se disparará una excepción de tipo `TypeError` en tiempo de ejecución, recordándonos que sólo se pueden concatenar cadenas con cadenas.

4.1.2. Repetición

En caso de que queramos repetir una cadena una determinada cantidad de veces, podemos usar el operador `*`, que opera una cadena con un entero:

```
1 s = 'Spam'*42 # repite 42 veces la cadena 'Spam'
2 print(s)
```

4.1.3. Longitud

La longitud de una secuencia se puede conocer mediante la función `len()`:

```
1 s = 'abc'
2 print(len(s)) # imprime 3
```

4.1.4. Indexación

Podemos acceder a los elementos de una cadena usando su posición mediante la operación de indexación, que tiene una sintaxis como la de C:

```
1 s = 'spam'
2 print(s[0],s[1],s[2],s[3])
```

La indexación especifica entre corchetes el desplazamiento desde el comienzo de la cadena, de modo que el primer elemento se indexa con el índice `0`. A diferencia de C, en Python podemos también acceder a los elementos de las secuencias usando un desplazamiento negativo. En este caso se entiende que este desplazamiento se resta a la longitud de la cadena para acceder al elemento:

```
1 s = 'spam'
2 print(s[-1],s[-2],s[-3],s[-4])
```

Si se intenta acceder a una posición de la secuencia que no existe, saltará una excepción de tipo `IndexError` en tiempo de ejecución, informándonos de un intento de acceso fuera de rango.

Para comprobar que una cadena es *inmutable*, podemos intentar hacer un acceso para modificar alguna de sus posiciones:

```
1 s = 'spam'
2 s[0] = 'S'
```

El resultado será otra excepción, en este caso de tipo `TypeError`, que nos indica que las cadenas no soportan la asignación de elementos.

4.1.5. Troceado

La operación de troceado básica permite extraer una sección contingua de elementos de la secuencia. Si `S` es una secuencia, el troceado se realiza con la sintaxis `S[i:j]`, donde `i` es el índice de la primera posición extraída y `j-1` el de la última posición extraída. Por ejemplo:

```
1 s = 'spam'
2 s = 'S'+s[1:4] # genera 'Spam'
```

Tanto el índice `i` como el `j` se pueden omitir en la operación de troceado. Si se omite `i`, se extrae desde la posición 0, mientras que si se omite `j`, se extrae hasta el final de la cadena. También es posible usar índices negativos en la operación de troceado.

Una versión extendida de la operación de troceado permite indicar un parámetro más: el *paso* o intervalo entre los elementos que se extraen. En la versión básica, este paso es +1. La versión extendida utiliza la sintaxis `S[i:j:k]`, donde `i` y `j` tienen la misma interpretación que en la versión básica, y `k` indica el paso entre los elementos extraídos. Por ejemplo:

```
1 s = 'abcdefghijklmnop'
2 print(s[1:10:2]) # imprime 'bdfhj'
```

Si el paso es negativo, la operación de troceado interpreta que los índices de comienzo y fin deben usarse de manera invertida:

```
1 s = 'abcdefghijklmnop'
2 print(s[5:1:-1]) # imprime 'fedc'
```

4.1.6. Conversión a cadena

En algunas ocasiones será necesario convertir un número u otro tipo de dato a cadena. Aunque en breve veremos una herramienta potente de Python para el formateo de cadenas, una primera alternativa para realizar la conversión a cadena es usar la función `str()`:

```
1 s = 'abc' + str(5) # forma la cadena 'abc5'
2 print(s)
```

4.1.7. Conversiones de caracteres

En Python no existe un tipo de dato para representar a un carácter. En su lugar se puede emplear una cadena de un único carácter. Para trabajar con el código numérico asociado a cada carácter, podemos emplear las funciones `ord()` y `chr()`. La primera recibe como argumento una cadena de un único carácter y devuelve su código numérico. La segunda realiza la operación contraria:

```
c = ord('ñ')
print(c) # imprime 241
s = chr(c)
print(s) # imprime ñ
```

4.1.8. Iteración con cadenas

Recuerda que, al tratar las sentencias `for-in` en la sección 2.5.3, la veíamos aplicada a una cadena para poder hacer una iteración entre sus caracteres:

```
a = 0
for c in 'supercalifragilisticoespialidoso':
        print(a,c)
        a += 1
```

4.1.9. Comprobación de subcadena

Suele resultar muy útil comprobar si una cadena es subcadena de otra. Y en general, resulta útil saber si una secuencia es subsecuencia de otra. Por esta razón, el lenguaje Python ofrece el operador `in` para realizar esta comprobación, devolviendo un valor `True` o `False`:

```
sub = 'fragil'
if sub in 'supercalifragilisticoespialidoso':
    print('La contiene')
```

4.2. Métodos de procesado de cadenas

Python 3.7 ofrece 45 métodos para realizar operaciones comunes con cadenas. Todos los métodos se invocan sobre la cadena en la que se quiere realizar la operación correspondiente. A continuación se describen los más importantes. Para una consulta más extensa, se puede acceder a la dirección `https://docs.python.org/3.7/library/stdtypes.html#string-methods`.

4.2.1. Métodos de búsqueda y sustitución

El modo más genérico de realizar búsquedas y sustituciones en cadenas es mediante el uso de expresiones regulares, que trataremos más adelante. Sin embargo, en algunos casos sencillos se pueden emplear algunos métodos de las cadenas.

En cuanto a los métodos de búsqueda, los más relevantes son:

- `str.find(sub[, start[, end]])`: devuelve la posición menor en la que se localiza la subcadena `sub` dentro de la cadena `str`, especificando opcionalmente la posición de comienzo `start` y de fin `end` del trozo de la cadena en el que se realiza la comprobación. Devuelve -1 si no se encuentra ninguna ocurrencia de la subcadena.

- `str.rfind(sub[, start[, end]])`: realiza la misma operación que `find` pero devuelve la posición mayor en la que se localiza la subcadena.
- `str.count(sub[, start[, end]])`: cuenta el número de ocurrencias de la subcadena `sub` en la cadena `str`, especificando opcionalmente la posición de comienzo `start` y de fin `end` del trozo de la cadena en el que se realiza el conteo.
- `str.startswith(prefix[, start[, end]])`: devuelve `True` si la cadena `str` comienza con el prefijo `prefix`, o `False` en caso contrario. Es posible especificar opcionalmente la posición de comienzo `start` y de fin `end` del trozo de la cadena en el que se realiza la comprobación.
- `str.endswith(suffix[, start[, end]])`: es similar al método anterior, pero sirve para realizar comprobaciones con sufijos de la cadena.

Para realizar sustituciones, podemos emplear los métodos:

- `str.replace(old, new[, count])`: devuelve una nueva cadena en la que las ocurrencias de la subcadena `old` en `str` se sustituyen por la subcadena `new`. Si se especifica el argumento `count`, éste indica el número máximo de ocurrencias que se sustituirán, empezando por la izquierda de la cadena `str`.
- `str.strip([chars])`: devuelve una nueva cadena en la que se eliminan los caracteres del comienzo y final de `str` que coincidan con los contenidos en la cadena `chars`. Si no se especifica `chars`, se eliminan los espacios en blanco al comienzo y final de `str`.
- `str.lstrip([chars])`: es una versión del método anterior en la que sólo se eliminan caracteres del comienzo.
- `str.rstrip([chars])`: es una versión del método `strip()` en la que sólo se eliminan caracteres del final.

4.2.2. Métodos de fragmentación

Para fragmentar una cadena en subcadenas se pueden emplear los métodos:

- `str.split(sep=None, maxsplit=-1)`: devuelve una lista[1] de subcadenas que resultan de fragmentar la cadena `str` usando la subcadena `sep` como separador. Si no se especifica `sep`, la separación se realiza en los espacios en blanco. El separador se elimina de las subcadenas fragmentadas. El argumento `maxsplit` permite indicar el número máximo de cortes que se llevarán a cabo en la cadena.
- `str.splitlines([keepends])`: realiza la misma operación que el método anterior, pero la fragmentación se realiza en los saltos de línea. Si se especifica `keepends` y su valor es `True`, entonces los saltos de línea se conservan en las subcadenas fragmentadas.

4.3. Formateo de cadenas

Hemos dejado en una sección aparte una de las herramientas más potentes incluidas en el lenguaje Python: el formateo de cadenas. Curiosamente, también es un caso en el que el zen de Python salta por los aires y deja de cumplirse clamorosamente: ¡hay dos modos de formatear cadenas! El

[1] En el siguiente capítulo se describe el funcionamiento de las listas de Python.

Código	Significado
s	Cadena (se aplica sobre cualquier tipo como una llamada a `str()`)
c	Carácter (`int` o cadena de un carácter)
d	Entero con signo en base 10
i	Igual que d (herencia de C)
u	Obsoleto: es equivalente a d
o	Entero con signo en base 8
x	Entero con signo en base 16
X	Igual que x pero con letras mayúsculas
e	Real con exponente en minúsculas
E	Real con exponente en mayúsculas
f	Real decimal (sin exponente)
F	Igual que f
g	Real escrito de la forma más compacta entre %e y %f
G	Real escrito de la forma más compacta entre %E y %F
%	Literal %

Tabla 4.1: Códigos para expresiones de formato de cadenas

modo más tradicional, con el cual hay miles y miles de líneas de código escritas, se basa en la filosofía de `printf` de C, y se denomina *expresiones de formateo de cadenas*. El segundo modo, aparecido en la versión 3.0 del lenguaje, sigue la corriente de C#/.NET y se apoya en el *método de formato de cadenas* `format()`, un método que se invoca sobre una cadena, de manera similar a los vistos en la sección anterior.

4.3.1. Expresiones de formateo de cadenas

Una expresión de formateo de cadenas tiene la sintaxis:

```
cadena_de_formato % (arg1,...,argn)
```

El operador `%` tiene a su izquierda una cadena de texto con el formato especificado en una notación similar a `printf`, incluyendo localizadores en los que se insertan los valores de una serie de argumentos, que aparecen en el lado derecho entre paréntesis y separados por comas. Los argumentos pueden ser de cualquier tipo. El resultado de la expresión de formateo es una nueva cadena en la que el formato queda aplicado a los argumentos. A diferencia de C, las expresiones de formateo de cadenas se pueden emplear en cualquier lugar en el que se puede usar una cadena, como puede ser el lado derecho de una asignación.

