

PYTHON PARA ANÁLISIS DE DATOS

ÍNDICE

TU RETO EN ESTA UNIDAD.....	5
1. INTRODUCCIÓN	7
1.1. ¿QUÉ ES PYTHON?.....	7
1.1.1. PYTHON Y EL ANÁLISIS DE DATOS.....	8
1.2. ¿PYTHON 2 O PYTHON 3?	9
1.3. CONFIGURACIÓN DEL ENTORNO DE TRABAJO.....	10
1.3.1. MATERIAL DE LA UNIDAD	10
1.3.2. INSTALACIÓN DE PYTHON Y ENTORNO DE DESARROLLO.....	10
1.3.3. INSTALACIÓN DE ANACONDA	11
1.3.4. JUPYTER NOTEBOOK.....	13
1.3.5. SPYDER	17
1.3.6. INTÉRPRETE DE LÍNEA DE COMANDOS - IPYTHON	18
1.3.7. EJECUCIÓN DE SCRIPTS	18
1.3.8. CONFIGURACIÓN DEL DIRECTORIOS.....	19
2. PRIMEROS PASOS EN PYTHON.....	20
2.1. SOBRE VARIABLES Y CONSTANTES.....	21
2.2. NÚMEROS.....	23
2.3. CADENAS DE CARACTERES.....	24
2.4. VALORES BOOLEANOS.....	29
2.5. NONE	30
3. CONTROL DE FLUJO	31
3.1. EJECUCIÓN CONDICIONAL.....	31
3.2. BUCLES	34
3.2.1. BUCLES WHILE	34
3.2.2. BUCLES FOR	35
3.3. BREAK, CONTINUE Y BUCLES ANIDADOS	37

4. ESTRUCTURAS DE DATOS	40
4.1. SECUENCIAS	40
4.1.1. LISTAS	41
4.1.2. TUPLAS	49
4.1.3. VOLVIENDO A LAS CADENAS.....	51
4.2. CONJUNTOS	52
4.3. DICCIONARIOS	55
4.4. DEFINICIONES POR COMPRENSIÓN.....	57
4.5. GENERADORES	61
5. FUNCIONES	63
5.1. RECURSIÓN	67
5.2. UN PEZ LLAMADO LAMBDA.....	68
5.2.1. ... Y FUNCIONES DE ORDEN SUPERIOR.....	71
6. MÓDULOS Y PAQUETES.....	79
6.1. CARGA DE MÓDULOS Y ESPACIOS DE NOMBRES	80
6.2. LOCALIZANDO LOS MÓDULOS.....	83
6.3. MÓDULOS EJECUTABLES.....	84
6.4. PAQUETES.....	85
6.5. ALGUNOS MÓDULOS INCLUIDOS DE SERIE.....	87
6.5.1. MÓDULOS DEL SISTEMA.....	88
6.5.2. MÓDULOS DE TIPOS DE DATOS	89
6.5.3. MÓDULOS ADICIONALES PARA CADENAS DE TEXTO	90
6.5.4. MÓDULOS MATEMÁTICOS.....	91
6.5.5. MÓDULOS PARA PROGRAMACIÓN FUNCIONAL	92
6.5.6. MÓDULOS PARA FICHEROS.....	94
6.6. INSTALACIÓN DE LIBRERÍAS EXTERNAS.....	94
7. ENTRADA Y SALIDA DE DATOS	96
7.1. LA ENTRADA Y SALIDA ESTÁNDAR.....	102
8. CLASES Y PROGRAMACIÓN ORIENTADA A OBJETOS	105
8.1. ¿ORIENTACIÓN A OBJETOS?.....	105
8.2. DEFINIENDO CLASES Y OBJETOS.....	106
8.2.1. CONSTRUCTORES.....	107
8.2.2. DEFINIENDO MÉTODOS	108
8.2.3. HERENCIA	109

¿QUÉ HAS APRENDIDO?	113
AUTOCOMPROBACIÓN.....	115
SOLUCIONARIO	119
BIBLIOGRAFÍA	121

TU RETO EN ESTA UNIDAD

En todo lo que llevamos de curso has aprendido a utilizar el lenguaje R para realizar todo tipo de tareas con datos, desde el análisis y manipulación básicas, pasando por la visualización y generación de gráficos, hasta construir y validar modelos estadísticos para estimar y predecir distintos fenómenos.

Pero no solo de R vive el científico de datos. En esta unidad y la siguiente vamos a enseñarte cómo un lenguaje de propósito general como Python se ha convertido, gracias a sus características propias y a unas potentes librerías desarrolladas por la comunidad de usuarios, en una de las herramientas más útiles y polivalentes para nuestro trabajo con datos.

En esta unidad repasaremos distintos aspectos de Python. ¡Pero no te asustes! La mayoría de conceptos de programación te serán familiares de otros lenguajes. El objetivo es darte una visión global de Python y que te familiarices con sus construcciones para que puedas manejarte, no que te aprendas todo de memoria y te conviertas en un experto. Esperamos que esta unidad te sirva también como referencia para consultar cuando tengas alguna duda más adelante.

Por cierto, ¿conoces las películas 'La vida de Brian' o 'Los caballeros de la mesa cuadrada'? Muy pronto te contaremos qué relación pueden tener estas películas y un lenguaje de programación (si no lo has averiguado tú ya, claro).

1. INTRODUCCIÓN

El objetivo en esta primera unidad de las dos dedicadas a Python es presentarte los fundamentos del lenguaje. Repasaremos las principales características y funcionalidades que ofrece el propio lenguaje Python y su enorme biblioteca de librerías. Nos centraremos en los conceptos más importantes para entender cómo funciona y aprender a aprovechar sus virtudes en nuestros programas.

1.1. ¿QUÉ ES PYTHON?

Python es uno de los lenguajes de programación más populares y extendidos hoy en día. Se trata de un lenguaje de propósito general, utilizado en infinidad de ámbitos y tareas, desde el desarrollo de aplicaciones y servicios web hasta en la programación de sistemas operativos.

Python es un lenguaje interpretado. Es decir, no es necesario compilar el código para obtener un programa ejecutable. El intérprete de Python se encarga de leer el código que nosotros introducimos y evaluarlo. Esta característica nos va a facilitar mucho poder escribir nuestro programa y probar qué resultado genera de forma interactiva. Además, también hace que podamos copiar nuestro mismo programa de un ordenador con Windows a un ordenador con Linux o un Mac OS y ejecutarlo sin cambiar nada, solo hace falta que esté instalado el intérprete. Decimos que Python es un lenguaje multiplataforma.

Python es un lenguaje de alto nivel, con una sintaxis que prima la sencillez y la legibilidad. Una de las ideas detrás de este lenguaje es simplificar la vida del usuario y permitirle programar en unas pocas líneas de código algoritmos bastante complejos de una forma clara.

Otra punto importante es la biblioteca estándar que incorpora de serie, un conjunto de librerías que proporciona gran cantidad de herramientas y componentes que abarcan múltiples funcionalidades (desde el acceso a ficheros, a conectarte a páginas web u otros ordenadores, o el procesamiento de cadenas de texto) y que sirven como base para ayudarte a construir nuevas aplicaciones.

Además, tendrás disponible una ingente colección de librerías y proyectos de código abierto aportadas por la enorme comunidad de usuarios de Python. Será muy difícil que no encuentres una librería que te ayude a solucionar alguna de tus necesidades.



Ojo al dato

Una curiosidad sobre Python. Guido van Rossum, principal creador del lenguaje, lo bautizó con este nombre en honor al grupo de humoristas británicos Monty Python, famosos por sus programas para TV *Monty Python's Flying Circus* o sus películas *La vida de Brian* o *Los caballeros de la mesa cuadrada*. ¿Cómo, no los conoces? Ya tienes una tarea, buscar y ver sus películas.

1.1.1. PYTHON Y EL ANÁLISIS DE DATOS

Todas las virtudes que acabamos de enumerar ya nos hacen ver que Python es un lenguaje muy potente con infinitad de aplicaciones, y que se adapta perfectamente para ayudarnos a resolver casi cualquier problema que se nos presente.

Si nos centramos en el análisis de datos, a estas virtudes sumamos que existen un número cada vez mayor de librerías avanzadas para la extracción y procesamiento de datos, cálculo matemático, análisis y modelado estadístico, aprendizaje automático (*machine learning*), inteligencia artificial y visualización gráfica.

Además, existen librerías para integrarlo con plataformas *big data* como Hadoop o Spark, y para conectarlo con todo tipo de fuentes y bases de datos, tanto SQL como NoSQL.

Por todos estos motivos, la popularidad de Python entre los campos de la ciencia de datos y el *big data* no para de crecer.

1.2. ¿PYTHON 2 O PYTHON 3?

Esta pregunta requiere una explicación.

Piensa en un lenguaje de programación como en un lenguaje humano cualquiera. Con el tiempo las formas en que usamos un idioma, las palabras, las expresiones, hasta las mismas reglas de la gramática o la ortografía sufren algunos cambios. El idioma evoluciona para adaptarse a nuevas necesidades, nuevos conceptos o formas que expresar. Con los lenguajes de programación ocurre igual.

Aunque Python no sea un lenguaje de programación tan "*veterano*" si lo comparamos con otros históricos (*Fortran* nace en 1955, *LISP* en 1958, o *C* en 1972), ciertamente sí es un lenguaje *adulto*; ya ha pasado por múltiples revisiones y cambios desde que se publicó la versión 1.0 allá por 1994. Python 2 apareció en el año 2000, y Python 3 (la última gran revisión del lenguaje) se lanzó en 2008.

Los cambios entre las versiones 2 y 3 del lenguaje fueron importantes en algunos puntos. Como ya te hemos contado, Python es un lenguaje interpretado; es el intérprete de Python quien se encarga de leer un programa, evaluar el código y traducirlo a instrucciones que la máquina puede ejecutar. Debido a algunos de estos cambios que se introdujeron de la versión 2 a la 3, se perdió la compatibilidad hacia atrás. Es decir, los intérpretes de la versión 2 no eran capaces de "*entender*" algunas de las nuevas expresiones de Python 3 y no podían ejecutar los programas escritos en la nueva versión.

Como había mucha (pero mucha) gente y empresas que habían creado sus programas y librerías usando Python 2, se decidió seguir manteniendo y mejorando esta versión del lenguaje en paralelo a Python 3 durante un periodo de tiempo. Así se daba un margen para poder ir migrando a la nueva versión. La última ver-

sión de Python 2, la 2.7, se publicó en 2010 (¡dos años después de que saliera Python 3!) y se ha decidido mantenerla hasta 2020. A partir de ahí, el equipo responsable del lenguaje dejará de desarrollar y resolver problemas para la versión 2.7.

Es cierto que todavía hay muchos programas y librerías escritas usando Python 2.7. Pero a día de hoy, más o menos un 95% de este software está disponible también escrito en Python 3. Y la mayor parte de todo el código nuevo ya se crea usando Python 3.

Así que volviendo a la pregunta que nos hacíamos, la respuesta es que para este curso vamos a usar Python 3. Y en concreto, trabajaremos con la versión más reciente, la 3.6.

De todas formas, si en algún momento te encuentras con código escrito en Python 2 no debes preocuparte. A nivel de un programador, apenas notarás diferencias, la mayor parte del código será igual. Los cambios más fundamentales afectan más al intérprete.

1.3. CONFIGURACIÓN DEL ENTORNO DE TRABAJO

1.3.1. MATERIAL DE LA UNIDAD

Para seguir el contenido de esta unidad tendrás que descargarte material auxiliar que tienes disponible en el campus. Incluye versiones interactivas de esta unidad para ejecutar en los entornos de desarrollo, así como ficheros de código y datos que usaremos en algunos ejemplos.

Te recomendamos que crees una carpeta para esta unidad (p.ej. dentro de tu carpeta de Documentos o en otra carpeta que estés usando para seguir el curso) y descargues todos los ficheros.

1.3.2. INSTALACIÓN DE PYTHON Y ENTORNO DE DESARROLLO


Ahora tenemos que instalar Python en nuestro ordenador.

Una opción sería descargar la versión de Python para nuestro sistema operativo de la página oficial (<https://www.python.org/downloads/>). Después de instalarlo tendríamos que instalar también todas las librerías externas que vamos a necesitar. Pero espera un poco...

Nosotros vamos a elegir una opción más cómoda, que consiste en utilizar Anaconda (<https://www.anaconda.com/>). Se trata de una distribución de Python especialmente orientada para la ciencia de datos. Viene de serie con un gran número de librerías, incluyendo las principales que necesitaremos para cálculo, análisis de datos, crear modelos estadísticos y generar gráficos de nuestros datos y resultados.

Además, la distribución de Anaconda viene con varias herramientas para obtener y configurar nuevos paquetes, programas y librerías de forma muy sencilla. ¡Incluso nos permite tener instaladas distintas versiones de Python a la vez sin que haya conflictos!

Una vez que hayamos instalado nuestra distribución de Python, tendremos que configurar también un entorno de desarrollo para poder ir creando y probando nuestros programas. Te presentamos dos opciones especialmente pensadas para trabajar en análisis de datos (aunque hay muchos más entornos disponibles en Internet).



Vídeo

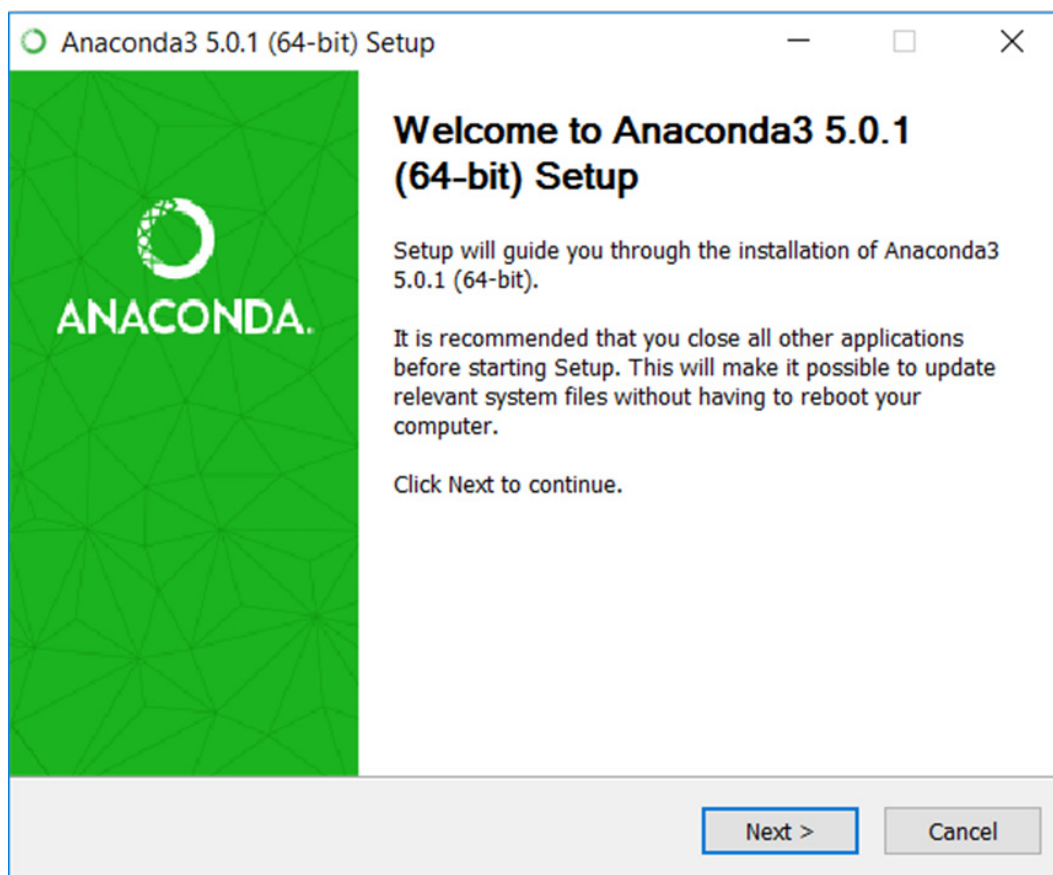
En el campus podrás encontrar vídeos de apoyo con la explicación paso a paso del proceso de instalación y un paseo por los distintos entornos de desarrollo.

1.3.3. INSTALACIÓN DE ANACONDA

La instalación de Anaconda es sencilla. Vamos a ver los pasos para S.O. Windows, pero está disponible para Linux y Mac OS.

- En tu navegador, accede a la página de descargas de Anaconda: <https://www.anaconda.com/download/>

- Selecciona tu sistema operativo y después elige la opción que incluye Python 3 (actualmente la versión 3.6). Si no lo detecta automáticamente, tendrás que seleccionar también si tu plataforma es de 32 o 64 bits. Procede a descargar el instalador (puede costarle unos minutos dependiendo de tu conexión a Internet).
- Una vez descargado, ejecuta el instalador. Sigue las instrucciones, manteniendo las opciones por defecto que te ofrezca en cada paso.



- Al terminar la instalación en Windows, podrás acceder a las distintas herramientas incluidas con Anaconda a través del menú de inicio. Entre otras cosas, tendrás los accesos directos a Jupyter y Spyder, dos entornos de desarrollo.

1.3.4. JUPYTER NOTEBOOK

Jupyter Notebook es una aplicación web que nos permite editar y ejecutar documentos (*notebooks*) en los que podemos combinar texto formateado con trozos de código Python que podemos ejecutar interactivamente y ver los resultados integrados también en el propio documento.

Al ser una aplicación web, el visor y editor de documentos se carga en nuestro navegador web como cualquier otra página. Pero para poder mostrar correctamente el contenido y ejecutar el código Python, debe conectarse a un proceso servidor, que es quien realmente ejecutará el código y devolverá el resultado.

Una de las principales ventajas de este formato de documentos es que permite compartir y distribuir nuestros análisis integrando textos explicativos, figuras y el código utilizado, lo que facilita la reproducibilidad de nuestro trabajo.

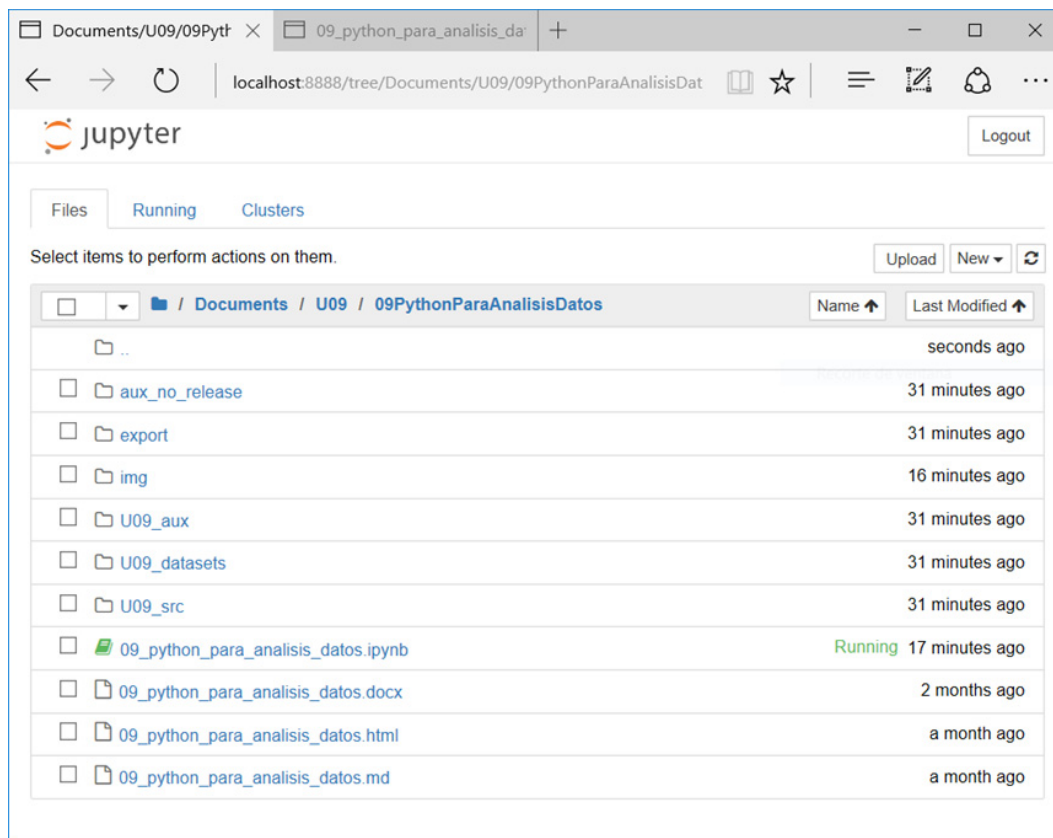
Jupyter se basa en IPython, un intérprete de Python con una consola interactiva que extiende y mejora las funcionalidades de la consola básica de Python, incluyendo autocompletado, depuración, resaltado de sintaxis, etc.

[Abrir Jupyter](#)

Tras instalar Anaconda, en Windows podemos lanzar Jupyter desde la lista de programas instalados en el menú de inicio. Si no, abrimos un terminal o consola de comandos y ejecutamos

```
> jupyter notebook
```

En cuanto el servidor complete su inicialización, se nos abrirá una nueva pestaña en el navegador con la interfaz del cliente Jupyter.



