



# UT04: EL LENGUAJE DE PROGRAMACIÓN PYTHON

Para la elaboración de algunos de los apartados de esta presentación se ha recurrido a la web El libro de Python, por lo que puedes recurrir a esta página si quieres ver una explicación más detallada.

<https://ellibrodepython.com>



# ÍNDICE

---

- 1.- Introducción a Python
- 2.- Instalación del intérprete Python
- 3.- Tipos de datos
- 4.- Control de flujo del programa
- 5.- Funciones

# 1

## INTRODUCCIÓN





El lenguaje de programación **Python** fue desarrollado a finales de los 80 por el holandés **Guido Van Rossum**.

Su objetivo era crear un lenguaje de programación fácil de leer, potente y enfocado en la productividad del desarrollador.

Como curiosidad, el nombre no proviene de la serpiente homónima, sino del grupo cómico británico Monty Python



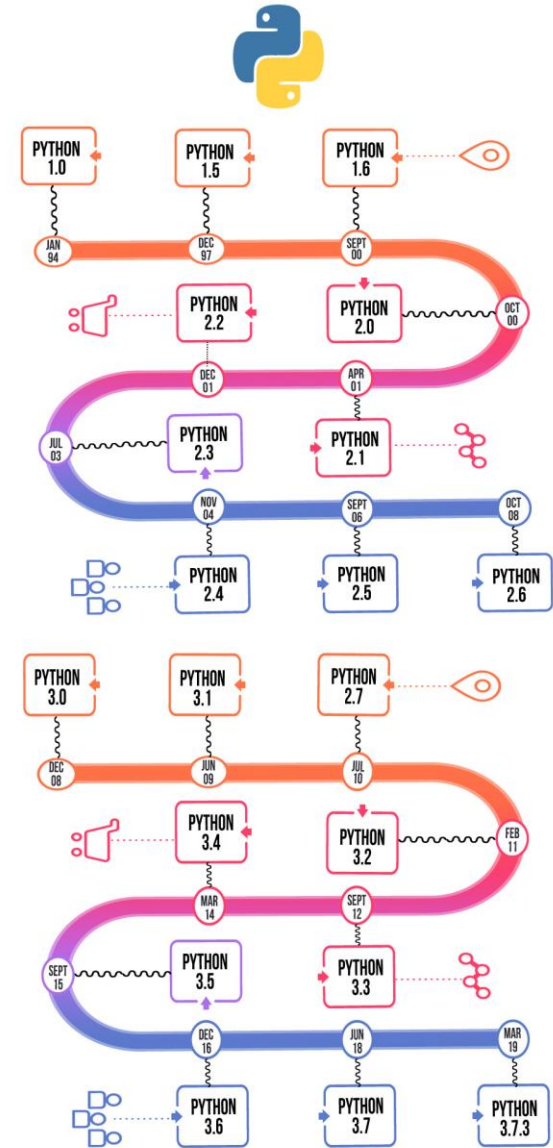
Python fue lanzado al público en febrero de 1991 como una versión alfa.

A lo largo de los 90 se publicaron diversas versiones a medida que fue ganando popularidad, especialmente en ámbitos académicos y científicos.

En el año 2000 se alcanzó un hito importante en su evolución con la versión 2.0.

En el año 2008, se lanzó **Python 3.0**, con cambios significativos en la sintaxis y la semántica para mejorar la consistencia y la claridad del lenguaje.

Aunque la transición ha sido muy lenta debido la incompatibilidad entre las dos versiones, en la actualidad Python 3 se ha consolidado como la rama principal de desarrollo.



Actualmente Python es ampliamente utilizado gracias a la enorme disponibilidad de librerías, utilizándose en campos como:

- **Desarrollo web:** para aplicaciones en el lado del servidor con *frameworks* como **Django** y **Flask**
- **Ciencia de datos y análisis:** su uso en este campo es muy extenso gracias a potentes librerías como son **numpy**, **Pandas** y **Matplotlib**
- **Inteligencia artificial y aprendizaje automático:** Python es líder en IA, con librerías como **TensorFlow**, **Keras** y **PyTorch**, que facilitan el desarrollo de modelos de IA y ML.
- **Automatización y scripting:** su sintaxis simple y legible le hace ideal para utilizarlo como lenguaje de script.
- **Desarrollo de juegos:** Python es cada vez más utilizado para desarrollo de juegos, tanto para prototipos rápidos como para juegos completos, gracias a librerías como **PyGame** y **Panda3D**