Las expresiones de formateo nos simplifican la vida en el sentido de que nos ahorran realizar un montón de operaciones de concatenado de cadenas y conversión de distintos tipos de dato a cadenas. Veamos un ejemplo sencillo:

```
1 n = 2
2 s = 'Hay %d maneras de %s una cadena en Python' % (n,'formatear')
3 print(s)
```

Los códigos usados para indicar el formato a la derecha de % aparecen indicados en la tabla 4.1. La especificación de los formatos puede precisarse algo más usando una indicación general que sigue la siguiente sintaxis:

`%[(clave)][flags][width][.precisión]código`.

El código es uno de los indicados en la tabla 4.1. Entre el carácter `%` y el código se pueden especificar algunos de los parámetros siguientes:

- Una `clave` para expresar qué valor se debe de sustituir en esta posición de formateo procedente de un diccionario[2].
- Una lista de `flags` o indicadores que expresan si se debe usar justificación a la izquierda (indicado con `-`), símbolo numérico (indicado con `+`), un espacio en blanco antes de un número positivo y un menos antes de un número negativo (expresado con un espacio en blanco), y un relleno de ceros (indicado con `0`).
- La longitud mínima total del texto que ocupa la posición de formateo.
- El número de dígitos de precisión después del punto decimal en los números reales.

Vamos a probar algunas de las posibilidades de las expresiones de formateo con unos ejemplos. Empezamos por el uso de enteros, con y sin indicación de tamaño mínimo, justificación a la izquierda y relleno de ceros:

```
1 n = 9876
2 s = '...%d...%-6d...%06d' % (n,n,n)
3 print(s) # Imprime '...9876...9876 ...009876'
```

Probamos ahora los códigos de formato para números reales:

```
1 x = 1.23456789
2 s = '%e | %f | %g' % (x,x,x)
3 print(s) # Imprime '1.234568e+00 | 1.234568 | 1.23457'
```

El uso de los parámetros de formateo da mucho juego cuando se emplean números reales:

```
1 x = 1.23456789
2 s = '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
3 print(s) # Imprime '1.23 | 01.23 | +001.2'
```

Por último, vamos a ver algún ejemplo del uso del parámetro *clave* en las expresiones de formato. Supongamos que disponemos de la información de una tabla de una base de datos que nos indica una cantidad de productos y una descripción de cada producto. Un registro de esta tabla se puede describir completamente como una estructura de información que especifica el valor de cada campo (cantidad y descripción). En Python se puede especificar este tipo de estructura usando un diccionario. Los diccionarios se especifican entre llaves de la siguiente forma:

```
1 dic = {'clave1' : valor1, ..., 'claveN' : valorN}
```

Podemos usar diccionarios en expresiones de formato especificando en las posiciones de formateo la clave del campo cuyo valor se sustituye en la posición. Por ejemplo:

```
1 s = '%(uds)d unidades de %(prod)s' % {'uds' : 5, 'prod' : 'spam'}
2 print(s) # Imprime '5 unidades de spam'
```

[2]Al igual que con las listas, el siguiente capítulo describe el uso de diccionarios en Python.

Esta utilidad puede resultar muy interesante en combinación con la función `vars()` de Python. Esta función devuelve un diccionario con las variables y valores definidos en el lugar en que se invoca. De este modo, podemos construir cadenas con los valores de las variables del ámbito en el que se encuentra la ejecución:

```
1 nombre = 'Mercedes'
2 grupo = 1
3 edad = 20
4 s = '%(nombre)s es profesora del grupo %(grupo)d' % vars()
5 print(s) # Imprime 'Mercedes es profesora del grupo 3'
```

4.3.2. Método de formateo de cadenas

No vamos a entrar en mucho detalle, porque con lo que hemos visto en el apartado anterior tenemos más que suficiente para las prácticas de Autómatas y Lenguajes Formales. Pero para completar la descripción de los dos modos de formatear cadenas en Python, vamos a describir brevemente el método `format()`. Se trata de un método que se aplica sobre cadenas. La cadena a la que se aplica va a contener el formato que queremos producir, especificando las posiciones de formateo de dos modos: con números o con nombres de parámetros. Los argumentos del método `format()` son los valores que se irán colocando en las posiciones de formateo. Un ejemplo del uso de números para las posiciones de formato:

```
1 plantilla = '{0}, {1} y {2}'
2 s = plantilla.format('spam', 'jamón', 'huevos')
3 print(s) # Imprime 'spam, jamón y huevos'
```

Y un ejemplo similar usando nombres en las posiciones de formateo:

```
1 plantilla = '{entrante}, {primero} y {segundo}'
2 s = plantilla.format(entrante='spam', primero='jamón', segundo='huevos')
3 print(s) # Imprime 'spam, jamón y huevos'
```

Esto es sólo una pequeña pincelada de la utilización del método `format()`. Te recomendamos que, si tienes interés, le eches un vistazo a la extensa documentación de Python sobre esta técnica de formateo: `https://docs.python.org/3.7/library/string.html#formatstrings`. ¡Buen provecho!

5

LISTAS, DICCIONARIOS, CONJUNTOS Y TUPLAS

SEGURO que tienes grandes recuerdos de los buenos ratos que has pasado programando tipos abstractos de datos en C. Los tipos abstractos de datos son fundamentales en el trabajo de los programadores, y uno respira con alivio cuando encuentra un lenguaje en el que generosamente se han preparado unos cuantos para usarlos con poco esfuerzo. Es el caso de Python, que nos ofrece una colección de tipos incluidos en el lenguaje realmente útiles para una gran cantidad de aplicaciones. Aunque hemos visto la mayoría de ellos en algunos ejemplos de páginas anteriores, vamos a analizarlos con más detenimiento a lo largo de este capítulo.

5.1. Listas

Las listas de Python son las secuencias ordenadas más flexibles del lenguaje. En el capítulo anterior se describieron las cadenas como ejemplo de secuencias, pero las listas van algo más allá en cuanto a flexibilidad: pueden contener instancias de cualquier tipo de dato, incluidas otras listas, y son mutables, es decir, podemos modificar cualquier posición de una lista, añadirle o borrarle elementos. Y lo mejor de todo, viniendo del mundo de C, es que no tenemos que preocuparnos por la memoria usada por la lista: Python se encarga de gestionarla, aumentando o reduciendo el espacio que haga falta.

5.1.1. Operaciones básicas

Vamos a empezar viendo la forma básica de crear listas:

```
1 lista_vacía = []
2 lista_no_vacía = [123, 'abc', 1.23, [1,2,3]]
```

Como puedes ver, en la sintaxis de Python relativa a las listas, se usan corchetes para denotar el comienzo y final de una lista, y en su interior se indican los elementos que contiene separándolos mediante comas.

Al tratarse de un tipo de secuencia, la longitud de una lista se puede consultar usando el método `len()`, y podemos manejar los operadores de concatenación `+` y repetición `*`:

```
print(len([1,2,3]))
l = [1,2,3]+[4,5,6] # Se genera la lista [1,2,3,4,5,6]
l = [1]*10 # Se genera la lista [1,1,1,1,1,1,1,1,1,1]
```

Recuerda que el operador de concatenación tiene que trabajar con operandos del mismo tipo. Por esta razón, puede resultar útil saber que es posible convertir una lista a una cadena y viceversa usando los métodos `str()` y `list()`:

```
s = str([1,2,3]) # Genera la cadena '[1,2,3]'
l = list('123') # Genera la lista ['1','2','3']
```

También podemos usar el operador `in` para verificar que una lista contiene un valor, o realizar una iteración entre sus elementos:

```
l = [1,2,3]
print(3 in l) # Imprime True
for x in l:
    print('Contiene',x)
```

La indexación de las listas funciona como la de las cadenas (también lanza `IndexError` si el índice de la indexación es incorrecto), al igual que el troceado:

```
l = ['um','Um','UM']
print(l[2]) # Imprime 'UM'
x = l[1:] # Extrae ['Um','UM']
```

Una forma común de representar matrices en Python es mediante listas anidadas. Por ejemplo, esto es una matriz 3x3:

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

Para acceder a las posiciones de la matriz, además de poder extraer la fila `i`-ésima mediante `m[i]`, también es posible extraer el elemento `i,j` de la matriz usando la idexación dos veces `m[i][j]`.

5.1.2. Modificación de listas

Tenemos dos modos de modificar las listas. El primero modo consiste en usar el operador de indexación o troceado a la izquierda de una asignación para modificar posiciones específicas de la lista. El segundo modo se basa en el empleo de métodos que se invocan sobre la lista y realizan modificaciones de la misma. Vamos a verlos por separado.

5.1.2.1. Modificación con indexación o troceado

Podemos modificar posiciones individuales o posiciones consecutivas de la lista empleando las operaciones de indexación y troceado:

```
c = ['zarangollo','spam','spam','paparajote']
c[1] = 'caldero'
c[1:3] = ['pisto','marinera']
```

Las operaciones anteriores alteran posiciones de la lista de acuerdo con los valores de los índices. También podemos realizar inserciones y eliminaciones de una lista usando la operación de troceado. Por ejemplo, podemos borrar sustituyendo por la lista vacía `[]`:

```
c = ['zarangollo','spam','spam','paparajote']
c[1:3] = [] # Elimina posiciones 1 y 2
```

Alternativamente podríamos usar el operador `del` para realizar esto mismo en posiciones individuales o con troceado:

```
c = ['zarangollo','spam','spam','paparajote']
del c[1] # Elimina la posición 1
del c[1:3] # Elimina las posiciones 1 y 2
```

También podemos insertar indicando una misma posición en los dos índices del troceado, y asignando una lista de elementos a introducir:

```
c = ['zarangollo','paparajote']
c[1:1] = ['pisto','caldero']
```

Para insertar una lista de valores `X` al principio de la lista `L` podríamos usar la asignación `L[:0]=X`. De forma similar, para concatenarla al final, podríamos usar `L[len(L):]=X`.