Desde aquí tendremos que navegar por las carpetas hasta llegar a donde se encuentren nuestros *cuadernos*. Los cuadernos o *notebooks* de Jupyter se distinguen por su extensión de fichero (*.ipynb*) y porque aparecen con un icono con forma de libro a la izquierda del nombre.

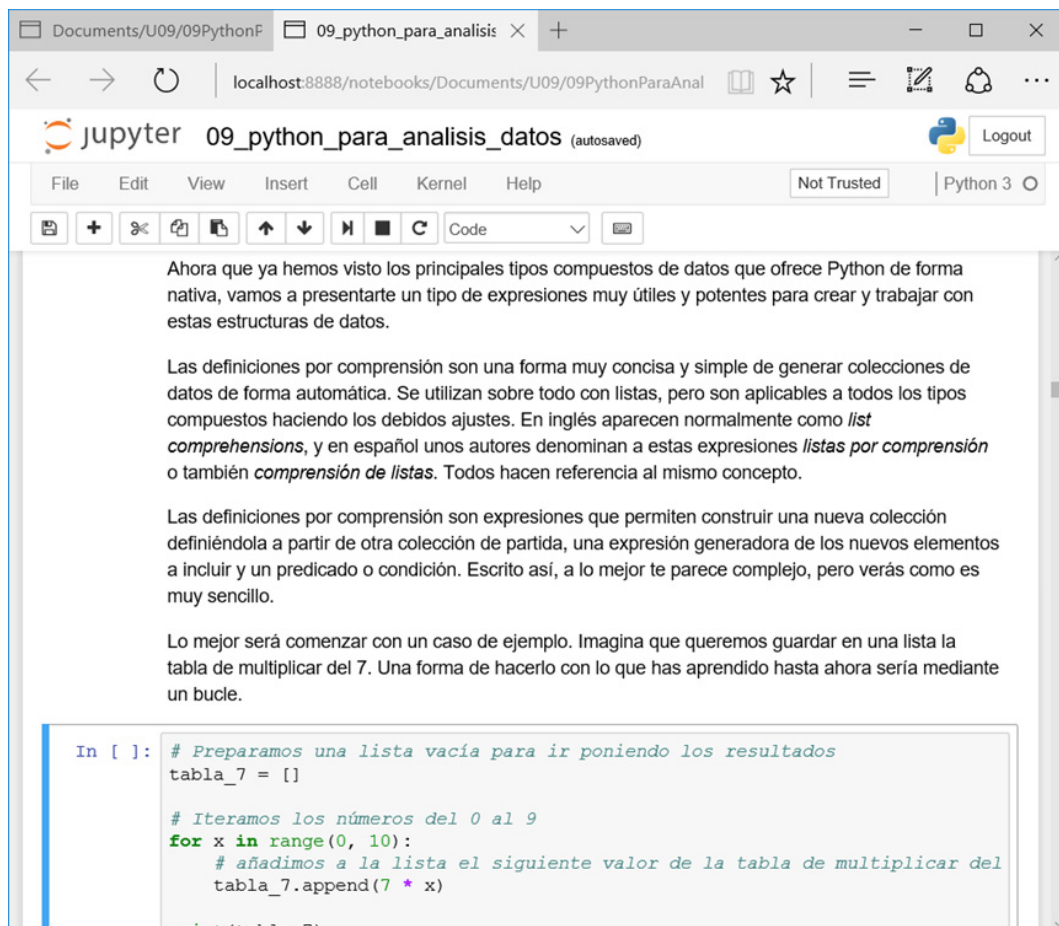
Si hacemos doble click en el cuaderno, su contenido se cargará en una nueva pestaña del navegador, a la que el servidor asociará una nueva sesión Python.

Navegando y ejecutando el contenido

El contenido de un cuaderno de Jupyter se compone de *celdas*. Una celda puede contener o bien texto o bien código interpretable.

Podemos ejecutar el contenido de una de estas celdas seleccionándola (quedará resaltada) y pulsando la combinación de teclas **Ctrl+Enter**. Los mensajes que se generen como resultado de la ejecución se imprimirán justo debajo de la celda.

Puedes editar el contenido de una celda (sea texto o código) seleccionándola y pulsando **Enter**, o haciendo doble click sobre ella.



Añadiendo nuevas celdas

Mediante la barra superior podemos acceder al menú *Insert*, que nos permite añadir una nueva celda en nuestro cuaderno.

Por defecto, la nueva celda será de tipo código (*code*). Estas celdas sirven para que introduzcamos las líneas con el fragmento de código que queremos probar y ejecutar en cada paso.

Podemos cambiar el tipo de contenido a través del menú *Cell*, seleccionando la opción *Cell Type -> Markdown*. De esta forma configuraremos la celda para que acepte contenido de texto formateado con la sintaxis de Markdown. Cuando pulsemos **Ctrl+Enter** para "ejecutar" el contenido de una celda tipo Markdown, se renderizará su contenido en formato HTML.

Controlando los resultados del código

Por defecto, al ejecutar una celda de código Python, el valor resultante que se mostrará será el de la última expresión evaluada. Si quieres que Jupyter muestre el resultado de todas las expresiones, nada más comenzar una sesión de trabajo tendrás que ejecutar este código.

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Ten en cuenta que esto no es necesario cuando ejecutemos cualquier comando para imprimir mensajes (p.ej. con la instrucción **print()**). Jupyter se encargará de imprimir en el cuaderno todos los mensajes sin necesidad de configurar nada más.

Cerrando una sesión en Jupyter

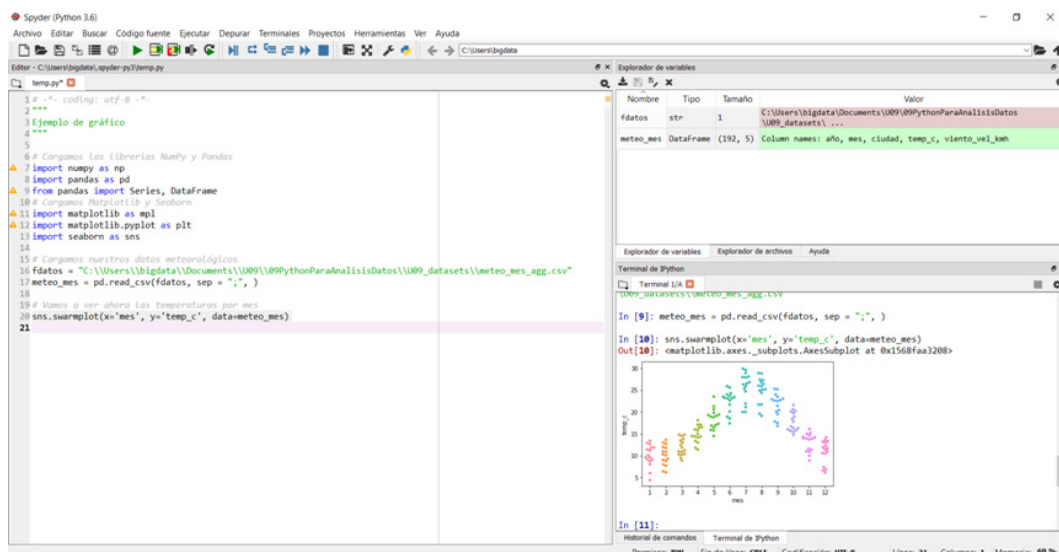
Cuando vayas a dejar de trabajar con un cuaderno en Jupyter, es conveniente que no cierres la pestaña del navegador directamente. En su lugar, ves al menú *File* y selecciona la opción *Close and Halt*. De esta forma, el proceso servidor se encarga de cerrar limpiamente el documento y la sesión Python asociada. Después recuerda cerrar también la ventana con el proceso Jupyter.

Para saber más...

<https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/>

1.3.5. SPYDER

Spyder (*Scientific PYTHON Development EnviRonment*) es un entorno de desarrollo para Python, pensado especialmente para su uso en tareas de programación científica y análisis de datos. Presenta un diseño y funcionalidades similares a las que has visto en RStudio, así que no te costará mucho esfuerzo habituarte a su manejo.



El panel principal (a la izquierda en la imagen) lo ocupa el editor de código. Aquí puedes abrir los ficheros de Python, examinarlos y editar su contenido. El editor incluye ayudas para autocompletar código (con **Tab**), resaltar sintaxis, control de errores, etc.

A la derecha tenemos dos paneles. En uno de ellos (el superior en la imagen) está el explorador de variables, el explorador de archivos y el visor de la ayuda.

En el panel inferior encontrarás la consola interactiva de Python, que utiliza IPython como motor (igual que Jupyter). En esta consola podrás introducir manualmente las instrucciones que quieras probar y ejecutar directamente. Pero también podrás ejecutar el código que tengas abierto en el editor. Para ello tan solo tienes que seleccionar las líneas que quieres evaluar y pulsar **Ctrl+Enter**.

Para saber más...

<https://pythonhosted.org/spyder/>

1.3.6. INTÉRPRETE DE LÍNEA DE COMANDOS - IPYTHON

Podemos lanzar una consola con el intérprete interactivo de Python simplemente abriendo un terminal y ejecutando el comando `python` (`python.exe` en Windows).

No obstante, con la instalación de Anaconda disponemos de un intérprete más versátil y cómodo de usar: IPython. Como ya te hemos adelantado, IPython es un intérprete de Python con una consola interactiva que extiende y mejora las funcionalidades de la consola básica de Python, incluyendo autocompletado, depuración, resaltado de sintaxis, etc.

Si trabajas en Windows, puedes acceder al intérprete de IPython abriendo el *Anaconda Prompt* que te aparecerá dentro del grupo de programas de Anaconda, accediendo desde el menú de inicio.

Para salir de cualquiera de estos intérpretes, puedes usar el comando `quit()`.

1.3.7. EJECUCIÓN DE SCRIPTS

Para programar y probar nuestro código es muy útil (casi imprescindible) utilizar alguno de estos entornos de desarrollo. Pero en algún momento tendremos que ejecutar algunos de nuestros desarrollos como programas de forma directa y autónoma.

Asumiendo que tenemos el código de nuestro programa en un fichero con extensión `.py`, para ejecutarlo simplemente llamamos al intérprete de Python con nuestro script.

```
> python.exe U09_src\hola_mundo.py
```

1.3.8. CONFIGURACIÓN DEL DIRECTORIOS

A lo largo de esta unidad veremos algunos ejemplos que requieren utilizar ficheros de código que incluimos con el material disponible en el campus. El código se encuentra dentro del subdirectorio `U09_src`.

Para que Python sepa acceder al directorio donde guardes los ficheros fuente, tendrás que ejecutar los siguientes comandos para añadirlo a las rutas donde busca el código fuente. A lo largo de esta unidad te explicaremos más sobre cómo busca y resuelve Python la ubicación de código, librerías y otros elementos.

```
import sys
import os.path

# Tan solo modifica la cadena de texto entre comillas
# de DIR_FICHEROS_CURSO
# Sustituye por la ruta completa hasta el directorio base
# donde descargues el material del campus
DIR_FICHEROS_CURSO = "."
DIR_U09 = os.path.join(DIR_FICHEROS_CURSO, "09PythonParaAnalisisDatos")
DIR_U09_SRC = os.path.join(DIR_U09, "U09_src")

sys.path.append(DIR_U09_SRC)
```

2. PRIMEROS PASOS EN PYTHON

Para comenzar, vamos a ver algunas características básicas del lenguaje mediante ejemplos. Empecemos por uno muy sencillo.

```
# Esto es un comentario.  
# En Python el inicio de un comentario se indica con el carácter almohadilla.  
# El comentario termina al saltar de línea.  
  
mensaje = "¡Hola mundo!" # Definimos la variable 'mensaje'. Un comentario puede ir detrás de una expresión  
  
print(mensaje)
```

```
¡Hola mundo!
```

Lo primero que vemos son comentarios. Como dice el propio ejemplo, en Python abrimos un comentario con el símbolo *almohadilla* (#). Estos comentarios terminan con el fin de línea, así que si quieres ampliarlo tendrás que ir añadiendo una nueva almohadilla al saltar de línea.

Después creamos una variable con el nombre `mensaje` y la inicializamos con un valor; en este caso una cadena de texto con un saludo cordial. Y como ves, también podemos añadir un comentario al final de una línea de código, detrás de una expresión.

Y terminamos utilizando la función `print` para imprimir el contenido de la variable. Hasta aquí, todo muy simple.

2.1. SOBRE VARIABLES Y CONSTANTES

Si has trabajado con otros lenguajes de programación como C++ o Java, tal vez te llame la atención que no hayamos necesitado declarar la variable de ninguna forma especial, ni definir previamente qué tipo de dato va a almacenar (¿un número, una cadena de texto?).

Python es un lenguaje con *tipado dinámico*. Esto quiere decir que él se encarga de averiguar el tipo que debe tener una variable en cada momento, en función del valor que le hayamos asignado. Fíjate que esto significa que podemos asignar a una misma variable un valor de otro tipo distinto en cualquier momento y Python no protestará. Simplemente asumirá que el tipo de la variable ha cambiado.



Atención

Que podamos hacer esto no significa que sea recomendable hacerlo en cualquier situación. Aunque en muchos casos podemos sacar provecho de esta característica para simplificar nuestros programas, en otras ocasiones será mejor no mezclar tipos en una misma variable para evitar errores y que el código sea más fácil de entender.

La asignación de un valor a una variable se hace con el signo igual '='. Python nos permite asignar el mismo valor a múltiples variables de una sola vez.

Aunque Python no impone ningún formato estricto, como recomendación de estilo los nombres de las variables se suelen escribir en minúsculas. Si se quiere usar un nombre compuesto, utilizaremos el guión bajo para separar las palabras. El nombre de una variable no puede empezar con un número.

```
nombre = "Papu"
nombre_completo = "Giorgi Papunashvili"
edad_jugador = 21
# Asignación multiple, mismo valor para las dos variables !!
goles_marcados = goles_fallados = 0
# Asignación simultánea, valores distintos a cada variable !!
tarjetas_amarillas, tarjetas_rojas = 1, 0
```

Las dos últimas líneas son ejemplos de asignación múltiple. En el primer caso, asignamos el mismo valor (cero) a las dos variables (`goles_marcados` y `goles_fallados`). En el segundo caso asignamos un valor distinto a cada variable (`tarjetas_amarillas` valdrá 1 y `tarjetas_rojas` valdrá 0).

Fíjate que la asignación múltiple de valores de forma simultánea nos permite hacer cosas como intercambiar valores de variables de forma muy sencilla

```
# Tenemos dos variables
x = 10
y = 20
print("x =", x, "y =", y)
```

```
x = 10 y = 20
```

```
# En otros lenguajes, utilizaríamos una variable auxiliar
# para intercambiar los valores
aux = x    # variable auxiliar, guardamos el valor original de "x"
x = y      # le damos a "x" el valor de "y"
y = aux    # y ahora le damos a "y" el antiguo valor de "x"
print("x =", x, "y =", y)
```

```
x = 20 y = 10
```

```
# Volvamos a las dos variables originales
x = 10
y = 20
print("x =", x, "y =", y)
```

```
x = 10 y = 20
```

```
# ¡En Python podemos hacer esto de una sola vez!
x, y = y, x    # intercambiamos los valores
print("x =", x, "y =", y)    # ¡Tachaaaaan!
```

```
x = 20 y = 10
```

En Python no existen las constantes tal cual, no hay una forma de especificar que una variable no puede cambiar su contenido una vez que ha sido asignada.

Como convención, lo que hacemos es utilizar nombres en mayúsculas para diferenciar los identificadores de las constantes que queramos definir.

Así nos resultará más sencillo reconocerlos en el código (aunque para Python no habrá ninguna diferencia con cualquier otra variable).


```
VERSION = "1.0"  
PI = 3.14159265359
```

2.2. NÚMEROS

Python nos permite trabajar con distintos tipos numéricos, como enteros (tipo *int*) o números reales o en punto flotante (tipo *float*). Disponemos de las operaciones matemáticas habituales con estos tipos. Vamos a verlo.

```
# Suma de enteros, devuelve un entero  
3 + 5
```

8

```
# Resta de enteros, devuelve un entero  
10 - 7
```

3

```
# Multiplicación de enteros, devuelve un entero  
3 * 4
```

12

```
# Pero si uno de los números tiene decimales (punto flotante),  
# el resultado también será un float  
3 + 5.5
```

8.5

```
12.3 - 2
```

10.3

```
3.0 * 4
```

12.0

```
# La división siempre devuelve un valor en punto flotante  
6 / 3
```

```
2.0
```

```
13 / 2
```

```
6.5
```

```
# Si lo que queremos es la división entera, se hace así  
13 // 2
```

```
6
```

```
# Módulo (resto de la división entera)  
13 % 2
```

```
1
```

```
# Potencia  
3 ** 2
```

```
9
```

```
# Valor absoluto  
abs(-3.2)
```

```
3.2
```

```
# Redondear un número en punto flotante  
round(2/3, 1) # una cifra decimal
```

```
0.7
```

```
round(2/3, 2) # dos cifras decimales
```

```
0.67
```

Si mezclamos en una operación valores enteros y en punto flotante, Python se encargará de "promocionar" o convertir todos los valores al tipo más general (en este caso de entero a punto flotante).

2.3. CADENAS DE CARACTERES

Python nos permite expresar cadenas de caracteres de distintas formas. Vamos a ver unos ejemplos de cómo definir una cadena de texto literal.

```

# Cadena de texto con comillas simples
texto_1 = 'Esto es una cadena de texto sencilla'

# Cadena de texto con comillas dobles
texto_2 = "Esto también es una cadena de texto válida"

# Podemos insertar unas comillas simples
# dentro de una cadena delimitada con comillas dobles
texto_3 = "Usar las comillas simples 'aquí' es correcto"

# O al revés, insertar comillas dobles
# en una cadena delimitada con comillas simples
texto_4 = 'Intercalar comillas "de esta otra forma" también es válido'

# También podemos crear cadenas de texto que ocupen varias líneas
texto_multiple_1 = """Podemos crear una cadena con múltiples líneas
rodeando el texto con comillas triples"""

texto_multiple_2 = '''Usar tres comillas simples
es igual de correcto que
usar tres comillas dobles'''

```

Podemos combinar las cadenas de texto de forma sencilla

```

# Podemos concatenar dos cadenas de texto
# usando el operador '+'
lenguaje = 'Py' + 'thon'

lenguaje + "esco"

```

```
'Pythonesco'
```

```

# También podemos repetir una cadena varias veces
# usando el operador '*'
"holá" * 3

```

```
'holaholahola'
```

Podemos contar los caracteres que forman una cadena, o seleccionar una parte de la cadena usando índices de posición entre corchetes ([]). A la hora de seleccionar un carácter o una parte de la cadena, tenemos que saber que en Python empezamos a contar posiciones en cero, así que el primer carácter está en el índice 0 (y no en el índice 1).

```
# Longitud de la cadena de texto  
len(lenguaje)
```

```
6
```

```
# Para seleccionar un carácter indicamos la posición entre corchetes  
lenguaje[0] # EL primer carácter lo obtenemos con el índice cero
```

```
'P'
```

```
lenguaje[5] # EL carácter para el índice 5
```

```
'n'
```

```
# AL empezar a contar en cero, el último carácter está en LEN - 1  
lenguaje[len(lenguaje) - 1]
```

```
'n'
```

```
# Si usamos índices negativos, empezamos a contar desde el final  
lenguaje[-1]
```

```
'n'
```

```
lenguaje[0:2] # También podemos seleccionar una "rebanada"
```

```
'Py'
```

```
# Fíjate que el caracter del primer índice si se incluye,  
# pero el segundo no  
lenguaje[2:6]
```

```
'thon'
```

```
# Seleccionamos desde la posición 2 (incluida) hasta el final  
lenguaje[2:]
```

```
'thon'
```

```
# Seleccionamos desde la penúltima posición (incluida) hasta el final  
lenguaje[-2:]
```

```
'on'
```

```
# Seleccionamos desde el inicio hasta la penúltima posición (excluida)  
lenguaje[:-2]
```

```
'Pyth'
```

```
# Como el primer índice siempre se incluye  
# y el segundo siempre se excluye,
```

```
# podemos reconstruir la cadena así
lenguaje[:2] + lenguaje[2:]
```

```
'Python'
```

Aunque podemos acceder a elementos individuales de una cadena, no podemos modificarlos directamente. Las cadenas en Python son *inmutables*.

```
# Esto produce un error
lenguaje[0] = 'c'
```

```
-----
-

TypeError                                Traceback (most recent
call last)

<ipython-input-55-98768c31a80e> in <module>()
      1 # Esto produce un error
----> 2 lenguaje[0] = 'c'

TypeError: 'str' object does not support item assignment
```