- **Aplicaciones de escritorio:** librerías como **PyQt** y **Tkinter** permiten crear aplicaciones multiplataforma con interfaces gráficas de usuario
- **Desarrollo de aplicaciones web:** aunque en este campo su uso es menos común, es posible crear aplicaciones para móviles con el framework **Kivy**
- **Automatización de pruebas:** es ampliamente utilizado en automatización de pruebas de software, ya sea unitarias, de integración o de aceptación, gracias a bibliotecas como **unittest** y **pytest**.
- **Ciberseguridad:** es ampliamente utilizado en este campo, especialmente para automatización de tareas de Red Team, con librerías como **Scapy** e **Impacket** (manipulación de paquetes de red), **Requests** (realización de solicitudes HTTP), **SQLAlchemy** (mapeo ORM con BBDD SQL), **Beautiful Soup** (web scrapping) o **PyCrypto** (algoritmos criptográficos).



Si nos centramos ya en el lenguaje en sí, podemos destacar las siguientes **características**:

## MULTIPARADIGMA

Python admite varios paradigmas de programación, incluyendo orientada a objetos, imperativa y programación funcional

### Imperativa

Se especifican explícitamente los pasos que debe realizar el programa para alcanzar el resultado, centrándose en el estado del programa y en la **manipulación directa de los datos** mediante instrucciones como asignaciones, bucles y condicionales.

### Orientada a objetos

Los programas se estructuran alrededor de **objetos**, que son **instancias de clases**. Estos objetos tienen propiedades y comportamientos asociados, y la interacción entre ellos se realiza a través de mensajes.

### Funcional

Se centra en las funciones como unidades de trabajo principales, tratando las funciones como **valores de primera clase** y evitando los cambios de estado y efectos secundarios. Se basa en la evaluación de expresiones y la aplicación de funciones.

## TIPADO DINÁMICO

Python es un lenguaje de tipado dinámico, lo que significa que las variables no están asociadas a un tipo de datos específico y pueden cambiar de tipo durante la ejecución del programa.

```
>>> a = "Hola"
>>> type(a)
<class 'str'>
>>> a = 123
>>> type(a)
<class 'int'>
>>> |
```

Creo la variable *a* y le asigno el valor "Hola", por lo que será de tipo *string*. Observa que no hace falta declaración de la variable

Ahora le asigno un valor diferente a la variable, en este caso un entero, por lo que la variable *a* pasara ahora a ser de tipo *int*.

Los lenguajes débilmente tipados son más flexibles y más sencillos de usar, ya que no hay que preocuparse por la sintaxis y reglas asociadas a la declaración de tipos, pero tienen el inconveniente de que es muy difícil detectar errores relacionados con tipos

## INTERPRETADO

Python es un lenguaje interpretado, lo que significa que se ejecuta línea a línea y no requiere una compilación previa antes de ejecutar el programa.

En realidad, esto no es totalmente cierto, ya que, cuando ejecutamos un programa en Python se compila a un formato intermedio denominado **bytecode**, el cual se ejecuta en la máquina virtual de Python (PVM).

Este paso intermedio permite aplicar optimizaciones propias de lenguajes compilados al ejecutar Python.

Let's look at an example. Here's a classic "Hello, World!" written in Python:

```
def hello()
    print("Hello, World!")
```

And here's the bytecode it turns into (translated into a human-readable form):