5.1.2.2. Modificación con métodos

Una segunda forma, quizás más explícita, de modificar una lista `L` consiste en realizar llamadas a métodos sobre dicha lista. Este es un primer grupo de métodos para alterar las posiciones de la lista:

- `L.insert(i,X)`: inserta el valor `X` en la posición `i` de la lista.
- `L.append(X)`: añade el valor `X` al final de la lista.
- `L.extend(L2)`: añade la lista de valores `L2` al final de la lista.
- `L.pop(i)`: elimina el valor que ocupa la posición `i`, y lo devuelve. Si no se indica el índice `i`, extrae el último elemento.
- `L.remove(X)`: elimina la primera ocurrencia del valor `X` en la lista. Atención, lanza una excepción `ValueError` si la lista no contiene el valor `X`.
- `L.clear()`: elimina todos los elementos de la lista.

5.1.3. Otras operaciones

Se pueden realizar otras operaciones sobre listas que pueden ser útiles:

- `L.index(X)`: devuelve la primera posición de la lista en la que aparece `X`; lanza `ValueError` si `X` no se encuentra en la lista.

- `L.count(X)`: devuelve la cantidad de veces que aparece `X` en la lista.
- `L.reverse()`: invierte la lista.
- `L.copy()`: genera una nueva lista que es copia de `L`.

Mención aparte merece el método `L.sort()`. Por defecto, este método ordena en orden ascendente los elementos de la lista. Si queremos que lo haga en orden descendente, podemos especificar el parámetro `reverse=True`:

```
l=[2,43,223,3,21,74,-1]
l.sort(reverse=True) # ordena en orden descendente
```

5.2. Diccionarios

Junto con las listas, los diccionarios son uno de los tipos incluidos en el lenguaje Python más útiles y flexibles. Un diccionario es una colección *no ordenada* – y, por tanto, no secuencial – de valores que se almacenan y recuperan mediante una *clave* en lugar de usar una posición. Los valores pueden ser de cualquier tipo de dato.

Mientras que una lista tiene la función de un array en otros lenguajes, los diccionarios son similares a los registros o estructuras. Están implementados para que las operaciones de consulta mediante las claves sean muy eficientes. Comparten con las listas el hecho de ser un tipo de dato *mutable*, y de ahorrar al programador el manejo de la memoria. Los diccionarios también reciben otras denominaciones, como *arrays asociativos* o *tablas de dispersión*.

5.2.1. Operaciones básicas

Empezamos, como en el caso de las listas, viendo cómo crear diccionarios de forma explícita. Si en las listas usábamos los corchetes como marcadores de comienzo y fin, en el caso de los diccionarios empleamos llaves. Además, cada valor aparece precedido de una clave y dos puntos:

```
X = {} # Diccionario vacío
D = { 'nombre':'Santiago Paredes', 'alias':'Chapu', 'dpto':'DIIC' }
```

La clave puede ser de cualquier tipo inmutable. En cuanto a los valores, no tienen ninguna limitación y, por ejemplo, podríamos encontrar diccionarios o listas dentro de diccionarios. Una segunda forma de crear diccionarios es mediante el método `dict()`, con dos variantes: especificando una lista de pares (nombre,valor), o bien una lista de argumentos nombrados:

```
D = dict([('nombre','Santiago Paredes'),('alias','Chapu'),('dpto','DIIC')])
D = dict(nombre='Santiago Paredes', alias='Chapu', dpto='DIIC')
```

La consulta de un diccionario se realiza con una sintaxis parecida a la de un array, pero en lugar de indicar la posición de la entrada que se quiere recuperar entre corchetes, se usa la clave asociada a la entrada. Por ejemplo, usando el diccionario `D` anterior:

```
print('%s es del departamento %s' % (D['alias'],D['dpto']))
```

Si la clave no existe en el diccionario, se lanza una excepción `KeyError`. Por tanto, puede resultar útil verificar si un diccionario contiene una entrada con una determinada clave antes de recuperarla, empleando el operador `in`:

```
if 'alias' in D:
    print(D['alias'])
```

Existe la alternativa con el método `get()` para recuperar un valor a partir de su clave, con la diferencia de que, en caso de que no exista la clave, no se lanza la excepción `KeyError` sino que se devuelve el valor `None`:

```
D = { 'uk':'Turing', 'eeuu':'Church', 'cz':'Gödel', 'de':'Hilbert'}
x = D.get('es')
print(x) # Imprime None
```

Podemos especificar un segundo parámetro del método `get()` a modo de valor por defecto que se devuelve en caso de que no se encuentre la clave en el diccionario:

```
x = D.get('es','Manolete')
print(x) # Imprime 'Manolete'
```

De forma similar al caso de las listas, se puede conocer la cantidad de entradas guardadas en un diccionario con el método `len()`. Pero ya hemos indicado que los diccionarios no son secuencias. Entonces, ¿no podemos hacer iteraciones con ellos? Sí podemos, gracias a dos métodos de los diccionarios que devuelven conjuntos para ser usados en la sentencia `for-in`. Se trata de los métodos `keys()`, que permite iterar sobre el conjunto de claves, y `values()`, que permite hacer lo mismo sobre los valores. Por ejemplo:

```
for k in D.keys():
    print('La clave %s tiene valor %s' % (k,D[k]))
```

Algo un poco raro de los métodos `keys()` y `values()` es que no devuelven listas sino, como se ha indicado, conjuntos. Además son conjuntos que están vinculados al diccionario de modo que si el diccionario se modifica, también lo hacen los conjuntos de claves o valores que hayamos obtenido[1]. Trataremos el tipo de dato conjunto en la sección 5.3.

También podemos realizar una copia de un diccionario usando el metodo `copy()`.

5.2.1.1. Modificación directa

El modo más sencillo para añadir o modificar una entrada del diccionario es accediendo a su clave a la izquierda de una asignación:

```
D = {}
D['nombre'] = 'Eduardo Martínez' # Añade una entrada
D['nombre'] = 'Pepe Juárez' # Modifica el valor de una entrada
```

Podemos eliminar una entrada del diccionario con el operador `del`:

```
D = {}
D['nombre'] = 'Eduardo Martínez' # Añade una entrada
del D['nombre'] # Y la elimina
```

Si la clave usada en la eliminación no existe, se lanza la excepción `KeyError`.

[1] Técnicamente hablando, los conjuntos devueltos por `keys()` y `values()` se denominan *vistas de diccionario*.

5.2.1.2. Modificación con métodos

Para concluir este apartado, vamos a ver una serie de métodos mediante los cuales podemos realizar modificaciones sobre un diccionario `D` dado:

- `D.clear()`: elimina todas las entradas del diccionario.
- `D.pop(clave,defecto?)`: extrae y elimina una entrada con la clave indicada como primer parámetro. En caso de que la entrada no exista, se lanza una excepción `KeyError`. Como alternativa, se puede indicar un segundo valor por defecto que se devuelve en caso de que la entrada no exista, evitando la excepción `KeyError`.
- `D.update(D2)`: inserta en el diccionario `D` todas las entradas del diccionario `D2`. En caso de que haya alguna clave en `D2` que ya esté en `D`, se modifica su valor con el que indique `D2`.

5.3. Conjuntos

En algunos casos puede interesar crear colecciones de valores no ordenadas en las que los valores sean únicos. Este tipo de colección se asemeja al concepto matemático de conjunto.

Python ofrece un tipo de dato mutable conjunto, que únicamente tiene una restricción: a diferencia de las colecciones que acabamos de ver, un conjunto en Python sólo puede contener valores inmutables. Eso significa que no podremos crear conjuntos de listas o de diccionarios. Pero sí de cualquier tipo de números o cadenas.

5.3.1. Operaciones básicas

Los conjuntos se crean de dos formas: o bien usando el método `set()`, o bien directamente listando entre llaves los elementos del conjunto. Este segundo método nos recordará a los diccionarios, y tiene su sentido: las claves de un diccionario deben ser de tipos inmutables, de modo que un conjunto es como un diccionario en el que las claves no tienen ningún valor asociado.

```
1 alfabeto1 = { 'a','b','c','d' }
2 alfabeto2 = set(['0','1'])
```

Una vez creados, podemos añadir elementos al conjunto con el método `add()`, y eliminarlos con `discard()` o `remove()`. Todos reciben como argumento el elemento que se añade o elimina. En el caso de `remove()`, si el elemento no existe se lanza la excepción `KeyError`.

```
1 alfabeto1.add('e')
2 alfabeto1.discard('f')
```

También es posible extraer un elemento cualquiera de un conjunto usando la operación `pop()` en caso de que el conjunto no sea vacío:

```
1 a = alfabeto1.pop()
```

Con el método `clear()` es posible vaciar completamente un conjunto. También podemos conocer la cardinalidad del conjunto mediante el método `len()`, y podemos comprobar la pertenencia de un elemento a un conjunto mediante el operador `in`:

```
1 if 'a' in alfabeto1:
2         print('a está en alfabeto1')
```

Podemos iterar con conjuntos, tal y como vimos aplicado al recorrido de claves de un diccionario en la sección anterior. Pero no podemos usar operaciones de indexación o troceado. Sin embargo, podemos convertir un conjunto en una lista usando el método `list()`:

```
l = list(alfabeto1)
```

5.3.2. Operaciones de conjuntos

Python incluye operadores binarios para realizar operaciones de unión, intersección y diferencia de conjuntos. El operador de unión es `|`, el de intersección es `&` y el de diferencia es `-`. Por ejemplo:

```
A = { 'violín', 'viola', 'violonchelo' }
B = { 'trompeta', 'trombón', 'trompa' }
C = { 'corneta', 'chirimía', 'sacabuche', 'bajón' }
D = A | B | C
```

Otra operación disponible es `^`, que permite hacer la diferencia simétrica de conjuntos (elementos que están en los conjuntos operados, salvo los que están en su intersección).