El tipo cadena en Python incluye de serie múltiples operaciones para poder manipular su contenido. Estas operaciones de manipulación no modifican la propia cadena, si no que devuelven una nueva versión modificada. Si quieres cambiar el contenido de una variable, tendrás que reasignarle el resultado de la función.

```
# Convertir a mayúsculas
lenguaje.upper()
```

```
'PYTHON'
```

```
# La operación anterior devolvió una nueva cadena modificada,
# pero no cambió el contenido de la variable
print(lenguaje)
```

```
Python
```

```
# Convertir a minúsculas
lenguaje.lower()
```

```
'python'
```

```
# Si quieres que la variable se actualice con la versión modificada,
# tendrás que asignar de nuevo
```

```
lenguaje = lenguaje.lower()
print(lenguaje)
```

```
python
```

```
# Contar cuántas veces aparece un caracter en la cadena
lenguaje.count('o')
```

```
1
```

```
# Reemplazar un caracter con otro
lenguaje.replace('p', 'c')
```

```
'cython'
```

```
autor = "Jacinto Benavente"
```

```
# Trocear una cadena tomando un caracter como delimitador de los trozos
autor.split(' ')
```

```
['Jacinto', 'Benavente']
```

```
# Si no indicamos separador, por defecto trocea en los espacios en blanco
# Podemos aprovechar la asignación múltiple para asignar cada trozo a una variable
nombre, apellido = autor.split()
print(nombre)
print(apellido)
```

```
Jacinto
Benavente
```

```
# También podemos usar una cadena como separador
# para unir una lista de elementos
";;;".join([apellido, nombre])
```

```
'Benavente;;;Jacinto'
```

```
# ¿La cadena de texto está compuesta por letras?
"abc".isalpha()
```

```
True
```

```
# ¿La cadena de texto está compuesta por dígitos?
"123".isdigit()
```

```
True
```

```
# Formatear una cadena de texto, insertando valores
# en las posiciones marcadas con {}
```

```
"{} ganó el premio Nobel en {}".format(autor, 1922)
```

```
'Jacinto Benavente ganó el premio Nobel en 1922'
```

```
# Para ver más operaciones disponibles sobre el tipo cadena de texto
help(str)
```

2.4. VALORES BOOLEANOS

En Python tenemos un tipo booleano para representar los valores *verdadero* y *falso*. Veamos cómo se codifican estos valores y las operaciones lógicas habituales.

```
soy_rico = False
soy_listo = True

# operación AND (Y) Lógica
soy_rico and soy_listo
```

```
False
```

```
# operación OR (O) Lógica
soy_rico or soy_listo
```

```
True
```

```
# negación
not soy_rico
```

```
True
```

También tenemos los operadores de comparación, que nos devuelven un valor de verdadero o falso

```
# Mayor que
10 > 3.0 # True
```

```
True
```

```
# Menor que  
10 < 3.0    # False
```

```
False
```

```
# Mayor o igual que  
10 >= 3.0   # True
```

```
True
```

```
# Menor o igual que  
10 <= 3.0   # False
```

```
False
```

```
# Igualdad  
6/3 == 2    # True
```

```
True
```

```
# Distintos  
"Rojo" != "Verde"  # True
```

```
True
```

2.5. `NONE`

En Python tenemos un valor especial para indicar que algo (una variable, un parámetro, un objeto) no tiene un valor definido, su valor está ausente o indeterminado. Este valor es `None`.

```
# podemos inicializar una variable con None  
# para indicar que aún no tiene valor asignado  
voluntario = None  
  
# comprobamos si tiene un valor asignado usando `is None`  
voluntario is None
```

```
True
```


3. CONTROL DE FLUJO

Para empezar a resolver tareas más complejas necesitaremos poder indicar no solo qué queremos calcular, también cuándo y cómo ejecutar los cálculos. En Python disponemos de instrucciones para controlar el flujo de ejecución, similares a las que encontramos en otros lenguajes de programación.

3.1. EJECUCIÓN CONDICIONAL

La primera estructura de control que vamos a ver es la ejecución condicional. Esta estructura nos permite decidir si ejecutamos o no una operación (o varias) dependiendo de si una condición se cumple o no.

En Python esto se consigue con la sentencia `if`, que sigue esta estructura básica:

```
if condicion:
    instruccion_1
    ...
    instruccion_n
```

```
precio_chaqueta = 59.95
precio_abrigo = 140
dinero_disponible = 100

print("Tengo {} euros.".format(dinero_disponible))

# Comprobamos la condición
if (precio_abrigo <= dinero_disponible):
    # Si es cierta, ejecutamos este bloque (1) de instrucciones
    print("El abrigo cuesta menos de {} eu-
```

```
ros".format(dinero_disponible))
    print("Puedo comprar el abrigo")
    # Este bloque (1) termina aquí
print("Voy a seguir mirando")

# Comprobamos otra condición
if (precio_chaqueta <= dinero_disponible):
    # Si es cierta, ejecutamos este bloque (2) de instrucciones
    print("La chaqueta cuesta menos de {} eu-
ros".format(dinero_disponible))
    print("Puedo comprar la chaqueta")
    # Este bloque (2) termina aquí
print("Ya no miro más")
```

```
Tengo 100 euros.
Voy a seguir mirando
La chaqueta cuesta menos de 100 euros
Puedo comprar la chaqueta
Ya no miro más
```

Sencillo, ¿verdad? Este ejemplo nos sirve además para explicar alguna característica adicional de Python. Prueba a ejecutarlo y repasemos lo que ocurre.

Después de declarar las variables que vamos a usar, nos encontramos con el primer `if` y la condición que tiene que comprobar. La condición puede ser cualquier expresión que al evaluarla devuelva un valor booleano (verdadero o falso). La sentencia termina seguida de dos puntos (:). Este símbolo marca el fin de la expresión a evaluar, y a continuación el inicio de un bloque de una o más instrucciones que se ejecutarán si la condición se cumple.

Como ves, el bloque de instrucciones dentro del `if` está indentado. En otros lenguajes de programación esto es solo cuestión de estilo y legibilidad. Sin embargo, en Python es obligatorio.



Importante

La sangría o indentación es la forma que tiene Python de delimitar y agrupar bloques de código. Python considera que todas las líneas con un mismo nivel de indentación pertenecen a un mismo bloque o cuerpo de código. En cuanto detecta una línea con menos indentación, asume que el bloque ha terminado.

Así pues, es muy importante que todas las líneas dentro de un mismo bloque estén indentadas de la misma forma. Se puede usar el tabulador o espacios, pero debe ser igual en todas las líneas.

¡Pero no te asustes pensando que vas a tener que andar contando espacios en blanco todo el tiempo! Hoy en día la mayoría de los entornos y editores de código para Python se encargan de insertar y controlar la indentación adecuada de forma automática. Si detecta que alguna línea no encaja bien, te lo indicará.

Volvamos a las sentencias `if`. En el ejemplo anterior comprobábamos una condición (`precio_abrigo <= dinero_disponible`) y después otra (`precio_chaqueta <= dinero_disponible`) de forma independiente. No hay ningún problema en hacerlo así. Pero en muchas ocasiones, las condiciones que tenemos que validar dependerán unas de otras o serán excluyentes entre sí. En Python podemos hacerlo así

```
if precio_abrigo <= dinero_disponible:
    # si se cumple la condición (1), ejecuta este bloque
    print("Me voy a comprar el abrigo")
elif precio_chaqueta <= dinero_disponible:
    # si no, si se cumple la condición (2) ejecuta este bloque
    print("Me voy a comprar la chaqueta")
else:
    # si no se cumplió ninguna, ejecuta este bloque
    print("Me voy con las manos vacías :-(")
```

```
Me voy a comprar la chaqueta
```

Generalizando, esta estructura de código seguiría este esquema

```
if condicion_1:
    bloque_1
elif condicion_2:
    bloque_2
...
elif condicion_n:
    bloque_n
else:
    bloque_por_defecto
```

En este esquema, las sentencias se van evaluando de forma secuencial hasta que se encuentre una condición que se cumpla. Cada sentencia **elif** **condición_i**: indica una condición alternativa a comprobar y sólo se evalúa si no se han cumplido ninguna de las (i - 1) condiciones previas.

La primera condición que resulte ser verdadera provocará que se ejecute su bloque correspondiente. Después se saltará hasta el final del grupo **if** **.. elif** **.. else**, sin comprobar ninguna otra condición.

La sentencia **else**: nos sirve para indicar qué hacer cuando ninguna de las condiciones previas se ha cumplido.

Podemos encadenar todas las sentencias **elif** que necesitemos. La sentencia **else** solamente puede aparecer una vez al final del grupo **if**.

3.2. BUCLES

Los bucles nos permiten repetir o iterar la ejecución de un bloque de código mientras se cumpla una condición o dispongamos de valores que procesar.

En Python disponemos de bucles **while** y bucles **for**.

3.2.1. BUCLES **WHILE**

Estos bucles sirven para repetir la ejecución de un conjunto de instrucciones *mientras que* una condición se siga cumpliendo.

```
i = 1
suma = 0

while i <= 10:
    suma = suma + i
    i = i + 1

print("La suma de los 10 primeros números naturales vale ", suma)
```

La suma de los 10 primeros números naturales vale 55

3.2.2. BUCLES FOR

Los bucles **for** nos permiten iterar por una secuencia o lista de elementos, repitiendo la ejecución de un bloque de código para cada uno de ellos.

```
lista_frutas = ["pera", "manzana", "ciruela", "cereza"] # Así definimos una lista, en un rato te contamos más
```

```
for fruta in lista_frutas:  
    print(fruta)
```

```
pera  
manzana  
ciruela  
cereza
```

En este ejemplo básico comenzamos por crear una lista de nombres de frutas. Podemos crear una lista incluyendo todos sus elementos separados por comas entre dos corchetes. Un poco más adelante en este tema te explicaremos más cosas sobre listas y otras estructuras de datos en Python.

Como acabamos de explicar, el bucle **for** va a recorrer los elementos de **lista_frutas** uno tras otro, haciendo que la primera variable (**fruta**) vaya tomando cada uno de los valores y ejecutando el bloque de código con cada valor. Una vez que se han iterado todos los valores, se sale del bucle.

Es común que nos encontremos con la necesidad de iterar sobre una secuencia de números. En estos casos contamos con una función muy útil que nos va a simplificar el trabajo: la función **range()**.

```
suma = 0  
  
for i in range(1, 10):  
    print(i)  
    suma = suma + i  
  
print(suma)
```

```
1
2
...
8
9
45
```

La función `range()` permite generar una secuencia de números entre dos valores de arranque y parada. Si has probado el ejemplo, habrás visto que el valor de arranque sí se incluye en la secuencia, mientras que el valor de parada queda excluido.



+ Info

Este comportamiento no es una decisión arbitraria. Tiene que ver con la forma en que funcionan los índices para acceder a los elementos de una secuencia (o a los caracteres de una cadena, como ya hemos visto). Profundizaremos un poco más enseguida, cuando estudiemos los tipos de datos compuestos.

Podemos usar la función `range()` con un único valor. En ese caso, lo tomará como si fuera el valor de parada y usará el cero como valor de arranque por defecto. También es posible añadir un argumento más para indicar el tamaño del paso o incremento para generar la secuencia de números.

```
# generamos una secuencia indicando solo el valor de parada
for i in range(5):
    print(str(i) * i)    # str(x) crea una cadena a partir de x
```

```
1
22
333
4444
```

```
# generamos una secuencia indicando que el paso debe ir
# de 2 en 2, en lugar de 1 en 1
for j in range(1, 10, 2):
    print(j)
```

```
1
3
5
7
9
```

3.3. BREAK, CONTINUE Y BUCLES ANIDADOS

Naturalmente podemos anidar bucles dentro de otros cuando lo necesitemos en nuestros programas.

```
for i in range(1, 5):          # bucle exterior
    for j in range(1, i+1):    # bucle interior
        # `end` indica el carácter a usar al final de la cadena
        print(j, end = '')
    print()                    # sin argumentos solo imprime un salto de li-
nea
```

```
1
12
123
1234
```

Sin embargo, en algunas ocasiones querremos alterar el flujo normal de ejecución de un bucle. Para ello disponemos de las sentencias **break** y **continue**.

La sentencia **break** sirve para interrumpir la ejecución del bucle más interno en el que se encuentre y devolver el flujo fuera del bucle inmediatamente a la siguiente instrucción. Lo entenderás más fácil con un ejemplo:

```
# calcular qué números son primos entre 10 y 19
for n in range(10, 20):        # Bucle Externo: iteramos de 10 a 19
    print("n =", n)
    # al inicio, aún no hemos encontrado un divisor de n
    tiene_divisor = False
    # Bucle Interno: nums entre 2 y (n-1) ¿son divisores de n?
```

```
for d in range(2, n):
    # si d es divisor de n, el resto es cero
    print("* d =", d)
    if n % d == 0:
        # en este caso n no es primo, tiene un divisor
        tiene_divisor = True
        # podemos pasar al siguiente num. n (saltar a bucle externo)
        break
    if not tiene_divisor: # si no encontramos divisor en el bucle interno
        print("\t>", n, 'es primo') # entonces n es primo!
```

La orden **break** interrumpe la ejecución y sale del bucle interno (donde buscamos divisores para *n*) en cuanto encontramos el primer divisor. Si probamos todos los posibles candidatos a divisor en el bucle interno y ninguno es un divisor válido, entonces el bucle interno terminará normalmente y podremos decir que *n* es primo.

Para casos así en los que queremos hacer algo cuando el bucle termina normalmente (al agotar todos los valores), Python nos permite usar la sentencia **else**. Sí, como lo oyes (bueno, como lo lees), los bucles en Python también pueden tener **else**. Vamos a usarlo para ajustar el ejemplo anterior

```
# calcular qué números son primos entre 10 y 19
for n in range(10, 20):          # Bucle Externo: iteramos de 10 a 19
    print("n =", n)
    for d in range(2, n):
        # Bucle Interno: nums entre 2 y (n-1) ¿son divisores de n?
        print("* d =", d)
        # si d es divisor de n, el resto es cero
        if n % d == 0:
            break # luego n no es primo, pasar al siguiente numero n
    else:
        # Else: solo llegamos al agotar la secuencia del Bucle Interno
        # entonces no hemos encontrado divisor, y n es primo!
        print("\t>", n, 'es primo')
```

¿Ves las diferencias en el código? Ya no necesitamos usar la variable **tiene_divisor** para controlar si hemos encontrado uno o no. Tampoco nos hace falta el **if** al terminar el bucle interno. La sentencia **else** ya se encarga de ejecutar su bloque solamente si hemos agotado la iteración del bucle hasta el final. El código es más sencillo, y el resultado es el mismo.

Para terminar con los bucles, la sentencia **continue** nos permite saltar a la siguiente iteración sin salir del bucle, pero descartando el resto de cálculos pendientes en el ciclo actual.

```
# queremos calcular la nota media de un alumno

# Tenemos una lista de notas de asignaturas
calificaciones = ["10", "notable", None, "8", "6"]
# algunas son numéricas, otras texto
# Hay un 'None', esto representa un valor ausente

suma = n = 0      # Inicializamos variables auxiliares

for nota in calificaciones:    # Recorremos las calificaciones
    if nota is None:           # si la nota está ausente,
        continue              # pasamos a la siguiente
    elif not nota.isnumeric(): # si la nota no es un número
        continue              # pasamos a la siguiente también
    else:                      # en caso contrario
        suma = suma + float(nota) # podemos computar la nota
        n = n + 1                # para la media

print("La nota media es", (suma/n))
```



El experto opina

Para algunos programadores, el uso de las sentencias **break** y **continue** es desaconsejable porque interrumpe el flujo normal del código, puede hacerlo más difícil de entender y en general siempre es posible hacer lo mismo añadiendo condicionales. En mi opinión, hay casos en los que resultan muy útiles y no solo no enturbian el código, si no que lo hacen más sucinto y claro de entender. Entonces, ¿cuándo usarlas? Bueno, es una cuestión de gustos y experiencia. Mi recomendación es que si simplifica el código y no lo vuelve más confuso para alguien que lo lea, no hay problema en utilizar **break** o **continue**.

Y si, un código legible y fácil de entender por otra persona es algo muy recomendable. Tus programas no solo los va a ejecutar una máquina. Normalmente tú mismo tendrás que volver a modificarlo tarde o temprano. Por no hablar de otras personas con las que colabores. Programar un código elegante y comprensible es una gran virtud. ¡Y Python es un lenguaje fantástico para conseguirlo!

4. ESTRUCTURAS DE DATOS

Python incorpora varios tipos compuestos de datos de forma nativa. Podemos ver un tipo compuesto como una estructura de datos que nos permite almacenar una colección de elementos o valores. Si has trabajado con otros lenguajes, un ejemplo sencillo sería un vector o *array* de elementos. Python nos ofrece más alternativas, con características diferentes dependiendo de cómo se almacenan los valores dentro de la estructura y de qué operaciones podemos hacer.

4.1. SECUENCIAS

Empezaremos por las estructuras de tipo *secuencial*. Son tipos de datos compuestos por una serie de elementos, preservando el orden o posición en el que se añaden a la estructura. Por este motivo, se dice que son tipos *ordenados*.



Importante

Como a veces se presta a confusión, vamos a recalcarlo. Aquí *ordenado* no significa que conforme añades elementos, estos se reordenen (p.ej. numéricamente o alfabéticamente). Solo hace referencia a que conservan el orden en el que son añadidos, es decir, su *orden posicional*.

Todos los subtipos de secuencias en Python tienen en común varias operaciones, como el acceso indexado o por posición, el *rebanado*, la concatenación o la pertenencia.

4.1.1. LISTAS

En Python, las listas son probablemente el tipo compuesto de uso más extendido por su versatilidad y sencillez. Una lista es una secuencia ordenada de elementos. A diferencia de los vectores o *arrays* de otros lenguajes, en una lista se pueden mezclar elementos de tipos distintos sin problema. Aunque lo habitual es que todos los elementos tengan el mismo tipo.

La forma común de definir una lista es incluyendo los elementos separados por comas entre un par de corchetes.

```
# creamos una lista vacía usando un par de corchetes
# sin ningún elemento dentro
lista_vacia = []

# podemos crear una lista de números
lista_nums = [1, 6, 2, 5, 3, 4]

# o una lista de cadenas de texto
lista_frutas = ["pera", "manzana", "ciruela", "cereza"]

# o una lista mezclando valores de cualquier tipo
lista_mezcla = [10, "veinte", 30.0, "cuarenta"]

# incluso podemos meter una lista como elemento dentro de otra lista
listas_anidadas = ["Aquí hay", ["listas", "anidadas"], [1, 2]]
```

Si te has fijado, la última lista de ejemplo tiene listas anidadas como elementos. ¿Cuántos elementos dirías que tiene la lista de nivel superior? ¿Serías capaz de identificarlos? Podemos usar lo que hemos aprendido de bucles para comprobarlo

```
for elemento in listas_anidadas:
    print(elemento)
```

```
Aquí hay
['listas', 'anidadas']
[1, 2]
```

Para acceder a los elementos de una lista por su índice o posición, ponemos el índice entre corchetes también.

El primer elemento de una lista (y de cualquier secuencia) tiene el índice cero. Así que si una lista tiene n elementos, para acceder al último tendremos que usar el índice $(n - 1)$.

```
# Longitud de la lista  
len(lista_nums)
```

6

```
# Longitud de la lista anidada:  
# cada "sub-lista" interna es vista como un único elemento  
len(listas_anidadas)
```

3

```
# Seleccionar un elemento  
lista_nums[0]           # EL primer valor lo obtenemos con el índice  
cero
```

1

```
lista_nums[5]           # EL elemento para el índice 5
```

4

```
lista_nums[len(lista_nums) - 1]  # EL último elemento está en LEN - 1
```

4

```
# AL seleccionar en una lista anidada  
listas_anidadas[1]       # EL elemento devuelto puede ser otra lista
```

```
['listas', 'anidadas']
```

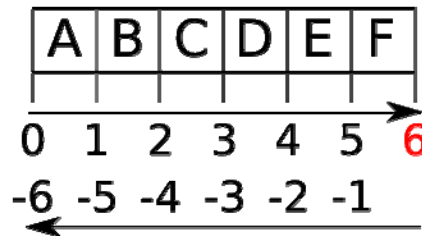
```
# Si usamos índices negativos, empezamos a contar desde el final  
lista_nums[-1]           # Devuelve el último elemento de la lista
```

4

```
lista_nums[-len(lista_nums)]  # Devuelve el primer elemento de la lista
```

1

Para aclarar mejor cómo funcionan los índices para acceder a un elemento, imagina que tenemos la lista ['A', 'B', 'C', 'D', 'E', 'F']. La siguiente figura muestra a qué posición referencia un índice positivo o un índice negativo.



En lugar de ver los índices *"apuntando"* a cada elemento, pensemos que los índices marcan las posiciones *entre elementos* de la secuencia. En este caso, el índice 6 referencia una posición fuera de los límites de la lista y produciría un error.