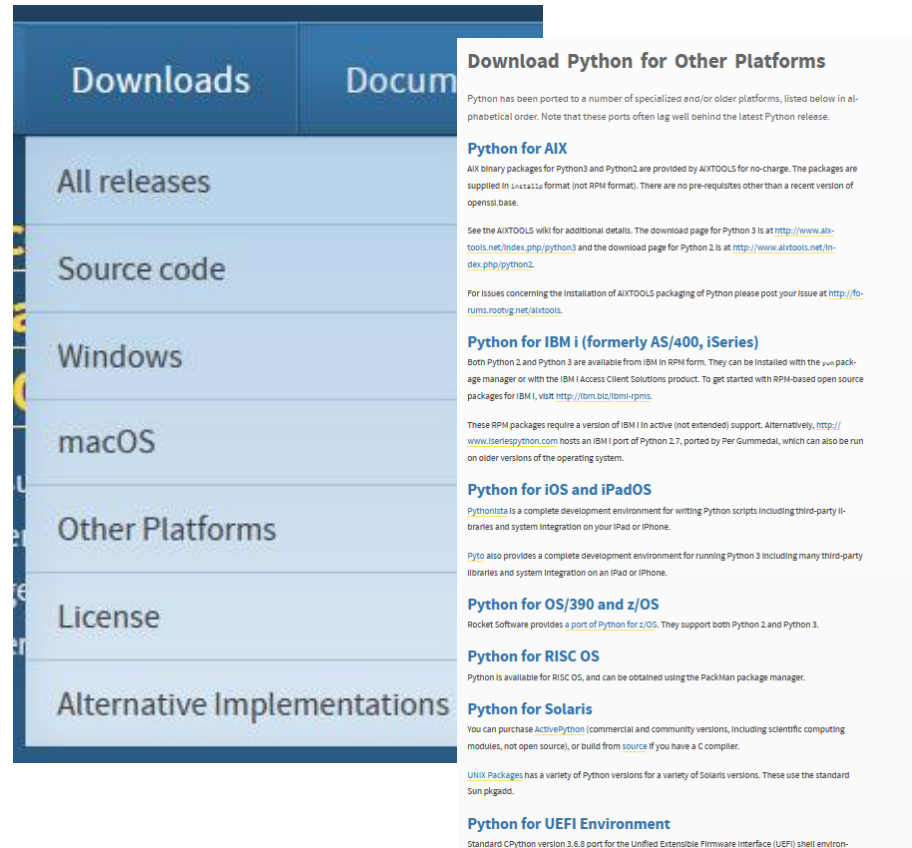
2	0 LOAD_GLOBAL	0 (print)
	2 LOAD_CONST	1 ('Hello, World!')
	4 CALL_FUNCTION	1

Ejemplo de conversión a bytecodes de Python, extraído de esta URL donde se explica muy bien el proceso

<https://opensource.com/article/18/4/introduction-python-bytecode>

## MULTIPLATAFORMA

Relacionado con el punto anterior, como Python no se ejecuta directamente sobre la máquina física, sino que lo hace sobre la máquina virtual, tiene la ventaja de que se puede ejecutar sobre cualquier sistema operativo que disponga de dicha máquina virtual.

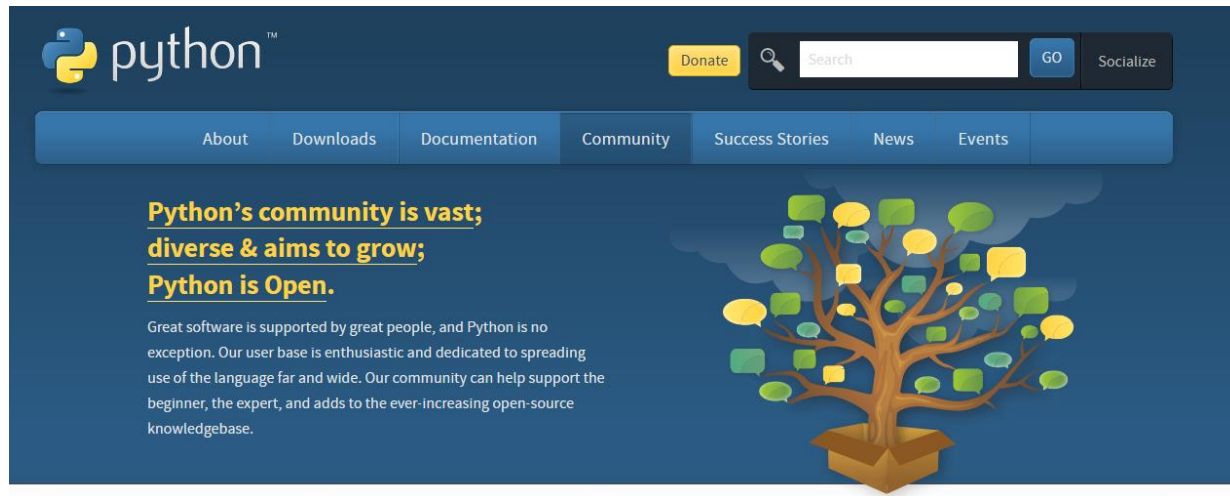


## GRAN BIBLIOTECA ESTÁNDAR

Python cuenta con una gran biblioteca estándar que ofrece módulos y funciones para realizar una gran variedad de tareas comunes, como manipulación de archivos, gestión de redes, acceso a bases de datos y procesamiento de datos.