Todos los operadores relacionales (ver 2.4.2) se pueden aplicar a conjuntos. El operador `>=` equivale a la comprobación de conjuntos $\supseteq$, mientras que `>` equivale a $\supset$. Los operadores relacionales en sentido contrario `<=` y `<` tienen su correspondiente interpretación $\subseteq$ y $\subset$.

5.4. Tuplas

Python ofrece un cuarto tipo de dato integrado cuyo uso es menos frecuente, pero conviene saber que existe: las *tuplas*. Una tupla es como una lista inmutable, es decir, una secuencia ordenada que no puede cambiar. Se suele emplear para manejar colecciones de ítems que son fijas, como los meses de un año, por ejemplo. Sintácticamente se codifican usando paréntesis en lugar de los corchetes de las listas. Soportan también anidamiento arbitrario y pueden almacenar cualquier tipo de dato.

```
N = ('do','re','mi','fa','sol','la','si')
N[0] = 'ut' # Lanza una excepción TypeError
```

Quizás te venga a la memoria, en plan *dejà vu*, esto de las tuplas. ¿Dónde lo hemos visto esto antes? Recordarás que las expresiones de formateo de cadenas (ver 4.3.1) tenían un listado de los valores que se sustituyen en la cadena, y que se indican entre paréntesis poniendo entre ellos comas, *et voilà!*

También se pueden crear tuplas usando la función `tuple()` que puede recibir como argumento una colección iterable (lista, conjunto u otra tupla). Todos los métodos expuestos en el apartado 5.1 son aplicables a las tuplas, salvo los relativos a la modificación del contenido.

6

ENTRADA Y SALIDA

PARA realizar aplicaciones medianamente útiles con Python tenemos que estudiar algunas herramientas más del lenguaje relativas a la entrada y salida de información de las aplicaciones. En este capítulo comenzamos viendo el modo de consultar los argumentos de un programa Python. A continuación comentaremos el uso de la entrada estándar, salida estándar y salida de errores. Para lograr cierta persistencia en los datos necesitaremos también almacenar y recuperar información de ficheros. Por esta razón, completaremos este capítulo con una explicación de la funcionalidad que ofrece Python para manejar el acceso a ficheros.

6.1. Argumentos del programa

Supongamos que queremos desarrollar una aplicación con Python que se va a ejecutar desde la consola del sistema operativo. Este tipo de aplicaciones suelen recibir información en forma de argumentos de la línea de comandos. Por ejemplo, si la aplicación está implementada en un módulo llamado `a.py` y recibe tres argumentos, se podría lanzar así:

```
$ python a.py arg1 arg2 arg3
```

El intérprete de Python coloca los argumentos con los que se ha invocado a la aplicación en la lista `sys.argv`.

```
import sys

if __name__ == '__main__':
    if len(sys.argv) != 4:
        print('Uso: %s arg1 arg2 arg3' % (sys.argv[0]))
    else:
        arg1 = sys.argv[1]
        arg2 = sys.argv[2]
        arg3 = sys.argv[3]
```

Al igual que sucede con los programas en C, el argumento `sys.argv[0]` contiene el nombre del script. Para indicar los argumentos desde PyCharm, tenemos que entrar en la configuración de ejecución o depuración del proyecto. Podemos verlo en los laboratorios, o escríbemos a los profesores de prácticas. Somos gente amable.

6.2. Entrada, salida y errores

El módulo `sys`, que hemos usado en el apartado anterior para consultar los argumentos del programa, también nos ofrece tres *flujos* (streams) abiertos para manejar la entrada estándar al programa, la salida estándar y la salida de errores. Se trata de `sys.stdin`, `sys.stdout` y `sys.stderr`.

Aunque es posible manejar estos flujos de entrada y salida directamente usando las herramientas de lectura y escritura de archivos, en esta sección comentamos el modo de emplearlos indirectamente a través de los métodos `input()` y `print()`.

6.2.1. Entrada estándar

El modo más sencillo para leer de la entrada estándar es la función `input()`, incluida en el propio lenguaje. La función lee una línea completa de la entrada, la convierte a una cadena (eliminando el salto de línea) y la devuelve. Opcionalmente podemos llamar a `input()` con un argumento que es el *promt* que se mostrará al usuario por salida estándar para invitarle a que escriba:

```
1 print("Indica tu sketch de los Monty Python favorito")
2 sketch = input("--> ")
3 best = ['The Spanish Inquisition','Four Yorkshiremen','Brain Specialist']
4 if sketch in best:
5         print("¡Ése es genial!")
6 else:
7     print("No está mal")
```

6.2.2. Salida estándar y salida de errores

Ya hemos visto el uso del método `print()` en bastantes ejemplos, pero podría indicarse algo más. Hasta ahora, hemos usado `print()` con uno o varios argumentos separados por comas, y el comportamiento que tiene por defecto consiste en imprimir las cadenas que resultan de invocar a `str()` sobre cada argumento, dejando un espacio en blanco como separador. Podemos modificar el separador usando el argumento nombrado `sep`:

```
1 print('Uno',2,['a','b',3],sep='.+.')
```

Un segundo argumento adicional del método `print()` es `file`, que permite especificar el flujo a través del cual se va a generar la salida. Por defecto toma el valor `sys.stdout`, pero podemos usarlo para producir mensajes de error a través de `sys.stderr`:

```
1 print('¡Catástrofe: excepción capturada!',file=sys.stderr)
```

Los mensajes enviados a la salida de error se muestran en rojo en la consola de PyCharm. Dependiendo del funcionamiento del intérprete de Python, los mensajes de la salida estándar y la salida de error se pueden mezclar. Esto se debe a que los accesos a la consola se intentan reducir

para acelerar la ejecución del programa. El texto pendiente de ser emitido por la consola se almacena en un buffer que, llegado el momento que decida el intérprete, se muestra por la salida. Si sólo se usa una salida (normal o de error), no hay problema en la ordenación de los mensajes. Si se emplean las dos, podemos encontrarnos con mezclas extrañas. Para obligar a que se vacíe un flujo en la consola, el método `print()` tiene otro argumento más: `flush`. Por defecto toma el valor `False`, indicando que no hay que imprimir inmediatamente la salida. Si se indica `True` se fuerza al intérprete a emitir por consola todo lo que tenga almacenado en el buffer asociado al flujo del argumento `file`.

Otro argumento que puedes usar con el método `print()` es `end`. Por defecto, el método genera un salto de línea como último caracter de la cadena que se imprime. Podemos modificar este comportamiento especificando otra cadena en `end`:

```
1 print('No imprimas un salto ',end='')
2 print('porque quiero seguir en la misma línea')
```

6.3. Ficheros

Python incluye una función integrada en el propio lenguaje para abrir archivos del sistema: `open()`. El resultado de la función es una instancia de un tipo de dato que actúa como *descriptor de archivo* y que ya hemos usado con los flujos de entrada estándar, salida estándar y error. Sobre los descriptores de archivo abiertos podremos invocar a una serie de métodos para realizar operaciones de lectura, escritura y cierre de los archivos.

6.3.1. Apertura

El uso más básico del método `open()` consiste en proporcionarle como argumento el nombre del archivo que queremos abrir. El nombre puede incluir una ruta de directorios delante. Si no la tiene, Python considera que está en el mismo directorio en el que se encuentra el módulo que realiza la apertura. Para usuarios de Windows es importante tener en cuenta que las rutas de archivos usan la contrabarra como separador de directorios. Por esta razón, será conveniente usar el prefijo `r` en la cadena, de modo que la contrabarra no se interprete como parte de un carácter de escape.

```
1 archivo = open(r'C:\spam.txt')
```

Por defecto, el modo de apertura es para *lectura*. Y sí, lo has adivinado: si el archivo no existe, excepción al canto. En este caso sería de tipo `FileNotFoundError`. Por tanto, si no te gusta el riesgo, pon un buen `try-except` rodeando la apertura del archivo.

Para indicar el modo de apertura, ya sea lectura o escritura, se puede pasar un segundo argumento al método `open` en forma de cadena de caracteres que tiene el significado siguiente:

- `r`: apertura en modo lectura, que es el modo por defecto. Se lee el fichero desde el comienzo.
- `w`: apertura en modo escritura. Si el fichero existe, lo vacía.
- `a`: apertura en modo escritura. Si el fichero existe, no lo vacía sino que añade lo que se escriba al final del mismo.
- `x`: apertura en modo escritura. Si el fichero existe, salta una excepción `FileExistsError`.

Para abrir un fichero en modo lectura y escritura simultáneamente, hay que añadir el carácter + al modo de apertura. Normalmente se empleará la operación `seek()` que luego se describe para situar la posición de la seguiente operación de lectura o escritura:

- `r+`: apertura en modo lectura y escritura sin vaciar el fichero. Inicialmente se escribe y lee por el principio.
- `w+`: apertura en modo lectura y escritura, vaciando el fichero si existe.
- `a+`: apertura en modo lectura y escritura. Si el fichero existe, inicialmente se escribe y lee al final del mismo.
- `x+`: apertura en modo lectura y escritura. Si el fichero existe, salta una excepción de tipo `FileExistsError`.

Detras de la indicación del modo de apertura, en la misma cadena, se puede indicar si se quiere manejar el fichero en modo texto o binario:

- `t`: apertura en modo texto, que es el modo por defecto.
- `b`: apertura en modo binario.

```
1 # Apertura en modo lectura/escritura para texto
2 archivo = open('log.txt','r+t')
```

El argumento con el modo de apertura tiene que indicarse en segundo lugar, o bien puede nombrarse con `mode=`. Un argumento que se nombra con `encoding` permite especificar cuál es el código de caracteres con el que se debe tratar el archivo. Si no se indica, se manejará el código de caracteres del sistema. Por ejemplo, en Windows se emplea `cp1252` (una extensión de `latin1`), pero otros sistemas como Mac OS X o Linux usan `utf8`:

```
1 # Apertura en modo escritura con latin1
2 archivo = open('salida.txt','w',encoding='latin1')
```

6.3.2. Lectura

Existen varias posibilidades al leer un archivo con Python: leerlo completamente en una única operación, leerlo línea a línea o leerlo en fragmentos de cierta cantidad de bytes. Vemos por separado cada alternativa.

