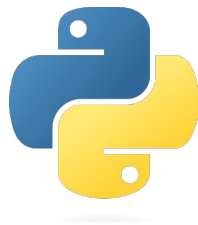


Programación en Python

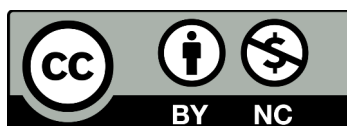
Fundamentos, orientación a objetos y persistencia



José Miguel Blázquez

Versión 1.1 – Septiembre 2024

Basado en los materiales del profesor Nacho Iborra
Licencia CC BY-NC



Sumario

1. Introducción a Python.....	4
1.1. Instalación de Python.....	4
1.2. Elegir un IDE apropiado.....	5
1.2. Nuestro primer programa Python.....	7
1.3. Otras herramientas.....	7
1.3.1. Google Colab.....	8
1.3.2. Jupyter.....	10
1.3.3. Anaconda.....	10
2. Elementos básicos del lenguaje.....	11
2.1. Comentarios.....	11
2.2. Tipos de datos básicos, variables y conversión.....	11
2.3. Operadores aritméticos, relacionales y lógicos.....	12
2.4. Entrada/Salida básica.....	14
3. Estructuras de control.....	16
3.1. Estructuras condicionales: if, if..else, if..elif..else.....	16
3.2. Estructuras repetitivas (bucles).....	17
3.3. Gestión de errores mediante excepciones.....	19
4. Arrays y listas.....	21
4.1. Definición y uso básico de listas.....	21
4.2. Otras operaciones con listas.....	23
5. Cadenas de texto.....	25
5.1. Operaciones básicas con cadenas.....	25
5.2. Otras operaciones sobre cadenas.....	25
6. Diccionarios y otras estructuras de datos.....	28
6.1. Tuplas.....	28
6.2. Diccionarios.....	29
6.3. Conjuntos.....	30
7. Programación modular: funciones.....	31
7.1. Definición de funciones.....	31
7.2. Parámetros.....	33
7.3. Operaciones avanzadas con listas.....	36
7.4. Nociones sobre programación funcional.....	36
7.5. Módulos.....	37
7.5.1. Descomponiendo nuestro código en módulos.....	38
7.5.2. Instalación de módulos adicionales.....	39
7.5.3. Más operaciones con módulos.....	41
7.5.4. Gestión de dependencias con entornos virtuales y Conda.....	41
8. Programación orientada a objetos.....	43
8.1. Definición de clases y objetos.....	43
8.2. Herencia.....	45
9. Acceso a ficheros.....	48
9.1. Gestión de ficheros de texto.....	48
9.2. Gestión del sistema de ficheros.....	50
10. Acceso a bases de datos.....	51
10.1. Acceso a bases de datos MySQL.....	51
10.2. Operaciones con la base de datos MySQL.....	52
10.3. Acceso a bases de datos NoSQL con MongoDB.....	54
10.4. Operaciones con la base de datos MongoDB.....	55

PREFACIO

Este documento contiene un curso sobre programación en lenguaje Python, sin embargo, no se trata de un curso para aprender fundamentos de programación en general, sino que está destinado para personas que ya tengan nociones en otros lenguajes y quieran aprender cómo funciona Python.

Explicaremos los conceptos básicos del lenguaje, suponiendo esa base previa de programación que ya hemos mencionado; hablaremos sobre variables, condiciones, bucles, arrays, objetos, etc... En caso de no tener esa base de programación, será conveniente adquirirla a través de otros cursos más detallados o bien mediante cursos de introducción sea con Python u otros lenguajes.

Este curso está pensado para introducir al alumnado ya titulado de ciclos formativos de grado superior de la familia de informática, tanto para los titulados de “Desarrollo de Aplicaciones Web” como de “Desarrollo de Aplicaciones Multiplataforma”; con el objetivo de que puedan cursar satisfactoriamente el curso de especialización de “Inteligencia Artificial y Big Data”, en caso de que no hayan trabajado con Python anteriormente en sus respectivos ciclos formativos.

El uso de estos materiales está sujeto a una licencia Creative Commons CC BY-NC.

1. Introducción a Python

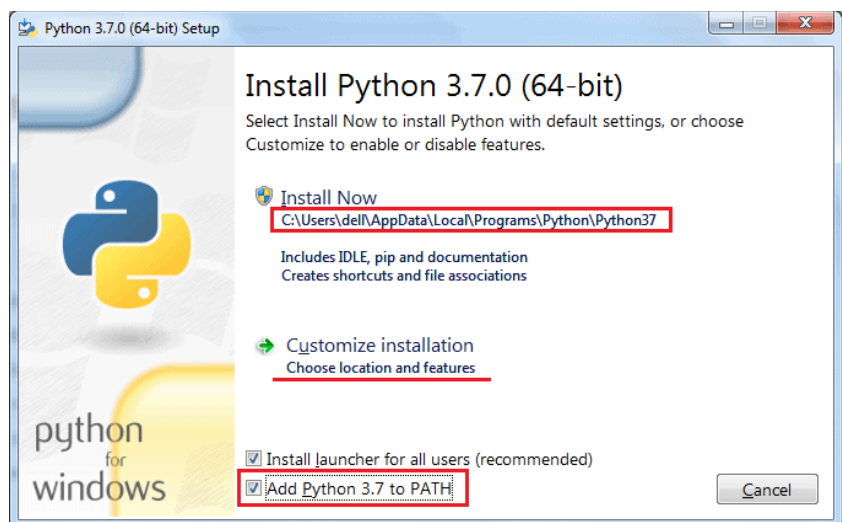
Python es un lenguaje de programación creado a principios de los años 90 por Guido van Rossum. Su nombre es un tributo al grupo de comedia *Monty Python*, y entre sus muchas virtudes, podemos destacar las siguientes:

- Se utiliza a menudo en etapas tempranas de aprendizaje de programación, porque es bastante sencillo de entender.
- Es un lenguaje multiplataforma, con el que podemos desarrollar aplicaciones de todo tipo (escritorio, web, etc) en diferentes sistemas (Windows, Mac, Linux).
- Es un lenguaje interpretado (no compilado), y puede utilizarse como un lenguaje de *script* en terminal, como ocurre con Perl, PowerShell u otros lenguajes de scripting.
- Su tipado es fuerte y dinámico, es decir, no existen tipos de datos implícitos, ni tenemos que indicarlos al declarar las variables en el programa; pero a medida que asignemos valores a las variables, éstas tomarán el tipo de dato adecuado.
- Podemos utilizar Python tanto desde una perspectiva orientada a objetos (usando clases y objetos) como sin dicha perspectiva (estructurado modular).
- Dispone de multitud de paquetes o librerías adicionales que podemos descargar para construir aplicaciones de todo tipo.

1.1. Instalación de Python

Para instalar Python basta con ir a la web oficial y descargar la versión apropiada para nuestro sistema. El intérprete de Python es normalmente un comando llamado `python` o `python3`, dependiendo del sistema operativo en el que estemos.

Cuando instalemos Python en Windows, debemos marcar en el asistente de instalación la casilla para añadir Python directamente al PATH del sistema. De lo contrario, tendremos que editar manualmente la variable de entorno para añadir la carpeta de instalación de Python.



En lo que respecta a Linux, primero podemos comprobar si ya tenemos Python instalado en nuestra distribución, con este comando:

```
python3 --version
```

De lo contrario, podemos actualizar a Python 3 fácilmente así:

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

Como hemos visto, podemos utilizar el parámetro `--version` o también `-V` para detectar la versión actualmente instalada de Python.

```
python -V
```

1.2. Elegir un IDE apropiado

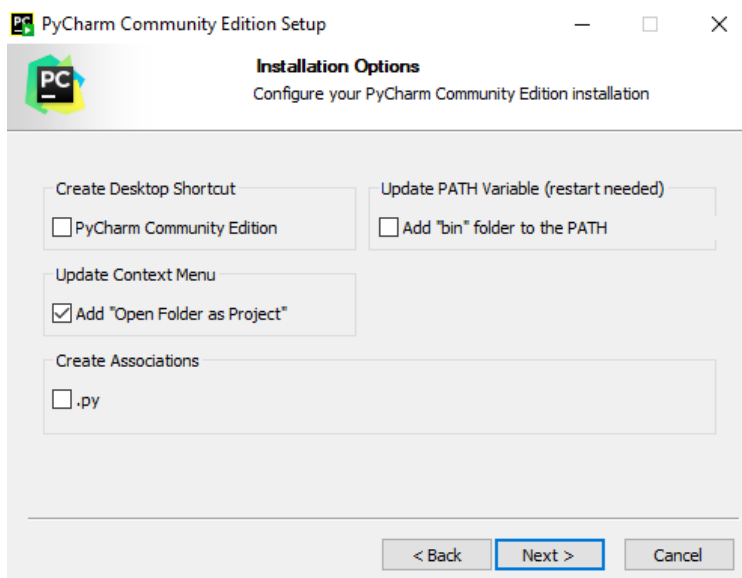
Una vez tengamos Python instalado en nuestro sistema, debemos elegir un IDE apropiado para desarrollar nuestras aplicaciones. Existe multitud de alternativas, como por ejemplo IDLE (el entorno que viene integrado en la instalación de Python), Eclipse, Visual Studio Code, PyCharm, Spyder... En este apartado vamos a dar unas breves nociones sobre los que consideramos más interesantes.

Utilizando IDLE

IDLE es un entorno que se descarga en la propia instalación de Python. Podemos ejecutarlo escribiendo `idle` o `idle3` en el terminal (dependiendo de nuestro sistema operativo y versión de Python). También podemos elegir el correspondiente acceso directo en el menú de Windows, o en la sección de *Aplicaciones* de Mac. Lo que veremos es un terminal de comandos Python. Podemos escribir comandos sueltos y se ejecutarán uno a uno, pero esto no es lo habitual, sino que necesitaremos editar un archivo de código fuente. Para ello, desde el menú *File > New File* podemos crear nuevos archivos de código Python, y guardarlos con el nombre adecuado (los archivos deben tener extensión `.py`, como por ejemplo *prueba.py*). Después, editamos el código fuente y podemos ejecutarlo desde el menú *Run > Run Module* de la ventana de edición, o bien pulsando la tecla F5. Sin embargo, este entorno se nos puede quedar algo corto o limitado si queremos gestionar varios archivos, y puede resultar algo engorroso estar cambiando entre el terminal y la ventana de edición.

PyCharm

Pasamos a analizar ahora otro IDE más potente, útil para desarrollar proyectos complejos. Hablamos de **PyCharm**, un IDE desarrollado por la empresa JetBrains, también responsable de otros IDEs populares para otros lenguajes como IntelliJ (Java) o PhpStorm (PHP). Podemos descargar PyCharm gratuitamente (versión *Community*) en su web oficial. En la instalación hay poco que configurar. En todo caso, para

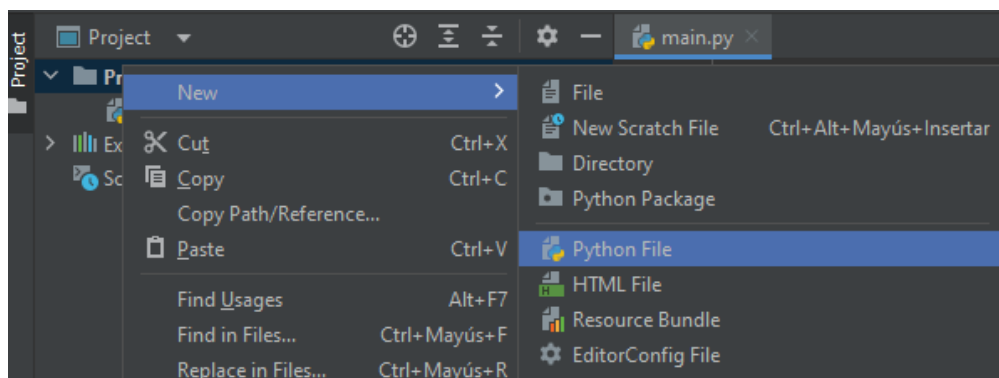


Windows, elegir la casilla para poder abrir carpetas enteras como proyectos con un simple clic derecho:

Una vez iniciado, crearemos un nuevo proyecto (*New project*). Por defecto los proyectos se guardarán en la subcarpeta *PyCharmProjects* de nuestra carpeta de usuario del sistema, aunque podemos cambiar la ubicación.

Es importante también destacar, en este paso, que deberemos elegir el entorno de ejecución de nuestro proyecto. Por defecto PyCharm creará un entorno virtual (*Venv*, *Virtual Environment*) para cada proyecto, lo que significará que nos instalará una serie de librerías predefinidas en el proyecto, y podremos añadir otras. Si no queremos hacer eso, deberemos elegir la casilla de *Previously configured interpreter* en la ventana anterior y elegir el intérprete del sistema.

De este modo, se creará una carpeta de código fuente vacía (con un archivo inicial llamado *main.py*), y podremos añadir nuevos archivos al proyecto haciendo clic derecho sobre él, y eligiendo *New > Python file*



Para ejecutar cualquiera de los archivos de nuestro proyecto, hacemos clic derecho sobre él y elegimos la opción de *Run*.

Visual Studio Code

Visual Studio Code ofrece unas funcionalidades similares a Geany, pero nos será de mayor utilidad cuando queramos gestionar un proyecto con varios archivos fuente. Podemos descargar VS Code de su web oficial. En el caso de Windows y Mac, simplemente ejecutamos el instalador. En el caso de Linux (Ubuntu), desde el terminal, ubicado en la carpeta donde hemos descargado el archivo, ejecutamos este comando:

```
sudo dpkg -i nombre_archivo.deb
```

donde *nombre_archivo.deb* será el archivo con extensión *.deb* que habremos descargado.

Para poder trabajar con Python desde Visual Studio Code, debemos abrir la carpeta donde vayamos a editar los archivos Python (menú *Archivo > Abrir carpeta*), y luego crear los archivos y carpetas que queramos desde el panel izquierdo del explorador. También es recomendable instalar la extensión *Code Runner* y configurarla para poder ejecutar los programas Python en el terminal integrado de Visual Studio Code.

Code-runner: Run In Terminal

☒ Whether to run code in Integrated Terminal.