Ahora es más fácil ver cómo podemos seleccionar una *"rebanada"* de elementos contiguos de una lista. Indicamos el índice o posición inicial desde donde empezamos la selección y la posición final hasta donde queremos llegar, escribiéndolos dentro de los corchetes separados por dos puntos (:). En las rebanadas, el elemento correspondiente al límite inicial siempre se incluye, pero el elemento en el límite final queda excluido.

```
letras = ['A', 'B', 'C', 'D', 'E', 'F']
```

```
# Seleccionamos una "rebanada" con los dos primeros elementos
```

```
letras[0:2]
```

```
['A', 'B']
```

```
letras[2:5]
```

```
['C', 'D', 'E']
```

```
# Si no especificamos el segundo índice, seleccionamos hasta el final
```

```
letras[2:]
```

```
['C', 'D', 'E', 'F']
```

```
# y si no especificamos el primer índice, seleccionamos desde el principio
```

```
letras[:4]
```

```
['A', 'B', 'C', 'D']
```

```
# También podemos usar la notación de índices negativos.  
# Seleccionamos desde la penúltima posición (incluida) hasta el final  
letras[-2:]
```

```
['E', 'F']
```

```
# Seleccionamos desde el inicio hasta la penúltima posición (excluida)  
letras[:-2]
```

```
['A', 'B', 'C', 'D']
```

```
# Como el primer índice siempre se incluye  
# y el segundo siempre se excluye,  
# podemos reconstruir la lista así  
letras[:2] + letras[2:]
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

Las listas sí son *mutables*, es decir, sí podemos modificar el contenido de los elementos de una lista

```
print(lista_frutas)
```

```
['pera', 'manzana', 'ciruela', 'cereza']
```

```
# modificamos un elemento individual  
lista_frutas[1] = "uva"  
  
print(lista_frutas)
```

```
['pera', 'uva', 'ciruela', 'cereza']
```

```
# también podemos reemplazar los valores de una "rebanada"  
lista_frutas[2:] = ["naranja", "aguacate"]  
  
print(lista_frutas)  
  
# o eliminar algunos valores, reemplazando con la lista vacía  
lista_frutas[1:3] = []  
  
print(lista_frutas)
```

```
['pera', 'cereza']
```

```
# o vaciarla entera  
lista_frutas[:] = []  
print(lista_frutas)
```

```
[]
```

También podemos concatenar listas o replicarlas

```
lista_1 = ['a', 'b', 'c']  
lista_2 = [100, 200, 300]
```

```
# concatenación  
lista_1 + lista_2
```

```
['a', 'b', 'c', 100, 200, 300]
```

```
# replicar la lista tres veces  
3 * lista_1
```

```
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

Y podemos extraer los elementos de una lista y asignarlos a distintas variables de forma sencilla con una sola instrucción

```
x, y, z = lista_2  
print(x)  
print(y)  
print(z)
```

```
100  
200  
300
```

Las listas en Python incluyen varios métodos para consultarlas y modificarlas

```
# añadir un elemento al final de la lista  
lista_1.append('d')  
print(lista_1)
```

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

```
# extraer el último elemento de la lista  
ultimo_valor = lista_1.pop()  
print(ultimo_valor)  
print(lista_1)
```

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

```
# insertar un elemento delante de una posición concreta
```

```
# inserta 'e' en la posición cero (al principio)
```

```
lista_1.insert(0, 'e')
```

```
# inserta 'f' en la posición 2
```

```
lista_1.insert(2, 'c')
```

```
print(lista_1)
```

```
['e', 'a', 'c', 'b', 'c']
```

```
# buscar en qué posición está un elemento
```

```
# (si no existe dará un error)
```

```
lista_1.index('b')
```

```
3
```

```
# contar el número de veces que aparece un elemento
```

```
lista_1.count('c')
```

```
2
```

```
# borrar el primer elemento de la lista
```

```
# que coincida con el valor dado
```

```
lista_1.remove('c')
```

```
print(lista_1)
```

```
['e', 'a', 'b', 'c']
```

```
# ordenar los valores de la lista
```

```
# (dependiendo del tipo: alfabéticamente, numéricamente...)
```

```
lista_1.sort()
```

```
print(lista_1)
```

```
['a', 'b', 'c', 'e']
```

```
# invertir el orden de la lista
```

```
lista_1.reverse()
```

```
print(lista_1)
```

```
['e', 'c', 'b', 'a']
```

```
# borrar un elemento indicando la posición
```

```
del lista_1[2]
```

```
print(lista_1)
```

```
['e', 'c', 'a']
```



```
# Limpiar todos los elementos de la lista
lista_1.clear()
print(lista_1)
```

```
[]
```

Como ves, no te engañábamos cuando decíamos que las listas son muy versátiles y fáciles de usar.

Más sobre variables, listas y mutabilidad

Acabas de ver cómo crear y manipular el contenido de listas en Python. Antes de continuar, vamos a aprovechar para explicar un detalle más sobre las variables, su contenido y lo que ocurre al hacer asignaciones en Python.

Imagina el siguiente caso habitual. Tienes una variable (pongamos **lista1**) y la inicializas con unos valores cualesquiera. Más adelante, tal vez tras hacer varios cálculos y operaciones con la primera variable, necesitas crear otra nueva variable (digamos **lista2**) e inicializarla con la misma lista que tenga la primera en ese momento. Así que le asignas la primera variable a la última.

```
# Creamos e inicializamos la primera variable
lista1 = [1, 2, 3]

# Hacemos nuestras operaciones ...

# Y ahora necesitamos crear una nueva variable
# e inicializarla con la misma lista que tenga lista1
lista2 = lista1
print(lista2)                # [1, 2, 3]
```

```
[1, 2, 3]
```

```
# Obviamente, ahora deben tener el mismo valor
lista1 == lista2                # True
```

```
True
```

Obviamente, tras la asignación, ambas variables tienen el mismo valor. Pero es que, en realidad, no solo tienen el mismo valor, si no que *apuntan* al mismo contenido. Ambas son dos nombres o *referencias* al mismo dato, a la misma lista, y no dos copias distintas con los mismos valores.

Una variable en Python no deja de ser eso, un nombre o *referencia* a un valor o estructura de datos que está almacenada de algún modo en memoria. En Python, la operación de asignación de una variable a otra *no copia* el contenido, si no que define un nuevo nombre o *referencia* al mismo contenido que la variable original.

Podemos ver que dos variables representan el mismo objeto con el operador `is`.

¿Y esto qué implica? Piénsalo... Si modificamos un elemento en la segunda variable, ¿qué ocurre con la primera?

Como ambas variables no son más que nombres o referencias para la misma lista en memoria, la primera variable mostrará también el nuevo valor.

```
# Comprobamos que las dos variables
# representan el mismo objeto
lista2 is lista1                # True
```

True

```
# Modificamos lista2
lista2[1] = 99

# Y vemos que lista1 nos devuelve
# el mismo contenido modificado
print(lista1)                  # [1, 99, 3]
```

[1, 99, 3]

No obstante, si en lugar de modificar el valor de los elementos de la lista, asignamos una lista distinta, estaremos definiendo un nuevo objeto o estructura de datos. La variable usada pasará a nombrar ese nuevo objeto independiente; las dos variables ya no serán *referencias* al mismo contenido.

```
# Si asignamos una nueva lista a la segunda variable
lista2 = [7, 8, 9]

# Dejan de referenciar al mismo contenido
lista2 is lista1                # False
```

False

```
# Incluso aunque usemos la primera variable
# en la expresión para definir de la segunda
lista2 = 2 * lista1          # Replicación de la primera lista

# Las estructuras de datos que referencian son distintas
lista2 is lista1            # False
```

```
False
```

A la hora de trabajar con estructuras de datos mutables (como las listas), deberás tener todo esto en cuenta para evitar modificar accidentalmente el contenido de una variable al manipular una segunda.

Si necesitas una copia idéntica pero independiente de una lista, puedes usar el método `copy()`.

```
lista1 = [1, 2, 3]

# Con copy() creas una nueva lista, copia exacta de la original,
# pero independiente
lista2 = lista1.copy()

# Los valores en las listas son iguales
lista2 == lista1          # True
```

```
True
```

```
# ¡Pero no son el mismo objeto!!
lista2 is lista1          # False
```

```
False
```

4.1.2. TUPLAS

Las tuplas son un tipo de datos similar a las listas. Como ellas, las tuplas están formadas por una secuencia de elementos ordenados. Las diferencias principales son dos: - Desde el punto de vista *sintáctico*, se definen usando paréntesis en lugar de corchetes (esta es fácil) - Desde el punto de vista *semántico*, **las tuplas son inmutables**.

¿Qué queremos decir con *inmutables*? Pues que una vez que has creado una tupla definiendo sus elementos, ya no puedes modificar los valores que contiene.

```
# construimos las tuplas poniendo sus elementos
# separados por comas entre dos paréntesis
palos_baraja = ("corazones", "diamantes", "tréboles", "picas")
valores_baraja = ('A', 2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K')

as_de_picas = (valores_baraja[0], palos_baraja[3])
reina_de_corazones = (valores_baraja[11], palos_baraja[0])

# podemos usar tuplas como elementos anidados dentro otra tupla
mano = (as_de_picas, reina_de_corazones)

print(mano)
```

```
(( 'A', 'picas'), ('Q', 'corazones'))
```

```
mano[0]
```

```
('A', 'picas')
```

```
mano[1]
```

```
('Q', 'corazones')
```

```
# también podemos usar listas como elementos,
# ¡cualquier objeto en realidad!
jugador = ("jugador_1", [as_de_picas, reina_de_corazones])
```

Si tratas de modificar la tupla, obtendrás un error

```
mano[0] = ('J', "corazones")
```

```
-----
TypeError                                Traceback (most recent
call last)
```

```
<ipython-input-149-993d0c940dfb> in <module>()
----> 1 mano[0] = ('J', "corazones")
```

```
TypeError: 'tuple' object does not support item assignment
```

En realidad, Python nos permite definir una tupla sin usar los paréntesis al crearla. Es decir, poniendo simplemente los elementos separados por comas

```
# tupla definida por enumeración, sin usar paréntesis
otra_tupla = 'uno', 'dos', 3
print(otra_tupla)
```

```
('uno', 'dos', 3)
```

De todas formas, te recomendamos utilizar siempre los paréntesis. Para empezar, porque así resaltas que es una tupla y entiendes más fácilmente tu código. Y después porque cuando uses tuplas anidadas o expresiones más complejas para definir los elementos, vas a necesitar incluir los paréntesis obligatoriamente. Así que, ¿para qué hacer distinciones? ¡Simplifiquemos!

Además, hay dos casos especiales de creación de tuplas: la tupla vacía y la tupla de un solo elemento.

Las tuplas vacías se crean con una pareja de paréntesis vacía, sin ningún elemento dentro.

Para crear una tupla con un único elemento, lo rodeamos entre paréntesis, pero es necesario añadir una coma justo detrás del elemento y antes del cierre de paréntesis. Parece raro, pero como los paréntesis también sirven para "encapsular" una expresión (p.ej. $(3 * 4)$), Python necesita una pista para distinguir si queremos construir una tupla con un solo valor o no.

```
# Creamos la tupla vacía con una pareja de paréntesis vacía, sin ningún elemento
tupla_vacia = ()
print(tupla_vacia)
```

```
()
```

```
# Creamos una tupla con un solo elemento
tupla_un_elemento = (1,)
print(tupla_un_elemento)
```

```
(1,)
```

4.1.3. VOLVIENDO A LAS CADENAS...

Si recuerdas, cuando presentamos las cadenas de caracteres vimos que podíamos extraer un elemento usando un índice entre corchetes, y que podíamos seleccionar también una "rebanada", o concatenar cadenas, o replicar una cadena n veces...

¿Adivinas a dónde quiero ir a parar? Efectivamente, las cadenas de caracteres son otro tipo de secuencia en Python. Unas secuencias con la particularidad de que sólo se componen de caracteres de texto, y que a la hora de representar-

las, se utiliza una notación más compacta y sencilla para los humanos, escribiendo simplemente el texto entre comillas. Además, son *inmutables* (como las tuplas) y llevan añadidas una serie de funciones o métodos especiales solo para cadenas.

4.2. CONJUNTOS

Ya has aprendido que en Python tenemos los tipos *lista*, *tupla* y *cadena de texto*, y que son tipos secuenciales, es decir, son colecciones *ordenadas* de elementos.

Bueno, dejemos ahora las secuencias. En Python tenemos también tipos compuestos de datos que no son ordenados. Es decir, que sus elementos no conservan el orden en el que son colocados.

El tipo compuesto *no ordenado* más básico es el conjunto (o **set** en Python). Un conjunto es una colección *no ordenada* de elementos, y donde además no puede haber elementos repetidos.

Un conjunto *sí* es modificable. Podemos eliminar o añadir nuevos elementos, siempre que no estén ya incluidos en el conjunto.

La utilidad de este tipo viene justamente cuando necesitamos controlar elementos duplicados o saber si un elemento ha sido incluido ya (*pertenece*) a un conjunto, sin importarnos la posición.

Para crear un conjunto escribimos los elementos entre un par de llaves y separados por comas. Si queremos crear un conjunto vacío utilizaremos la función **set()**, y no un par de llaves vacías (**{}**) como podrías pensar. Un poco más adelante te mostraremos que un par de llaves vacías crean un *diccionario* vacío. Luego te lo contamos.

```
# Vamos a controlar qué objetos
# añadimos en nuestra mochila para una excursión

# creamos un conjunto vacío
# (solo por probar, para este ejemplo no nos haría falta)
en_mochila = set()

# 'inicializamos' : incluimos varios objetos
```

```
en_mochila = {"bocadillo", "agua", "linterna", "agua", "cuerda"}
```

```
# si imprimimos, veremos que el "agua"  
# no aparece duplicada - ya sabemos que llevamos agua  
print(en_mochila)
```

```
{'cuerda', 'linterna', 'bocadillo', 'agua'}
```

```
# anotamos que metemos un cuchillo  
en_mochila.add("cuchillo")  
print(en_mochila)
```

```
{'agua', 'cuchillo', 'bocadillo', 'cuerda', 'linterna'}
```

```
# para comprobar si un elemento está en el conjunto  
# usamos `in`  
"bocadillo" in en_mochila
```

```
True
```

```
"cerillas" not in en_mochila
```

```
True
```

```
# también podemos quitar un elemento del conjunto  
en_mochila.discard("cuerda")  
print(en_mochila)
```

```
{'agua', 'cuchillo', 'bocadillo', 'linterna'}
```

También disponemos de las operaciones habituales entre conjuntos: unión, diferencia, intersección...

```
a = {1, 2, 3, 5, 8}  
b = {1, 2, 4, 8, 16}
```

```
# unión de a y b  
a.union(b)
```

```
{1, 2, 3, 4, 5, 8, 16}
```

```
# también podemos hacerlo así: elementos que están en a ó en b  
a | b
```

```
{1, 2, 3, 4, 5, 8, 16}
```

```
# intersección de a y b  
a.intersection(b)
```

```
{1, 2, 8}
```

```
# también podemos hacerlo así: elementos que están en a y en b  
a & b
```

```
{1, 2, 8}
```

```
# diferencia entre a y b  
a.difference(b)
```

```
{3, 5}
```

```
# también podemos hacerlo así: elementos que están en a pero no están en b  
a - b
```

```
{3, 5}
```

```
# diferencia simétrica o excluyente entre a y b  
a.symmetric_difference(b)
```

```
{3, 4, 5, 16}
```

```
# también podemos hacerlo así: elementos que estan exclusivamente bien en a o bien en b (pero no en ambos)  
a ^ b
```

```
{3, 4, 5, 16}
```

```
s = {1, 2}
```

```
# ¿s es un subconjunto de a?  
s.issubset(a)
```

```
True
```

```
# o dicho de otro modo: todos los elementos de s están en a  
s <= a
```

```
True
```

```
# ¿a es un superconjunto de s?  
a.issuperset(s)
```

```
True
```

```
# o dicho de otro modo: a incluye todos los elementos que están en s  
a >= s
```

```
True
```


4.3. DICCIONARIOS

Los diccionarios son otro de los tipos de datos compuestos más útiles, y que Python proporciona de forma nativa.

La principal característica de los diccionarios es que almacenan parejas de elementos formadas por una *clave* o elemento identificador y el *valor* que queremos asociarle. Si piensas en un diccionario de los que usas normalmente para consultar la definición de una palabra, tendríamos que las *claves* son los términos o palabras, y los *valores* asociados serían las definiciones de esas palabras.



Importante

Al contrario de lo que pueda parecer, los diccionarios no son un tipo *ordenado* (o no tienen por qué serlo). Lo importante en un tipo diccionario es poder añadir una *clave* y su *valor* asociado, y que luego buscar por la *clave* para obtener su *valor* sea una operación rápida y eficiente. Para conseguirlo, el lenguaje gestiona y organiza internamente las *claves* para optimizar el funcionamiento. Cómo lo hace es algo más complejo de explicar. Por ahora, simplemente necesitas saber que las claves no tienen por qué estar en el orden que tú pensarías después de añadir datos nuevos. ¡Ten esto en cuenta cuando uses los diccionarios!

Las claves en un diccionario deben ser únicas, no puede haber claves repetidas. Además, una clave tiene que estar formada por un elemento *inmutable*, no puede cambiar una vez que se ha añadido junto con su valor al diccionario. Por ejemplo, una cadena de texto, un número o una tupla pueden usarse como claves. Pero una lista no, porque es posible modificar los elementos de la lista *a posteriori*.

Para crear un diccionario inicialmente vacío utilizamos una pareja de llaves '{}'. Si quieres añadir elementos en la propia inicialización, sólo tienes que poner parejas **clave : valor** separadas por comas dentro de las llaves. Fíjate que separamos la clave de su valor utilizando dos puntos (':'). Para acceder a un valor, utilizamos los corchetes como haríamos en una lista o una tupla, solo que en este caso indicamos la *clave* y no un índice de posición.

Lo vemos más fácil con ejemplos.

```
# Si queremos crear un diccionario vacío,  
# usamos una pareja de llaves  
libreta_telefonos = {}  
  
# También podemos incluir elementos en la inicialización,  
# poniendo pares clave : valor  
libreta_telefonos = { "Carlos" : 5556045, "Luis" : 5556048 , "Javier" :  
5556051 }  
  
# Para acceder a un valor, utilizamos la clave entre corchetes  
libreta_telefonos["Luis"]
```

```
5556048
```

```
# Podemos añadir una nueva clave y asignarle un valor  
libreta_telefonos["Daniel"] = 5556056  
  
# Si asignamos un valor a una clave existente,  
# reemplazaremos el valor antiguo  
libreta_telefonos["Carlos"] = 5556033  
  
print(libreta_telefonos)
```

```
{'Carlos': 5556033, 'Luis': 5556048, 'Javier': 5556051, 'Daniel':  
5556056}
```

```
# Podemos comprobar si una clave está en el diccionario  
"Luis" in libreta_telefonos
```

```
True
```

```
# Y podemos eliminar una pareja clave valor indicando la clave  
del libreta_telefonos["Luis"]  
  
print(libreta_telefonos)
```

```
{'Carlos': 5556033, 'Javier': 5556051, 'Daniel': 5556056}
```

Tenemos distintas formas de iterar un diccionario. Podemos iterar sobre las claves del diccionario de forma directa. También podemos iterar sobre las parejas (*clave, valor*).

```
# Podemos iterar directamente sobre las claves de un diccionario así
for nombre in libreta_telefonos:
    print(nombre, "=", libreta_telefonos[nombre])
```

```
Carlos = 5556033
Javier = 5556051
Daniel = 5556056
```

```
# O podemos iterar sobre las parejas clave,valor
for nombre, telefono in libreta_telefonos.items():
    print(nombre, "=", telefono)
```

```
Carlos = 5556033
Javier = 5556051
Daniel = 5556056
```

```
# Y si lo que queremos es iterar solo los valores, podemos hacerlo así
for telefono in libreta_telefonos.values():
    print(telefono)
```

```
5556033
5556051
5556056
```

4.4. DEFINICIONES POR COMPRESIÓN

Ahora que ya hemos visto los principales tipos compuestos de datos que ofrece Python de forma nativa, vamos a presentarte un tipo de expresiones muy útiles y potentes para crear y trabajar con estas estructuras de datos.

Las definiciones por comprensión son una forma muy concisa y simple de generar colecciones de datos de forma automática. Se utilizan sobre todo con listas, pero son aplicables a todos los tipos compuestos haciendo los debidos ajustes. En inglés aparecen normalmente como *list comprehensions*, y en español unos autores denominan a estas expresiones *listas por comprensión* o también *comprensión de listas*. Todos hacen referencia al mismo concepto.

Las definiciones por comprensión son expresiones que permiten construir una nueva colección definiéndola a partir de otra colección de partida, una expresión generadora de los nuevos elementos a incluir y un predicado o condición. Escrito así, a lo mejor te parece complejo, pero verás como es muy sencillo.

Lo mejor será comenzar con un caso de ejemplo. Imagina que queremos guardar en una lista la tabla de multiplicar del 7. Una forma de hacerlo con lo que has aprendido hasta ahora sería mediante un bucle.

```
# Preparamos una lista vacía para ir poniendo los resultados
tabla_7 = []

# Iteramos Los números del 0 al 9
for x in range(0, 10):
    # añadimos a la lista el siguiente valor
    # de la tabla de multiplicar del 7
    tabla_7.append(7 * x)

print(tabla_7)
```

```
[0, 7, 14, 21, 28, 35, 42, 49, 56, 63]
```

Hemos tenido que crear una variable con una lista vacía al principio, y después utilizar un bucle **for** para iterar por los valores de 0 a 9 e ir modificando en cada paso la lista, para añadir cada valor de la tabla de multiplicar.