6.3.2.1. Lectura completa

El modo más sencillo para leer un archivo consiste en invocar al método `read()` sobre el mismo una única vez:

```
1 archivo = open('archivo.txt')
2 contenido = archivo.read()
```

En la variable `contenido` se queda almacenada una cadena con todo el contenido del archivo. Sobre esta cadena se pueden usar los métodos de indexación, troceado o fragmentación que sean necesarios para procesar el contenido.

6.3.2.2. Lectura por líneas

En muchas ocasiones, el procesamiento de un archivo se debe realizar línea a línea. Python nos da tres alternativas para leer una tras otra todas las líneas de un archivo: el método `readline()` que se invoca sobre el archivo abierto, un iterador sobre el archivo, o bien el método `readlines()`, que devuelve una lista con todas las líneas del archivo.

El método `readline()` devuelve sucesivamente las líneas del fichero abierto, incluyendo el salto de línea. Cuando no quedan más líneas, el método devuelve la cadena vacía:

```
archivo = open(nombre)
n = 1
while True:
    línea = archivo.readline()
    if not línea:
        break # Final del archivo
    else:
        print('Línea %d: -%s-' % (n,línea.rstrip()))
        n += 1
```

Recuerda que la estructura del `while` anterior, sólo apta para estómagos fuertes, es una forma de escribir un *do-while* en Python. Una alternativa más elegante para hacer lo mismo consiste en usar un `for-in` sobre el archivo abierto, que en Python es un objeto sobre el que se puede implementar una iteración:

```
archivo = open(nombre)
n = 1
for línea in archivo:
    print('Línea %d: -%s-' % (n,línea.rstrip()))
    n += 1
```

Y vamos con la tercera alternativa: el método `readlines()`. Mediante este método podemos obtener todas las líneas del archivo en una lista:

```
archivo = open(nombre)
líneas = archivo.readlines()
n = 1
for línea in líneas:
    print('Línea %d: -%s-' % (n,línea.rstrip()))
    n += 1
```

Es cuestión de gustos el escoger una u otra.

6.3.2.3. Lectura por caracteres o bytes

Una posibilidad más a la hora de leer los archivos – especialmente útil cuando se trata de archivos binarios – es la lectura de bloques de bytes o caracteres. Para ello tenemos que emplear una versión de `read()` que tiene como argumento el número de bytes o caracteres que se quiere leer:

```
tamaño_cabecera = 10
archivo = open(nombre,"rb")
cabecera = archivo.read(tamaño_cabecera)
for b in cabecera:
```

```
5         print('%x' % (b), end=' ')
```

6.3.3. Escritura

Lo sentimos por los programadores indecisos: también tenemos varias formas de escribir en un fichero. Podemos emplear los métodos `write()` y `writelines()` del fichero abierto en modo escritura, o bien podemos usar `print()` especificando en el parámetro `file` el fichero.

El método `write()` es el clásico. Tiene como argumento la cadena que se quiere imprimir, que es una cadena de caracteres en caso de que se haya abierto en modo texto, o bien una cadena binaria si se ha abierto en modo binario:

```
1 archivo = open('salida.txt','w',encoding='latin1')
2 archivo.write('De minimis non curat praetor\n')
3 archivo.write('Aquila non capit muscas\n')
4 archivo.write('Beati hispani quibus vivere bibere est\n')
```

Como alternativa, en caso de que tengamos una lista de cadenas y queramos volcarlas sobre el archivo en el orden en el que se encuentran en la lista, podemos emplear `writelines()`:

```
1 archivo = open('salida.txt','w',encoding='latin1')
2 latinajos = []
3 latinajos += ['De minimis non curat praetor\n']
4 latinajos += ['Aquila non capit muscas\n']
5 latinajos += ['Beati hispani quibus vivere bibere est\n']
6 archivo.writelines(latinajos)
```

Y como tercera alternativa, tenemos el método `print()` con el argumento `file`:

```
1 archivo = open('salida.txt','w')
2 print('%.2f*%.2f+%.2f=%.2f' % (3.14,2.0,5.1,3.14*2.0+5.1), file=archivo)
```

6.3.4. Otros métodos

Para concluir la descripción de las operaciones sobre archivos, comentamos brevemente tres métodos más. En primer lugar, el método de cierre: `close()`. Como hemos indicado en varias ocasiones, Python emplea un *recolector de basura* para liberar la memoria que ya no se referencia en nuestro programa. Esta recolección incluye a los ficheros abiertos que ya no se pueden usar porque el código ha pasado a otro ámbito y no son alcanzables. Python cierra automáticamente estos archivos antes de liberar los recursos que emplean. Pero nosotros podemos realizar el cierre de forma explícita:

```
1 archivo.close()
```

El cierre lleva a cabo un vaciado del buffer de salida que el intérprete de Python puede estar usando para reducir el acceso al disco. También podemos forzar ese vaciado sin cerrar el archivo, invocando al método `flush()`:

```
1 archivo.flush()
```

Por último, cuando un archivo se abre en modo lectura y escritura (incluyendo + en el modo de apertura), es muy posible que necesitemos situarnos en diferentes posiciones del archivo en función de la operación que queramos hacer. Sabiendo el desplazamiento `N` desde el comienzo del fichero, podemos indicar la posición de la siguiente lectura o escritura usando `seek()`:

```
1 archivo.seek(N)
```

7

CLASES

VAMOS a hacer una mínima incursión en el terreno de la programación orientada a objetos con Python. Tiene que ser mínima por dos razones: para aprender el paradigma de programación orientado a objetos ya tenéis la asignatura correspondiente, y no es muy aconsejable mezclar ideas de aquí y allá cuando se está empezando porque la indigestión puede ser importante; y en segundo lugar, si nos metemos a fondo con la orientación a objetos en Python, estos apuntes van a perder su intención inicial de ser pequeños. Por tanto, sólo vamos a contar lo estrictamente necesario y así tendréis la oportunidad de leer con ganas los libros que aparecen indicados en la bibliografía al respecto de esta parte del lenguaje.

7.1. Introducción

Vamos a enfocar el uso de la orientación a objetos como una vuelta de tuerca más de la estructuración del código. Como vimos en el capítulo 3, Python está muy preparado para que resulte fácil organizar el código de un proyecto en módulos. Un módulo contiene una serie de definiciones que puedo importar en otro módulo, pero atención, sólo puedo hacerlo una vez. Hasta ahora hemos visto módulos que contienen definiciones de variables y de funciones.

Imagina que quieres hacer un módulo que contenga variables y funciones para representar rectángulos. Aunque se puede hacer, no sería muy estético que con un único módulo, que sólo se puede importar una vez, pudieses representar muchas instancias de rectángulos distintos. Aquí es donde viene en ayuda la definición de clases.

Una clase permite definir un tipo de dato nuevo del cual puedo crear cuantas instancias me apetezca, de modo similar a la forma de crear muchas listas o diccionarios que ya hemos visto. Cada instancia, u *objeto*, tiene su propia memoria, con sus valores de variables, y puedo invocar a métodos sobre cada instancia que trabajen con esos valores y no los de otra instancia.

7.1.1. Definición

Vamos a empezar viendo cómo definir una clase. Dentro de un mismo módulo se pueden crear múltiples clases. Por ejemplo, supongamos que definimos una clase para representar un rectángulo. Podemos hacerlo en un módulo `figuras2D.py`, dentro del cual definimos la clase empezando con la palabra clave `class` seguida del nombre de la clase `Rectángulo`[1]. Normalmente los nombres de clases se inician con una letra en mayúscula en Python. La clase crea un espacio de nombres nuevo, y todas las definiciones de variables o métodos de la clase se especifican en un bloque indentado a continuación de la línea `class`.

```
1 class Rectángulo:
2     # Definición de métodos y variables
```

Una clase tiene un primer método especial, porque es el encargado de crear instancias nuevas de la clase. Se llama *inicializador*, y puede recibir argumentos con los valores que permiten definir la nueva instancia. Por ejemplo, un rectángulo se define con el alto y el ancho. El método inicializador siempre se llama igual en Python: `__init(self,...)__`:

```
1 class Rectángulo:
2     def __init__(self,alto,ancho):
3         self.alto = alto
4         self.ancho = ancho
```

Como ves, además de los argumentos que sirven para definir la instancia nueva, siempre hay un primer argumento en el inicializador y en cualquier otro método de la clase, que se llama `self`. Se trata de una referencia a la instancia de la clase. En el inicializador, puedo definir variables de las instancias usando la notación `self.X`, donde `X` es el nombre de la variable que quiero definir. Normalmente los parámetros que se pasan al inicializador sirven para dar valor a algunas de estas variables. En cualquier método adicional puedo consultar y modificar los valores de estas variables usando `self.X`.

Podemos añadir un método para calcular el área del rectángulo, siguiendo siempre la convención de poner como primer argumento a `self`:

```
1 class Rectángulo:
2     def __init__(self,alto,ancho):
3         self.alto = alto
4         self.ancho = ancho
5     def área(self):
6         return self.alto*self.ancho
```

7.1.2. Creación y uso de instancias

Una vez que hemos definido la clase, podemos probarla con varias instancias y llamadas a sus métodos. Para instanciar un objeto de una clase hay que invocar a su inicializador, usando el nombre de la clase. Una vez que están instanciados los objetos, podemos invocar a métodos sobre cada uno de ellos. Si la clase está en un módulo distinto a aquél que va a usarla, nos hará falta empezar haciendo la importación correspondiente:

```
1 from figuras2D import Rectángulo
```

[1] Los nombres de clases se escriben comenzando con mayúsculas: `class DestructorFacultad`.

También podemos probar la clase dentro del módulo `figuras2D.py`, con algún código de primer nivel que instancie varios objetos e invoque a sus métodos para comprobar que todo va bien:

```
if __name__ == '__main__':
    r1 = Rectángulo(5,4) # Crea un rectángulo con alto=5 y ancho=4
    r2 = Rectángulo(8.3,20.1)
    print('Area de r1:',r1.área())
    print('Area de r2:',r2.área())
```

En muchas ocasiones se crean instancias de una clase dentro de otra clase. Este uso se conoce como *composición* en orientación a objetos.