Intérpretes online

Una opción alternativa para desarrollar programas en Python, especialmente cuando son relativamente simples y cortos, es utilizar algún editor online. Simplemente debemos escribir el código en el panel correspondiente, y hacer clic en el botón para ejecutar la aplicación, viendo el resultado en la consola derecha. Ejemplo: myCompiler.io

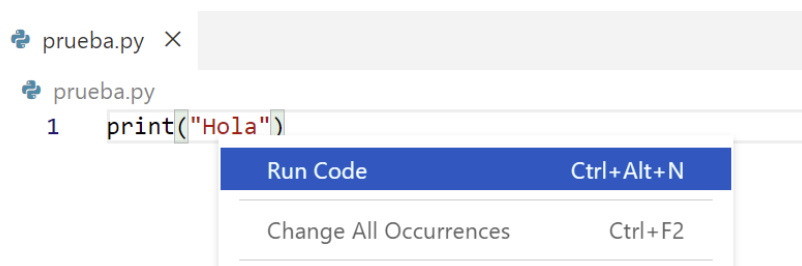
1.2. Nuestro primer programa Python

Vamos a crear nuestro primer programa Python que muestre un saludo por pantalla. Podemos emplear cualquiera de los IDEs comentados anteriormente. Creamos un archivo fuente Python llamado, por ejemplo, `prueba.py` en la carpeta donde queramos trabajar, y lo guardamos con este contenido:

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

A la hora de ejecutarlo:

- Si usamos IDLE, pulsamos F5
- Si usamos el editor online de *replIt*, pulsamos el botón de ejecutar o *Run*
- Si usamos PyCharm, hacemos clic derecho sobre el archivo y elegimos la opción de *Run*
- Si utilizamos Visual Studio Code y hemos instalado la extensión *Code Runner*, hacemos clic derecho sobre el código fuente y elegimos la opción *Run Code* que aparecerá en el menú contextual



El resultado en cualquier caso será el mismo, viendo por terminal el texto que hemos indicado en la instrucción `print`:

```
Hola
```

1.3. Otras herramientas

Además de los IDEs tradicionales para trabajar con Python existen otras posibilidades de trabajo, impulsadas en gran parte por la aplicación que ha tenido este lenguaje en la inteligencia artificial.

Son especialmente útiles para trabajo colaborativo online, o para trabajo con *machine learning* y tratamiento de datos.

1.3.1. Google Colab

Google Colab es una de las muchas herramientas que ofrece Google en la nube. Nos permite definir documentos donde intercalar texto con código Python (y algún otro lenguaje, como R o Julia), de forma que podemos hacer tutoriales guiados. Además, facilita la integración con multitud de librerías ya incorporadas en Colab, como *NumPy* o *Matplotlib*, lo que permite trabajar fácilmente con conjuntos de datos y obtener representaciones gráficas en el mismo documento. Estos datos podemos incorporarlos de varias fuentes de forma automática, tales como hojas de cálculo de nuestra cuenta de Google Drive, o incluso repositorios GitHub.

Primeros pasos

Para comenzar a trabajar con Colab, debemos acceder a su web oficial con nuestra cuenta de Google (lo que asociará Colab con el espacio en Google Drive de esa cuenta). Una vez dentro, veremos que Colab trabaja en base a unos documentos llamados **cuadernos**, que se guardan en un formato especial *ipynb*, compatible con otras herramientas como Jupyter, que veremos a continuación.

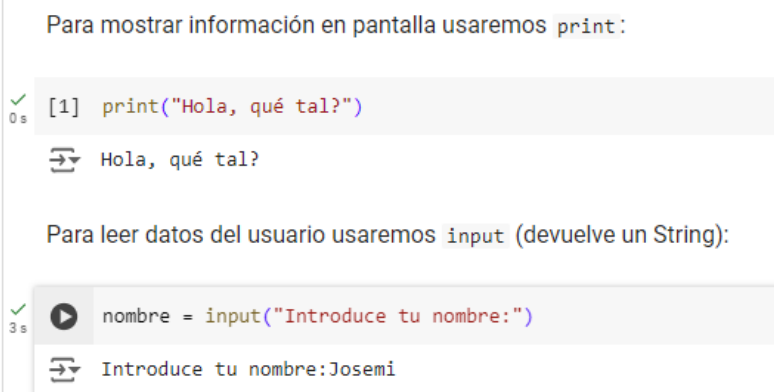
Desde el menú *Archivo* podemos abrir un cuaderno que tengamos previamente guardado en Drive, crear nuevos cuadernos, o subir archivos *ipynb* nuevos a nuestra cuenta.

NOTA: si accedemos a un cuaderno Colab de una cuenta que no es la nuestra, es recomendable copiarlo a nuestra cuenta de Drive, usando el menú *Archivo > Guardar una copia en Drive*, ya que de lo contrario los cambios que hagamos no se van a guardar.

También es posible crear nuevos documentos Colab desde nuestro panel principal de Google Drive, con el botón izquierdo *Nuevo*. Si no aparece la opción de *Colab*, podemos añadirla desde el apartado de *Conectar más aplicaciones*.

Trabajo con cuadernos. Operaciones básicas

Dentro de un cuaderno abierto, desde el menú *Insertar* podemos insertar encabezados de sección para dividir la estructura de nuestro documento, y en cada sección podemos incluir bloques de texto explicativo, o bloques de código que podemos ejecutar pulsando en la flecha a la izquierda del bloque o la combinación de teclas *Control+Intro*. El código se ejecuta en una máquina virtual remota gestionada por Google Colab.



```
Para mostrar información en pantalla usaremos print:
```

```
[1] print("Hola, qué tal?")
```

```
Hola, qué tal?
```

```
Para leer datos del usuario usaremos input (devuelve un String):
```

```
nombre = input("Introduce tu nombre:")
```

```
Introduce tu nombre:Josemi
```


También podemos aprovechar los bloques de código para ejecutar instrucciones en el sistema operativo de la máquina remota donde se está ejecutando el código, anteponiendo un `!` delante:

```
!python -V
```

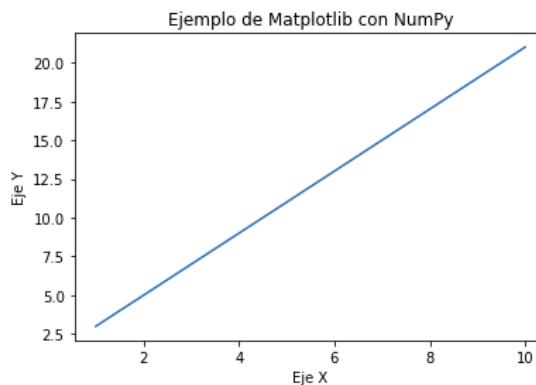
Finalmente, desde el menú *Archivo* anterior también podemos guardar los cambios, tanto en Drive (opción por defecto) como en GitHub si lo preferimos.

Como decíamos, una de las ventajas que ofrece Colab es que nos permite trabajar directamente en la nube con ciertas librerías ya instaladas y que pueden resultar muy útiles para el tratamiento de datos, tales como *NumP* para tratamiento de arrays o *Matplotlib* para generación de gráficas, en el caso de Python. De este modo, se puede tener en un solo documento la explicación, los datos con los que trabajar y los resultados que se obtienen.

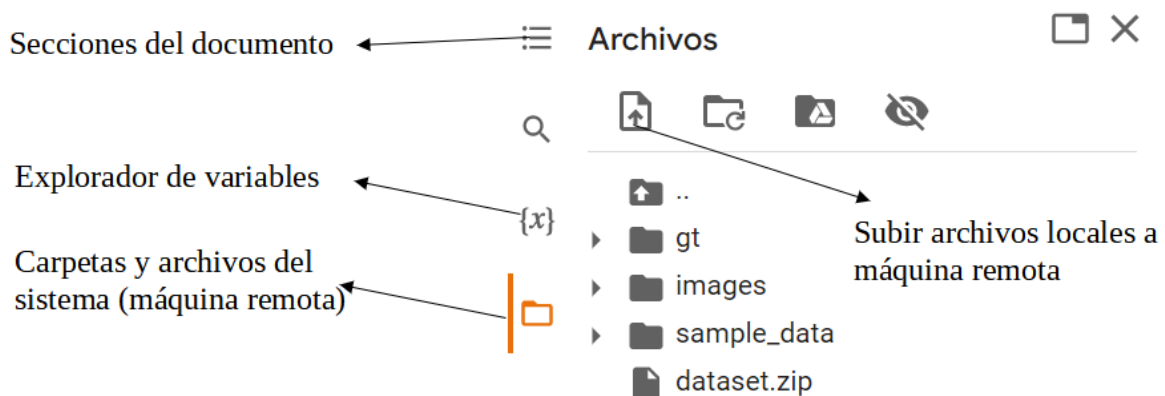
Aquí vemos un ejemplo de cómo trabajar con las librerías NumPy y Matplotlib, i

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1,11)
y = 2 * x + 1
plt.title("Ejemplo de Matplotlib con NumPy")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.plot(x,y)
plt.show()
```



En la parte izquierda tenemos unos iconos de utilidad, para poder, por ejemplo, ver el árbol de secciones y subsecciones del documento actual, o también los archivos y carpetas que se han generado en la máquina virtual remota (y donde podemos también añadir archivos propios, o descargar archivos generados en dicha máquina).



Limitaciones y otras consideraciones

En cuanto a sus **limitaciones**, en la versión gratuita sólo se permite el uso de una CPU de forma “ilimitada”, unos minutos de uso de una GPU (para procesamiento paralelo, útil en el trabajo con redes neuronales), una RAM limitada y una actividad continuada de hasta 12 horas por sesión. Para activar la GPU deberemos hacerlo desde el menú *Entorno de ejecución - Cambiar tipo de entorno de ejecución*.

1.3.2. Jupyter

Jupyter es un conjunto de herramientas que ofrecen una funcionalidad similar a Colab (gestionar cuadernos interactivos que intercalan texto y código Python), pero que debe instalarse de forma local en el sistema. De hecho, Colab es básicamente un servidor Jupyter instalado en Google.

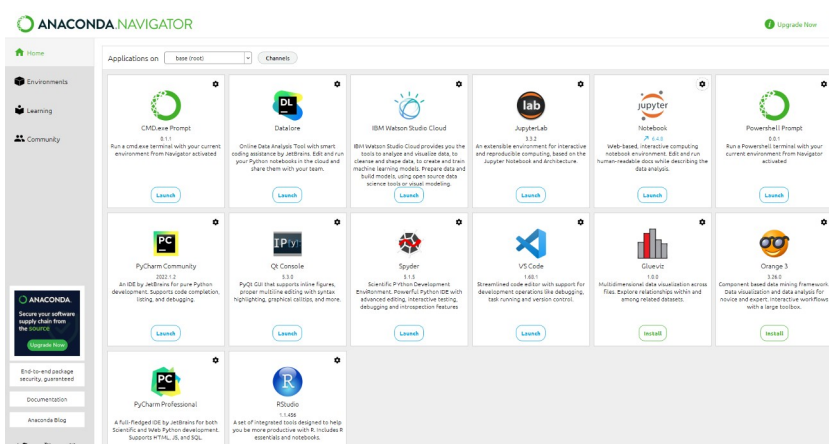
Se distinguen dos aplicaciones:

- *Jupyter Notebook*, que es algo más básico para editar archivos Jupyter (*ipynb*), y definir bloques de texto, código, etc.
- *JupyterLab*, que integra una serie de herramientas adicionales para trabajo más avanzado en *machine learning* o *data science*.

El principal inconveniente de estas herramientas es que se instalan en local, con lo que, por un lado, perdemos ese elemento colaborativo online que ofrece *Colab*, y por otro, tenemos que instalar manualmente en el sistema las librerías que necesitemos para nuestros trabajos (*NumPy*, *Matplotlib*, etc).

1.3.3. Anaconda

Anaconda es un pack de software que integra diferentes herramientas orientadas al trabajo con Python y otros lenguajes (como R), en ámbitos como *data science* o *machine learning*. Entre otras cosas, podemos instalar el propio entorno Python (o R), junto con algunos editores predefinidos (como PyCharm), así como el propio *Jupyter Notebook* y también algunas librerías típicas de *data science* o *machine learning* tales como *NumPy* o *TensorFlow*.



Como principal ventaja sobre el entorno *Jupyter* anterior, podemos decir que Anaconda ya incorpora Jupyter, y también incorpora las principales librerías para trabajo con *data science* o *machine learning*, con lo que nos facilita bastante el comenzar a trabajar en estos entornos. También incorpora su propia versión de Python y/o lenguaje R, así que no interfiere con la que podamos tener instalada en el sistema previamente.

2. Elementos básicos del lenguaje

Veamos en este documento algunos elementos básicos del lenguaje para empezar a hacer nuestros primeros programas. Veremos qué tipos de datos maneja Python, cómo declarar variables para almacenar información, y cómo hacer algunas operaciones aritméticas básicas. También veremos cómo mostrar información por pantalla y pedírsela al usuario por teclado.

2.1. Comentarios

Los comentarios en Python pueden ser de una sola línea o de varias, como en otros lenguajes. Para los comentarios de una sola línea, debemos comenzarlos con el símbolo de la almohadilla `#`. Por ejemplo:

```
# Mostramos un mensaje  
print ("Hola mundo")
```

También podemos definir comentarios de múltiples líneas. En este caso, comienzan y terminan por una triple comilla simple `'''`:

```
'''  
Esto es un comentario de varias líneas  
Segunda línea del comentario  
'''  
print ("Hola mundo")
```

2.2. Tipos de datos básicos, variables y conversión

Existen tres tipos de datos básicos en Python:

- **Números:** pueden ser enteros (3), reales (4.33) e incluso complejos (5 + 3j)
- **Textos:** todo son cadenas (string). No se dispone del tipo carácter (*char*), así que para los textos podemos emplear indistintamente comillas dobles ("Hola mundo") o simples ('Hola mundo'). También podemos emplear las secuencias de escape habituales, como el salto de línea `\n` o la tabulación `\t` ("Hola\tmundo")
- **Booleanos:** pueden ser `True` o `False`.

Declaración e inicialización de variables

Debido a que Python es un lenguaje dinámicamente tipado (es decir, no es necesario definir el tipo de dato que vamos a almacenar en una variable), simplemente declaramos las variables y les asignamos un valor. Dependiendo del valor asignado, la variable toma el tipo de dato adecuado. Por ejemplo:

```
edad = 23  
mensaje = "Hola mundo"
```

Python es un lenguaje que ofrece particularidades que no muchos lenguajes tienen. Por ejemplo, si queremos intercambiar el valor de dos variables, en la mayoría de lenguajes necesitaríamos una

tercera variable auxiliar que almacene temporalmente uno de los dos datos a intercambiar. Así nos quedaría el código en Java, por ejemplo, para intercambiar los valores de *n1* y *n2*:

```
int n1 = 3, n2 = 8;
...
int aux = n1;
n1 = n2;
n2 = aux;
```

Sin embargo, en Python es tan sencillo como hacer esto:

```
n1 = 3
n2 = 8
n1, n2 = n2, n1
```

Conversión de datos

Si queremos convertir de un tipo básico a otro, podemos emplear estas instrucciones útiles:

- `int(valor)`: convierte a entero el valor indicado, que puede ser un número real o un texto, por ejemplo
- `float(valor)`: funciona como `int`, pero convierte a número real el valor indicado.
- `str(valor)`: obtiene una representación textual del valor indicado

```
texto = "23"
numero = int(texto) # 23
edad = int(input("Introduce tu edad: ")) # importante con input
```

2.3. Operadores aritméticos, relacionales y lógicos

Veremos a continuación de forma resumida los principales operadores que ofrece el lenguaje Python, así como el funcionamiento básico de las operaciones. Empecemos por los operadores **aritméticos**:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
**	Potencia
/	División
//	División entera
%	Resto o módulo

```
>>> 5/3
1.6666666666666667
>>> 5//3
1
```

Al sumar, restar o multiplicar números enteros, el resultado es entero.

Al hacer operaciones en las que intervienen números enteros y decimales, el resultado es siempre decimal. En el caso de que el resultado no tenga parte decimal, Python escribe 0 como parte decimal para indicar que el resultado es un número decimal:

```
>>> 4.5 * 3
13.5
>>> 4.5 * 2
9.0
```

Al dividir números enteros, el resultado es siempre decimal, aunque sea un número entero. Cuando Python escribe un número decimal, lo escribe siempre con parte decimal, aunque sea nula:

```
>>> 9 / 2
4.5
>>> 9 / 3
3.0
```

A la hora de comparar dos elementos tenemos los típicos **operadores relacionales** que ofrecen la mayoría de lenguajes de alto nivel: ==, !=, <, <=, >, >=

Los **operadores lógicos** que ofrece Python para poder enlazar comprobaciones simples y formar otras más complejas son:

- **and**: (Y lógica, que en muchos lenguajes se representa con &&)
- **or**: (O lógica, que en muchos lenguajes se representa con doble barra vertical | |)
- **not**: (negación lógica, que en muchos lenguajes se representa con !)

Por ejemplo:

```
>>> 3 > 2 and 5 > 1
True
```

Como pasa en cualquier lenguaje de programación de alto nivel, las expresiones se evalúan de la manera tradicional.

Es decir, siguiendo la jerarquía de operaciones, cuyo orden completo podemos ver en la siguiente tabla:

Operator	Description
**	Exponent
~ + -	Bitwise NOT, unary plus, and minus
* / % //	Multiplication, division, modulus and floor division
+ -	Addition and subtraction
>> <<	Bitwise right shift and bitwise left shift
&	Bitwise AND
^	Bitwise XOR and regular OR
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //=- += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

2.4. Entrada/Salida básica

Veamos ahora qué mecanismos ofrece Python para sacar información por pantalla y pedirle datos al usuario

Salida con *print*

Hemos visto que la instrucción `print` puede utilizarse para mostrar mensajes por pantalla. Podemos mostrar un mensaje simple...

```
print ("Hola mundo")
```

... o combinar varias partes de un mensaje, separadas por comas (Python añade un espacio entre cada parte):

```
print ("La suma de", num1, "y", num2, "es", resultado)
```

También podemos proporcionar algún formato de salida, de forma similar a como funciona la instrucción *printf* en lenguajes como C o Java. Por ejemplo, de este modo definimos un hueco o espacio horizontal para un número (3 espacios de hueco)

```
print ("Mi nombre es Nacho y tengo %3d años" % (edad))
```

O podemos especificar cuántos dígitos enteros y decimales mostrar en una expresión o variable real. En el siguiente ejemplo mostramos el número con 3 dígitos enteros (si no hay se rellenan con espacios) y 2 decimales (si no hay se rellenan con ceros).

```
print ("El resultado final es %3.2f" % (resultado))
```

Si necesitamos mostrar más de un dato, podemos especificarlo entre paréntesis, separados por comas:

```
print ("La suma de %d y %d es %5d" % (num1, num2, resultado))
```

Como ocurre con C o Java, podemos emplear los símbolos especiales `%d`, `%f` or `%s` para representar datos enteros, reales o textos en la expresión a mostrar con *print*, respectivamente.

Alternativamente, podemos anteponer una `f` al texto que queremos mostrar, y eso nos va a permitir intercalar variables dentro, encerradas entre llaves. En el caso de que alguna de las variables necesite un formato específico (más o menos cifras enteras o decimales) lo especificamos con dos puntos, y una sintaxis similar a la anterior.

```
print(f'La suma de {num1} y {num2} es {resultado:5d}')
```

Debemos tener en cuenta, no obstante, que por defecto `print` salta a la siguiente línea después de mostrar la información. Si no queremos que sea así, necesitamos añadir un parámetro llamado `end` en la instrucción, indicando con qué carácter queremos terminar. Por ejemplo, de este modo finalizamos con una cadena vacía, para no pasar a la siguiente línea:

```
print ("Hola", end="")
print ("Adiós")
#Muestra "HolaAdiós"
```

Obtener la entrada del usuario

Para obtener la entrada del usuario por teclado, empleamos la instrucción `input`. Esta instrucción recoge todo lo que escribe el usuario hasta que pulse *Intro* o *Enter*, y lo obtiene como información textual. Si le hemos pedido un número entero o real, por ejemplo, deberemos convertirlo al dato correspondiente usando las instrucciones `int` o `float` vistas antes:

```
print("Escribe un número entero:")
numero = int(input())
```

Alternativamente, la propia instrucción `input` también admite entre paréntesis el texto que le queremos mostrar al usuario para explicarle lo que tiene que hacer. Así, el código anterior puede abreviarse de este modo:

```
numero = int(input("Escribe un número entero:"))
```

Ejercicio 1:

Crea un programa llamado `Porcentajes.py` que le pregunte al usuario cuántos chicos y chicas hay en su clase, y calcule el porcentaje de chicos y chicas. **PISTA:** para sacar el símbolo del porcentaje en un texto, debemos duplicarlo `%`.

Ejercicio 2:

Crea un programa llamado `Saludo.py` que le pregunte al usuario su nombre y edad, y muestre un mensaje de saludo personalizado, de este modo: `Hola Nacho, tienes 43 años`

Ejercicio 3:

Crea un programa llamado `Media.py` que le pida al usuario 4 números enteros y calcule su media (real). La media debe mostrarse en pantalla con 3 cifras decimales.

3. Estructuras de control

Las estructuras de control nos permiten crear programas con múltiples caminos posibles a seguir, dependiendo de ciertas condiciones a comprobar. Estas estructuras son también habituales en otros lenguajes de programación... hablamos de *if*, *while* o *for*.

3.1. Estructuras condicionales: *if*, *if..else*, *if..elif..else*

Si queremos ejecutar un conjunto de instrucciones si se cumple una determinada condición, debemos añadir esta condición en una cláusula *if*, y las instrucciones vinculadas a esa condición deben quedar **tabuladas** respecto a ese *if*. Por ejemplo:

```
numero = int(input("Escribe un número positivo:"))
if numero > 0:
    numero = numero + 1
    print ("El siguiente número es", numero)
print ("Fin del programa")
```

Podemos también distinguir entre dos posibles caminos con la estructura *if . . else*. En este caso, debemos tabular el bloque de instrucciones que vaya asociado a cada parte (*if* o *else*):

```
numero = int(input("Escribe un número positivo:"))
if numero > 0:
    numero = numero + 1
    print ("El siguiente número es", numero)
else:
    print ("El número no es positivo")
print ("Fin del programa")
```

Si necesitamos tener más de dos caminos diferentes, podemos anidar estructuras *if . . else* unas dentro de otras...

```
numero = int(input("Escribe un número positivo:"))
if numero > 0:
    numero = numero + 1
    print ("El siguiente número es", numero)
else:
    if numero < -10:
        print ("El número es demasiado bajo")
    else:
        print ("El número no es positivo")
print ("Fin del programa")
```

... pero, en lugar de anidar estas estructuras, también podemos utilizar la cláusula *if . . elif* para especificar más de un bloque de condiciones.

Podemos enlazar tantas cláusulas `elif` como necesitemos, y también concluir con una cláusula `else` si queremos, para el último camino a distinguir:

```
numero = int(input("Escribe un número positivo:"))
if numero > 0:
    numero = numero + 1
    print ("El siguiente número es", numero)
elif numero < -10:
    print ("El número es demasiado bajo")
else:
    print ("El número no es positivo")
print ("Fin del programa")
```

La cláusula switch-case no existe con ese nombre en Python, pero sí hay un equivalente muy poderoso: match-case. Sin embargo, no lo abordaremos en este manual.

Uso abreviado de `if..else`

La estructura `if..else` se puede usar de modo abreviado en una línea para asignar un valor u otro a una variable. Sería el equivalente al *operador ternario* `?:` que existe en otros lenguajes, y que no está disponible en Python. Por ejemplo, esta instrucción asigna a la variable `par` un valor `True` si `n` es par, y `False` en caso contrario:

```
par = True if n % 2 == 0 else False
```

3.2. Estructuras repetitivas (bucles)

El bucle `while`

La estructura `while` tiene una sintaxis similar a `if`. El bloque de instrucciones asociado se va a ejecutar repetidamente mientras se cumpla la condición indicada. Por ejemplo, este código le pide números al usuario repetidamente hasta que escribe un número negativo o cero:

```
numero = int(input())
while numero > 0:
    print ("Has escrito", numero)
    numero = int(input())
print ("Fin del programa")
```

No hay una estructura `do..while` en Python, que sí existe en otros lenguajes, ya que cualquier bucle `do..while` se puede representar como `while` con algunos pequeños cambios.

El bucle `for`

Sin embargo, como en otros muchos lenguajes, sí hay una cláusula `for` en Python, y se puede utilizar de varias formas:

- Podemos, por ejemplo, explorar una secuencia determinada de valores. En este ejemplo, la variable `valor` tomará los distintos valores indicados en la lista (2, 4 y 5):

```
for valor in [2, 4, 5]:
    print (valor)
```

- También podemos hacer un uso más “tradicional”, e iterar desde un valor inicial hasta uno final. En este ejemplo, mostramos por pantalla los números del 1 al 4 (inclusive). Si sólo especificamos un número en el rango, entonces Python cuenta del 0 hasta ese número (sin incluir dicho número).

```
for valor in range(1, 5):
    print (valor)
```

- En el caso de que sólo queramos indicar un número de repeticiones sin intención de usar el contador, podemos especificar un único valor en `range`:

```
for i in range(4):
    ...
```

- Finalmente, podemos establecer un incremento distinto de 1, especificando un tercer parámetro dentro de la opción `range`. Por ejemplo, aquí contamos del 0 al 100 con un incremento de 10 en 10:

```
for valor in range(0, 101, 10):
    print (valor)
```

Ejercicio 1:

Crea un programa llamado `Notas.py` que le pida al usuario 3 notas, y calcule la nota final según estas reglas:

- Si ninguna nota es mayor que 4, la nota final es 0
- Si algunas notas son mayores que 4 (pero no todas), la nota final es 2
- Si todas las notas son mayores que 4, la nota final será el 30% de la primera más el 20% de la segunda más el 50% de la tercera

Ejercicio 2:

Crea un programa llamado `Factura.py` que le pida al usuario precios para una factura, hasta que escriba 0. Entonces, el programa debe mostrar el total de la factura con 2 dígitos decimales.

Ejercicio 3:

Crea un programa llamado `MayorMenor.py` que le pida al usuario que introduzca una secuencia de N números positivos (primero el usuario deberá indicar cuántos números va a introducir). Al final del proceso, el programa deberá mostrar por pantalla el valor del número mayor y el menor introducidos por el usuario. Por ejemplo:

```
Dime cuántos números vas a introducir:
3
Escribe 3 números:
3
7
2
El mayor es 7
El menor es 2
```

Ejercicio 4:

Crea un programa llamado `MCD.py` que le pida al usuario dos números $n1$ y $n2$ y utilice el algoritmo de Euclides para calcular su máximo común divisor (MCD). Este número es el divisor mayor que tienen en común los dos números. Aplicando el algoritmo de Euclides, se calcula de la siguiente forma:

1. Dividir el mayor de $n1$ y $n2$ entre el menor
2. Si la división es exacta (resto 0), el MCD es el número menor
3. Si no, se sustituye el número mayor por el resto de la división, y se vuelve al paso 1

Por ejemplo, para 20 y 12 haríamos algo así:

- Dividimos $20 / 12$. No es exacta, y el resto es 8. Reemplazamos 20 por 8
- Dividimos $12 / 8$. No es exacta, y el resto es 4. Reemplazamos 12 por 4
- Dividimos $8 / 4$. Es exacta, con lo que el MCD es 4.

Ejercicio 5:

Crea un programa llamado `InvertirNumero.py` que le pida al usuario un número entero y construya otro en otra variable que sea el original dado la vuelta. Por ejemplo, si el número inicial es 2356, debe construir el 6532.

3.3. Gestión de errores mediante excepciones

Al igual que ocurre en otros lenguajes como Java, en Python podemos controlar los errores no deseados de un programa para que, en lugar de que éste termine de forma anómala y muestre un mensaje confuso al usuario, “atrapar” el error producido, mostrar un mensaje más conciso y seguir ejecutando el programa.

Para ello debemos incorporar el código que puede provocar el error dentro de un bloque `try`. Si dicho código se ejecuta con normalidad, al final del mismo se saldrá del bloque `try` y se continuará con la ejecución del programa. De producirse algún error, se acudirá al bloque `except` que pondremos justo a continuación. En este segundo bloque podemos mostrar el mensaje de error que queramos, y luego seguir con la ejecución del programa. También es habitual recibir un objeto en el bloque `except` que recoge el error producido, con lo que podemos mostrar directamente el mensaje de error almacenado en dicho objeto.

El siguiente ejemplo trata de dividir los dos números escritos por el usuario. Se producirá un error si el segundo número es 0, y lo mostraremos en el bloque `except`:

```
try:
    n1 = int(input("Escribe el dividendo:\n"))
    n2 = int(input("Escribe el divisor:\n"))
    resultado = n1 / n2
    print("El resultado es:", resultado)
except Exception as e:
    print("Error:", str(e))
```

También se pueden provocar o lanzar excepciones en un momento determinado con la instrucción `raise`. Esto permite, entre otras cosas, centralizar los errores producidos en un único punto. Por ejemplo, si no quisiéramos dividir números negativos en el ejemplo anterior podríamos provocar una excepción dentro del `try` para que se capture:

```
try:
    n1 = int(input("Escribe el dividendo:\n"))
    n2 = int(input("Escribe el divisor:\n"))
    if n1 < 0 or n2 < 0:
        raise Exception("No se admiten números negativos")
    resultado = n1 / n2
    print("El resultado es:", resultado)
except Exception as e:
    print("Error:", str(e))
```

Ejercicio 6:

Crea un programa llamado `Impares.py` que pida al usuario un número entero positivo y muestre por pantalla todos los números impares desde 1 hasta ese número, separados por comas. Si el número introducido no es un valor numérico entero, o no es positivo, se lo deberá volver a pedir las veces que sean necesarias antes de hacer el conteo, mostrando el mensaje de error correspondiente (por ejemplo, *Número no válido*).

4. Arrays y listas

Un array es una estructura estática, de tamaño prefijado, que permite almacenar dentro diferentes datos, a los que podemos acceder por su posición numérica en la secuencia (empezando normalmente por la posición 0). En Python no existen arrays propiamente dichos; en su lugar se utilizan **listas**, con un comportamiento similar pero flexible, donde podemos añadir y quitar elementos fácilmente en/de cualquier posición. Similar al ArrayList de Java.

4.1. Definición y uso básico de listas

Una lista en Python será una secuencia de elementos que *pueden ser de distintos tipos*. Se representa con los elementos separados por comas, entre corchetes. Por ejemplo:

```
datos = ['Nacho', 'Pepe', 2015, 2013]
```

Además, podemos crear una lista sin valores iniciales, usando corchetes vacíos o la instrucción `list`:

```
datos = []  
datos = list()
```

Acceder a los elementos de una lista

Como hemos dicho, los elementos de una lista se referencian por un índice o posición numérica, empezando en cero. Así, si tenemos una lista como esta:

```
datos = ['Nacho', 'Pepe', 2015, 2013]
```

y ejecutamos la instrucción `print(datos[1])`, se imprimirá por pantalla el segundo elemento de la lista, que es *Pepe*. Además, Python ofrece la posibilidad de imprimir una lista entera usando la misma instrucción `print`. En este caso, se mostrarían los datos tal cual están almacenados, separados por comas, y entre corchetes:

```
print(datos[1])    # Pepe  
print(datos)      # ['Nacho', 'Pepe', 2015, 2013]
```

Finalmente, Python ofrece la posibilidad de acceder a los elementos desde el final de la lista, usando índices negativos. Así, `datos[-1]` en el ejemplo anterior obtendría el último elemento de la lista (2013 en este caso).

Para recorrer los elementos de una lista, podemos utilizar un `for` que recorra sus elementos...

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

... o bien un `for` que recorra las posiciones:

```
for i in range(0, len(datos)):
    print(datos[i])
```

Listas multidimensionales

Una lista puede tener tipos simples (textos, números, etc) o tipos complejos (como por ejemplo, tuplas, u otras listas, u objetos de clases como veremos en otros documentos del curso). Aquí vemos una lista que internamente contiene otras listas:

```
datos = [
    ["Nacho", 40, 233],
    ["Pepe", 70, 231],
    ...
]
```

En este caso (cuando una lista contiene otras listas, o tuplas), la posición numérica dentro de la lista nos llevará a la lista o tupla que ocupa esa posición, con lo que necesitaremos otro índice numérico para indicar qué dato de esa lista o tupla nos interesa. Así, para el ejemplo anterior, `datos[0][0]` haría referencia al primer elemento de la primera lista (el nombre “Nacho”), y `datos[1][2]` al tercer dato de la segunda lista (el valor 231).

Añadir, modificar y borrar elementos

Si queremos añadir un elemento **al final** de la lista usamos la instrucción `append`. En el caso de querer insertar un elemento **en una posición determinada**, usamos la instrucción `insert`, indicando en qué posición queremos insertar, y el dato que queremos insertar. Automáticamente, todos los elementos a la derecha de esa posición se desplazarán para hacer hueco al nuevo elemento.

También podemos **actualizar el valor** de un dato de la lista, simplemente indicando su posición y el nuevo valor que queremos asignar:

Finalmente, si queremos **eliminar** un dato de la lista, usamos la instrucción `del`, seguida del elemento que queremos borrar. Alternativamente, podemos usar la instrucción `remove` de la propia lista, indicando el dato que queremos borrar. Se borrará la primera ocurrencia de ese dato

Aquí vemos un ejemplo completo de estas instrucciones

```
datos = [1, 2, 3]
datos.append(1000) # [1, 2, 3, 1000]
datos.insert(1, 20) # [1, 20, 2, 3, 1000]
del datos[2] # [1, 20, 3, 1000]
datos.remove(20) # [1, 3, 1000]
```

4.2. Otras operaciones con listas

Hay otras operaciones que nos pueden resultar útiles sobre una lista. Por ejemplo, la instrucción **len(lista)** devuelve el número de elementos actualmente almacenado en la lista (tamaño de la lista). Esto nos puede servir para recorrer tanto listas simples como multidimensionales

```
datos = [1, 2, 3, 4]
print(len(datos))      # 4
datos2 = [
    [1, 2, 3],
    [4, 5, 6, 7]
]
print(len(datos2))     # 2 (cantidad de filas)
print(len(datos2[0]))  # 3 (cantidad de datos de la primera fila)
```

La instrucción **list(valor)** convierte un elemento en una lista de valores.

```
datos = list('123')
print(datos)      # ['1', '2', '3']
```

La operación **lista1 + lista2** concatena los datos de dos listas

```
datos1 = [1, 2, 3, 4]
datos2 = [4, 5, 6]
datosTotales = datos1 + datos2  # [1, 2, 3, 4, 4, 5, 6]
```

La operación **lista * N** genera una nueva lista donde los elementos de la lista original aparecen repetidos N veces, como si de una secuencia se tratase:

```
datos = [4, 5, 6]
datosRepetidos = datos * 3  # [4, 5, 6, 4, 5, 6, 4, 5, 6]
```

La expresión **n in lista** comprueba si el dato *n* está en la lista

```
datos = [4, 5, 6]
if 4 in datos:
    print("Existe el dato 4")
```

Las instrucciones **max(lista)** y **min(lista)** obtienen el mayor / menor valor de la lista, respectivamente. La función **sum** calcula la suma total de los elementos indicados.

```
datos = [4, 5, 6]
print(max(datos))  # 6
print(sum(datos))  # 15
```

La instrucción **lista.count(objeto)** obtiene el número de apariciones del objeto en la lista

```
datos = [4, 5, 6, 4]
print(datos.count(4))  # 2
```

La instrucción **lista.index(objeto)** obtiene la posición donde aparece por primera vez el objeto en la lista. Si el elemento no se encuentra, se produce una excepción en el programa. Podemos pasarle como segundo dato desde qué posición queremos seguir buscando.

```
datos = [4, 5, 6, 4]
print(datos.index(4))      # 0
print(datos.index(4, 1))  # 3
```

La instrucción **lista.sort(funcion)** ordena una lista según el criterio especificado en la función indicada. Para listas de datos simples (listas de enteros, de strings...) no hace falta indicar ninguna función: las listas se ordenan automáticamente de menor a mayor. Si queremos una ordenación inversa, usamos un parámetro adicional llamado **reverse**:

```
datos = [4, 2, 7, 5]
datos.sort()          # [2, 4, 5, 7]
datos.sort(reverse=True) # [7, 5, 4, 2]
```

Finalmente, la instrucción **lista.reverse()** invierte el orden de la lista. Este método NO devuelve una nueva lista, sino que afecta a la original. Si queremos mantener el orden original y que la lista invertida sea otra nueva, podemos usar la instrucción **lista[::-1]** y asignar el valor a otra variable

```
datos = [1, 2, 3, 4]
datos.reverse()          # [4, 3, 2, 1]
datosInvertidos = datos[::-1] # [1, 2, 3, 4]
```

Ejercicio 1:

Crea un programa llamado **ListaInvertida.py** que le pida al usuario que introduzca un conjunto de nombres separados por comas, y le muestre por pantalla la misma lista en orden inverso.

Ejercicio 2:

Crea un programa llamado **BuscaNumeros.py** que le pida al usuario que escriba números. El programa los irá añadiendo uno tras otro a una lista hasta que el usuario escriba 0. Entonces, le pedirá que diga un número y le indicará en qué posiciones de la lista aparece ese número.

Ejercicio 3:

Crea un programa llamado **Identidad.py** que le pida al usuario un tamaño de tabla N y luego le deje rellenar los datos de N filas y N columnas de enteros. Al finalizar, le deberá decir si la tabla que ha rellenado se corresponde o no con una matriz identidad. Una matriz identidad es aquella que tiene unos en su diagonal principal y ceros en el resto. Por ejemplo (para un tamaño 3 x 3):

```
1 0 0
0 1 0
0 0 1
```


5. Cadenas de texto

Ya hemos comentado anteriormente que Python permite manejar cadenas de texto utilizando indistintamente comillas simples o dobles. Podemos crearlas con un texto predefinido en el programa, o pedir las al usuario a través de la instrucción `input`:

```
texto = "Hola"
texto2 = 'Buenas'
texto3 = input("Dime tu nombre: ")
```

5.1. Operaciones básicas con cadenas

Además, podemos hacer algunas operaciones básicas sobre las cadenas de texto, tales como concatenar con `+`:

```
nombre = "Nacho"
texto = "Hola, " + nombre
```

Podemos convertir también cualquier dato simple a cadena usando la instrucción `str`:

```
edad = 43
texto = "Tengo " + str(edad) + " años"
```

Podemos repetir un texto un número determinado de veces, usando el operador `*`:

```
texto = "Hola" * 3      # HolaHolaHola
```

Finalmente, podemos acceder a cada uno de los caracteres de la cadena usando los corchetes (desde la posición 0) y podemos determinar el tamaño de la cadena con la instrucción `len(cadena)`:

```
texto = "Hola";
print(texto[0])    # H (letra en la primera posición)
print(len(texto))  # 4 (tamaño de la cadena)
```

5.2. Otras operaciones sobre cadenas

Además de estas operaciones básicas, también podemos utilizar algunas opciones que también ofrecen otros lenguajes. Por ejemplo, tenemos la instrucción **`split`** que devuelve una lista con los elementos que ha podido **extraer de una cadena**, a partir de un delimitador especificado:

```
texto = "Uno,Dos,Tres"
partes = texto.split(",") # ["Uno", "Dos", "Tres"]
```

El paso inverso, es decir, **unir** partes de un texto con un separador, se puede hacer a través de la instrucción **`join`**

```
partes = ["Uno", "Dos", "Tres"]
texto = ','.join(partes) # Uno,Dos,Tres
```

La instrucción **replace** reemplaza todas las ocurrencias de un texto (primer dato) por otro texto (segundo dato), devolviendo una cadena con el resultado final:

```
texto = "Java es el mejor lenguaje"
texto2 = texto.replace("Java", "Python") # Python es el mejor lenguaje
```

Las instrucciones **lower** y **upper** convierten todo el texto en **minúsculas** / **mayúsculas**, respectivamente, devolviendo una cadena con el resultado:

```
texto = "Hola, buenas"
textoMayus = texto.upper() # HOLA, BUENAS
```

La instrucción **find** permite **buscar** si un texto está contenido dentro de otro, y en qué posición aparece por primera vez. Obtendrá un índice negativo si el texto no se encuentra. Alternativamente, podemos utilizar la expresión **in** para ver si un texto está dentro de otro, sin importar la posición.

```
texto = "Hola, buenas"
posicion = texto.find("buenas")
if posicion >= 0:
    print("Texto encontrado en posición", posicion)
if "buenas" in texto:
    print("El texto contiene \"buenas\"")
```

Si queremos **extraer una subcadena** a partir de una cadena principal, debemos indicar entre corchetes el índice a partir del cual queremos empezar a cortar (inclusive), y el índice donde queremos terminar de cortar (exclusive), separados por dos puntos. Si no indicamos este segundo índice, se corta hasta el final de la cadena.

```
texto = "Hola, buenas tardes"
subcadena = texto[6:12] # buenas
subcadena2 = texto[6:] # buenas tardes
```

Estos índices pueden tomarse desde el inicio de la cadena (valores positivos) o desde el final (valores negativos). Estas dos instrucciones son equivalentes para la cadena de entrada indicada:

```
texto = "Hola, buenas tardes"
subcadena = texto[6:12] # buenas
subcadena2 = texto[6:-7] # buenas
```

A la hora de **comparar** cadenas para ver cuál es mayor o menor alfabéticamente, podemos emplear los operadores de comparación.