No es que sea un código muy complicado ni largo. Pero Python nos permite hacer esto de forma más sencilla y breve.

```
tabla_7 = [ 7 * x for x in range(0, 10) ]

print(tabla_7)
```

```
[0, 7, 14, 21, 28, 35, 42, 49, 56, 63]
```

¿Qué te parece? ¿Se entiende bien? Vamos a examinar la expresión trozo a trozo para aclarar cómo funciona.

Para empezar, la expresión está entre corchetes. Esto indica que vamos a construir una lista, rellenándola con los elementos que genere la expresión. Enseguida te contamos también cómo hacer para crear un conjunto, o una tupla o un diccionario.

Vamos con el código dentro de los corchetes. La primera parte (**7 * x**) es la expresión que genera los nuevos elementos a partir de los valores que tome **x**, mientras que la expresión **for** nos dice cuál es el *dominio* de valores de entrada que va a tomar **x** (en este caso la secuencia de 0 hasta 9).

Podemos ampliar la definición incorporando una o más condiciones con una cláusula **if**. Supongamos que queremos guardar los valores de la tabla de multiplicar del 7 solo cuando el factor multiplicador es un número par (7*2, 7*4, 7*6...). Fácil.

```
# usamos un if para comprobar si x es par
# (utilizando el operador módulo o resto, ¿lo recuerdas?)
tabla_7_pares = [ 7 * x for x in range(0, 10) if x % 2 == 0 ]
```

Como ves en el ejemplo, simplemente hemos añadido la cláusula condicional '**if x%2 == 0**' al final de la construcción. Para cada valor de **x** en el rango especificado (de 0 a 9 en este caso), se comprueba si se cumple la condición (**x % 2 == 0**, es decir, que el resto de dividir **x** entre 2 sea cero, luego que **x** sea par).

Si se cumple el predicado, entonces se genera un nuevo valor (**7 * x**) para la lista; y si no, ese valor de **x** se descarta.

El dominio de valores de entrada puede ser cualquier objeto o expresión *iterable*.

```
lista_frutas = ["pera", "manzana", "ciruela", "cereza"]

[ fruta.upper() for fruta in lista_frutas if fruta.startswith("c") ]
```

```
['CIRUELA', 'CEREZA']
```

Podemos combinar dos o más cláusulas **for** con sus dominios de entrada para construir la colección.

```
[ x * y for x in (1, 2, 3) for y in (4, 5, 6) ]
```

```
[4, 5, 6, 8, 10, 12, 12, 15, 18]
```

Esta expresión se evalúa para todas las combinaciones de valores de **x** e **y**. Si lo piensas, es como si tuvieras dos bucles **for** anidados. Vamos a incluir en cada elemento generado los valores de **x** e **y**, así quedará más claro.

```
[ (x, y, x * y) for x in (1, 2, 3) for y in (4, 5, 6) ]
```

```
[(1, 4, 4),
 (1, 5, 5),
 (1, 6, 6),
```

```
(2, 4, 8),  
(2, 5, 10),  
(2, 6, 12),  
(3, 4, 12),  
(3, 5, 15),  
(3, 6, 18)]
```

En este caso hemos hecho que la expresión generadora devuelva una tupla con los valores de **x**, **y** y de **x * y**. Puede utilizarse cualquier expresión, todo lo complejo que se necesite, siempre que devuelva un elemento válido.

Si en lugar de generar una lista queremos construir un conjunto, solamente tenemos que reemplazar los corchetes por llaves.

```
{ x for x in range(10) if x % 2 != 0 }
```

```
{1, 3, 5, 7, 9}
```

Si queremos una tupla, no será suficiente con utilizar paréntesis. Tendremos que utilizar el constructor **tuple()**

```
tuple( x for x in range(5) )
```

```
(0, 1, 2, 3, 4)
```

Y para construir un diccionario utilizando el mecanismo de comprensión, además de utilizar las llaves en lugar de los corchetes, la expresión generadora deberá tener la forma **clave : valor**

```
# aquí la clave es el factor multiplicador, y el valor es el resultado de la multiplicación  
dict_tabla_7 = { num : 7 * num for num in range(0, 10) }  
  
print(dict_tabla_7[3])
```

```
21
```

4.5. GENERADORES

Un par de párrafos más arriba acabamos de ver que si queremos crear una tupla definida por comprensión no basta con reemplazar los corchetes por paréntesis, tenemos que usar el constructor `tuple()`.

Bueno, ¿y qué obtenemos entonces si usamos solamente los paréntesis?

En este caso, en lugar de devolver inmediatamente una secuencia completa con todos los valores, Python nos devuelve un tipo especial de objeto que denominamos *generador*.

Digamos que un *generador* es un objeto que se encarga de hacer los cálculos para *generar* los valores resultantes uno a uno conforme los necesitemos y los vayamos pidiendo, *iterando* sobre la colección de entrada, en lugar de construir la secuencia entera de golpe.

La utilidad de este mecanismo es que así se ahorra espacio de memoria (no tenemos que almacenar todos los resultados) y resulta más eficiente en los casos en los que no necesitamos la secuencia resultado completa de una vez, por ejemplo si queremos iterar los valores en un bucle.

```
# construimos un generador para la tabla de multiplicar del 7
gen_tabla_7 = ( 7 * x for x in range(0, 10) )

# si intentamos imprimir el contenido de 'gen_tabla_7'...
print(gen_tabla_7)
# no veremos una secuencia, si no un mensaje
# indicando que es un objeto de tipo 'generator'
```

```
<generator object <genexpr> at 0x7f9d9438dca8>
```

```
# vamos a usarlo en un bucle
for v in gen_tabla_7:
    print(v)

# el generador va devolviendo un nuevo valor
# cada vez que se le pide
# hasta que agota todos los valores de su dominio de entrada
```

```
0
7
14
21
28
35
42
49
56
63
```

Si tenemos un generador y queremos extraer sus valores a una lista, podemos utilizar el constructor `list`

```
# si tenemos un generador
gen_tabla_3 = ( 3 * x for x in range(0, 10) )
# y queremos pasar todos sus elementos a una lista, usamos 'list'
lista_tabla_3 = list(gen_tabla_3)
print(lista_tabla_3)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Fíjate que si intentas volver a utilizar el generador después de haber iterado todos sus elementos (en un bucle o al extraerlos a una lista), no obtendrás nada, excepto posiblemente un error. El generador ha quedado vacío, está agotado, ya no tiene valores que devolver.

```
otra_lista_tabla_3 = list(gen_tabla_3)

# obtienes una lista vacía []
print(otra_lista_tabla_3)
```

```
[]
```


5. FUNCIONES

En lenguajes de programación, una función es un conjunto de operaciones que sirven para completar un cálculo o tarea específica y que se agrupan y encapsulan como unidades de código independientes para poder ser reutilizadas a lo largo de nuestro programa siempre que sea necesario.

El uso de funciones nos permite no tener que reescribir el mismo código en distintas partes de nuestro programa, sustituyéndolo por *llamadas* a una función. No solo tiene la ventaja de la reutilización y evitarnos escribir mucho más código. También hace los programas más modulares y fáciles de leer, depurar y modificar.

En Python indicamos que vamos a *definir* una función utilizando la palabra reservada **def**. Veamos un ejemplo

```
# vamos a definir la función
def mi_primera_funcion():
    print("Soy la primerita función")
```

Bueno, no es la función más útil del mundo, pero para empezar nos sirve.

Podemos dividir la definición de la función en dos partes: el encabezado y el cuerpo.

El encabezado empieza con la palabra clave **def**, seguida del nombre de la función (**mi_primera_funcion**). Para los nombres de funciones se aplican las mismas reglas que para los nombres de variables. Después del nombre de la función se colocan los paréntesis de apertura y cierre, y finalmente los dos puntos (:) para indicar el final de la cabecera.

Justo después del encabezado, en la línea siguiente, comienza el cuerpo de la función, que es donde incluiremos todas las instrucciones que componen el cálculo o tarea que queremos implementar. En este ejemplo, el cuerpo lo forma únicamente la sentencia **print**. Al igual que vimos con las sentencias de control de flujo, este cuerpo va indentado respecto a la cabecera. La primera línea que vuelva al mismo nivel de indentación que el encabezado nos marcará el fin de la función.

Una vez que tenemos definida la función, ya podemos utilizarla. Para *invocar* una función simplemente utilizamos su nombre seguido de los paréntesis.

```
mi_primera_funcion()
```

Soy la primerita función

Como hemos visto, es una buena costumbre incluir comentarios en nuestro código para documentar qué estamos haciendo, de forma que luego sea más fácil de seguir por otras personas (¡y por nosotros mismos!).

En el caso de las funciones, Python permite añadir una documentación específica añadiendo una cadena de texto justo entre el encabezado y el cuerpo de la función. A estas cadenas en Python se las conoce como **docstring**.

Las *docstrings* son opcionales, pero es muy recomendable incluirlas. No tienen ningún efecto en el código que se ejecuta. Normalmente se utilizan cadenas rodeadas por comillas triples, porque así la cadena puede ocupar varias líneas consecutivas. Utiliza las *docstrings* para describir qué es lo que hacen tus funciones y qué parámetros necesitan.

```
def mi_primera_funcion():  
    """Función trivial de ejemplo.  
  
    Imprime un mensaje de prueba.  
    """  
    print("Soy la primerita función")
```

Podemos hacer que la función devuelva un valor como resultado utilizando la sentencia **return**. Además, podemos añadir *parámetros* a la función para utilizarlos en sus cálculos. Indicamos los parámetros que necesita una función añadiéndolos entre los paréntesis de la definición, como nombres de variables.

```

# definimos una nueva función
# que devuelve el cuadrado del número x que pasamos como argumento
def cuadrado(x):
    """Devuelve el cuadrado del número x"""
    return x**2

# y probamos que funciona
print("El cuadrado de 7 es", cuadrado(7))
print("El cuadrado de 12 es", cuadrado(12))

```

```

El cuadrado de 7 es 49
El cuadrado de 12 es 144

```

En esta nueva función **cuadrado** hemos incluido un parámetro **x**. Dentro del cuerpo de la función podemos utilizar el parámetro como si fuera una variable normal. Cuando llamamos a la función dentro de la sentencia **print**, lo hacemos indicando entre paréntesis cuál es el valor que queremos dar a **x** en esta ejecución; en este caso queremos ejecutar la función **cuadrado** haciendo que **x** tome el valor **12**. Estamos haciendo algo similar a una *asignación* de variables. Solo que el parámetro **x** es una especie de variable *local*, únicamente va a existir y ser visible dentro de la función.

Por lo demás, el cuerpo de la función simplemente devuelve (**return**) el resultado de elevar el valor que tenga **x** al cuadrado (**x**2**).

Una función puede recibir varios parámetros, cada uno del tipo que sea necesario. También puede devolver cualquier tipo de elemento.

```

# una función con dos parámetros
# y que devuelve una lista definida por comprensión
def llama_a_la_puerta_de(persona, n_veces):
    """Llama a la puerta de la persona
    Las veces que haga falta"""
    return [ "Toc,toc,toc! " + persona + "!" for i in range(n_veces) ]

llama_a_la_puerta_de("Penny", 3)

```

```

['Toc,toc,toc! Penny!', 'Toc,toc,toc! Penny!', 'Toc,toc,toc!
Penny!']

```

Al definir una función, también podemos especificar un valor *por defecto* para cualquiera de los parámetros. Se indica como si hiciéramos una asignación del parámetro en el encabezado. El valor *por defecto* o *por omisión* solamente se aplicará si al llamar a la función no incluimos un valor explícito para ese parámetro (*omitimos* su valor).

```
# redefinimos la función con dos parámetros
# indicando un valor por omisión para 'n_veces'
def llama_a_la_puerta_de(persona, n_veces=3):
    """Llama a la puerta de la persona
    las veces que haga falta"""
    return [ "Toc,toc,toc! " + persona + "!" for i in range(n_veces) ]

# ahora podemos usar la función sin especificar 'n_veces'
# y tomará el valor por defecto
llama_a_la_puerta_de("Penny")

# si incluimos un valor para el parámetro,
# se ignora el valor por omisión
llama_a_la_puerta_de("Amy", 1)
```

```
['Toc,toc,toc! Penny!', 'Toc,toc,toc! Penny!', 'Toc,toc,toc! Penny!']
```

```
['Toc,toc,toc! Amy!']
```

Al definir una función, los parámetros para los que no se especifique un valor por defecto deben ir delante de los parámetros con valor por omisión.

Al llamar a una función es obligatorio dar valores para todos los parámetros que no tengan valor por omisión especificado. Por defecto, deben proporcionarse los valores en el mismo orden en el que se han definido los parámetros.

No obstante, podemos llamar a una función indicando los valores para cada parámetro por su nombre, como pares **nombre_parametro=valor**. Haciéndolo así, podemos dar valor a los parámetros en el orden que queramos.

```
llama_a_la_puerta_de(n_veces=5, persona="Howard")
```

```
[ 'Toc,toc,toc! Howard!',  
  'Toc,toc,toc! Howard!',  
  'Toc,toc,toc! Howard!',  
  'Toc,toc,toc! Howard!',  
  'Toc,toc,toc! Howard!']
```

5.1. RECURSIÓN

En el cuerpo de una función se puede llamar a otras funciones. Y naturalmente una función también se puede llamarse a sí misma. Es lo que se llama una definición recursiva.

Este tipo de construcciones son habituales en definiciones de series o progresiones matemáticas, o para recorrer determinados tipos de estructuras de datos.

Tomemos por ejemplo la conocida ***Sucesión de Fibonacci***, que se define así

$$Fib(0) = 0$$

$$Fib(1) = 1$$

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

El elemento n -ésimo de la sucesión (para $n \geq 2$) se construye sumando los elementos para $(n-1)$ y $(n-2)$. Esto es una definición recursiva. Escribamos unos cuantos términos:

$$Fib(0) = 0$$

$$Fib(1) = 1$$

$$Fib(2) = Fib(1) + Fib(0) = 1 + 0 = 1$$

$$Fib(3) = Fib(2) + Fib(1) = 1 + 1 = 2$$

$$Fib(4) = Fib(3) + Fib(2) = 2 + 1 = 3$$

$$Fib(5) = Fib(4) + Fib(3) = 3 + 2 = 5$$

$$Fib(6) = Fib(5) + Fib(4) = 5 + 3 = 8$$

En Python podemos construir una función que implemente esta definición recursiva

```
def fibonacci(n):  
    """Calcula el n-ésimo elemento de la serie de Fibonacci"""  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)  
  
fibonacci(3)  
fibonacci(6)
```

2

8

¿Cuáles son las ventajas de las funciones recursivas? Bueno, como puedes ver el código es sencillo y fácil de entender. Si quisieramos hacer lo mismo usando bucles tendríamos que utilizar más variables para iterar y para almacenar resultados intermedios. Las funciones recursivas encajan muy bien para este tipo de casos.

Sin embargo, también tienen inconvenientes. Las funciones recursivas suelen consumir mucha más memoria y tiempo de ejecución cuando el número de veces que se tienen que llamar a sí mismas es grande. Esto hace que además sean más complicadas de depurar si hay algún error. Es más, si no se tiene cuidado al programarlas podemos caer en una recursión infinita, de la que no se pueda salir sin un error o una interrupción externa.

5.2. UN PEZ LLAMADO `LAMBDA...`

En ocasiones, necesitamos utilizar una pequeña función para realizar algún cálculo en un punto específico de nuestro programa y nada más. En casos como éste y otros que te vamos a presentar, sería más cómodo si pudiéramos expresar la función de una forma más sencilla, sin tener que definir el encabezado con su `def`, su nombre, ...

En Python podemos conseguir esto utilizando las funciones *anónimas*, también llamadas funciones o expresiones *lambda*, por la cláusula que se utiliza para definir las.

Una función *anónima* efectivamente comienza con la palabra clave **lambda** (en lugar de **def** como las funciones normales) y no tiene un nombre identificador (de ahí lo de *anónima*). Una función *lambda* puede tener cualquier número de argumentos, pero su cuerpo solamente puede constar de una expresión que devuelva un valor.

```
# creamos una función anónima
# y se la asignamos a la variable 'triple'
triple = lambda x: 3 * x

triple(5)
```

15

Después de la palabra clave **lambda** colocamos todos los parámetros separados por comas. Fíjate que no ponemos paréntesis. Después de la lista de parámetros se ponen dos puntos (:) y seguidamente la expresión que calcula el resultado a devolver.

En este ejemplo, hemos asignado la función anónima que hemos creado a una variable. Después utilizamos esta variable para llamar a la función. Probablemente estarás preguntándote *"¿y para esto hace falta crear las funciones lambda? ¿No hacen ya este papel las funciones normales?"*

Si es así, ¡estarás haciendo muy bien!

El ejemplo anterior era solamente una forma sencilla de mostrarte la sintaxis para definir una función anónima.

Sin embargo, la verdadera utilidad viene cuando tenemos funciones que pueden recibir otras funciones como parámetros. O funciones que devuelven como resultado otra función. Veamos un ejemplo para entender de qué hablamos.

```
def genera_multiplicador(n):  
    """Esta función crea y devuelve a su vez  
    una nueva función que multiplicará  
    cualquier valor por n"""  
    return lambda x: x * n  
  
# vamos a generar una nueva función  
# que multiplique su entrada por 2  
doble = genera_multiplicador(2)  
  
# La variable 'doble' apunta ahora  
# a una función anónima que devuelve  
# su entrada multiplicada por 2  
print( doble(3) )  
print( doble(5) )
```

```
6  
10
```

```
# ahora generamos una nueva función  
# que multiplique su entrada por 3  
triple = genera_multiplicador(3)  
  
# La variable 'triple' apunta ahora  
# a una función anónima que devuelve  
# su entrada multiplicada por 3  
print( triple(7) )  
print( triple(9) )
```

```
21  
27
```

¿Un poco complicado? Repasemos el código paso a paso.

Empezamos definiendo la función `genera_multiplicador`. Es una función normal, con su `def` y que recibe un parámetro `n`.

Lo complejo viene en el cuerpo. En lugar de devolver un valor *"normal"*, como habíamos visto hasta ahora, esta función devuelve una expresión `lambda`, devuelve *otra función*.

En este caso, devuelve una función *lambda* que, a su vez, lo que hace es tomar su parámetro `x` y devolverlo multiplicado por `n`.

Imaginemos este ejemplo de otra manera. Supón que tenemos un almacén en el que vendemos herramientas preparadas para construir formas geométricas. Tenemos una máquina configurable con la que podemos preparar estas herramientas a medida para construir las formas geométricas que necesiten los clientes. Viene un cliente y nos pide una herramienta para construir triángulos de cualquier tamaño. Nosotros vamos a nuestra máquina, la configuramos y producimos una nueva herramienta que está ajustada para construir triángulos, el cliente solo tendrá que indicar el tamaño. Otro cliente nos pide después una herramienta para construir cuadrados de cualquier tamaño. Volvemos a configurar nuestro aparato y obtenemos una nueva herramienta ajustada para que el cliente construya cuadrados del tamaño que le plazca.

Algo parecido es lo que ocurre en este ejemplo. La función `genera_multiplicador` sería nuestra máquina "configurable" (mediante el parámetro `n`) y las funciones anónimas que devuelve serían las "herramientas" preparadas para cada caso concreto.

Solamente hay una sutileza más, que tal vez ya te haya rondado la cabeza. Si el parámetro `n` en realidad está definido en la función `genera_multiplicador`, ¿cómo es que la función `lambda` se acuerda del valor correcto de `n` que tiene que utilizar? En el ejemplo llamamos primero a `genera_multiplicador` con `n=2` y asignamos la función anónima a la variable `doble`. Y después volvemos a llamar a `genera_multiplicador` con `n=3` y asignamos la función anónima devuelta a la variable `triple`.

Muy bien pero, ¿por qué la función anónima que hay en `doble` mantiene el valor `n=2` (lo "recuerda") y no ha cambiado a `n=3`?

De esta especie de "magia" se encarga Python. En realidad, al devolver la función `lambda` dentro de `genera_multiplicador`, Python no solo devuelve una función anónima, si no que la "empaqueta" con un *contexto*, es decir, con los valores que tenía cada variable visible en ese momento.

5.2.1. ... Y FUNCIONES DE ORDEN SUPERIOR

A las funciones que reciben como parámetro otra función, o que devuelven como resultado otra función, se las denomina *funciones de orden superior*.

Existen algunas funciones de orden superior cuyo uso está tan extendido que suelen estar ya incluidas en muchos lenguajes de programación. Se trata de las funciones **map**, **filter** y **reduce**. En Python 2 todas están soportadas de forma nativa. En Python 3 las dos primeras siguen estando incluidas, mientras que la función **reduce** se ha colocado en una librería aparte.

Vamos a explicar cada una de ellas.

map

La función **map** sirve para aplicar (o *mapear*) otra función *f* (que recibe como parámetro) a todos los elementos de una secuencia o colección.