## COMUNIDAD ACTIVA

Python cuenta con una gran y activa comunidad de desarrolladores que contribuyen con bibliotecas, *frameworks* y herramientas que amplían la funcionalidad del lenguaje y lo mantienen actualizado con las últimas tendencias tecnológicas.



## FACILIDAD DE INTEGRACIÓN

Python se puede integrar fácilmente con otros lenguajes de programación, como C, C++, Java y .NET, lo que permite a los desarrolladores aprovechar el código existente y utilizar las fortalezas de cada lenguaje según sea necesario.



# 2

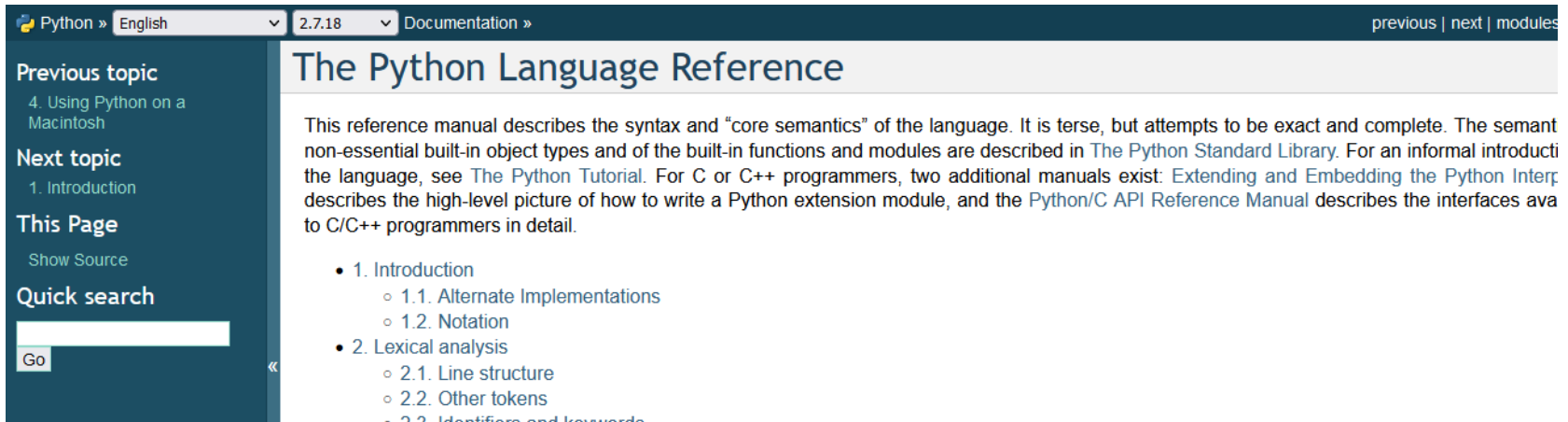
## INSTALACIÓN



Para ejecutar programas en Python únicamente es necesario tener el intérprete instalado en nuestro sistema. Sin embargo, si buscas un poco en internet, verá que existe CPython, JPython, IronPython, ... ¿A qué se debe esta disparidad y en qué nos afecta?

Para explicarlo voy a resumir este muy interesante artículo: <https://www.toptal.com/python/por-que-hay-tantos-pythons> donde se explica mucho más detallado.

Lo primero que tenemos que distinguir es que **Python es una interfaz**, es decir, hay una especificación que indica cómo es el lenguaje Python, por lo que cualquiera de las implementaciones que haya tiene que ceñirse a dicha especificación.