```
texto1 = "Hola"
texto2 = "buenas"
if texto1 < texto2:
    print("Es menor \"Hola\"")
```

También nos puede resultar útil **limpiar** una cadena: eliminar espacios innecesarios al inicio o al final. Para ello podemos emplear la instrucción **strip** (elimina espacios a ambos lados) o bien **lstrip** y **rstrip** para eliminar sólo por la izquierda o derecha, respectivamente:

```
texto1 = "\tHola  "  
texto2 = texto1.strip() # "Hola"
```

Ejercicio 1:

Crea un programa llamado SumaSecuencia.py que le pida al usuario una secuencia de números separados por espacios y calcule la suma total de esos números.

Ejercicio 2:

Crea un programa llamado CuentaTexto.py que le pida al usuario un texto, y luego le diga cuántas veces aparece la palabra Python en ese texto.

6. Diccionarios y otras estructuras de datos

En este apartado vamos a analizar otras estructuras de datos soportadas por Python que, si bien no son las más habituales, nos van a permitir organizar la información de cierta forma para poder acceder a ella.

6.1. Tuplas

Las tuplas son un tipo especial de datos en Python, que se parecen algo a las estructuras o *structs* de C o C#. Permiten definir un conjunto de datos heterogéneos, pero esos datos, una vez se definen, son **inmutables**, es decir, no podemos modificar su valor. Cada dato dentro de la tupla tiene un índice (empezando por cero), para poder acceder a cada dato.

Para definir una tupla, especificamos sus datos (entre paréntesis si queremos, aunque no es obligatorio):

```
nombres = ("John", "Helen", "Mary")
nombres2 = "Susan", "Adam", "Peter"
```

Como decimos, podemos especificar también diferentes datos en cada posición. Por ejemplo, para almacenar información personal de una persona:

```
datosPersonales = ("John Doe", 36, "611223344")
```

Después, podemos acceder a cada elemento con un índice entre corchetes, empezando por el 0:

```
print(datosPersonales[0]) # John Doe
```

Las tuplas ofrecen mucha flexibilidad a la hora de acceder a sus elementos, o asignarlos a variables separadas. Observemos este ejemplo:

```
datos = 20, "Nacho", 55.4
print(datos) # (20, "Nacho", 55.4)
elem1, elem2, elem3 = datos
print(elem2) # Nacho
elem2 = "May"
print(elem2) # May
datos[1] = "May" # Error: tupla immutable
```

Como podemos ver, podemos sacar datos de una tupla de forma individual, y luego modificar esas variables. O bien tratar la tupla como un todo (variable `datos` anterior), en cuyo caso los valores almacenados son inmutables.

Ejercicio 1:

Crea un programa llamado **Tuplas.py** donde le pidas al usuario que rellene su dirección postal, que estará formada por su nombre de calle (texto), número de puerta (entero) y número de piso (entero). Almacena los datos en una tupla y luego muestra por pantalla el resultado (campo a campo).

6.2. Diccionarios

Los diccionarios, también denominados mapas o tablas *hash* en muchos lenguajes de programación, permiten almacenar información en forma de pares *clave-valor*, donde cada valor almacenado en la colección tiene asociada una clave, y accedemos a dicho valor a través de su clave.

Para definir un diccionario inicialmente vacío, se utiliza una pareja de llaves:

```
datos = {}
```

Después, para acceder a cada posición del diccionario y darle un valor o consultar el valor que tiene, debemos hacerlo siempre por su clave. Por ejemplo, imaginemos que estamos gestionando un diccionario con un catálogo de productos de una tienda. La clave para identificar cada producto será su código (alfanumérico, por ejemplo), y el valor asociado serán los datos del producto, que podemos representar como una tupla con su código, título y precio, por ejemplo. De este modo podríamos crear e inicializar el diccionario:

```
productos = {}
productos['111A'] = ('111A', 'Monitor LG 22 pulgadas', 99.95)
productos['222B'] = ('222B', 'Disco duro 512GB SSD', 109.45)
productos['333C'] = ('333C', 'Ratón bluetooth', 19.35)
```

Para acceder a un elemento en concreto, siempre tendrá que ser con su clave. Podemos utilizar el operador **in** para verificar previamente si existe la clave en el diccionario:

```
if '111A' in productos:
    print(productos['111A'][1]) # Monitor LG 22 pulgadas
```

También podemos recorrer todo el diccionario. En este caso podemos usar algunas alternativas:

```
# Recorrer todas las claves y con ellas sacar los valores
for clave in productos:
    print(productos[clave][1]) # Títulos de los productos

# Recorrer el diccionario sacando clave y valor
for clave, valor in productos.items():
    print(valor[1])           # Títulos de los productos
```

Existen otras operaciones habituales sobre diccionarios, como por ejemplo:

- **pop** elimina la clave que indiquemos del diccionario, junto con su valor asociado.

```
productos.pop('111A')
```

- Las operaciones **len** y **clear** sirven, respectivamente, para obtener el número de elementos del diccionario, o para borrarlos todos.

```
print("El diccionario tiene", len(productos), "productos")
productos.clear()
```

Ejercicio 2:

Crea un programa llamado **DiccionarioTuplas.py** donde le pidas al usuario que rellene direcciones de 4 usuarios, identificados por su clave que será el DNI. Así, para cada usuario rellenará dicho DNI, y luego los datos de la dirección como en el ejercicio anterior (nombre de calle, número de puerta y número de piso). Almacenará los datos en un diccionario (asociando cada DNI con su dirección) y luego le pedirá al usuario que escriba un DNI y mostrará los datos de su dirección, o el mensaje “El DNI no se encuentra almacenado” si no existe dicha clave.

6.3. Conjuntos

Los conjuntos son un tipo de colección que no admite duplicados. Se pueden crear directamente pasando los elementos entre llaves, y luego con las operaciones `add` y `remove` añadimos o quitamos elementos al conjunto, respectivamente. También podemos crear conjuntos a partir de listas usando la función `set`, que se encarga automáticamente de eliminar duplicados.

Conviene tener presente que en los conjuntos no existe un orden ni una secuencia numerada de elementos, como sí ocurre en las listas. Veamos un ejemplo:

```
elementos = ["Uno", "Dos", "Dos", "Tres"]
conjunto = set(elementos)
print(conjunto)
# {"Dos", "Uno", "Tres"}

conjunto.add("Uno")
conjunto.add("Cuatro")
print(conjunto)
# {"Dos", "Uno", "Tres", "Cuatro"}

conjunto.remove("Tres")
print(conjunto)
# {"Dos", "Uno", "Cuatro"}
```

7. Programación modular: funciones

Las funciones nos permiten agrupar el código en bloques reutilizables. De este modo evitamos repetir innecesariamente el código y, además, podemos reutilizarlo en diferentes partes del programa.

7.1. Definición de funciones

A la hora de definir una función en Python, comenzamos con la palabra `def` seguida del nombre de la función y los parámetros que tendrá, entre paréntesis. Para cada parámetro sólo debemos especificar su nombre (recuerda que en Python no se especifican los tipos de datos explícitamente).

Igual que ocurre con otras estructuras como `if` o `while`, el código perteneciente a una función debe estar tabulado. Además, si queremos que la función devuelva algún valor, podemos emplear la cláusula `return` como en otros lenguajes, aunque no es obligatoria si no queremos devolver nada. También podemos definir un `return` vacío para indicar que no se devuelve nada.

Veamos algunos ejemplos.

- Esta función recibe dos parámetros y devuelve el mayor de ellos

```
def maximo(num1, num2):  
    if num1 > num2:  
        return num1  
    else:  
        return num2
```

- Esta función recibe un texto como parámetro y lo saca por pantalla:

```
def imprimeTexto(texto):  
    print(texto)  
    return # Esta línea se podría omitir
```

A la hora de llamar a estas funciones desde otras partes del código, se hace igual que en otros lenguajes:

```
print ("Escribe dos números: ")  
n1 = int(input())  
n2 = int(input())  
print ("El máximo es: ", maximo(n1, n2))  
  
texto = input("Escribe un texto:")  
imprimeTexto(texto)
```

Más sobre el valor de retorno

Hemos visto que con la instrucción `return` podemos hacer que la función devuelva un resultado. Este resultado puede ser un valor simple (por ejemplo, un número) o un dato complejo, o compuesto de varios elementos. En este último caso, podemos hacer que la función devuelva:

- Una lista de valores
- Un mapa o diccionario de datos
- Una tupla
- ...

La siguiente función devuelve una lista con los datos que recibe como parámetros:

```
def lista(n1, n2, n3):  
    return [n1, n2, n3]  
  
datos = lista(1, 2, 3)  
print(datos[1]) # 2
```

Un uso curioso de esta particularidad es el trabajo con tuplas: podemos hacer que una función devuelva una secuencia de datos (varios datos), separados por comas, y asignar el resultado de la llamada a un conjunto de variables, también separadas por comas. Por ejemplo, la siguiente función devuelve un número y un nombre de persona. Al llamarla, podemos obtener de golpe los dos valores devueltos, y asignarlos a dos variables independientes:

```
def mi_funcion():  
    return 20, "Nacho"  
  
numero, nombre = mi_funcion()  
print(numero)    # 20  
print(nombre)    # Nacho
```

La instrucción *pass*

En algunas ocasiones nos puede interesar definir la cabecera de una función y no implementar (aún) su código. En este caso, para no dejar la función vacía (lo que daría un error de ejecución) podemos emplear la instrucción vacía **pass** como única instrucción de la función (que no tiene ningún efecto), y ya la completaremos más adelante:

```
def mi_funcion():  
    pass
```


7.2. Parámetros

Veamos a continuación algunos aspectos relevantes sobre los parámetros que pasamos a las funciones.

Paso por valor y por referencia

En Python todos los parámetros simples (números, booleanos y textos) se pasan por valor, con lo que no podemos modificar el valor original del dato (se pasa una copia del mismo), y todos los tipos complejos (listas, u objetos) se pasan por referencia. Esto último implica que, siempre que se mantenga la referencia original, podemos modificar el valor del parámetro de forma persistente (se aplica a la variable original utilizada como parámetro). Por ejemplo, si empleamos esta función:

```
def anyadirValores(lista):  
    lista.append(30)  
    print ("Valores en la función:", lista)  
    return
```

y llamamos a la función de este modo:

```
lista1 = [10, 20]  
anyadirValores(lista1)  
print ("Valores fuera de la función:", lista1)
```

Entonces la variable *lista1* y el parámetro *lista* almacenan los mismos valores finales: [10, 20, 30]. Sin embargo, si usamos esta otra función:

```
def anyadirValores(lista):  
    lista = [30, 40]  
    print ("Valores en la función:", lista)  
    return
```

y llamamos a la función del mismo modo que antes, entonces la variable original *lista1* tendrá los valores [10, 20] al finalizar, y el parámetro *lista* tendrá los valores [30, 40] dentro de la función, pero este cambio se pierde fuera de la misma, porque se ha modificado la referencia de la variable (la hemos reasignado entera en la función), y por tanto hemos creado una nueva referencia distinta a la original, que no modifica entonces su contenido.

Tipos de parámetros

Los parámetros definidos en una función pueden ser de distintos tipos, y se pueden especificar de distintas formas. Veremos aquí algunos ejemplos.

Por un lado tenemos los parámetros **obligatorios**. Es el modo normal de pasar parámetros; si simplemente definimos el nombre de cada parámetro, entonces ese parámetro es obligatorio, y debemos darles valor al llamarles, en el mismo orden en que están definidos. Aquí podemos ver un ejemplo (el mismo visto anteriormente):

```
def maximo(num1, num2):  
    if num1 > num2:  
        return num1  
    else:  
        return num2
```

También podemos invocar a una función usando los nombres de los parámetros como **palabras clave**. De este modo no tenemos por qué seguir el mismo orden que cuando se definió dicha función. Por ejemplo:

```
def imprimirDatos(nombre, edad):  
    print ("Tu nombre es", nombre, "y tu edad es", edad)  
    return  
...  
imprimirDatos(edad = 28, nombre = "Juan")
```

Además, podemos asignar **valores por defecto** a los parámetros que queramos. Así, si queremos llamar a la función, podemos omitir los parámetros que tengan un valor por defecto asignado, si queremos. Por ejemplo:

```
def imprimirDatos(nombre, edad = 0):  
    print ("Tu nombre es", nombre, "y tu edad es", edad)  
    return  
...  
imprimirDatos("Juan") # Imprime "Tu nombre es Juan y tu edad es 0"
```

NOTA: es importante que los parámetros que tengan valores por defecto se coloquen todos al final de la lista de parámetros (tras los obligatorios), para que así no queden huecos si queremos llamar a la función omitiendo parámetros. También es importante que, cuando omitamos un parámetro, los que vayan detrás también se omitan para que no se desplace el orden y se asignen por error a otros parámetros.

Funciones con un número variable de parámetros

Las funciones en Python también admiten un **número variable de parámetros**. Esto lo podemos especificar como último parámetro de la función un tipo especial que permite pasar tantos parámetros como necesitemos. Por ejemplo:

```
def imprimirTodo(num1, *numeros):  
    print("Primer número:", num1)  
    for num in numeros:  
        print num  
    return
```

Lo que hará la función en este caso es recibir un primer parámetro obligatorio (*num1*) y el resto, opcionales, se recibirán en forma de **tupla** con sus valores. Podemos invocar a la función así:

```
imprimirTodo(1, 2, 3, 4) # La tupla sería (2, 3, 4) en este caso
```

De forma alternativa podemos indicar un doble asterisco en ese último parámetro:

```
def imprimirTodo(num1, **valores):
    print("Primer número:", num1)
    for num in valores:
        print(valores[num])
    return
```

En este otro caso lo que se recibe como parámetro adicional es un **mapa** donde a cada parámetro (valor) se le asocia un nombre (clave). Podríamos invocar a la función de este modo:

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

Podemos **combinar ambas cosas** en una función que admita primero una secuencia de valores y luego una secuencia de valores con nombre asociado, por ejemplo:

```
def imprimirTodo(*numeros, **valores):
    ...
```

Y la podríamos invocar así:

```
imprimirTodo(1, 2, 3, a = 4, b = 5)
# El primer parámetro recogería la tupla (1, 2, 3)
# El segundo parámetro recogería el mapa {"a": 4, "b": 5}
```

Ejercicio 1:

Crea un programa llamado `Funciones.py` con las siguientes funciones:

1. Una función llamada `mcd` que reciba dos enteros a y b como parámetros y devuelva el máximo común divisor de esos parámetros. El máximo común divisor es el número más alto por el que se pueden dividir los dos números.
2. Una función llamada `esPrimo` que reciba un número como parámetro y devuelva un booleano indicando si el número es primo o no

Desde el programa principal, llama a la función `mcd` para calcular el máximo común divisor de 20 y 12 (debería dar 4), y usa la función `esPrimo` para sacar los números primos que haya del 1 al 50.

Paso de parámetros al programa principal

A pesar de que en Python no existe una función principal `main` como la que sí existe en otros lenguajes como C, Java, C#... sí es posible pasar parámetros al programa desde el terminal cuando lo ejecutamos. Para ello, importamos el elemento `sys`, que hace referencia al sistema sobre el que se ejecuta el programa. Dentro, disponemos de un array predefinido llamado `argv`, similar al que existe en C o C++, con los datos que le llegan al programa. El primero de ellos, igual que ocurre en C o C++ es el nombre del propio ejecutable, y el resto son los parámetros adicionales.