```
cuadrados = map(lambda x: x**2, [1,2,3,4,5])  
  
list(cuadrados)
```

[1, 4, 9, 16, 25]

Como ves, a **map** le pasamos como primer argumento una función. En este caso, una función anónima usando una expresión *lambda* (**lambda x: x**2**) que devuelve su valor de entrada elevado al cuadrado. El segundo argumento de **map** es una colección de elementos. Aquí le pasamos una lista de números ([1,2,3,4,5]) que queremos elevar al cuadrado.

Lo que va a hacer la función **map** es ir tomando uno a uno los valores de la secuencia y ejecutar la función que le hemos pasado como primer argumento con cada uno de estos valores.

map no devuelve inmediatamente la lista con los valores resultantes. En Python 3 lo que devuelve **map** es un tipo especial de objeto que se denomina *iterador*, que es una variante de los *generadores* que explicamos anteriormente.

Como recordatorio, es un objeto que se encarga de hacer los cálculos para *generar* los valores resultantes uno a uno conforme los necesites y se los vayas pidiendo, *iterando* sobre la colección de entrada, en lugar de construir la secuencia entera de golpe, lo que permite ahorrar espacio de memoria ya que no tenemos que almacenar todos los resultados de una vez.

Si realmente queremos tener una lista con todos los valores resultado, basta con hacer lo que aparece en la última línea del ejemplo, construir una lista con `list`.

`filter`

La función `filter` sirve para seleccionar (o *filtrar*) aquellos elementos de una secuencia o colección que cumplen una condición determinada. La condición se le pasa a `filter` en forma de otra función.

```
pares = filter(lambda x: x % 2 == 0, [1,2,3,4,5])  
  
list(pares)
```

[2, 4]

Al igual que `map`, a `filter` le pasamos como primer argumento una función. En el ejemplo, le pasamos una función anónima (`lambda x: x % 2 == 0`) que devuelve *verdadero* si su valor de entrada es un número par, y *falso* en caso contrario. La función que le pasemos a `filter` siempre debe devolver valores `True` o `False`. El segundo argumento de `filter` es de nuevo una colección de elementos.

Lo que va a hacer `filter` en este caso es ir tomando cada uno de los valores de la secuencia y ejecutar la función que le pasamos como primer argumento con cada uno de dichos valores, devolviendo solamente los elementos para los que el resultado sea `True`.

En realidad, `filter` devuelve también un objeto de tipo *iterador*, como ocurre con `map`. Por eso utilizamos `list` al final del ejemplo para obtener una lista con los resultados.

`reduce`

La función `reduce` sirve para ejecutar otra función `f` de forma acumulativa sobre una colección de elementos. Dicho así, a lo mejor no te hemos aclarado demasiado. Mucho mejor con un ejemplo

```
import functools

suma = functools.reduce(lambda x,y: x + y, [1,2,3,4,5])

print(suma)
```

15

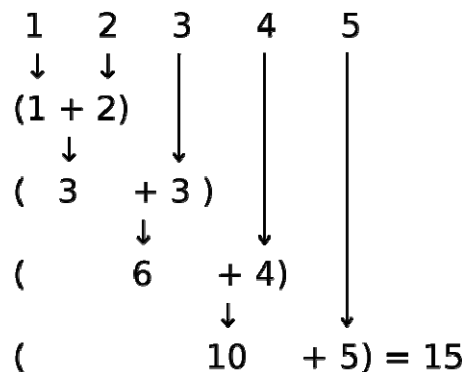
En la primera línea estamos cargando el *módulo* **functools**. Como te decíamos al empezar a hablar de las funciones de orden superior, la función **reduce** estaba incluida por defecto en Python 2, pero en Python 3 se pasó a una librería aparte, en este caso al módulo **functools**. Para cargar componentes de una librería en Python utilizamos la palabra reservada **import**. Más adelante te contaremos más acerca de las librerías en Python.

En la segunda línea ya usamos la función **reduce** del módulo **functools**. Si te fijas, el primer argumento de **reduce** es nuevamente una función (en el ejemplo, la función anónima **lambda x,y: x + y**). El segundo argumento vuelve a ser una colección de valores.

La diferencia aquí reside en que la función que le pasemos a **reduce** debe ser una función que tome *dos argumentos* y devuelva *un solo valor*. En este caso, la función anónima recibe dos valores en **x** e **y** y devuelve la suma de ambos.

¿Y qué hace exactamente **reduce**? Pues empieza tomando los dos primeros elementos de la lista de entrada (**[1,2,3,4,5]**) y ejecuta la función. A continuación, toma el resultado y el siguiente elemento de la lista y vuelve a ejecutar la función. A continuación, toma el nuevo resultado y el siguiente elemento de la lista y vuelve a ejecutar la función. A continuación... Bueno, ya ves como sigue. **reduce** continuará con el proceso hasta agotar todos los elementos de la lista. Al terminar, devolverá el resultado final que haya dado la última evaluación de la función.

Tal vez este diagrama explique más claramente la secuencia de operaciones.



Utilidad de las funciones de orden superior

Las funciones de orden superior pueden ser unas herramientas muy útiles y potentes para ayudarte a resolver algunos problemas.

No obstante, en buena parte de los casos Python te ofrece formas alternativas de conseguir el mismo resultado de una forma más sencilla y legible, utilizando algunas de las construcciones que ya hemos visto a lo largo de esta unidad.

Fijémonos en los ejemplos de `map`, `filter` o `reduce` que hemos mostrado. ¿Se te ocurre cómo podrías resolver cada caso aplicando lo que ya has aprendido?

Empecemos con `map`. En el ejemplo, queríamos elevar los elementos de una lista al cuadrado.

```
# calcular cuadrados usando 'map'
cuadrados = map(lambda x: x**2, [1,2,3,4,5])
print( list(cuadrados) )
```

```
[1, 4, 9, 16, 25]
```

```
# calcular cuadrados usando un generador
cuadrados_2 = ( x**2 for x in [1,2,3,4,5] )
print( list(cuadrados_2) )
```

```
[1, 4, 9, 16, 25]
```

```
# si queremos directamente una lista,
# podemos definirla por comprensión
lista_cuadrados = [ x**2 for x in [1,2,3,4,5] ]
print(lista_cuadrados)
```

```
[1, 4, 9, 16, 25]
```

Podemos conseguir el mismo resultado utilizando una expresión de tipo *generador* con una expresión por comprensión. O si queremos directamente guardar el resultado como una lista, usar una lista por comprensión.

Si recuerdas cuando hablamos de los *generadores*, si intentas volver a extraer valores de un *generador* que ya ha sido *agotado*, obtendrás una lista vacía (`[]`). Esto mismo te ocurrirá con el *iterador* devuelto por `map`. Una vez que ya los has usado, *generando* todos los elementos hasta agotar la secuencia de entrada, ya no hay más valores que devolver.

En cambio con la lista por comprensión, efectivamente, obtienes una lista (no un *generador*) y la tienes disponible mientras la necesites. Tenlo en cuenta cuando vayas a decidir si usar una estructura u otra.

Pasemos al ejemplo con `filter`. El objetivo era quedarnos con los números pares de una lista.

```
# extraer Los números pares usando 'filter'
pares = filter(lambda x: x % 2 == 0, [1,2,3,4,5])
print( list(pares) )
```

```
[2, 4]
```

```
# hacemos Lo mismo usando un generador
pares_2 = ( x for x in [1,2,3,4,5] if x % 2 == 0 )
print( list(pares_2) )
```

```
[2, 4]
```

```
# o si queremos una lista,
# la definimos por comprensión
lista_pares = [ x for x in [1,2,3,4,5] if x % 2 == 0 ]
print(lista_pares)
```

```
[2, 4]
```

Otra vez conseguimos lo mismo utilizando definiciones por comprensión, solo que esta vez añadiendo una cláusula `if` con la misma condición que en la función de filtrado.

Y por último, el ejemplo de `reduce`. En este caso queremos calcular la suma de todos los elementos de una lista.

```
import functools

# calcular la suma de elementos usando 'reduce'
suma = functools.reduce(lambda x,y: x + y, [1,2,3,4,5])
print(suma)
```

15

```
# la forma alternativa de aplicar una función
# de forma acumulativa
# es usando un bucle
otra_suma = 0
for x in [1,2,3,4,5]:
    otra_suma = otra_suma + x

print(otra_suma)
```

15

Si queremos aplicar una función a una colección de valores, acumulando o combinando los resultados, la forma general de hacerlo será mediante un bucle.

Claro que este tipo de operaciones de cálculo acumulado o de reducción a un único valor, como obtener la suma, el mínimo o el máximo, son tan comunes que Python proporciona directamente estas funciones.

```
print( sum([1,2,3,4,5]) )
```

15

```
print( min([1,2,3,4,5]) )
```

1

```
print( max([1,2,3,4,5]) )
```

5

Visto que en Python tenemos alternativas *a priori* más simples y fáciles de entender que las funciones de orden superior, la cuestión ahora es ¿cuándo usar las funciones **map**, **filter** o **reduce**?

A diferencia de otros lenguajes, Python incorpora el mecanismo de definiciones por comprensión, que como hemos visto resulta muy potente a la par que fácil de utilizar.

Los ejemplos que hemos visto son casos típicos para resolverlos utilizando estos generadores o listas por comprensión en Python. De hecho, es la solución más habitual y extendida.

Sin embargo, también habrá casos en los que los cálculos o funciones a aplicar no sean tan simples. Las funciones **map**, **filter** y **reduce** no solo aceptan funciones *lambda*, también aceptan las funciones habituales, siempre que respeten el número de parámetros y el tipo de resultado a devolver.

Es más, la potencia más grande viene normalmente cuando hay que combinar estas operaciones de filtrado, *mapeo* y reducción o acumulación. Se trata de un esquema bastante típico en procesamiento de datos, y en el que podemos aplicar las funciones **map**, **filter** y **reduce** de forma encadenada.

Así que, como en otras ocasiones, la respuesta a cuándo usarlas es "*depende*". Depende del tipo de problema, se adaptará mejor un tipo de solución u otra.



El experto opina

¿Recuerdas las recomendaciones sobre el estilo de programación que hemos ido haciendo? Cuando tengas que decidir entre varias formas de resolver la misma tarea, intenta elegir la solución que simplifique el código aprovechando las herramientas que tienes, haciéndolo también fácil de entender. Python ofrece alternativas para programar de forma elegante, concisa y comprensible.

No obstante, en caso de duda y si el tiempo apremia, utiliza la técnica que mejor domines. Si no estás totalmente seguro de cómo implementar algo con una técnica alternativa, te costará más esfuerzo y es más probable que cometas un error. Si tienes tiempo, entonces ¡adelante, prueba, equivócate y aprende!

6. MÓDULOS Y PAQUETES

Conforme empezamos a desarrollar programas con estructuras cada vez más complejas (variables, funciones, clases y objetos), tenemos la necesidad de poder guardar nuestro código de una forma organizada para poder reutilizarlo, sin tener que rehacer cosas que ya habíamos resuelto.

Los módulos son la forma de organizar nuestro código, agrupando definiciones y funcionalidades relacionadas en unidades que podemos cargar y reutilizar en nuevos programas.

En Python, cada archivo de código terminado con el sufijo `.py` es un módulo. Un módulo de Python puede contener declaraciones de variables y constantes, definiciones de funciones y clases, comentarios.

A continuación te mostramos el contenido de un módulo de ejemplo, incluyendo algunas de las definiciones de función que hemos utilizado hasta ahora.



Atención

Puedes crear un fichero llamado `mimodulo.py` y copiar dentro el código siguiente. O si lo prefieres, puedes encontrar el fichero `mimodulo.py` en el material que acompaña a la unidad. Asegúrate de que el fichero se encuentre en el directorio de fuentes (`U09_src`), y que lo añadiste al *path* como te explicamos al inicio de la unidad, para que esté accesible para la sesión de Python.

```
# %Load U09_src/mimodulo.py
"""
Ejemplo de módulo con definicion de funciones
y declaración de variables
"""

# Definimos funciones del módulo

def fibonacci(n):
    """Calcula el n-ésimo elemento de la serie de Fibonacci"""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

def genera_multiplicador(n):
    """Esta función crea y devuelve a su vez
    una nueva función que multiplicará cualquier valor por n"""
    return lambda x: x * n

# Declaramos variables del módulo

# Asignamos a esta variable la función multiplicadora por 2
doblar = genera_multiplicador(2)

# Asignamos a esta variable la función multiplicadora por 3
triplicar = genera_multiplicador(3)
```

6.1. CARGA DE MÓDULOS Y ESPACIOS DE NOMBRES

Para cargar el contenido de un módulo y tener acceso a sus funciones, clases, variables, etc. utilizamos el comando **import** seguido del nombre del módulo.

```
# vamos a importar nuestro primer módulo
import mimodulo

# ya podemos utilizar sus funciones
mimodulo.fibonacci(10)
```

Como ves, al hacer el `import` no ponemos el nombre del fichero completo con su extensión. Podemos el nombre *del módulo*. En Python el nombre del módulo se toma de la primera parte del nombre del fichero, sin el `.py`.

Una vez que hemos cargado el módulo de esta forma, podemos acceder a su contenido a través del nombre del módulo, siguiendo con un punto (`.`) y el nombre de la función o variable que queremos usar.

¿Pero por qué tenemos que anteponer el nombre del módulo?

Piensa que en nuestras aplicaciones fácilmente acabaremos teniendo muchos módulos, y es posible que tengamos funciones o variables o cualquier otro objeto con el mismo nombre en módulos distintos. ¿Cómo hacer para que no haya confusión al importarlos y usarlos, cómo hacer que no *colisionen*?

Para eso Python utiliza el concepto de *espacios de nombres* (o *namespaces* en inglés). Un espacio de nombres es algo así como un directorio en el que se van anotando los nombres de todos los objetos que vamos creando (variables, funciones, etc.). No puede haber dos objetos con el mismo nombre dentro de un *namespace*.

Cuando iniciamos una nueva sesión de Python con el intérprete interactivo, o al ejecutar un programa o *script*, se crea un nuevo espacio de nombres *global* asociado a esa sesión. En este espacio de nombres se añaden a su vez todos los símbolos, todos los nombres de los objetos predefinidos integrados en el lenguaje (funciones, variables, constantes... como `len`, `sum`, `list` o `map`). Y también se añadirán todos los objetos que nosotros vayamos definiendo directamente en la sesión.

Volviendo a los módulos, en Python cada uno tiene su propio espacio de nombres, que contiene los identificadores de todos los elementos definidos dentro del módulo. Así que puedes tener dos variables o dos funciones con el mismo nombre en distintos módulos. Cada objeto estará incluido en su *namespace* correspondiente y no habrá conflictos entre ellos.

Al importar un módulo también podemos asociarle un alias, normalmente para abreviar el nombre original del módulo.

```
# importamos nuestro módulo, pero con un alias más cortito
import mimodulo as m

# ahora podemos usar el alias 'm' para acceder a las funciones
m.fibonacci(5)
```

También es posible importar directamente un elemento del módulo deseado a nuestro espacio de nombres actual. La sintaxis en este caso sería

```
from nombre_modulo import nombre_elemento

# vamos a importar únicamente la definición de 'doblar' de 'mimodulo'
# incluyéndola en nuestro espacio de nombres actual
from mimodulo import doblar

# ahora podemos usar 'doblar' como si la hubiéramos definido en este nivel
doblar(7)
```

14

Esta variante incluye el objeto indicado (la función `doblar`) en nuestro espacio actual, sin añadir el espacio de nombres del módulo (en este caso, `mimodulo` no estaría definido para acceder a otras funciones).

Hay que tener cuidado al utilizar esta forma de importar elementos, porque puede que ya exista otro objeto con el mismo nombre en nuestro contexto o espacio de nombres actual. Siempre podemos asignar un alias en estos casos.

```
# tenemos una versión local de la función 'triplicar'
def triplicar(x):
    """Función triplicar local"""
    return 3 * x

# vamos a cargar la versión del módulo, pero con un alias
# para que no haya conflictos de nombres
from mimodulo import triplicar as triple

# podemos usar ambas versiones
print( triplicar(5) )
print( triple(5) )
```

15
15

Por último, también podemos importar todos los nombres definidos dentro de un módulo sin utilizar su *namespace*, aunque sin poder asignarles un alias. Para ello, en lugar de la lista de elementos del módulo que queremos importar, utilizamos un asterisco (*) a modo de comodín.

```
from mimodulo import *
```

No obstante, esta última opción es muy desaconsejable, ya que reemplaza todas las definiciones de los nombres ya existentes. Además, dificulta entender el código, al no saber de dónde viene una función o una variable, y complica su mantenimiento.

6.2. LOCALIZANDO LOS MÓDULOS

Cuando le pedimos a Python que importe un módulo, primero tiene que encontrarlo. Para ello sigue una serie de pasos.

- Primero comprueba si se trata de uno de los módulos predefinidos incluidos con el lenguaje.
- Si no existe ningún módulo del lenguaje con ese nombre, busca el fichero con el módulo en el directorio que contiene el *script* de Python actual, o el directorio de ejecución de la sesión actual si estamos en modo interactivo.
- Si no lo encuentra en el directorio actual, consulta la variable entorno `PYTHONPATH`. Se trata de una variable que hay que definir a nivel de sistema operativo, y donde se puede indicar una lista de directorios donde buscar módulos y librerías instaladas.
- Por último, busca en el directorio donde esté instalado Python.

Podemos ver la secuencia de directorios donde buscará de la siguiente forma

```
import sys
sys.path
```

Cuando desarrollemos nuestros módulos, tendremos que asegurarnos que sean accesibles desde el directorio actual o bien añadir los directorios donde se encuentren a la secuencia de búsqueda, bien mediante la variable de entorno `PYTHONPATH` o bien modificando la variable `sys.path` dentro del código.

6.3. MÓDULOS EJECUTABLES

Podemos ejecutar un módulo como si fuera un fichero de *script* normal.

```
$ python mimodulo.py
```

Si el módulo únicamente contiene definiciones de variables, funciones, clases, etc. lo único que ocurrirá es que se cargarán todas ellas en un *namespace*, y seguidamente terminará la ejecución.

Cuando se lanza la ejecución de un *script* o un módulo, Python prepara el *namespace* para la ejecución y le da el nombre especial `"__main__"`.

Si queremos incluir en un módulo un código para que se ejecute solamente cuando lo lancemos como *script*, podemos consultar cuál es el nombre del *namespace* actual utilizando la variable global `__name__`.

Veamos el siguiente ejemplo de módulo con sección *main*.

```
# %Load U09_src/mimoduloexec.py
"""
Ejemplo de módulo ejecutable
"""

# Definición de funciones

def llama_a_la_puerta_de(persona, n_veces=3):
    """Llama a la puerta de la persona las veces que haga falta"""
    return [ "Toc,toc,toc! " + persona + "!" for i in range(n_veces) ]

# Cuerpo principal ejecutable

# si estamos dentro del namespace __main__
if __name__ == "__main__":
    # ejecuta cosas!!
    print("Ejecutamos el main")
    print(llama_a_la_puerta_de("Penny"))
```

Prueba a ejecutar el anterior módulo.

```
$ python mimoduloexec.py
```

Deberías ver cómo se imprimen los distintos mensajes.

En cambio, si lo importas desde una sesión interactiva o desde otro módulo, simplemente cargará las definiciones, sin ejecutar nada más.

```
import mimoduloexec
```

6.4. PAQUETES

En Python los paquetes son la forma de organizar un conjunto de módulos relacionados entre sí.

Igual que un módulo se corresponde con un archivo de código, los paquetes se estructuran como directorios o carpetas. Solo deben cumplir con un requisito adicional. Para que Python reconozca que se trata de un paquete de módulos, dentro del directorio debe incluirse un archivo especial de inicialización con el nombre `__init__.py`.

De entrada, no es necesario que el archivo `__init__.py` contenga nada, podemos dejarlo completamente vacío. Conforme vayamos ampliando el contenido de nuestro paquete, podremos incluir dentro de `__init__.py` código de configuración e importación de otros módulos que se necesiten en el paquete.

Veamos un ejemplo de estructura de directorios y ficheros para una colección de paquetes y módulos que hemos creado (puedes acceder a estos ficheros en el material que acompaña a esta unidad)

```
paqtema09/  
├── __init__.py  
├── modulo0901.py  
└── subtema09  
    ├── __init__.py  
    ├── modulo0901.py  
    └── modulo0902.py
```

Aquí hemos creado un paquete llamado `paqtema09`. Como corresponde, hemos añadido un fichero `__init__.py` dentro del directorio para que Python lo reconozca. También hemos añadido un módulo llamado `modulo0901`, creando su correspondiente fichero `.py`.

Como puedes ver, también es posible crear *subpaquetes*, es decir, paquetes anidados dentro de paquetes. Es como cualquier organización de directorios o carpetas, no hay nada especial. Aquí tenemos el *subpaquete* `subtema09` con su `__init__.py` dentro. Además, le hemos añadido dos módulos, `modulo0901` y `modulo0902`.

¿Te fijaste? Tenemos dos módulos con el mismo nombre (`modulo0901`), pero no hay problema porque uno está en el paquete `paqtema09` y otro en el *subpaquete* `subtema09`. Es como tener ficheros con el mismo nombre en carpetas distintas, no hay colisión.

Ahora veamos como cargar un paquete y su contenido. Utilizamos `import`, de la misma forma que con los módulos.