7.1.3. Variables de clase

Además de definir variables de instancia en el inicializador de la clase, también se pueden definir variables de clase que son compartidas por todas las instancias de la clase. Para ello únicamente hay que definir las variables fuera de los métodos de la clase, sin usar `self` delante:

```
class Rectángulo:
    tipo = 'rectángulo' # Variable compartida por todas las instancias
    ...
```

7.1.4. Visibilidad

En Python, por defecto, todas las variables y métodos son accesibles para cualquier código que utilice la clase. Existen algunos modos de ocultar las definiciones pero, como se suele decir, queda fuera del ámbito de este documento. Únicamente os recomendamos que sigáis la convención de nombrar con un prefijo de dos guiones bajos `__X` a los métodos y variables que no queráis que se usen desde fuera de la clase.

7.2. Herencia

Además de la composición, la otra gran ventaja de la orientación a objetos es la *herencia* que consiste en definir clases a partir de otras previamente creadas. La herencia requiere un diseño meditado del código, porque no es algo que se improvise a la ligera. Por ejemplo, si quisiéramos ampliar el módulo `figuras2D.py` con cuadrados, círculos y demás, nos daríamos cuenta de que todas las clases comparten alguna funcionalidad común, como el método para calcular el área. Podríamos definir una clase base que definiese este método y, mediante herencia, crearíamos los distintos tipos de figuras extendiendo la clase base mediante herencia.

Vamos a ver aplicado el concepto de herencia a una jerarquía de clases que definen puntos en espacios tridimensionales, bidimensionales y unidimensionales. Creamos tres clases `Punto3D`, `Punto2D` y `Punto1D`, y vamos a hacer que la primera de ellas esté en la raíz de la jerarquía de herencia, ya que un punto bidimensional se puede considerar como otro tridimensional en el que la coordenada `z` es cero; de modo similar, un punto de unidimensional es como uno bidimensional en el que la coordenada `y` es cero:

```
class Punto3D:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z
```

```
4       def módulo(self):
5           return (self.x**2+self.y**2+self.z**2)**0.5
6
7   class Punto2D(Punto3D):
8       def __init__(self, x, y):
9           super().__init__(x, y, 0)
10
11  class Punto1D(Punto2D):
12      def __init__(self, x):
13          super().__init__(x, 0)
14
15  if __name__ == '__main__':
16      p3 = Punto3D(4,2,2)
17      p2 = Punto2D(3,4)
18      p1 = Punto1D(6)
19      print('%.2f %.2f %.2f' % (p3.módulo(),p2.módulo(),p1.módulo()))
```

Como vemos en el código anterior, la forma de indicar que una clase hereda de otra consiste en indicar junto al nombre de la clase, entre paréntesis, la clase de la que hereda, también llamada clase *base*. Por ejemplo `Punto2D(Punto3D)` significa que la clase `Punto2D` hereda de la clase `Punto3D`. La herencia implica que todos los métodos y variables de la clase base pasan a ser métodos y variables de la nueva clase, salvo el inicializador. Por ejemplo, el método `módulo()` se puede invocar también sobre objetos de la clase `Punto2D`.

En la clase `Punto2D` se ha definido un inicializador distinto, que sólo tiene argumentos `x` e `y`. Desde este inicializador se invoca al de la clase base mediante `super()`, que permite recuperar una referencia a dicha clase base. Como se puede observar, el argumento `z` en la invocación de la línea 9 es 0, como corresponde a un punto bidimensional. De forma similar se continúa la jerarquía de herencia a `Punto1D`.

7.3. Comprobaciones con objetos

Cuando creamos objetos y los guardamos en variables o cualquier tipo de contenedor, lo que estamos haciendo realmente es guardar una referencia a la posición de memoria donde se ha creado el objeto. Python es bueno con nosotros y nos oculta todos los detalles de bajo nivel que en lenguajes más espartanos, como C, tendríamos que manejar nosotros mismos.

Si una variable que contiene una referencia a un objeto se asigna a otra variable, siempre estaremos copiando la referencia, no estaremos creando una copia del objeto en sí. Y recuerda que un objeto puede tener internamente referencias a otros objetos que usa por composición.

Para comprobar si una referencia que tenemos por un lado apunta al mismo objeto que otra, podemos usar el operador relacional `is`:

```
p1 = Punto3D(4,2,2)
p2 = p1 # Copiamos referencias
print(p1 is p2) # Imprime True
```

Si lo que queremos hacer es comprobar si dos objetos son iguales en el sentido de que contienen exactamente la misma información, aunque sean instancias distintas de una misma clase, podemos usar el operador relacional ==:

```
p1 = Punto3D(4,2,2)
p2 = Punto3D(4,2,2)
print(p1 == p2) # Imprime True
print(p1 is p2) # Imprime False
```

También podemos comprobar si un objeto es de cierta clase, usando la función `isinstance()` que ya vimos en la sección 2.2.8. Esta función recibe como argumento la referencia al objeto y la clase o tipo. Debido a la herencia, `isinstance()` devuelve `True` si comprobamos que un objeto es instancia de toda la jerarquía hasta la clase base:

```
p1 = Punto1D(4)
print(isinstance(p1,Punto1D)) # Imprime True
print(isinstance(p1,Punto2D)) # Imprime True
print(isinstance(p1,Punto3D)) # Imprime True
```

Además de las comprobaciones con tipos básicos y clases, `isinstance()` también permite verificar si una instancia es una lista (`list`), un diccionario (`dict`), un conjunto (`set`) o una tupla (`tuple`).

8

AUTÓMATAS EN PYTHON

En este capítulo iniciamos la descripción de los aspectos que están más relacionados con la asignatura Autómatas y Lenguajes Formales de la FIUM. Concretamente, vamos a centrarnos en el modo de manipular los autómatas finitos deterministas (AFDs) desde el código Python. Los autómatas que podremos manejar tendrán que haber sido editados con la herramienta JFLAP, bien en su versión 7 como en la 8beta. No entraremos en la descripción del formalismo de los AFDs, sino en el funcionamiento del código disponible para usarlos desde Python.

8.1. Instalación del paquete `jflap`

En los recursos de la asignatura en el Aula Virtual puedes encontrar la carpeta `Python`. Dentro de ella hay un fichero `jflap.zip`. Al descomprimirlo se genera una carpeta `jflap` dentro de la cual encontrarás los siguientes tres ficheros:

- `__init__.py`: fichero de inicialización del paquete.
- `Afd.py`: es el módulo principal que contiene la clase `Afd` que manejaremos para manipular el autómata desde Python.
- `Transiciones.py`: es un módulo auxiliar de `Afd.py`. No es necesario importarlo.

Mueve la carpeta `jflap` al directorio del proyecto de PyCharm en el que estés trabajando. De los tres ficheros anteriores, únicamente será necesario importar la clase `Afd` del fichero `Afd.py`. Para esto, tendremos que usar la siguiente línea de importación desde nuestro código:

```
1 from jflap.Afd import Afd
```

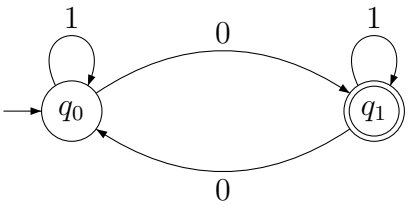


Figura 8.1: Autómata `prueba.jff`

8.2. Instanciación de un Afd

Dependiendo de la versión de JFLAP que estés usando, cambia ligeramente el modo de instanciar un objeto `Afd`. Puedes ver la versión de JFLAP seleccionando el menú *Help*, opción *About*. Necesitamos dos argumentos:

- Ruta del fichero JFLAP que contiene el autómata. Si has usado la versión 7, el fichero tendrá como extensión `.jff`; si has usado la versión 8beta tendrá como extensión `.jflap`. Indica la ruta completa, recordando que si estás en Windows conviene que uses `r'...'` para evitar problemas con las contrabarras que separan los directorios.
- Versión del fichero. La versión del fichero es un número. Debes usar el 6 (sí, no es un error, 6 en lugar de 7) para los ficheros `.jff` y el 8 para los ficheros `.jflap`. Si no indicas nada, se usa 6 por defecto.

No estaría mal que empleases un `try-except` similar a éste:

```
1 from jflap.Afd import Afd
2 from sys import stderr
3 ruta = r'C:\prueba.jff'
4 try:
5     autómata = Afd(ruta)
6 except FileNotFoundError:
7     print('Me temo que la ruta está mal',file=stderr)
8 except Exception as error:
9     print('Problema analizando el fichero:',error,file=stderr)
```

La excepción `FileNotFoundError` se lanza desde el constructor del objeto `Afd` cuando el fichero pasado como primer argumento no existe. En un programa que solicite el autómata al usuario desde la consola, lo normal sería pedirlo nuevamente. La segunda línea `except` captura cualquier excepción. Por ejemplo, si el fichero no se puede analizar con la versión especificada como segundo parámetro, se recibirá una excepción que indicará el problema. Igualmente, si la versión especificada no es ni la 6 ni la 8, se lanzará una excepción advirtiéndonos de esto.

En las siguientes secciones consideramos que estamos trabajando con un autómata como el indicado en la figura 8.1.

8.3. Uso de Afd

En esta sección se proporciona un pequeño manual de uso de un objeto `Afd`. En primer lugar, veremos cómo imprimir la información del autómata, que nos puede servir en la fase de pruebas.

Seguidamente estudiaremos los métodos que son necesarios para simular con el autómata, es decir, para ir recorriendo transiciones desde el estado inicial, usando un símbolo en cada paso.