The screenshot shows the Python 2.7.18 documentation page for 'The Python Language Reference'. The page has a dark blue header with navigation links: 'Python »', 'English', '2.7.18', 'Documentation »', and 'previous | next | modules'. On the left side, there is a sidebar with 'Previous topic' (4. Using Python on a Macintosh), 'Next topic' (1. Introduction), 'This Page' (Show Source), and 'Quick search' (Go). The main content area is titled 'The Python Language Reference' and contains a paragraph describing the reference manual. Below the paragraph is a table of contents with two main sections: '1. Introduction' and '2. Lexical analysis', each with sub-sections.

Python » English 2.7.18 Documentation » previous | next | modules

**Previous topic**  
4. Using Python on a Macintosh

**Next topic**  
1. Introduction

**This Page**  
Show Source

**Quick search**  
Go

## The Python Language Reference

This reference manual describes the syntax and “core semantics” of the language. It is terse, but attempts to be exact and complete. The semantic non-essential built-in object types and of the built-in functions and modules are described in [The Python Standard Library](#). For an informal introduction to the language, see [The Python Tutorial](#). For C or C++ programmers, two additional manuals exist: [Extending and Embedding the Python Interpreter](#) describes the high-level picture of how to write a Python extension module, and the [Python/C API Reference Manual](#) describes the interfaces available to C/C++ programmers in detail.

- 1. Introduction
  - 1.1. Alternate Implementations
  - 1.2. Notation
- 2. Lexical analysis
  - 2.1. Line structure
  - 2.2. Other tokens
  - 2.3. Identifiers and keywords

Cualquier intérprete compilará el código a Python (siguiendo la especificación) a *bytecodes* que se ejecutarán en la máquina virtual de Python.

Ya ahí es donde están las diferencias, ya que hay diferentes máquinas virtuales de Python.

Lo habitual es utilizar **CPython**, una máquina virtual implementada en lenguaje C.

Otra alternativa bastante habitual es **Jython**, una implementación en Java que produce bytecode para ser ejecutado en la JVM. Esto tiene ventajas, por ejemplo, puedes importar clases Java directamente en tu código Python

```
[Java HotSpot(TM) 64-Bit Server VM (Apple Inc.)] on java1.6.0_51
>>> from java.util import HashSet
>>> s = HashSet(5)
>>> s.add("Foo")
>>> s.add("Bar")
>>> s
[Foo, Bar]
```

Otra implementación es **IronPython**, escrita en C# y que se ejecuta en el **Common Language Runtime (CLR)** de Microsoft, lo que permite importar clases C# en tu código.

En la siguiente tabla (extraída de la web anterior) puedes ver las diferentes implementaciones que hay de Python.

Implementación	Máquina Virtual	Ej) Lenguaje Compatible
CPython	CPython VM	C
Jython	JVM	Java
IronPython	CLR	C#
Brython	Motor Javascript(Por Ej., V8)	JavaScript
RubyPython	Ruby VM	Ruby





Todo lo anterior es simplemente como curiosidad, ya que para instalar Python en nuestro equipo no es necesario saber todo eso.

Simplemente vamos a la página oficial de Python (<https://www.python.org/downloads/windows/>) y descargamos el instalador para nuestro sistema operativo.