```
import paqtema09
```

En este caso concreto, esto tampoco es muy útil. El paquete en sí no contiene nada, las definiciones están dentro de los módulos. Si en `__init__.py` incluyéramos definiciones y carga de módulos, sí que los tendríamos accesibles a través de `paqtema09`.

En su lugar, probemos a cargar los módulos. En los tres módulos que hay definimos una constante `CONSTANTE_T09`, con un valor distinto en cada módulo.

```
import paqtema09.modulo0901
import paqtema09.subtema09.modulo0901
import paqtema09.subtema09.modulo0902

print(paqtema09.modulo0901.CONSTANTE_T09)
print(paqtema09.subtema09.modulo0901.CONSTANTE_T09)
print(paqtema09.subtema09.modulo0902.CONSTANTE_T09)
```

Definida en `paqtema09.modulo0901`

Definida en `paqtema09.subtema09.modulo0901`

Definida en `paqtema09.subtema09.modulo0902`

Como ves, aquí entran de nuevo los espacios de nombres. Igual que en los módulos, los paquetes tienen asociado su *namespace* propio. Python utiliza esta jerarquía de *namespaces* para resolver los nombres de los objetos y saber cuál tiene que usar.

Podemos utilizar los alias para abreviar

```
import paqtema09.modulo0901 as m1a
import paqtema09.subtema09.modulo0901 as m1b
import paqtema09.subtema09.modulo0902 as m2

print(m1a.CONSTANTE_T09)
print(m1b.CONSTANTE_T09)
print(m2.CONSTANTE_T09)
```

Definida en paqtema09.modulo0901

Definida en paqtema09.subtema09.modulo0901

Definida en paqtema09.subtema09.modulo0902

O utilizar la construcción `from .. import ..` para cargar directamente un módulo en nuestro espacio de nombres, ahorrándonos tener que escribir la jerarquía completa de paquetes (eso sí, recuerda tener cuidado para que no colisionen nombres de objetos).

```
from paqtema09.subtema09 import modulo0901
from paqtema09.subtema09 import modulo0902

print(modulo0901.CONSTANTE_T09)
print(modulo0902.CONSTANTE_T09)
```

Definida en paqtema09.subtema09.modulo0901

Definida en paqtema09.subtema09.modulo0902

6.5. ALGUNOS MÓDULOS INCLUIDOS DE SERIE

Python viene de serie con una ingente colección de paquetes y módulos conocida como *Python Standard Library* (biblioteca estándar de Python), y que ofrecen gran cantidad de herramientas y funcionalidades muy útiles para ayudarnos en nuestros programas.

Vamos a revisar unas pocas de ellas, pero no veremos todo ni entraremos en mucho detalle. Puedes investigar más en los siguientes enlaces:

- <https://docs.python.org/3/library/index.html>
- <http://docs.python.org.ar/tutorial/3/stdlib.html>
- <http://docs.python.org.ar/tutorial/3/stdlib2.html>

En cualquier caso, no hace falta que te aprendas todos estos módulos y sus funcionalidades de memoria. Es suficiente con que te vayas familiarizando con ellos y con lo que te ofrecen puedes. Siempre podrás consultar la documentación en cualquier momento para resolver tus dudas y refrescar cómo usarlos.

6.5.1. MÓDULOS DEL SISTEMA

os

Este módulo incluye funciones básicas para acceder e interactuar al sistema operativo.

```
import os
```

	Descripción
<code>os.environ</code>	Permite acceder a las variables de entorno; p.ej. <code>os.environ['HOME']</code>
<code>os.chdir()</code>	Cambiar el directorio de trabajo de la sesión actual
<code>os.mkdir()</code>	Crear nuevos directorios
<code>os.rmdir()</code>	Borrar directorios
<code>os.listdir()</code>	Devuelve un listado con el contenido de un directorio
<code>os.open()</code>	Abrir un fichero
<code>os.close()</code>	Cerrar un fichero

os.path

Este módulo nos permite realizar consultas sobre nombres de ficheros y directorios

```
import os.path
```

	Descripción
<code>os.path.basename()</code>	Devuelve el nombre del último elemento (directorio/fichero) de una ruta
<code>os.path.dirname()</code>	Devuelve la cadena de directorios hasta el último elemento de la ruta
<code>os.path.abspath()</code>	Devuelve la ruta absoluta completa hasta un directorio o fichero
<code>os.path.isdir()</code>	Comprueba si una ruta corresponde a un directorio
<code>os.path.isfile()</code>	Comprueba si una ruta corresponde a un fichero

sys

Este módulo da acceso a variables y funciones relacionadas con el intérprete y su estado

```
import sys
```

	Descripción
<code>sys.path</code>	Listado de directorios de búsqueda de módulos
<code>sys.modules</code>	Diccionario con los módulos cargados actualmente
<code>sys.version</code>	Versión de Python ejecutándose
<code>sys.argv</code>	Listado de parámetros pasados al <i>script</i> Python por línea de comandos
<code>sys.stdin</code>	Referencia a la entrada estándar de datos
<code>sys.stdout</code>	Referencia a la salida estándar de datos
<code>sys.stderr</code>	Referencia a la salida estándar de errores
<code>sys.exit()</code>	Terminar la ejecución de Python y salir al sistema operativo

6.5.2. MÓDULOS DE TIPOS DE DATOS

datetime

Este módulo incluye clases y funciones básicas para manipular fechas y horas

```
import datetime as dt
```

	Descripción
<code>dt.date()</code>	Crea un objeto que representa una fecha (con año, mes y día)
<code>dt.time()</code>	Crea un objeto que representa una hora del día (con horas, minutos y segundos)
<code>dt.datetime()</code>	Combinación de date y time
<code>dt.date.today()</code>	Devuelve un objeto date con la fecha de hoy
<code>dt.datetime.now()</code>	Devuelve un objeto datetime que representa el instante actual

calendar

Este módulo permite formatear y hacer operaciones con fechas de calendario

6.5.3. MÓDULOS ADICIONALES PARA CADENAS DE TEXTO

string

Este módulo proporciona constantes y funciones adicionales para formatear cadenas de texto

```
import string
```

	Descripción
<code>string.ascii_lowercase</code>	Secuencia de caracteres alfabéticos en minúscula
<code>string.ascii_uppercase</code>	Secuencia de caracteres alfabéticos en mayúscula
<code>string.digits</code>	Secuencia de caracteres numéricos
<code>string.Formatter</code>	Clase para dar formatos avanzados a cadenas de texto

re

Este módulo proporciona las funcionalidades para trabajar con expresiones regulares en Python. Las expresiones regulares son una forma de expresar patrones de búsqueda mediante una secuencia de caracteres. Se trata de una herramienta muy potente para realizar búsquedas, extracciones, sustituciones y otras operaciones de edición sobre cadenas de texto. Si no las has usado nunca y no conoces cómo funcionan, no te preocupes, tarde o temprano te topará con ellas. Solo recuerda que Python te permite usarlas.

```
import re
```

	Descripción
<code>re.compile()</code>	Compila una expresión regular para poder utilizarla en búsquedas
<code>re.search()</code>	Busca la primera ocurrencia de una expresión regular
<code>re.match()</code>	Busca la expresión regular y devuelve los elementos con correspondencia al patrón
<code>re.split()</code>	Divide una cadena de texto en porciones en base al patrón

6.5.4. MÓDULOS MATEMÁTICOS

`math`

Este módulo ofrece varias funciones matemáticas de uso extendido

```
import math
```

	Descripción
<code>math.ceil()</code>	Devuelve el número entero más pequeño que es mayor o igual que un valor
<code>math.floor()</code>	Devuelve el número entero más grande que es menor o igual que un valor
<code>math.gcd()</code>	Devuelve el máximo común divisor de dos números
<code>math.isinf()</code>	Comprueba si es un valor infinito
<code>math.trunc()</code>	Devuelve la parte entera de un número en punto flotante
<code>math.exp()</code>	Devuelve el valor de e^x
<code>math.log()</code>	Devuelve el logaritmo de x (en base e)
<code>math.sqrt()</code>	Devuelve la raíz cuadrada de un valor
<code>math.cos()</code>	Devuelve el coseno de un valor
<code>math.sin()</code>	Devuelve el seno de un valor
<code>math.tan()</code>	Devuelve la tangente de un valor
<code>math.pi</code>	Constante con el valor del número π

	Descripción
<code>math.e</code>	Constante con el valor del número e
<code>math.inf</code>	Constante que representa Infinito

random

Este módulo proporciona funcionalidades para generar números pseudo-aleatorios, trabajar con diferentes distribuciones de probabilidad y simular procesos y muestreos estocásticos.

```
import random
```

	Descripción
<code>random.seed()</code>	Inicializar la <i>semilla</i> del generador de números aleatorios
<code>random.randint()</code>	Devuelve un número entero aleatorio entre dos valores
<code>random.choice()</code>	Devuelve un elemento elegido aleatoriamente de una secuencia
<code>random.shuffle()</code>	Mezcla aleatoriamente los elementos de una secuencia
<code>random.sample()</code>	Extrae una muestra aleatoria de k elementos de una secuencia
<code>random.random()</code>	Devuelve un número aleatorio en punto flotante entre 0 y 1
<code>random.uniform()</code>	Devuelve un número aleatorio a partir de una distribución uniforme
<code>random.expovariate()</code>	Devuelve un número aleatorio a partir de una distribución exponencial
<code>random.gauss()</code>	Devuelve un número aleatorio a partir de una distribución normal

6.5.5. MÓDULOS PARA PROGRAMACIÓN FUNCIONAL

functools

Este módulo incluye utilidades para facilitar el trabajo con funciones de orden superior

```
import functools as fn
```

	Descripción
<code>fn.reduce()</code>	Implementa la función de orden superior de <i>reducción</i>

itertools

Este módulo ofrece una serie de *iteradores* muy útiles para construir ciertos bucles de forma eficiente

```
import itertools as it
```

	Descripción
<code>it.cycle()</code>	Itera indefinidamente de forma cíclica sobre los elementos de una secuencia
<code>it.repeat()</code>	Itera repitiendo un mismo valor
<code>it.accumulate()</code>	Itera sobre los valores acumulados de una secuencia
<code>it.product()</code>	Itera sobre el producto cartesiano de dos secuencias
<code>it.permutations()</code>	Itera sobre las permutaciones de k elementos de una secuencia
<code>it.combinations()</code>	Itera sobre las combinaciones únicas ordenadas de k elementos de una secuencia

operator

Este módulo proporciona operadores comunes para su uso con funciones de orden superior, en lugar de expresiones *lambda*. Enumeramos solo algunos de los que incluye.

```
import operator as op
```

	Descripción
<code>op.le</code>	Operación $x \leq y$
<code>op.eq</code>	Operación $x = y$
<code>op.abs</code>	Operación valor absoluto
<code>op.add</code>	Operación $x + y$

<code>op.sub</code>	Operación $x - y$
<code>op.mul</code>	Operación $x * y$
<code>op.pow</code>	Operación x^y

6.5.6. MÓDULOS PARA FICHEROS

`fileinput`

Este módulo proporciona funciones para iterar leyendo líneas de uno o más ficheros de entrada.

```
import fileinput as fin
```

	Descripción
<code>fin.input()</code>	Itera leyendo línea a línea de todos los ficheros de entrada indicados

`csv`

Este módulo proporciona funciones para leer y escribir ficheros en formato CSV.

```
import csv
```

	Descripción
<code>csv.reader()</code>	Devuelve un objeto para leer el contenido de un fichero CSV determinado
<code>csv.writer()</code>	Devuelve un objeto para escribir datos en un fichero con formato CSV

6.6. INSTALACIÓN DE LIBRERÍAS EXTERNAS

En muchos casos, Python y su biblioteca estándar nos proporcionarán todos los elementos que necesitamos para construir nuestros programas.

Sin embargo, conforme tengamos que resolver tareas más complejas, acabaremos necesitando incluir librerías y módulos desarrollados por terceros. Como ya comentamos al principio de esta unidad, Python cuenta con una enorme comunidad de usuarios que colaboran y aportan nuevos módulos de forma libre para que todo el mundo pueda beneficiarse. Puedes empezar a explorar toda la variedad de librerías disponibles en la web de Pypi (*The Python Package Index* - <https://pypi.python.org/pypi>).

Para instalar y administrar los módulos que tenemos en nuestra máquina, contamos con la herramienta **pip**. Actualmente, **pip** está incluida ya en las instalaciones de Python a partir de las versiones 2.7.9 (para Python 2) y 3.4 (para Python 3). En versiones anteriores, es posible instalar **pip** siguiendo las instrucciones incluidas en la página web <https://pip.pypa.io/en/stable/installing/>

Una vez que la herramienta está instalada, utilizar **pip** es muy sencillo. Se trata de una utilidad de línea de comandos, así que tendremos que abrir una consola o terminal.

Para instalar un nuevo módulo utilizamos el comando **install** de **pip**:

```
$ pip install ModuloNuevo
```

Como ves, ni siquiera es necesario descargar los ficheros del módulo. Este comando automáticamente se conecta al repositorio de Pypi para obtener el módulo y procede a instalarlo. Si el nuevo módulo a su vez depende de otros para funcionar, estas dependencias también se descargarán e instalarán de forma automática.

En la tabla siguiente tienes un listado de los comandos más habituales de **pip**

Comando	Descripción
<code>pip install Modulo</code>	Instalar un módulo obtenido del repositorio
<code>pip install --upgrade Modulo</code>	Actualizar la versión de un módulo ya instalado
<code>pip uninstall Modulo</code>	Desinstalar un módulo
<code>pip list</code>	Listar los módulos instalados
<code>pip show Modulo</code>	Mostrar información sobre un módulo instalado
<code>pip search patron</code>	Buscar información sobre módulos en el repositorio

7. ENTRADA Y SALIDA DE DATOS

En la mayoría de los programas es necesario poder cargar unos datos de entrada y almacenar los resultados de salida. En esta sección vamos a ver cómo utilizar los mecanismos básicos de manejo de ficheros en Python para realizar estas tareas.

Python nos permite trabajar con un fichero a través de un objeto **File**. Un objeto **File** representa una conexión a un fichero determinado, y nos permite leer su contenido o escribir en él.

Veamos un ejemplo sencillo.

```
# Ajusta La ubicación del directorio `U09_aux`  
# incluido en el material de la unidad  
# que descargaste del campus !!  
DIR_U09_AUX = os.path.join(".", "U09_aux")  
  
nombre_fichero = os.path.join(DIR_U09_AUX, "lorem_ipsum.txt")  
  
# Abrimos un fichero de texto con la función open()  
f = open(nombre_fichero, "r")  
  
# Tenemos un objeto file en la variable f  
# Leemos una línea del fichero usando readline()  
linea_1 = f.readline()  
print(linea_1)  
  
# Cuando hemos terminado, debemos cerrar el fichero  
f.close()
```

Lorem ipsum dolor sit amet,

Para empezar, tenemos que abrir el fichero. Para ello utilizamos la función `open()`. El primer argumento de esta función es el nombre del fichero que queremos abrir. Si el fichero está en un directorio diferente de donde se ejecuta la sesión de Python, tendremos que indicar la ruta completa de directorios. El segundo argumento indica *en qué modo* queremos abrir el fichero, esto es, si queremos abrirlo solo para lectura (pero no podremos escribir en él), o para lectura y escritura, si queremos vaciar su contenido antes de empezar a escribir, o queremos añadir contenido al final...

A continuación indicamos los códigos que se utilizan para los distintos *modos* aceptados

Modo	Descripción
<code>r</code>	Abrir solo para lectura, desde el inicio del fichero
<code>r+</code>	Abrir para lectura y escritura, desde el inicio del fichero
<code>w</code>	Abrir solo para escritura. Si el fichero no existe, lo crea primero. Si ya existe, sobrescribe el contenido
<code>w+</code>	Abrir para escritura y lectura. Si el fichero no existe, lo crea primero. Si ya existe, sobrescribe el contenido
<code>a</code>	Abrir para añadir al final del fichero. Si el fichero no existe, lo crea primero.
<code>a+</code>	Abrir para añadir al final del fichero y para lectura. Si el fichero no existe, lo crea primero.

Estos modos sirven para trabajar con ficheros cuyo contenido esté codificado como texto. Si queremos trabajar con datos binarios (por ejemplo, una imagen), añadimos el modificador `b` a la primera letra del modo (p.ej. `"rb"`, `"rb+"` o `"wb+"`).

La función `open()` nos devuelve un objeto `file` que usaremos para acceder y manipular su contenido.

Para leer el contenido, disponemos de varias funciones

Función	Descripción
<code>f.read(n)</code>	Lee <code>n</code> caracteres o bytes del fichero
<code>f.readline()</code>	Lee una nueva línea del fichero
<code>f.readlines()</code>	Lee todas las líneas del fichero

Veamos un ejemplo de cómo leer el contenido completo de un fichero de texto.

```
# Abrimos un fichero de texto con la función open()
f = open(nombre_fichero, "r")

# Leemos todas las líneas del fichero
lineas = f.readlines():
print(lineas)

# Cuando hemos terminado, debemos cerrar el fichero
f.close()
```

La función `f.readlines()` lee todas las líneas y las devuelve en una lista. En realidad, si lo que necesitamos es iterar y procesar secuencialmente cada una de las líneas, cargar todo el contenido del fichero de golpe en memoria no suele ser la mejor opción, sobre todo si el fichero es grande. Una forma más eficiente y rápida es la siguiente.

```
# Abrimos un fichero de texto con la función open()
f = open(nombre_fichero, "r")

# Iteramos sobre las líneas del fichero
for linea in f:
    print(linea)

# Cerramos el fichero
f.close()
```

Para escribir datos en un fichero podemos usar las siguientes funciones

Función		Descripción	
<code>f.write(datos)</code>		Escribe el contenido de datos en el fichero	
<code>f.writelines(lineas)</code>		Escribe el contenido de la lista de lineas en el fichero	

Cuando utilicemos estas funciones tenemos que saber que no añaden los caracteres delimitadores de líneas al final de cada una de forma automática. Si el propio contenido no incluye los caracteres de fin de línea, tendremos que añadirlos nosotros antes (si es que estamos escribiendo ficheros de texto, claro).

```

# Abrimos el fichero con Los datos de entrada
f_in = open(nombre_fichero, "r")
# y un fichero en el que guardar Los resultados
f_out = open("resultado.txt", "w")

# Iteramos cada línea del fichero de entrada
for linea in f_in:
    # Dividimos cada línea en palabras
    palabras = linea.split()
    # Contamos cuantas palabras hay
    n_palabras = len(palabras)
    # Escribimos el número de palabras en el fichero de salida
    f_out.write(str(n_palabras))
    # Añadimos el carácter de fin de línea
    f_out.write("\n")

# Cuando hemos terminado, debemos cerrar Los fichero
f_in.close()
f_out.close()

```

Cuando ejecutes este fragmento de código, busca y abre el fichero **resultado.txt** y comprueba su contenido. Fíjate también en cómo escribimos en el fichero el número de palabras que hemos contado en cada línea. No escribimos directamente la variable **n_palabras**. Convertimos el valor a cadena de texto utilizando **str()**. La razón es que si abrimos un fichero para escritura en modo texto ("w"), debemos pasar cadenas de texto a la función **f.write()**. De lo contrario, se producirá un error.

Como habrás visto, cada nueva operación de lectura y escritura que ejecutamos comienza a partir de la última posición accedida del fichero. Esto es el comportamiento que necesitaremos habitualmente. En algunos casos, puede que queramos volver al inicio del fichero, o saber en qué posición del fichero nos encontramos. Para eso tenemos las funciones **f.seek()** y **f.tell()**.

Función	Descripción
f.seek(posicion [, origen])	Desplazarse a una posicion dentro del fichero, relativa al origen
f.tell()	Devolver la posición actual dentro del fichero

En la función `f.seek()` el argumento `origen` es opcional y puede tomar los siguientes valores

Origen en <code>f.seek()</code>	Descripción
0	Desplazarse desde el inicio del fichero
1	Desplazarse desde la posición actual del fichero
2	Desplazarse desde el final del fichero

En los dos últimos casos, el argumento `posicion` de `f.seek()` puede tomar valores negativos. Así le indicamos que queremos desplazarnos hacia atrás desde la posición actual (`origen = 1`) o desde el final del fichero (`origen = 2`). Veamos un ejemplo sencillo.