8.3.1. Mostrar el Afd

Para poder depurar un programa que utiliza un objeto `Afd`, puede ser interesante imprimir toda la información que contiene para verificar que coincide con el autómata que hemos indicado al instanciar el objeto. Para ello, podemos usar el método `mostrarAfd()`:

```
autómata = Afd(ruta,versión)
# Depuramos
autómata.mostrarAfd()
```

El resultado es un volcado por consola de la información que contiene. Por ejemplo, para el autómata mostrado en la figura 8.1, veríamos lo siguiente:

```
Número de estados: 2
Número de transiciones: 4
El estado inicial es: 'q0'
Los estados finales son: ['q1']
El alfabeto del autómata es: ['0', '1']
El estado 'q0' tiene estas transiciones:
{'1' : 'q0', '0' : 'q1' }
El estado 'q1' tiene estas transiciones:
{'0' : 'q0', '1' : 'q1' }
```

8.3.2. Obtener el estado inicial

Para validar una cadena con un autómata, necesitamos comenzar el recorrido de los símbolos de la cadena desde el estado inicial. Para recuperar el estado inicial, simplemente tenemos que invocar al método `getEstadoInicial()` del autómata:

```
autómata = Afd(ruta,versión)
# Inicio de la simulación
estadoActual = autómata.getEstadoInicial()
print(estadoActual)
```

8.3.3. Consultar una transición

Supongamos que disponemos de una variable `simboloActual` que contiene el siguiente símbolo a procesar de la cadena. Necesitaremos comprobar si, desde el estado actual, se puede avanzar a un nuevo estado recorriendo alguna transición con el símbolo actual. Disponemos del método `estadoSiguiente()` para realizar esta consulta:

```
estadoSig = autómata.estadoSiguiente(estadoActual,simboloActual)
print(estadoSig)
```

Este método recibe como argumentos el estado actual y el símbolo actual, y devuelve el estado al que se llega o `None` en caso de que la transición no esté definida.

8.3.4. Comprobar si un estado es final

Para verificar si una cadena se valida, debemos comprobar si el último estado al que se llega tras recorrer todos sus símbolos es final. Para realizar esta comprobación disponemos del método `esFinal()`:

```
estadoSig = autómata.estadoSiguiente(estadoActual,simboloActual)
print(autómata.esFinal(estadoSig))
```

Devuelve `True` en caso de que sí lo sea, `False` en caso contrario.

8.3.5. Obtener el alfabeto del autómata

Se puede consultar el alfabeto que emplea el autómata con el método `getAlfabeto()`:

```
# Conjunto de símbolos
alfabeto = autómata.getAlfabeto()
```

9

EXPRESIONES REGULARES

MUCHAS de las aplicaciones que aprovechan de forma práctica los principios teóricos de la asignatura Autómatas y Lenguajes Formales, emplean *expresiones regulares* (ER) para la validación, búsqueda y sustitución de cadenas de caracteres. La validación la podemos encontrar en cualquier aplicación que reciba una entrada que puede contener errores de formato, como los campos de un formulario rellenados por una persona; un DNI o un código bancario IBAN tienen una estructura perfectamente definida que se debe verificar antes de aceptar la entrada del usuario. Las búsquedas forman parte de infinidad de aplicaciones, como los procesadores de texto, las herramientas de consulta de bases de datos o los *spiders* que recorren la web o las redes sociales y extraen información que aparece en determinados campos de páginas HTML o contenidos XML. Por último, las operaciones de sustitución son la base para la construcción de traductores, es decir, programas que reciben una entrada en un determinado formato y generan la salida en otro formato. La interconexión de muchas aplicaciones desarrolladas por empresas distintas sería imposible sin estas traducciones entre distintos formatos.

En este capítulo analizamos cómo emplear las expresiones regulares extendidas en Python con la finalidad de realizar las tres operaciones indicadas anteriormente. Como complemento a este capítulo, conviene que tengas a mano el documento que puedes encontrar en el Aula Virtual `Tutorial_sintaxis_ER.pdf`.

9.1. Paquetes `re` y `regex`

Toda la funcionalidad de Python relativa a expresiones regulares se encuentra en uno de los paquetes estándares de la distribución de Python, llamado `re`. Vaya por adelantado la dirección en la que puedes resolver todas las dudas que se te ocurran, y más, sobre el funcionamiento de este paquete: `https://docs.python.org/3/library/re.html`.

Sin embargo, el paquete `re` no tiene soporte para expresiones regulares del tipo `\p{L}`. Por esta razón, conviene que instalemos el paquete `regex` mediante la herramienta `pip` en nuestro sistema:

```
pip install regex
```

Todos los métodos definidos en el paquete `re` están en `regex`, ampliando el soporte de expresiones regulares. Puedes hacer así la importación del paquete `regex`:

```
import regex as re
```

Y puedes encontrar informacion sobre `regex` aquí: `https://pypi.org/project/regex/`.

9.1.1. Representación de la expresión regular

En primer lugar, y a riesgo de ser pesados, conviene recordaros que las expresiones regulares hay que representarlas en Python con cadenas *raw*, ya que es habitual usar contrabarras en la notación de expresiones regulares extendida:

```
patrón_dni = r'\d{8}[A-Z]' # Patrón para DNI
```

9.1.2. Compilación de la expresión regular

Antes de usar una expresión regular conviene compilarla. El proceso de compilación implica el análisis de la expresión regular y su conversión a un autómata finito. Como es un proceso costoso, especialmente en el caso de expresiones regulares complejas, hay que evitar en la medida de lo posible la repetición de la operación de compilación de una misma expresión regular:

```
patrón_dni = r'\d{8}[A-Z]' # Patrón para DNI
er_dni = re.compile(patrón_dni) # Objeto ER
```

9.1.3. Validación de cadenas

Para comprobar que una cadena completa se valida con una expresión regular, disponemos del método `fullmatch()` que podemos invocar sobre el objeto ER previamente compilado:

```
import regex as re

if __name__ == '__main__':
    patrón_dni = r'\d{8}[A-Z]' # Patrón para DNI
    er_dni = re.compile(patrón_dni) # Objeto ER
    cadena = input('Introduce un DNI: ')
    result = er_dni.fullmatch(cadena.rstrip())
    if result:
        print('Cadena validada')
    else:
        print('Cadena rechazada')
```

El resultado del método `fullmatch()` es `None` en caso de que la cadena no se valide, y en caso de que sí se valide es un objeto `Match`. La utilidad de este objeto la veremos inmediatamente con otros métodos. En el caso de `fullmatch()`, al hacer una comprobación sí/no, únicamente sirve a modo de resultado booleano.

Python incluye un segundo método de validación de cadenas algo más laxo que `fullmatch()`: el método `match()` valida si algún prefijo de la cadena proporcionada como argumento verifica el

patrón de la ER. En el ejemplo anterior, si introducimos 12345678ALF, también se devuelve un objeto Match. En este caso, podemos saber el prefijo que ha verificado la ER usando los métodos start() y end() del objeto Match devuelto:

```
result = er_dni.match(cadena.rstrip())
if result:
    print('Validado',cadena[result.start():result.end()])
```

Opcionalmente, los métodos fullmatch() y match() pueden tener otros dos parámetros más para indicar desde qué posición de inicio y hasta qué posición de final de la cadena se debe hacer la verificación:

```
er_digitos = re.compile(r'\d+')
result = er_digitos.fullmatch('abc123def',3,6)
```

9.1.4. Búsqueda de cadenas

Tratamos a continuación dos formas de buscar subcadenas que cumplen un determinado patrón dentro de una cadena mayor.

Si sólo nos interesa encontrar la primera aparición de la subcadena, se puede emplear el método search() sobre la expresión regular compilada. Este método recibe como argumento la cadena en la que se busca y devuelve un objeto *match* en caso de que se haya encontrado una aparición del patrón, o None en caso de que no aparezca:

```
er_digitos = re.compile(r'\d+')
n = 1
cadena = 'AB 12 DE 34'
m = er_digitos.search(cadena)
if m:
        print('Resultado %d: %s' % (n,cadena[r.start():r.end()]))
else:
        print('No encontrado')
```

Para localizar las subcadenas no solapadas que cumplen un determinado patrón dentro de una cadena mayor, haciendo un recorrido de izquierda a derecha de la misma, lo más conveniente es usar un iterador que podemos obtener con el método finditer():

```
er_digitos = re.compile(r'\d+')
n = 1
cadena = 'AB 12 DE 34'
for r in er_digitos.finditer(cadena):
        print('Resultado %d: %s' % (n,cadena[r.start():r.end()]))
        n += 1
```

El iterador asigna a la variable indicada en el for un objeto Match. Al igual que con los métodos fullmatch() y match(), los métodos search() y finditer() pueden tener dos parámetros opcionales para indicar la posición de inicio y fin de la búsqueda.

9.1.5. Sustituciones

Otra operación común que se puede implementar fácilmente con la ayuda de las expresiones regulares es la búsqueda y sustitución de cadenas. En Python se puede usar el método sub() para

esta operación. El primer argumento es la cadena de remplazo que sustituye a las ocurrencias de la ER; el segundo argumento es la cadena sobre la que se realiza la búsqueda y sustitución; un tercer argumento permite indicar el número máximo de sustituciones (si no se indica, no hay número máximo). El resultado de la llamada a `sub()` es la cadena con las sustituciones ya realizadas:

```
er_digitos = re.compile(r'\d+')
cadena = 'AB 12 DE 34'
nueva = er_digitos.sub('-',cadena)
print(nueva) # Imprime 'AB - DE -'
```

Si no es posible realizar ninguna sustitución, se devuelve la misma cadena que se pasó como segundo argumento.

Una posibilidad bastante útil del método `sub()` consiste en que el primer argumento, en lugar de ser una cadena estática, sea el nombre de un método. Este método recibirá como argumento un objeto `Match` correspondiente a la coincidencia previa a la sustitución, y debe devolver la cadena que lo sustituye. Por ejemplo:

```
def sust(m):
    suma = 0
    for i in m.group(0):
        suma += int(i)
    return str(suma)

if __name__ == '__main__':
    er_digitos = re.compile(r'\d+')
    cadena = 'AB 12 DE 34'
    nueva = er_digitos.sub(sust,cadena)
    print(nueva) # Imprime 'AB 3 DE 7'
```

El método `sust()` llama al método `group(0)` el objeto Match para recuperar la cadena que se va a sustituir. En la siguiente sección se trata en detalle el uso de grupos.