### Stable Releases

#### ▪ [Python 3.11.8 - Feb. 6, 2024](#)

**Note that Python 3.11.8 *cannot* be used on Windows 7 or earlier.**

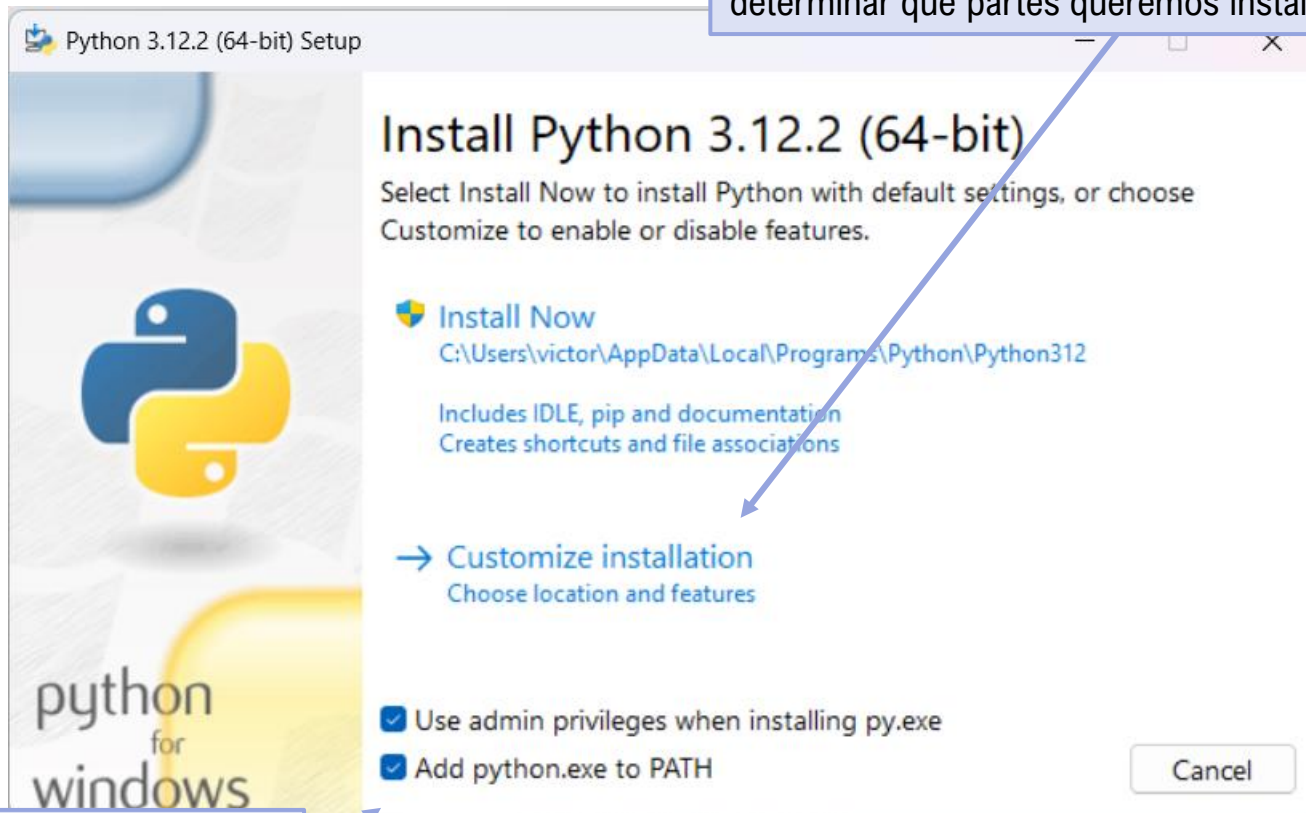
- Download [Windows installer \(64-bit\)](#)
- Download [Windows installer \(ARM64\)](#)
- Download [Windows embeddable package \(64-bit\)](#)
- Download [Windows embeddable package \(32-bit\)](#)
- Download [Windows embeddable package \(ARM64\)](#)
- Download [Windows installer \(32-bit\)](#)

#### ▪ [Python 3.12.2 - Feb. 6, 2024](#)

**Note that Python 3.12.2 *cannot* be used on Windows 7 or earlier.**

- Download [Windows installer \(64-bit\)](#)
- Download [Windows installer \(ARM64\)](#)
- Download [Windows embeddable package \(64-bit\)](#)
- Download [Windows embeddable package \(32-bit\)](#)
- Download [Windows embeddable package \(ARM64\)](#)
- Download [Windows installer \(32-bit\)](#)

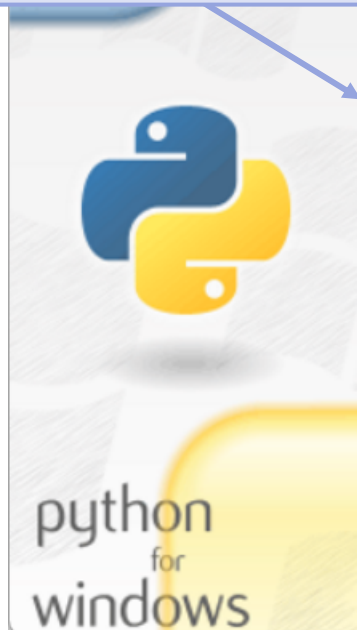
Si le damos a *Customize installation* podremos determinar qué partes queremos instalar.