```
# Abrimos un fichero de texto con la función open()
f = open(nombre_fichero, "r")

# Nada más abrir el fichero para lectura estaremos en la posición cero
pos_0 = f.tell()
print("Pos. inicial:", pos_0)
```

```
Pos. inicial: 0
```

```
# Leemos la primera línea del fichero
linea_1 = f.readline()
pos_1 = f.tell()
print("Línea 1: ", linea_1)
print("Tras leer línea 1, pos:", pos_1)
```

```
Línea 1: Lorem ipsum dolor sit amet,
Tras leer línea 1, pos: 28
```

```
# Leemos la segunda línea del fichero
linea_2 = f.readline()
pos_2 = f.tell()
print("Línea 2: ", linea_2)
print("Tras leer línea 2, pos:", pos_2)
```

```
Línea 2: consectetur adipiscing elit,
Tras leer línea 2, pos: 57
```

```
# Nos volvemos a colocar al inicio del fichero
f.seek(0)
print("Volvemos al inicio del fichero")
```

```
0
Volvemos al inicio del fichero
```

```
# Leemos la primera línea de nuevo
linea_1_bis = f.readline()
pos_1_bis = f.tell()
print("Linea 1 (bis):", linea_1_bis)
print("Tras leer linea 1 (bis), pos:", pos_1_bis)
```

```
Linea 1 (bis): Lorem ipsum dolor sit amet,
Tras leer linea 1 (bis), pos: 28
```

```
# Ahora nos desplazamos a continuación de la segunda línea
f.seek(pos_2)
print("Saltamos tras la linea 2")
linea_3 = f.readline()
pos_3 = f.tell()
print("Linea 3: ", linea_3)
print("Tras leer linea 3, pos:", pos_3)
```

```
57

Saltamos tras la linea 2
Linea 3:  sed eiusmod tempor incididunt

Tras leer linea 3, pos: 85
```

```
# Cuando hemos terminado, debemos cerrar el fichero
f.close()
```

A lo largo de todos estos ejemplos hemos visto que debemos asegurarnos de cerrar un fichero una vez que hemos terminado de trabajar con él. Existe una forma alternativa de trabajar con ficheros en Python, más elegante y que se encarga de cerrar automáticamente el fichero, utilizando la cláusula **with**. Veamos cómo.

```
# Abrimos el fichero dentro de un bloque `with`
with open(nombre_fichero, "r") as f:
    # Dentro del bloque, podemos usar el objeto File de forma normal
    for linea in f:
        print(linea)
```

```
# Al salir del bloque `with`, el fichero se cierra automáticamente
print("¿El fichero está cerrado?", f.closed)
```

```
Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed eiusmod tempor incididunt  
ut labore et dolore magna aliqua.
```

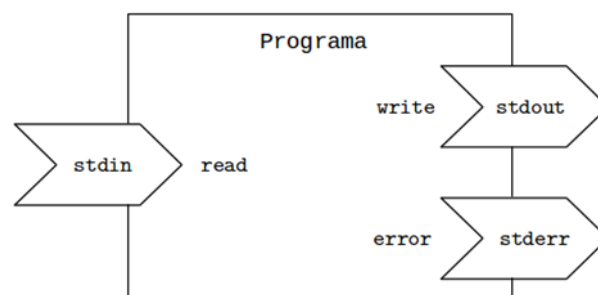
```
¿El fichero está cerrado? True
```

La cláusula **with** nos permite definir e inicializar un *alias* a partir de un objeto o expresión, de forma que podamos utilizarlo dentro del bloque de código que sigue a la cláusula. Al terminar el bloque, se ejecutan de forma automática las operaciones de finalización que tenga asociadas el objeto.

En este caso, usamos la cláusula **with** asociando el objeto **File** devuelto por la función **open()** al *alias* **f**. Dentro del bloque **with** podemos acceder al fichero usando **f** como una variable normal, de las formas que ya hemos visto. La diferencia viene cuando se terminan de ejecutar todas las operaciones del bloque **with**. Cuando se llega al final del bloque, este se encarga de hacer el **close()** sobre el fichero de forma automática.

7.1. LA ENTRADA Y SALIDA ESTÁNDAR

Si eres un usuario habitual de Linux o Unix (o incluso de Mac OS), probablemente ya estarás familiarizado con los conceptos de la entrada y salida estándar de un proceso.



Sin embargo, si solo has sido usuario de Windows o de Mac OS sin utilizar mucho la consola o terminal de línea de comandos, es fácil que no sepas a qué nos estamos refiriendo.

Podemos ver la entrada y salida estándar como unas conexiones a ficheros que se abren de forma automática para cualquier proceso del que se lance la ejecución.

La entrada estándar (comunmente **stdin**, por *standard input*) es una conexión solo de lectura a través de la que un programa puede recibir y leer datos generados desde el entorno en el que se lanzó. Por ejemplo, cuando lanzamos un programa desde el terminal o consola de texto, la conexión **stdin** recibe la entrada generada desde el teclado.

La salida estándar (**stdout**, por *standard output*) es la conexión solo de escritura en la que se escribe por defecto cualquier mensaje o resultado de salida, cuando no se utiliza explícitamente otro fichero o conexión. Por ejemplo, cuando hacemos un **print()** en Python, se escribe en **stdout**. Si estamos ejecutando el programa desde un terminal o consola de texto, veremos la salida en el terminal.

Existe además una tercera conexión, la salida de error estándar o **stderr**. Es una conexión similar a **stdout**, solo de escritura, y que se utiliza como el canal por defecto en el que se escriben los mensajes de error, excepciones, etc.

En Python podemos acceder a estos ficheros a través del módulo **sys**.

Objeto	Funciones
sys.stdin	<code>sys.stdin.read()</code> <code>sys.stdin.readline()</code> <code>sys.stdin.readlines()</code>
sys.stdout	<code>sys.stdout.write()</code> <code>sys.stdout.writelines()</code>
sys.stderr	<code>sys.stderr.write()</code> <code>sys.stderr.writelines()</code>

El programa **loop_lee_stdin.py** es un pequeño ejemplo de cómo utilizar **stdin** y **stdout**.

```
# %load U09_src/loop_lee_stdin.py
"""
Ejemplo de lectura de datos de la entrada estándar,
capturada desde teclado al lanzar el script por línea de comandos
"""

import sys
```

```

# Escribimos un mensaje inicial para el usuario en STDOUT
sys.stdout.write("Prueba a escribir (una línea vacía sirve para salir)
...\n")

while True:

    # Leemos de STDIN
    x = sys.stdin.readline()

    # Comprobamos si no es una línea vacía
    if (len(x) > 0) and (x != '') and (x != '\n'):
        mensaje = "He leído {}".format(x)
        sys.stdout.write(mensaje)
    else:
        # Es una línea vacía, salimos
        sys.stderr.write("No hay más datos\n")
        break

```

Para probar el programa, abre un terminal, ve al directorio con el material de la unidad donde se encuentran los ficheros con código fuente y ejecuta

```
$ python loop_lee_stdin.py
```

Al ejecutarlo, el programa mostrará en la consola el mensaje inicial y se quedará esperando a que escribas algo utilizando el teclado. Cuando pulses INTRO, se completará la línea y la función `readline()` de `sys.stdin` devolverá la cadena a la variable `x`, para escribir a continuación un mensaje en la salida estándar con el contenido leído. Esto se repetirá infinitamente, hasta que introduzcas una línea vacía.



+ Info

Aunque este ejemplo no te parezca muy útil, la entrada y salida estándar se utilizan en multitud de programas para procesar datos. Especialmente para encastrar la salida de un proceso (`stdout`) y usarla como entrada de otro proceso (`stdin`) y construir secuencias de procesamiento de datos complejas a partir de procesos individuales muy sencillos. Esta técnica recibe el nombre de *data pipelines* o tuberías de datos, y es una de las bases del procesamiento *Big Data*.

8. CLASES Y PROGRAMACIÓN ORIENTADA A OBJETOS

Python es un lenguaje orientado a objetos (OOL - *Object Oriented Language*). Como tal, incorpora las herramientas para definir nuestras propias clases, instanciar y operar con objetos, o gestionar aspectos como la herencia o el polimorfismo.

8.1. ¿ORIENTACIÓN A OBJETOS?

Si todos estos conceptos te resultan totalmente desconocidos, vamos a tratar de explicarlo por encima.

Puedes ver un objeto como una representación de un objeto del mundo real, con ciertas propiedades y capacidades o funcionalidades concretas.

Una clase no deja de ser una abstracción, un patrón de cierta familia de elementos, que nos dice cuáles son las propiedades y funcionalidades comunes que comparten todos los objetos de dicha clase.

En programación orientada a objetos, a las propiedades las llamamos *atributos* y a las funcionalidades o capacidades las llamamos *métodos*. Y al proceso de crear un nuevo objeto a partir de una clase lo denominamos *instanciación*.

8.2. DEFINIENDO CLASES Y OBJETOS

Para crear una nueva clase empleamos la palabra reservada `class`.

```
class Coche:
    '''Esta clase representa un coche'''

    # Atributo de clase
    ruedas = 4

    # Método de clase
    def comprueba_coche():
        print("Soy un coche.")

print(Coche.ruedas)
Coche.comprueba_coche()
```

```
4
Soy un coche.
```

Como ves, es simple. Una vez que definimos una nueva clase, su nombre queda registrado y se le asigna un espacio de nombres propio. A través de su nombre podemos acceder a la clase y a sus elementos.

Aquí hemos definido un atributo de clase (`ruedas`) cuyo valor compartirán todos los objetos de clase `Coche`. Estos atributos se definen dentro de la clase igual que una variable normal.

Del mismo modo, hemos definido un método de clase, cuya funcionalidad compartirán también todos los objetos de clase `Coche`. Un método de clase se define igual que una función común.

¿Cómo creamos nuevos objetos de una clase? Invocando la clase como si fuera una función.

```
c1 = Coche()
print(c1)
```

```
<__main__.Coche object at 0x7f9d9433c0b8>
```

8.2.1. CONSTRUCTORES

Los constructores son métodos especiales de una clase que permiten configurar distintas propiedades de los objetos durante su instanciación. En Python, el constructor de una clase se define siempre con una función con el nombre `__init__`.

```
class Coche:
    '''Esta clase representa un coche'''

    # Atributo de clase
    ruedas = 4

    def __init__(self, potencia, velocidad_max, color):
        '''Constructor de Coche'''
        # Inicializamos atributos propios del nuevo objeto
        self.potencia = potencia
        self.velocidad_max = velocidad_max
        self.color = color

    # Método de clase
    def comprueba_coche():
        print("Soy un coche.")
```

El método constructor recibe siempre como primer argumento una referencia al nuevo objeto (argumento `self`). Adicionalmente, podemos añadir más argumentos arbitrarios. El propósito principal de un constructor es declarar e inicializar los atributos del objeto.

La forma de definir los atributos es creándolos a través del objeto `self`, como nuevas propiedades.

```
# Creamos un nuevo objeto
coche_1 = Coche(120, 200, "rojo")

# Accedemos a sus atributos directamente
print(coche_1.color)
```

```
rojo
```

Ten en cuenta que los atributos de dos instancias distintas no comparten el valor de sus atributos.

```
# Creamos un nuevo objeto
coche_2 = Coche(150, 220, "azul")

# Accedemos a sus atributos directamente
print(coche_2.color)
```

```
azul
```

8.2.2. DEFINIENDO MÉTODOS

Para definir métodos aplicables a los objetos solo tenemos que añadir primero el argumento para pasar la referencia al objeto (**self**).

```
class Coche:
    '''Esta clase representa un coche'''

    # Atributo de clase
    ruedas = 4

    def __init__(self, potencia, velocidad_max, color):
        '''Constructor de Coche'''
        # Inicializamos atributos propios del nuevo objeto
        self.potencia = potencia
        self.velocidad_max = velocidad_max
        self.color = color
        self.velocidad = 0

    def acelera(self, incremento):
        '''Método para acelerar el coche `self`'''
        self.velocidad = self.velocidad + incremento
        if (self.velocidad > self.velocidad_max):
            print("Voy a todo gas!!")
            self.velocidad = self.velocidad_max
        else:
            print("Voy a %d" % self.velocidad)

    # Método de clase
    def prueba_coche():
        print("Soy un coche.")
```

```
coche_1 = Coche(120, 200, "rojo")
coche_1.acelera(50)
coche_1.acelera(60)
coche_1.acelera(100)
```

```
Voy a 50
Voy a 110
Voy a todo gas!!
```

Un método de instancia puede acceder a los atributos particulares del objeto a través de la referencia **self**. En cambio, un método de clase solamente tiene acceso a los atributos de clase (los compartidos por todos los objetos). Es más, un método de clase no se puede ejecutar a través de una instancia, únicamente a través de su clase.

8.2.3. HERENCIA

Una de las características más importantes de la orientación a objetos es el concepto de *herencia*. Esencialmente consiste en definir una nueva clase *derivándola* de una existente, de la que hereda por defecto todas sus características, métodos y atributos.

La forma de especificar que una clase hereda de otra clase *base* es indicando su nombre entre paréntesis.

```
# Definimos la clase Familiar
# que hereda de la clase Coche
class Familiar(Coche):

    def __init__(self, potencia, velocidad_max, color, maletero):
        '''Constructor para coche familiar'''
        # primero inicializamos los atributos heredados
        # llamando al constructor de la superclase
        super().__init__(potencia, velocidad_max, color)
        # y ahora inicializamos atributos propios
        # de esta clase unicamente
        self.maletero = maletero
        self.carga = 0

    def pon_carga(self, peso):
        '''Método para cargar el maletero'''
```

```
# En un coche familiar tenemos más capacidad de carga
self.carga = self.carga + peso
if (self.carga > self.maletero):
    print("No cabe nada más!!")
    self.carga = self.maletero
else:
    print("Llevo %d kg" % self.carga)

fam_1 = Familiar(130, 180, 'gris', 300)
print(fam_1)
```

```
<__main__.Familiar object at 0x7f9d942d60f0>
```

Como hemos dicho, una clase derivada hereda todos los métodos y atributos de la clase *padre* (también conocida como *superclase*).

Sin embargo, la potencia de este mecanismo viene al ampliar o redefinir los elementos heredados de la *superclase*.

En este ejemplo, ampliamos los atributos para nuevas instancias de esta clase, añadiendo el atributo **maletero** en el constructor. Además, en el constructor de **Familiar** empezamos por inicializar los atributos heredados de la clase *padre*, llamando a su constructor. Decimos que hemos *sobrecargado* el constructor.

También aprovechamos para añadir un nuevo método (**pon_carga()**) para los objetos de la clase **Familiar**. A esto lo llamamos *extender* la clase. Eso sí, seguimos teniendo acceso a todos los demás atributos y métodos heredados.

```
fam_1.acelera(40)
fam_1.pon_carga(50)
```

```
Voy a 40
Llevo 50 kg
```

Redefinición o *sobrecarga* de métodos

No solo podemos *sobrecargar* el constructor de una clase derivada. La *sobrecarga* aplica a cualquier método. De esta forma alteramos el comportamiento heredado para adaptarlo a las características de la nueva clase.


```

# Definimos la clase Familiar
# que hereda de la clase Coche
class Familiar(Coche):

    def __init__(self, potencia, velocidad_max, color, maletero):
        '''Constructor para coche familiar'''
        # primero inicializamos los atributos heredados
        # llamando al constructor de la superclase
        super().__init__(potencia, velocidad_max, color)
        # y ahora inicializamos atributos propios
        # de esta clase unicamente
        self.maletero = maletero
        self.carga = 0

    def pon_carga(self, peso):
        '''Método para cargar el maletero'''
        # En un coche familiar tenemos más capacidad de carga
        self.carga = self.carga + peso
        if (self.carga > self.maletero):
            print("No cabe nada más!!")
            self.carga = self.maletero
        else:
            print("Llevo %d kg" % self.carga)

    def acelera(self, incremento):
        '''Método sobrecargado para acelerar un familiar'''
        self.velocidad = self.velocidad + 0.5*incremento
        if (self.velocidad > 120):
            print("Dónde vas, Fitipaldi!!")
            self.velocidad = self.velocidad_max
        else:
            print("He subido solo a %d" % self.velocidad)

fam_2 = Familiar(120, 180, "negro", 250)
fam_2.acelera(70)
fam_2.acelera(80)
fam_2.acelera(100)

```

```

He subido solo a 35
He subido solo a 75
Dónde vas, Fitipaldi!!

```

Para decidir qué atributo o método debe usar, Python usa un mecanismo de resolución con el que va explorando desde la clase más baja en el árbol de herencia hacia las clases superiores. Es decir, comienza buscando un método en la propia clase del objeto. Si no lo encuentra, asciende a sus clases *padre* para continuar la búsqueda.

¿QUÉ HAS APRENDIDO?

Con esta unidad ya tienes los conocimientos y herramientas necesarias para desarrollar tus propios programas en Python y poder analizar y entender el funcionamiento de código escrito en Python por otras personas.

Esto te va a ser muy útil, para empezar, en la siguiente unidad, donde profundizaremos en las principales librerías para trabajar en ciencia de datos con Python.

Recuerda que Python es un lenguaje de propósito general, y uno de los más extendidos actualmente. Te animamos a hacer nuevas pruebas y programas por tu cuenta, profundizando en tu dominio del lenguaje. Seguro que te abrirá nuevas puertas

Y por último, no dudes en volver a esta unidad y utilizarla como referencia siempre que lo necesites.

AUTOCOMPROBACIÓN

1. Python:

- a) Es un lenguaje de programación interpretado de propósito general.
- b) Es un lenguaje de programación interpretado especializado para computación científica.
- c) Es un lenguaje de programación compilado multiplataforma.
- d) Es un lenguaje de programación científica multiplataforma.

2. Anaconda:

- a) Es una versión de Python especial para ciencia de datos.
- b) Es una distribución independiente de Python que incluye de serie un gran número de librerías para ciencia de datos.
- c) Es un editor para programar en Python especial para ciencia de datos.
- d) Es una herramienta que viene de serie con Python orientada a la ciencia de datos.

3. ¿Cuál es el resultado de la expresión `abs(-1.0 * (3 ** 2))`?

- a) Error por mezclar tipos enteros y de punto flotante.
- b) El valor entero 9.
- c) El valor entero -9.
- d) El valor en punto flotante 9.0.

4. La expresión `"a:bc".split(':')` devuelve:

- a) Un error.
- b) Una lista con dos cadenas `['a', 'bc']`.
- c) La cadena después del carácter `:`, `'bc'`.
- d) La cadena sin el carácter `:`, `'abc'`.

5. La indentación de código en Python:

- a) Se recomienda para mejorar la legibilidad del código.
- b) Es la forma para delimitar bloques de código, agrupando líneas contiguas con el mismo nivel de indentación.
- c) Se debe hacer exclusivamente con tabuladores.
- d) Conviene hacerse con múltiplos de cuatro espacios.

6. ¿Cuál de las siguientes expresiones imprime los números de 0 a 5, ambos incluidos?

- a) `for x in range(0, 5): print(x)`
- b) `for x in (0:5): print(x)`
- c) `for x in range(6): print(x)`
- d) `while i in range(6): print(i)`

7. Si tenemos el siguiente código:

```
x1 = [1,2,3]
x2 = x1
x1[1] = 'b'
```

¿qué valor tiene x2?

- a) `[1,2,3]`
- b) `['b',2,3]`
- c) `['b']`
- d) `[1,'b',3]`

8. Dado el siguiente código:

```
import functools as fn
fn.reduce(lambda x,y: x+y, map(lambda x: x**2, range(10)))
```

¿Cuál de las siguientes expresiones es equivalente?

- a) `sum(map(lambda x: x**2, range(10)))`
- b) `sum((x**2 for x in range(10)))`
- c) Ambas expresiones anteriores.
- d) Ninguna de las anteriores.

9. Si abrimos un fichero de la siguiente manera:

```
f = open(nombre_fichero, "r")
```

¿Cuál es la forma más eficiente de leer las líneas del fichero una a una?

- a) `for l in f.read().split('\n'): print(l)`
- b) `for l in f.readline(): print(l)`
- c) `for l in f.readlines(): print(l)`
- d) `for l in f: print(l)`

10. Para definir los métodos aplicables a un objeto:

- a) Debemos incluir como primer argumento en la declaración de la función la referencia *self* al propio objeto.
- b) Debemos incluir como primer argumento el nombre de la clase.
- c) Debemos declarar las funciones justo después del constructor.
- d) Debemos preceder el nombre de la función con el nombre de la clase.

SOLUCIONARIO

1.	a	2.	b	3.	d	4.	b	5.	b
6.	c	7.	d	8.	c	9.	d	10.	a

BIBLIOGRAFÍA

Información general sobre Python

- Página principal del lenguaje Python:
<https://www.python.org>
- Tutorial oficial de Python (inglés):
<https://docs.python.org/3/tutorial/index.html>
- Tutorial de Python (traducción al español):
 - Online: <http://docs.python.org.ar/tutorial/3/index.html>
 - PDF: <http://docs.python.org.ar/tutorial/pdfs/TutorialPython3.pdf>
- Python para Principiantes:
 - Online: <https://librosweb.es/libro/python/>
 - PDF: <http://www.iaa.es/python/curso-python-para-principiantes.pdf>
- Dive into Python 3: <http://www.diveintopython3.net/>

Anaconda

- Página principal de Anaconda:
<https://www.anaconda.com/>
- Documentación del proyecto:
<https://docs.anaconda.com/>

- Referencia rápida de comandos:

https://conda.io/docs/_downloads/conda-cheatsheet.pdf

Jupyter

- Página principal del proyecto:

<http://jupyter.org/>

- Guía rápida:

<https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/>

Spyder

- Página principal del proyecto:

<https://github.com/spyder-ide>

- Documentación:

<https://pythonhosted.org/spyder/>

Biblioteca Estándar de Python

- The Python Standard Library:

<https://docs.python.org/3/library/index.html>

- Pequeño paseo por la Biblioteca Estándar (parte I):

<http://docs.python.org.ar/tutorial/3/stdlib.html>

- Pequeño paseo por la Biblioteca Estándar (parte II):

<http://docs.python.org.ar/tutorial/3/stdlib2.html>

Referencias a otras librerías

- The Python Package Index (PyPI):

<https://pypi.python.org/pypi>