9.2. Grupos

Una expresión regular puede definir grupos usando paréntesis. Por ejemplo:

```
patrón_dni = r'(\d{8})([A-Z])' # Patrón para DNI con dos grupos
```

La expresión regular anterior define dos grupos, o lo que es lo mismo, dos fragmentos de la expresión regular global indicados entre paréntesis: los dígitos del DNI forman el primer grupo, y la letra mayúscula forma el segundo grupo. Cualquiera de las operaciones vistas en la sección anterior, de validación, búsqueda y sustitución, puede aprovechar el uso de los grupos.

9.2.1. Grupos en validación y búsqueda

En el caso de las operaciones de validación y búsqueda, podemos extraer la porción de la cadena de entrada que se ha reconocido con cada grupo mediante el objeto `Match` devuelto por la operación, y el método `group()` del mismo. Este método recibe como argumento un entero. Si es 0, el método devuelve la cadena completa reconocida con toda la expresión regular. Si es 1, devuelve la porción reconocida a través del primer grupo, y así sucesivamente:

```
patrón_dni = r'(\d{8})([A-Z])' # Patrón para DNI con dos grupos
er_dni = re.compile(patrón_dni) # Objeto ER
cadena = input('Introduce un DNI: ')
result = er_dni.fullmatch(cadena.rstrip())
if result:
    print('DNI',result.group(0),'validado')
    print('Dígitos:',result.group(1))
    print('Letras:',result.group(2))
```

Dependiendo de la expresión regular, no todos los grupos pueden haber sido usados en la validación de la cadena. En caso de que un grupo no se use durante la validación, el método `group()` devuelve `None`. Por ejemplo:

```
patrón = r'(\d+)|(\p{L}+)|([ \r\t\n]+)'
er = re.compile(patrón)
cadena = 'AB 12 DE 34'
for r in er.finditer(cadena):
    if r.group(1):
        print('Dígitos',r.group(1))
    elif r.group(2):
        print('Letras',r.group(2))
    else:
        print('Espacios en blanco')
```

Como alternativa al método `group()` se puede usar el objeto `Match` con indexación:

```
patrón = r'(\d+)|(\p{L}+)|([ \r\t\n]+)'
er = re.compile(patrón)
cadena = 'AB 12 DE 34'
for r in er.finditer(cadena):
    if r[1]:
        print('Dígitos',r.group(1))
    elif r[2]:
        print('Letras',r.group(2))
    else:
        print('Espacios en blanco')
```

9.2.2. Grupos en sustituciones

Las cadenas de remplazo usadas en las sustituciones pueden usar la información de los grupos obtenida en la operación de búsqueda previa al remplazo. La subcadena reconocida a través del grupo `N` se puede referenciar desde la cadena de remplazo mediante `\N`:

```
er_digitos = re.compile(r'[A-Z]{2}(\d+)')
cadena = 'AB12 DE34'
nueva = er_digitos.sub(r'-\1-',cadena)
print(nueva) # Imprime '-12- -34-'
```

9.2.3. Otras formas de referenciar grupos

Si una expresión regular es muy extensa, es posible que necesitemos usar paréntesis no sólo para denotar grupos, sino simplemente para agrupar partes de la expresión regular con el fin de que

los operadores de la misma se combinen como nosotros queramos. Puede llegar a ser una auténtica pesadilla ir contando los paréntesis para ver cuáles representan un grupo y cuáles no.

Una primera forma de simplificar el problema consiste en marcar de una forma especial los paréntesis que *no* denotan un grupo. Esto se consigue usando la notación `(?:...)`, es decir, añadiendo `?:` detrás del paréntesis de apertura. Estos paréntesis no cuentan a la hora de enumerar los grupos:

```
patrón = r'((?:a|b|c)+)|(\d+)'
er = re.compile(patrón)
cadena = 'aaba12ab'
for r in er.finditer(cadena):
    if r.group(1):
        print('Letras:'+r.group(1))
    elif r.group(2):
        print('Números:'+r.group(2))
```

Una segunda forma de abordar el uso de grupos consiste en darles un nombre, en lugar de un número que depende del orden de apertura y cierre de paréntesis. Los nombres se indican con la notación `(?P<nombre>...)`. El método `group()` del objeto `Matcher` puede recibir como argumento el nombre del grupo que queremos recuperar:

```
patrón_dni = r'(?P<digitos>\d{8})(?P<letra>[A-Z])'
er_dni = re.compile(patrón_dni) # Objeto ER
cadena = input('Introduce un DNI: ')
result = er_dni.fullmatch(cadena.rstrip())
if result:
    print('DNI',result.group(0),'validado')
    print('Dígitos:',result.group('digitos'))
    print('Letras:',result.group('letra'))
```

En las cadenas de sustitución podemos usar \g<nombre> como referencia a un grupo nombrado en la expresión regular.

10

DOCUMENTACIÓN DE PYTHON

FINALIZAMOS estos apuntes de Python con una explicación breve sobre el modo habitual de documentar el código. Si vas hacia atrás unas cuantas hojas, y más concretamente a la sección 2.1.1 en la que tratábamos los comentarios de código en Python, podrás ver que el esqueleto para `main.py` incluía al comienzo una cadena de múltiples líneas que comenzaba y terminaba con tres comillas. Este tipo de cadena se conoce como *docstring*. Hay ciertas partes del código de Python en las que podemos introducir estas cadenas para documentar lo que estamos haciendo. Por supuesto, a nadie le gusta documentar el código. ¡Que el que lo lea se esfuerce en entenderlo! Pero cuando te toca a ti darle vueltas a lo que ha escrito otro, no viene nada mal algunas palabritas sobre lo que ha hecho.

En el código Python, al menos, deberíamos describir los módulos, las clases y las funciones. La documentación del módulo debe aparecer justo al comienzo del fichero, mientras que la de una clase o un método se indica justo a continuación de la línea `class` o la línea `def`. El esquema de la documentación es el siguiente:

```
'''
Documentación del módulo
'''

class X:
    '''
    Documentación de la clase
    '''

    def metodo(args):
        '''
        Documentación de la función
        '''
```

En la siguiente sección indicamos sugerencias sobre lo que puede especificarse en cada cadena de documentación. No vamos a usar ningún sistema complejo para documentar el código, del estilo

de *reStructured*. Ten en cuenta que en esta asignatura no tienes que entregar una memoria de prácticas. A cambio, los profesores de la asignatura evaluamos que seas generoso en la documentación del código.

10.1. Formato de la documentación

La documentación de módulos, clases y funciones sigue un formato muy libre. Se realiza con cadenas multilínea (con tres comillas simples o dobles), y se recomienda que tenga las siguientes partes:

1. Tras la apertura de la cadena, escribimos una línea en forma de resumen o título.
2. Dejamos una línea en blanco después del resumen.
3. A continuación escribimos una serie de líneas adicionales en las que nos podemos extender lo que queramos, evitando superar los 72 caracteres por línea.
4. Después del cierre de la cadena, se deja una línea en blanco antes del código que viene a continuación.

10.1.1. Documentación de una clase

Cuando describimos una clase, al menos deberíamos indicar lo siguiente:

- Un breve resumen del propósito de la clase.
- Un listado de los atributos (de clase o instancia).
- Un listado de los métodos que queremos que se usen desde el código cliente.

Por ejemplo:

```
1  class Afd:
2      '''
3      Clase que representa un fichero JFLAP en Python
4
5      Atributos
6      ---------
7      nestados: int
8          Número de estados del autómata
9      ntransiciones: int
10         Número de transiciones del autómata
11
12     Métodos
13     -------
14     getEstadoInicial() : str
15         Devuelve el nombre del estado inicial
16     esFinal(estado : str) : bool
17         Devuelve True si el estado es final
18     estadoSiguiente(estado : str, símbolo : str) : str
19         Devuelve el destino de una transición
20     '''
```

10.1.2. Documentación de un método

La documentación de los métodos puede ser más explícita a la hora de describir lo que estos hacen. Por ejemplo, se puede indicar:

- Una breve descripción de lo que hace el método y cómo se usa.
- Una descripción de los argumentos, tanto requeridos como opcionales o nombrados.
- En el caso de los atributos opcionales, el valor por defecto que toman.
- Una descripción de cualquier excepción que se puede lanzar desde el método y en qué circunstancias.

Este podría ser un ejemplo:

```
def estadoSiguiente(self, estado, símbolo)
    '''
    Devuelve un str con el destino de una transición desde un estado
    y con un símbolo dados. Si no existe una transición con esas
    características, devuelve None.

    Parámetros
    ----------
    estado: str
        Estado desde el que se inicia la transición
    símbolo: str
        Símbolo que etiqueta la transición
```

10.1.3. Documentación de un módulo

En la parte superior de un módulo es conveniente escribir una descripción general del mismo que puede contener, entre otras cosas, lo siguiente:

- Título del módulo y fecha de creación.
- Autor indicado tras `@author:`.
- Descripción general de las clases o métodos agrupados en el módulo y su funcionalidad común.
- Indicación de las dependencias del módulo, como los paquetes que deben estar instalados para poder usarlo.
- Indicación de la funcionalidad del módulo a través del código de primer nivel, en caso de que tenga un punto de entrada `__main__`.

10.2. Uso de la documentación

Al margen de permitir al que lee el código fuente una comprensión mayor sobre lo que realiza, podemos usar alguna herramienta que saca provecho de los *docstrings* que hemos introducido en el código. Una de ellas es `help`. Desde el intérprete interactivo de Python podemos importar un módulo y consultar su documentación usando dicho comando. Por ejemplo:

```
>>> import jflap.Afd
>>> help(jflap.Afd)
```

BIBLIOGRAFÍA

[1] Mª Ángeles Fernández de Sevilla Vellón. *Introducción práctica a la programación con Python*. Textos Universitarios Tecnología, 2016.

[2] Sébastien Chazallet. *Python 3. Los fundamentos del lenguaje*. Ediciones ENI, 2016.

[3] Allen Downey. *Think Python*. Green Tea Press, 2012.

[4] Mark Lutz. *Learning Python*. O'Reilly, 2013.