```
import sys

for i in range(1, len(sys.argv)):
    # Recorremos los parámetros quitando el 0 (que es el ejecutable)
    print(sys.argv[i])
```

Ejercicio 2:

Crea un programa llamado `Contar.py` que reciba como parámetros del programa principal dos datos (numéricos) y realice un conteo desde el primero hasta el segundo. Si no se reciben los dos datos mostraremos un mensaje de error y finalizaremos.

7.3. Operaciones avanzadas con listas

En este apartado veremos algunas operaciones algo más complejas que se pueden realizar con listas, y que requieren definir alguna función adicional.

Ordenación de listas

Ya hemos visto en documentos anteriores que la instrucción **sort** permite ordenar automáticamente listas simples. Pero si queremos ordenar algunos datos más complejos (como objetos, o tuplas) debemos proporcionar una función que indique el criterio de comparación. Por ejemplo, esta lista de tuplas queda ordenada ascendentemente por su edad (segundo campo):

```
def ordenarPorEdad(persona):  
    return (persona[1])  
  
gente = [("John Doe", 36, "611223344"),  
         ("Mary Stewart", 54, "733445566"),  
         ...]  
gente.sort(key=ordenarPorEdad)
```

Mapeo de listas

La instrucción **map** aplica una función de transformación a una lista y devuelve los elementos transformados. Estos elementos pueden formar de nuevo una lista usando la instrucción **list**. Este ejemplo obtiene una lista con los cuadrados de los números de la lista original:

```
def cuadrado(x):  
    return (x * x)  
  
lista = [1, 2, 3, 4]  
cuadrados = list(map(cuadrado, lista))
```

Filtrado de listas

La instrucción **filter** aplica una función de filtrado a una lista y devuelve los elementos que cumplen la condición o pasan el filtro. Como ocurre con *map*, podemos formar una nueva lista con ellos usando la instrucción **list**. El siguiente ejemplo se queda con los números pares de una colección:

```
def par(x):  
    return x % 2 == 0  
  
lista = [1, 2, 3, 4]  
pares = list(filter(par, lista))
```

7.4. Nociones sobre programación funcional

Algunas funciones son bastante simples y ocupan una simple línea de código. Por ejemplo, echemos un vistazo a un fragmento de código anterior que transforma un array de números en sus cuadrados:

```
def cuadrado(x):  
    return (x * x)  
  
lista = [1, 2, 3, 4]
```

```
cuadrados = list(map(cuadrado, lista))
```

Cuando la función es así de simple, se puede reemplazar por una **expresión lambda**. Estas expresiones se utilizan en muchos lenguajes para implementar funciones normalmente cortas, de forma que ocupan una línea de código. Además, tienen la peculiaridad de que se pueden definir en el mismo punto en que se quieren utilizar. Empleando una expresión lambda, el código anterior quedaría así:

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

Como vemos, la notación consiste en utilizar la palabra **lambda** seguida de los parámetros que necesita la función (separados por comas), dos puntos y el resultado a devolver. Aplicando esto mismo al ejemplo anterior que filtra los números pares usando *filter*, podría quedar un código así:

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

A la hora de ordenar colecciones de datos, también podemos definir con lambdas el criterio de ordenación. El siguiente ejemplo ordena una lista de tuplas por el segundo campo de dichas tuplas (edad numérica):

```
gente = [("John Doe", 36, "611223344"),
         ("Mary Stewart", 54, "733445566"),
         ...]
gente.sort(key=lambda x: x[1])
```

Ejercicio 3:

Crea un programa llamado `Loteria.py` que le pida al usuario que introduzca los 6 números que juega a la lotería (separados por espacios). Entonces, deberá crear una lista con ellos, ordenarla ascendentemente e imprimirla en pantalla. Además, el programa debe indicar si es una lista válida (es decir, los números deben estar entre 1 y 49, inclusive, sin repetirse). Por ejemplo:

```
Introduce los 6 números de la lotería separados por espacios
1 20 12 20 6 50
[1, 6, 12, 20, 20, 50]
La lista NO es válida:
Hay números repetidos
Hay números menores que 1 o mayores que 49
```

7.5. Módulos

Python es un lenguaje que se puede (y muchas veces debe) modularizar, es decir, dividir su código en distintos módulos reutilizables. De hecho, el propio núcleo de Python que instalamos en el sistema ya está modularizado, y podemos incorporar algunos de esos módulos a nuestros programas con la instrucción `import`. También podemos instalar módulos adicionales en el sistema (o en el proyecto) e incorporarlos con esta misma instrucción. Veremos cómo hacer esto en este documento.

Importando módulos propios de Python

Como decíamos, la instrucción `import` nos permite incorporar a nuestro programa módulos del núcleo de Python para poderlos utilizar. En el siguiente ejemplo importamos el módulo `sys` para acceder a los parámetros que se pasan al programa principal (`sys.argv`).

```
import sys

for i in range(1, len(sys.argv)):
    # Recorremos los parámetros quitando el 0 (ejecutable)
    print(sys.argv[i])
```

En la instrucción `import`, podemos añadir una partícula `as` para dar un nombre alternativo al módulo a la hora de utilizarlo:

```
import sys as sistema

for i in range(1, len(sistema.argv)):
    # Recorremos los parámetros quitando el 0 (ejecutable)
    print(sistema.argv[i])
```

Alternativamente, también podemos elegir importar sólo una parte (o partes, separadas por comas) del módulo en cuestión. Por ejemplo, de este modo importamos únicamente la constante `pi` y la función `sqrt` del módulo `math`:

```
from math import pi, sqrt
print(pi)
print(sqrt(pi * 3))
```

7.5.1. Descomponiendo nuestro código en módulos

A medida que el código de nuestro programa crece, puede ser necesario dividirlo en distintos ficheros fuentes. Al hacer esto, podemos emplear la instrucción `import` para “cargar” o utilizar unos módulos dentro de otros. Por ejemplo, supongamos que incluimos en un archivo llamado *modulo.py* el siguiente contenido:

```
constantePi = 3.141592

def sumar(a, b):
    return a + b
```

Si queremos utilizar estos elementos desde otro fichero, basta con importarlo:

```
import modulo

print(modulo.sumar(3, 4))
print(modulo.constantePi * 8)
```

Podemos utilizar esta otra sintaxis alternativa para importar directamente unos elementos seleccionados:

```
from modulo import constantePi, sumar

print(sumar(3, 4))
print(constantePi * 8)
```

En el caso de trabajar con clases, si tenemos cada clase en un archivo, será conveniente utilizar también la cláusula `from` para indicar que de ese archivo se importe la clase, y así no tener que poner ningún prefijo. Por ejemplo, si tenemos la clase `Persona` en el fichero `Persona.py`:

```
class Persona:
    def __init__(self, ...):
        ...
```

Y queremos utilizarla desde otro archivo, en lugar de hacer `import Persona` (que nos obligaría después a usar `Persona.Persona` para referirnos a la clase), podemos hacer:

```
from Persona import Persona
```

Trabajando con varios archivos fuente

Si nuestro proyecto empieza a ser complejo y queremos descomponerlo en distintos archivos fuente, necesitamos un IDE que nos permita trabajar cómodamente con todos estos archivos. Por ejemplo, podemos emplear el IDE **PyCharm**, del que ya hemos hablado. Basta con instalarlo, crear un proyecto e ir colocando nuestro código fuente en distintos archivos dentro de ese proyecto.

7.5.2. Instalación de módulos adicionales

Como comentábamos al inicio de este documento, podemos enriquecer nuestras aplicaciones Python añadiendo módulos de terceros en nuestro sistema. Esto puede llevarse a cabo con la herramienta `pip` (*Package Installer for Python*). Esta herramienta viene ya incorporada en las últimas versiones de Python. Podemos utilizarla de este modo:

```
pip install <nombre_modulo>
```

NOTA: en algunas versiones de Linux y Mac el comando se llama `pip3` en lugar de `pip`.

Podemos consultar los paquetes o módulos que tenemos instalados con el siguiente comando:

```
pip list
```

Ejemplo: Pillow

Pillow es una versión simplificada de una librería llamada PIL (*Python Imaging Library*) que permite manipular imágenes desde Python. Por ejemplo, escalarlas, rotarlas, cambiarles el formato, etc. Muy utilizada cuando trabajamos con redes neuronales convolucionales, por ejemplo.

Instalación, primeros pasos y operaciones básicas

Para utilizar Pillow en nuestro sistema, deberemos instalar la librería con un comando como éste:

```
pip install Pillow
```

Después, deberemos incorporar (importar) la librería en los archivos fuente que la necesiten. Normalmente basta con incorporar el módulo `Image`:

```
from PIL import Image
```

Una vez importada la librería, podemos abrir imágenes con el método `open` del elemento `Image`:

```
imagen = Image.open("fichero.png")
```

Podemos generar miniaturas de una imagen con el método `thumbnail` indicando el tamaño final como una tupla:

```
tamano = (64, 64)
miniatura = imagen.thumbnail(tamano)
```

Podemos reescalar la imagen a cualquier tamaño con `resize` (también pasando una tupla con el tamaño deseado):

```
tamano2 = (300, 300)
redimensionada = imagen.resize(tamano2)
```

O rotarla con el método `rotate` (indicando el ángulo de rotación en grados):

```
rotada = imagen.rotate(45)
```

También podemos hacer operaciones de transposición, como voltear la imagen horizontal o verticalmente:

```
volteoHoriz = imagen.transpose(Image.Transpose.FLIP_LEFT_RIGHT)
```

Finalmente, podemos guardar los cambios en otro archivo con el método `save`, indicando el nombre del archivo destino y la extensión o formato de salida (opcional):

```
rotada.save("rotada.png")
volteoHoriz.save("volteada.jpg", "JPEG")
```

Puedes encontrar más información sobre esta librería en su página oficial

Ejercicio 2:

Haz un programa que le pida al usuario un nombre de imagen, la cargue y, si existe, le pida continuamente tamaños a los que la quiera escalar (ancho y alto), y guarde una copia de la imagen original con ese tamaño. El nombre de cada imagen escalada tendrá el patrón *ancho_alto_nombreFicheroOriginal*. El proceso terminará cuando el usuario ponga la anchura o altura a 0.

NOTA: para saber si un archivo existe o no puedes usar el módulo `os.path` del núcleo de Python, y su método `isfile`:

```
import os.path

if (os.path.isfile(nombreFichero)):
    ...
```


7.5.3. Más operaciones con módulos

Además de instalar módulos, existen otras operaciones básicas que podemos necesitar hacer con ellos.

Actualización de módulos

Podemos actualizar módulos previamente instalados, e incluso la propia herramienta *pip*. Para actualizar el propio *pip*, ejecutamos el siguiente comando:

```
pip install pip -U
```

Para actualizar cualquier módulo instalado, basta con añadir el parámetro `-U`:

```
pip install Pillow -U
```

Consultar versión actual

Podemos consultar la versión actual de *pip* con:

```
pip --version
```

También podemos obtener información sobre uno de los módulos instalados con `pip show`:

```
pip show Pillow
```

Eliminar módulos

Para eliminar un módulo instalado ejecutamos el siguiente comando:

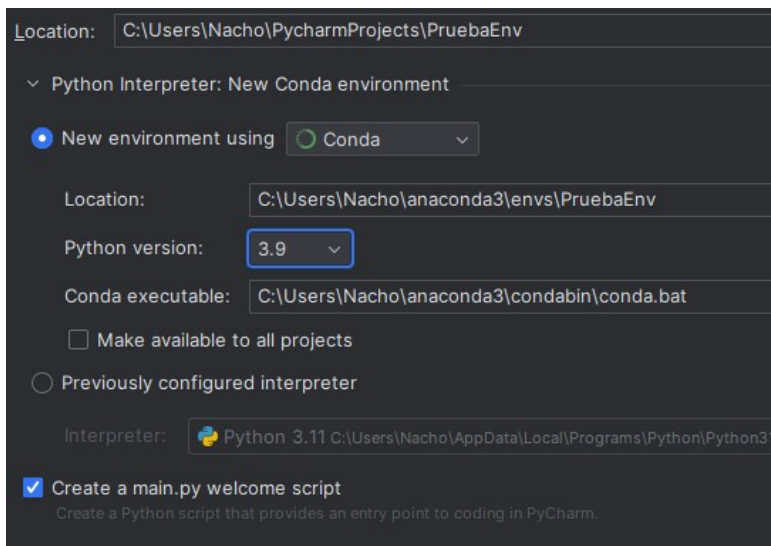
```
pip uninstall nombre_modulo
```

7.5.4. Gestión de dependencias con entornos virtuales y Conda

En algunos casos es posible que nos interese instalar ciertas librerías o módulos que no son compatibles con la versión de Python que tenemos actualmente instalada, o que queramos instalar un conjunto de módulos interdependientes entre sí con versiones específicas. Hacer eso en la misma máquina en la que conviven otras instalaciones puede ser algo complicado pero, afortunadamente, podemos crear entornos separados usando la herramienta **Conda**.

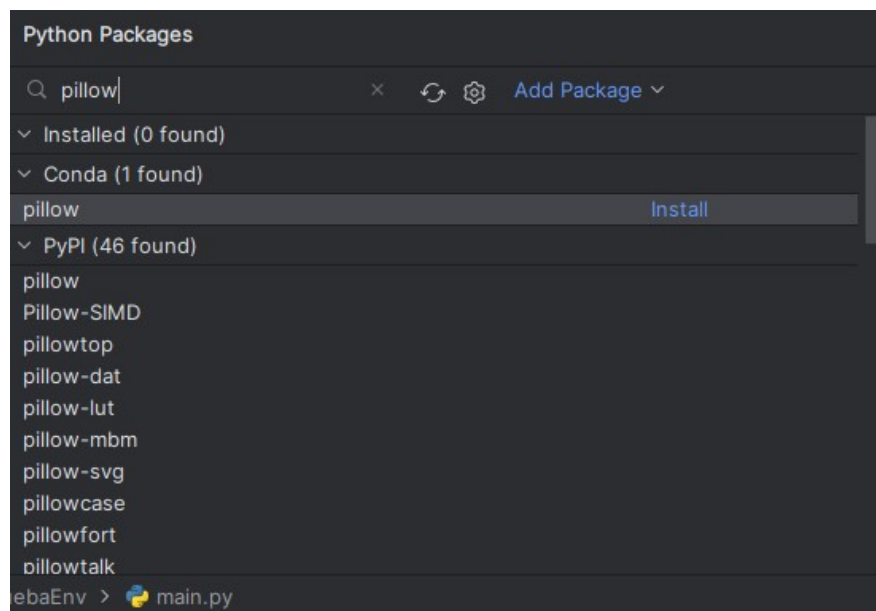
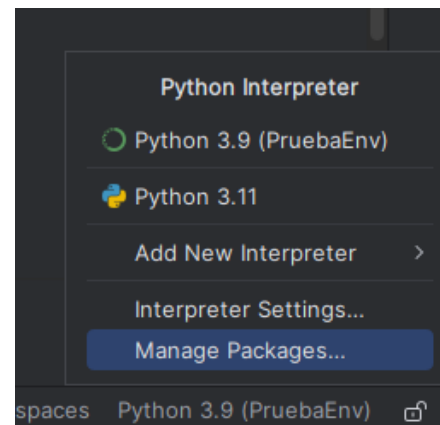
Existen varias formas de instalar el gestor *Conda*, aunque una de las más habituales es descargando e instalando la herramienta *Anaconda*. Al instalarla, además de todas las herramientas que incorpora, también tendremos disponible el gestor *Conda*.

El siguiente paso será crear un **entorno virtual** usando Conda. Podemos emplear el IDE PyCharm para ello. Creamos un nuevo proyecto, y elegimos *New environment*, empleando Conda como gestor del entorno. Notar que también podemos elegir qué versión de Python añadir al entorno (cualquiera, aunque no la tengamos instalada específicamente en nuestro sistema):



Una vez se ha creado el proyecto con el entorno, podremos gestionar los paquetes o módulos que forman parte del entorno desde la esquina inferior derecha de la ventana de PyCharm, eligiendo la opción *Manage Packages*:

Aparecerá un panel donde podremos buscar los paquetes o módulos a instalar, y hacer clic en el enlace *Install*. Podremos elegir también qué versión instalar del módulo en cuestión:



NOTA: es **IMPORTANTE** recalcar que los módulos que instalemos sólo estarán presentes en el entorno virtual creado para el proyecto específico, y no para otros proyectos (deberemos repetir los mismos pasos si queremos tener lo mismo en otros proyectos).

8. Programación orientada a objetos

Python es un lenguaje multiparadigma, lo que significa que podemos utilizarlo enfocado a distintos paradigmas o formas de resolver problemas. En los apartados anteriores nos hemos centrado en un paradigma estructurado y modular, donde descomponíamos el código en funciones, y llamábamos a unas u otras en un orden determinado para resolver el problema.

Además de lo anterior, Python también permite estructurar nuestro código en clases, de forma que cada clase es una entidad que representa un conjunto de elementos del programa (clientes, productos, pedidos, etc.), y podremos crear objetos de esas clases que interactuarán entre sí.

8.1. Definición de clases y objetos

Las clases en Python se definen con la palabra `class` seguida del nombre de la clase. Después, con el código tabulado, indicamos los diferentes elementos de la clase: constructor métodos, etc. Por ejemplo, de este modo definiríamos una clase `Persona` que almacene datos básicos de personas, como su nombre, edad o teléfono:

```
class Persona:

    # Constructor
    def __init__(self, n, e, t):
        self.nombre = n
        self.edad = e
        self.telefono = t

    # Otro método de ejemplo
    def mostrar(self):
        print(self.nombre)
```

Como podemos ver, el constructor de una clase en Python se llama `__init__`. El primer parámetro que recibe (`self`) es una partícula especial que servirá para hacer referencia a cualquier elemento de la clase (atributos o métodos). De este modo, `self.nombre` hace referencia al atributo `nombre` (que no es necesario definir previamente, como en otros lenguajes). El resto de parámetros del constructor son los valores que vamos a asignar a cada atributo respectivamente: nombre (*n*), edad (*e*) y teléfono (*t*).

Adicionalmente, podemos especificar otros métodos que podamos necesitar, como en este caso el método `mostrar` que muestra por pantalla el nombre de la persona.

Instanciación de objetos

Para crear objetos de una clase, basta con usar su constructor, y para ello escribimos el nombre de la clase y le pasamos los parámetros que necesita el constructor (excluyendo el parámetro `self`, que va implícito). Así crearíamos un objeto de tipo `Persona`, y llamaríamos a su método `mostrar`:

```
p1 = Persona("María", 52, "677889900")
p1.mostrar()
```

Del mismo modo, podemos añadir objetos dentro de, por ejemplo, listas:

```
personas = [
    Persona("Juan", 70, "611223344"),
    Persona("Ana", 40, "675849301")
]
personas.append(Persona("María", 52, "677889900"))
```

Visibilidad pública o privada

Por defecto todos los elementos de una clase en Python tienen visibilidad pública. Eso quiere decir que, en el ejemplo anterior, podríamos acceder directamente a los atributos de la clase desde fuera de la misma:

```
p1 = Persona("María", 52, "677889900")
p1.edad = 53
print(p1.telefono)
```

En el caso de no querer que sea así, se deben nombrar los elementos privados anteponiéndoles el símbolo del subrayado por partida doble `__`. La clase anterior quedaría así, dejando los atributos privados:

```
class Persona:

    # Constructor
    def __init__(self, n, e, t):
        self.__nombre = n
        self.__edad = e
        self.__telefono = t

    # Otro método de ejemplo
    def mostrar(self):
        print(self.__nombre)
```

En este caso, convendría definir unos métodos públicos de acceso a los elementos privados; los típicos *getters* y *setters* de otros lenguajes, que en Python tienen una nomenclatura particular, ya que debemos utilizar ciertas anotaciones para especificar que ciertos métodos son *getters* o *setters*. Así, por ejemplo, definiríamos el *getter* y el *setter* para la edad en la clase anterior:

```
class Persona:

    # Constructor
    def __init__(self, n, e, t):
        self.__nombre = n
        self.__edad = e
        self.__telefono = t

    def mostrar(self):
        print(self.__nombre)

    @property      # Getter
    def edad(self):
```

```

        return self.__edad

    @edad.setter    # Setter
    def edad(self, e):
        if e >= 0:
            self.__edad = e

```

Desde el programa principal, podemos usarlos de este modo:

```

p1 = Persona("María", 52, "677889900")
p1.edad = 30
print(p1.edad)

```

Ejercicio 1:

Escribe un programa en Python llamado **Jugadores.py** que defina una clase llamada **Jugador**, con atributos dorsal (numérico) y nombre (texto). Define el constructor para darles valor y un método que muestre la información de un jugador con el formato **dorsal. Nombre..** Por ejemplo: **16. Pau Gasol**. En el programa principal, crea un par de jugadores con sus datos, y muestra su información por pantalla.

Ejercicio 2:

Escribe una nueva versión del ejercicio anterior en un programa llamado **Equipos.py** donde, además de la clase **Jugador** haya una segunda clase llamada **Equipo**, cuyo único atributo será el nombre del equipo (texto), junto con un constructor para darle valor. Haz que cada jugador pueda pertenecer a un equipo añadiendo un atributo *Equipo* a la clase *Jugador*. En el programa principal, crea un jugador y un equipo, y asigna el equipo al jugador. Muestra por pantalla la información del jugador, incluyendo el equipo.

8.2. Herencia

Para ilustrar el mecanismo de herencia en Python, vamos a suponer que disponemos de una clase **Persona** con un nombre y una edad como atributos (además de constructores y métodos) y de ella vamos a heredar para crear una clase **Programador**, que incorporará todos los elementos de su clase base (**Persona**) y añadirá como atributo propio el lenguaje en que programa.

- Para indicar que una clase **hereda de** otra clase, se indica el nombre de la clase a heredar entre paréntesis, tras el nombre de la clase hija.
- Para **acceder desde una clase hija a los elementos de una clase padre** se emplea el método `super()`. Se puede utilizar tanto en el constructor (para llamar al constructor de la clase padre y pasarle los parámetros que necesite) como desde cualquier otro método.
- Para **redefinir un método de la clase padre en la hija** no es necesario marcarlo de ninguna forma especial. Simplemente se vuelve a definir el método con su nuevo código (que se puede apoyar en el del padre, si lo necesita).

Veamos todo esto en el ejemplo:

```
# Clase padre
class Persona:

    def __init__(self, n, e):
        self.nombre = n
        self.edad = e

    def mostrar(self):
        return nombre + ", " + str(edad) + " años"

    # Otros métodos...

# Clase hija
class Programador(Persona):

    def __init__(self, n, e, l):
        super().__init__(n, e)
        self.lenguaje = l

    def mostrar(self):
        return super().mostrar() + "\nPrograma en " + self.lenguaje
```

Herencia múltiple en Python

En Python se **admite herencia múltiple**, de forma que una clase puede heredar de más de una clase padre. Supongamos el caso anterior, donde la clase **Programador** hereda tanto de **Persona** como de **Empleado**. De esta última clase, incorpora el nombre de la empresa donde trabaja y el salario mensual.

```
# Clase padre 1: Persona
class Persona :
    # Mismo código que en el ejemplo anterior

# Clase padre 2: Empleado
class Empleado:

    def __init__(self, e, s):
        self.empresa = e
        self.salario = s

    # Otros métodos...

# Clase hija que hereda de ambos
class Programador (Persona, Empleado):

    def __init__(self, nombre, edad, empresa, salario, lenguaje):
        Persona.__init__(nombre, edad)
        Empleado.__init__(empresa, salario)
        self.lenguaje = lenguaje

    def mostrar(self):
        return self.nombre + "\nPrograma en " + self.lenguaje
```

Notar cómo, en el constructor, se antepone el nombre de cada clase delante del método al que llamar de dicha clase. En este caso, para llamar al constructor de **Persona** o de **Empleado**, con los parámetros correspondientes.

Ejercicio 3:

En un hospital hay diferentes tipos de personal: médicos, enfermeros y administrativos. De todos guardamos su información básica (dni, nombre, dirección y teléfono), de los médicos almacenamos también su especialidad, y de los enfermeros la planta en la que trabajan.

Al hospital acuden pacientes. De todos ellos se guarda un historial con su dni, nombre, dirección, teléfono, y un conjunto de pruebas y consultas que hayan hecho en el hospital. De cada prueba o consulta guardamos la fecha en que se hizo, y el médico que le atendió

Define las clases necesarias para el enunciado propuesto en un programa llamado **Hospital.py**. Define un programa principal que cree un array de personal de hospital con varios médicos y enfermeros. Define un paciente con sus datos, y dale de alta diversas pruebas realizadas por varios médicos. Finalmente, trata de mostrar por pantalla los datos completos del paciente, incluyendo su historial de pruebas.

9. Acceso a ficheros

En este documento veremos algunas nociones elementales de cómo extraer, procesar y/o guardar información en ficheros de texto empleando el lenguaje Python. También veremos algunas operaciones útiles sobre el sistema de ficheros: crear carpetas, listar ficheros y carpetas de una ruta, copiar ficheros...

9.1. Gestión de ficheros de texto

Apertura de ficheros

Lo primero que debemos hacer es abrir el fichero correspondiente con la función `open`. Como primer parámetro indicaremos el nombre del fichero a abrir (con una ruta relativa a donde esté el ejecutable Python), y como segundo parámetro para qué lo vamos a abrir: `r` para lectura, `w` para escritura, `r+` para lectura-escritura combinadas... Aquí vemos unos ejemplos:

```
lectura = open("fichero1.txt", "r")
escritura = open("fichero2.txt", "w")
lectura_escritura = open("fichero3.txt", "r+")
```

En el caso de que queramos leer o escribir en un formato determinado, añadimos un parámetro adicional `encoding` para especificar dicho formato. Esto permitirá procesar correctamente algunos símbolos especiales de ciertos alfabetos, como los acentos en el alfabeto latino. Por ejemplo:

```
lectura = open("fichero1.txt", "r", encoding="utf-8")
```

Al abrir para escritura según el ejemplo anterior, se borrará el contenido previo que pudiera tener el fichero. Si queremos mantenerlo para añadir contenido nuevo al final, empleamos la opción `a` en lugar de `w`. Esta opción creará el fichero si no existe, o respetará su contenido previo si ya existía:

```
escrituraAppend = open("fichero3.txt", "a")
```

Operaciones básicas sobre ficheros de texto

Las dos operaciones básicas que podemos realizar sobre ficheros de texto son la escritura y la lectura (dependiendo del modo en que hayamos abierto el fichero previamente).

Para **escribir datos** en el fichero de salida, empleamos su método `write`:

```
n = 3
escritura.write("El número vale " + str(n))
```

Para **leer datos** del fichero de entrada, lo más habitual es leer el fichero con la función `readlines`, que nos da ya una lista con todas las líneas para ir las procesando:

```
lineas = lectura.readlines()
for linea in lineas:
    # Procesamiento de la línea en cuestión
```


Existen otros métodos de lectura alternativa, como el método `read`, que se emplea para leer bytes de ficheros binarios (imágenes, por ejemplo).

Cierre de fichero

Es importante cerrar el fichero una vez finalizado el trabajo con él, empleando la instrucción `close` (tanto para lectura como para escritura):

```
lectura.close()
escritura.close()
```

Uso de la cláusula *with*

La declaración `with` en Python se utiliza para crear un contexto de administración (*context manager*) que garantiza que ciertos recursos se gestionen adecuadamente, como la apertura y el cierre de archivos, conexiones a bases de datos o la liberación de recursos. Podemos emplearlo, por ejemplo, para abrir un fichero (para lectura o escritura) y despreocuparnos de su cierre:

```
with open("archivo.txt", "r") as archivo:
    # Realizar operaciones de lectura en el archivo
    contenido = archivo.readlines()
    # El archivo se cerrará automáticamente al salir del bloque 'with'

# Fuera del bloque 'with', el archivo ya está cerrado
```

Ejemplo

El siguiente ejemplo lee todas las líneas de un fichero “*datos.txt*”. En cada línea hay números separados por espacios. Tras leer cada línea, procesará todos los números que hay contenidos en ella y mostrará su suma:

```
lectura = open("datos.txt", "r")
lineas = lectura.readlines()
numLinea = 0
for linea in lineas:
    numLinea += 1
    suma = 0
    numeros = linea.split(" ")
    for numero in numeros:
        suma += int(numero)
    print("La línea %d suma %d" % (numLinea, suma))
```

Ejercicio 1:

Crea un programa llamado **FicheroPersonas.py** que lea información de personas (nombre y edad) de un fichero de texto, y muestre por pantalla los datos de la persona más joven y más vieja del fichero. El formato del fichero será como el siguiente, y se deberá almacenar en una lista antes de procesar la información.

```
Nacho;44
Juan;70
Mario;9
Laura;6
Ana;40
```

9.2. Gestión del sistema de ficheros

Veremos a continuación algunas instrucciones útiles para realizar operaciones típicas sobre el sistema de ficheros. Casi todas ellas forman parte del paquete `os` de Python, por lo que deberemos incorporarlo al inicio de nuestro código, con `import os`. Otras funciones que usaremos pertenecen a otras librerías, como `shutil`, también disponibles en el núcleo de Python.

- Para **crear un directorio o carpeta** usaremos la función `os.mkdir(ruta)`
- Para **listar ficheros y carpetas de una carpeta** usamos la función `os.listdir(ruta)`. Se mostrará el listado de recursos que están directamente en la carpeta indicada (no en subcarpetas)
- Para **borrar un fichero** usaremos el método `os.remove(ruta)`. En el caso de una carpeta, usaremos `os.rmdir(ruta)`, siempre que la carpeta esté vacía.
- Para **copiar un fichero o carpeta** de una ruta a otra, usaremos el método `copy` del paquete `shutil`, indicando ruta origen y destino a copiar.
- Para **comprobar si un recurso existe**, disponemos de `os.path.exists(ruta)`
- Para **determinar si un recurso es fichero o carpeta** podemos usar los métodos `is_dir` e `is_file` del elemento `Path` del módulo `pathlib`. También aquí disponemos del método `iter_dir` para iterar el contenido de una carpeta.

Aquí vemos un ejemplo de algunas de estas funciones:

```
import os
import shutil
from pathlib import Path

# Creamos carpeta D:\pruebas
# La 'r' delante del texto indica que se interprete como
# un texto literal, respetando la barra \
if not os.path.exists(r'D:\Pruebas'):
    os.mkdir(r'D:\Pruebas')

# Listamos ficheros y carpetas de C:\Usuarios\Nacho
path = Path(r'C:\Users\Nacho')
for elemento in path.iterdir():
    if elemento.is_dir():
        print('Carpeta', elemento)
    else:
        print('Fichero', elemento)

# Copiamos fichero "prueba.txt" de un lugar a otro
origen = r'C:\Users\Nacho\prueba.txt'
destino = r'D:\Pruebas\prueba_copia.txt'
shutil.copy(origen, destino)

# Borramos fichero "prueba.txt" de una carpeta
os.remove(r'C:\Users\Nacho\prueba.txt')
```

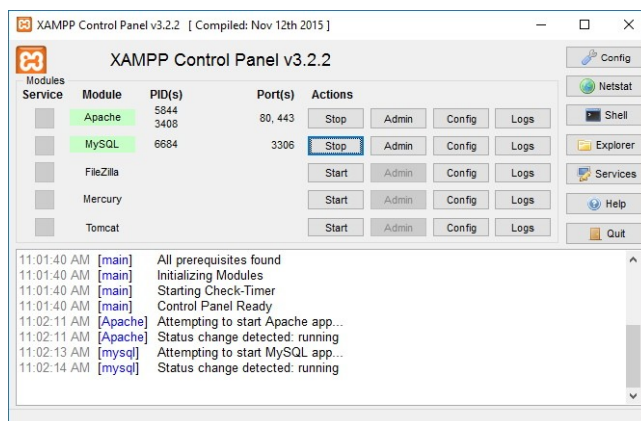
10. Acceso a bases de datos

En este documento daremos unas nociones básicas de cómo acceder a diferentes sistemas gestores de bases de datos desde Python. En concreto nos centraremos en un ejemplo de base de datos relacional como MySQL, y en otro de base de datos No-SQL, como MongoDB. Como ocurre en muchos otros casos, necesitaremos instalar la librería adecuada para poder trabajar con la base de datos en cuestión, y después utilizar los métodos correspondientes para conectar, insertar, listar, etc.

10.1. Acceso a bases de datos MySQL

Instalación del servidor MySQL

Para poder trabajar con bases de datos MySQL, el primer paso será tener instalado un servidor MySQL. Podemos instalar uno de forma manual, y luego configurar los usuarios, bases de datos y tablas desde el terminal, o bien podemos disponer de un entorno algo más visual para poder gestionar el servidor y la base de datos de forma más cómoda. En nuestro caso, ya que no se trata de un curso para administrar servidores de bases de datos, optaremos por esta segunda opción a través de la herramienta XAMPP. Al instalarla dispondremos de un servidor web Apache, y un servidor MySQL. Podremos poner ambos en marcha desde la herramienta *XAMPP Manager* que viene incorporada, y poner en marcha ambos servidores.



La ventaja que tiene utilizar esta herramienta es que también disponemos de una aplicación web llamada *phpMyAdmin*, que se pone en marcha sobre Apache y nos permite gestionar las bases de datos que tengamos instaladas: crearlas, definir las tablas, hacer copia de seguridad, etc.



Instalación de las librerías de Python

Para poder acceder a MySQL desde Python necesitamos utilizar alguna librería externa que nos facilite los mecanismos de conexión y gestión de esa base de datos. Una de las más utilizadas es `mysql-connector-python`, que instalamos con el correspondiente comando `pip`:

```
pip install mysql-connector-python
```

Después importaremos el elemento `connector` en nuestro programa:

```
import mysql.connector
```

10.2. Operaciones con la base de datos MySQL

Veremos a continuación un ejemplo sencillo de cada una de las operaciones habituales con MySQL.

Conexión a una base de datos

La primera operación que tendremos que realizar será conectar con la base de datos que queramos. Supondremos que ya la tendremos previamente creada, a través de *phpMyAdmin* alguna otra herramienta. En cualquier caso, la propia librería `mysql.connector` dispone de mecanismos para crear bases de datos y tablas, pero no los veremos aquí.

La instrucción para conectar es como sigue:

```
import mysql.connector

bd = mysql.connector.connect(
    host="localhost",
    user="usuario",
    password="password",
    database="miBD"
)
```

NOTA: en la instrucción anterior deberemos reemplazar los valores de los parámetros por los reales. Lo normal es que conectemos desde el mismo ordenador donde está la base de datos, con lo que el *host* suele ser *localhost*. Pero el usuario y contraseña pueden cambiar. La instalación por defecto de XAMPP deja un usuario *root* con contraseña vacía. Finalmente, el parámetro *database* apuntará al nombre de la base de datos que queramos utilizar.

Operaciones de inserción, borrado y actualización

Para realizar operaciones INSERT, DELETE o UPDATE, a partir del objeto obtenido al conectar, crearemos un *cursor* que nos permita acceder a la base de datos, y usaremos su instrucción `execute` para ejecutar la instrucción correspondiente. Por ejemplo, si tenemos una tabla *personas* con unos campos *nombre* y *edad*, podemos insertar una nueva persona así:

```
# ... Código anterior para conectar

# Obtenemos el cursor
cursor = bd.cursor()
```

```
# Planteamos la instrucción SQL
sql = "INSERT INTO personas (nombre, edad) VALUES(%s, %s)"
valores = ("Nacho", 45)
cursor.execute(sql, valores)
bd.commit()
```

Notar que definimos unos comodines %S en la instrucción SQL para luego reemplazarlos por los valores que queramos. La propia librería ya se encarga de reemplazar y guardar cada dato con su tipo adecuado (en este caso, un texto y un entero).

Del mismo modo podemos insertar múltiples datos, especificando un vector de tuplas:

```
sql = "INSERT INTO personas (nombre, edad) VALUES(%s, %s)"
valores = [
    ("Nacho", 45),
    ("Ana", 41),
    ("Juan", 70)
]
cursor.execute(sql, valores)
bd.commit()
```

En inserciones que generen un *id* autonumérico podemos recuperarlo tras la inserción, a través del cursor. También podemos obtener otras propiedades, como el número de filas insertadas:

```
bd.commit()
filas_insertadas = cursor.rowcount
ultimo_id = cursor.lastrowid
```

También de forma similar podemos lanzar borrados o actualizaciones, y obtener el número de filas afectadas:

```
sql = "DELETE FROM personas WHERE nombre = %s"
valores = ("Jaime", )
cursor.execute(sql, valores)
bd.commit()
print("Se han borrado", cursor.rowcount, "elementos")

sql = "UPDATE personas SET edad=50 WHERE nombre=%s"
valores = ("Pepe", )
cursor.execute(sql, valores)
bd.commit()
print("Se han actualizado", cursor.rowcount, "elementos")
```

Consultas

Para lanzar consultas (SELECT) construimos la instrucción SQL de forma similar, pero usamos el método `fetchall` para obtener los resultados, en lugar de `commit`. Por ejemplo, así obtenemos las personas mayores que la edad que diga el usuario previamente:

```
edad_usuario = # ... Le pedimos la edad al usuario

sql = "SELECT nombre, edad FROM personas WHERE edad > %s"
valores = (edad_usuario,)
```

```
# Ejecutar la consulta
cursor.execute(sql, valores)

# Obtener los resultados
resultados = cursor.fetchall()

# Imprimir los resultados
for r in resultados:
    print("Nombre:", r[0])
    print("Edad:", r[1])
```

Adicionalmente, disponemos del método `fetchone` en lugar de `fetchall` si sólo queremos acceder al primer elemento que cumpla la selección.

Cierre de la conexión

Tras finalizar las operaciones necesarias en la base de datos, conviene cerrar tanto el cursor como la conexión a la base de datos:

```
cursor.close()
bd.close()
```

Ejercicio 1:

Descarga [este backup](#) de una base de datos MySQL llamada *contactos* e impórtalo en tu servidor. Creará una base de datos *contactos* con una tabla *contactos* con una serie de registros insertados. Crea un programa `ContactosMySQL.py` y prueba a hacer una consulta que muestre los contactos cuyo nombre contenga el texto indicado por el usuario.

10.3. Acceso a bases de datos NoSQL con MongoDB

Instalación del servidor MongoDB

Como ocurría en el caso anterior, si queremos trabajar con bases de datos MongoDB necesitamos disponer de un servidor instalado. Nuevamente, al no ser éste el propósito principal del curso, dejamos en manos del lector el cómo instalarlo y qué herramienta(s) utilizar para gestionarlo.

Instalación de las librerías de Python

La librería que usaremos en este caso para acceder a Mongo será `pymongo`, que podemos instalar con el correspondiente comando:

```
pip install pymongo
```

La incluiremos en nuestro código fuente de este modo:

```
import pymongo
```

10.4. Operaciones con la base de datos MongoDB

Veremos a continuación un ejemplo sencillo de cada una de las operaciones habituales con MongoDB.

Conexión y creación de una base de datos

Para conectar a un servidor Mongo usamos esta expresión:

```
import pymongo

cliente = pymongo.MongoClient("mongodb://localhost:27017")
bd = cliente["nombreBD"]
```

La primera instrucción conecta con el servidor indicado (en nuestro caso, *localhost*), por el puerto por defecto en que suele escuchar Mongo (27017). La segunda instrucción crea una base de datos *nombreBD* si no existe y, en caso contrario, accede a ella. Realmente la base de datos no va a existir hasta que añadamos contenido.

Crear una colección y añadir documentos.

Las bases de datos en MongoDB se componen de colecciones, del mismo modo que una base de datos en MySQL se compone de tablas. Para crear una colección simplemente la nombramos desde la base de datos.

```
...
bd = cliente["nombreBD"]
# Accedemos a la coleccion "personas" de la BD
coleccion = bd["personas"]
```

Nuevamente, la colección no se creará hasta que añadamos contenido en ella (documentos). Para **insertar documentos** en una colección debemos definir los campos que tendrán los documentos, en forma de mapa. Después podemos utilizar el método `insert_one` de la colección para insertar el documento.

```
persona = {'nombre': 'Nacho', 'edad': 45}
coleccion.insert_one(persona)
```

En el caso de bases de datos Mongo, a cada documento se le asigna un identificador alfanumérico automático en un campo interno llamado `_id`. Para poderlo recuperar debemos guardarnos en una variable el objeto insertado, para luego acceder a esa información con la propiedad `inserted_id`:

```
persona = {'nombre': 'Nacho', 'edad': 45}
nueva_persona = coleccion.insert_one(persona)
print("Persona insertada con id", nueva_persona.inserted_id)
```

Si queremos añadir múltiples documentos, los añadimos a un vector y utilizamos el método `insert_many`:

```
personas = [
    {'nombre': 'Nacho', 'edad': 45},
```

```
{'nombre': 'Ana', 'edad': 41},  
{'nombre': 'Juan', 'edad': 70}  
]  
datos = coleccion.insert_many(personas)  
print("Ids insertados:", datos.inserted_ids)
```

Búsquedas

Para buscar documentos en una colección empleamos los métodos `find_one` o `find`, dependiendo de si queremos encontrar uno o varios.

```
resultado1 = coleccion.find_one({'nombre': 'Nacho'})  
print(resultado1)  
  
resultado2 = coleccion.find({'nombre': 'Nacho'})  
for r in resultado2:  
    print(r)
```

También podemos acceder a un dato concreto de cada documento como si fuera un diccionario. Por ejemplo:

```
...  
for r in resultado2:  
    print(r['nombre'])
```

Como podemos ver, se pueden especificar criterios de filtrado o búsqueda como primer parámetro de los métodos anteriores. También se pueden especificar criterios de búsqueda más complejos, como se puede consultar en la documentación de pymongo.

Actualizaciones y borrados

Para eliminar documentos de una colección emplearemos los métodos `delete_one` o `delete_many`, dependiendo de la cantidad:

```
coleccion.delete_one({'nombre': 'Nacho'})  
coleccion.delete_many({'edad': 30})
```

NOTA: si no especificamos criterios de filtrado en `delete_many` se borrarán TODOS los documentos de la colección.

De forma análoga, podemos emplear los métodos `update_one` y `update_many` para actualizar los campos de documentos:

```
coleccion.update_one({'_id': 1}, {'$set': {'nombre': 'Nombre  
modificado', 'edad': 33}})
```

En este caso, los métodos reciben dos parámetros: el criterio para buscar el documento (o documentos) a actualizar, y el conjunto de nuevos datos a asignar, que suele venir en un bloque denominado `$set`, donde se asigna a cada campo su valor.

Ejercicio 2:

Crea un programa `BibliotecaMongo.py` que cree una base *biblioteca* en tu servidor MongoDB, con una colección llamada *libros*. Inserta en ella estos dos libros con sus campos:

- *titulo*: La tabla de Flandes, *autor*: Arturo Pérez Reverte, *paginas*: 312
- *titulo*: El juego de Ender, *autor*: Orson Scott Card, *paginas*: 452

Después prueba a buscar los libros que tengan más de 400 páginas, para ver si muestra datos correctos. Busca en Internet cómo especificar un rango en el filtrado (libros cuyo número de páginas sea *mayor que X*).