Agregamos el **PATH** para poder ejecutarlo desde la línea de comandos

**PIP (Python Package Index)** es el sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software instalados en Python.

**IDLE (Integrated Development Environment for Python)** es el entorno gráfico que permite editar y ejecutar programar en Python.



## Optional Features

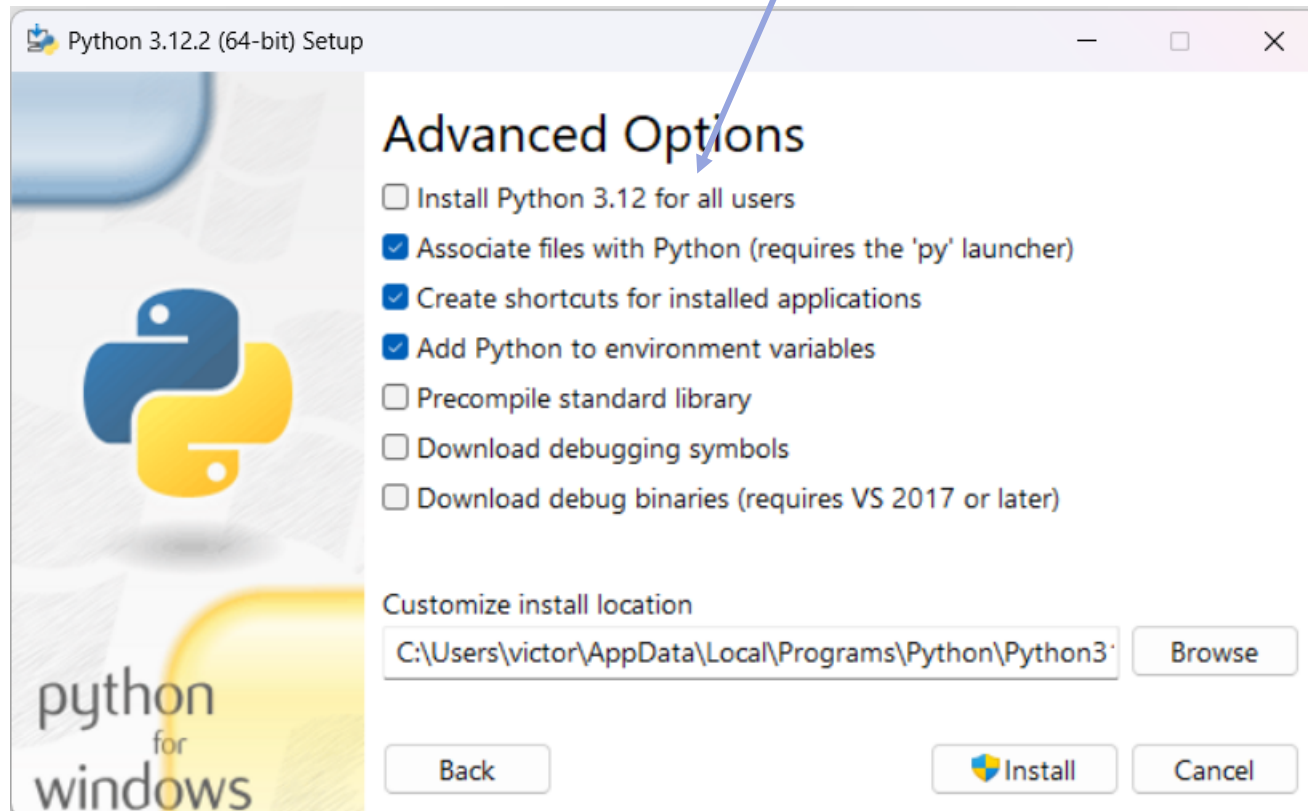
- ☒ Documentation  
Installs the Python documentation files.
- ☒ pip  
Installs pip, which can download and install other Python packages.
- ☒ td/tk and IDLE  
Installs tkinter and the IDLE development environment.
- ☒ Python test suite  
Installs the standard library test suite.
- ☒ py launcher ☒ for all users (requires admin privileges)  
Installs the global 'py' launcher to make it easier to start Python.

Back

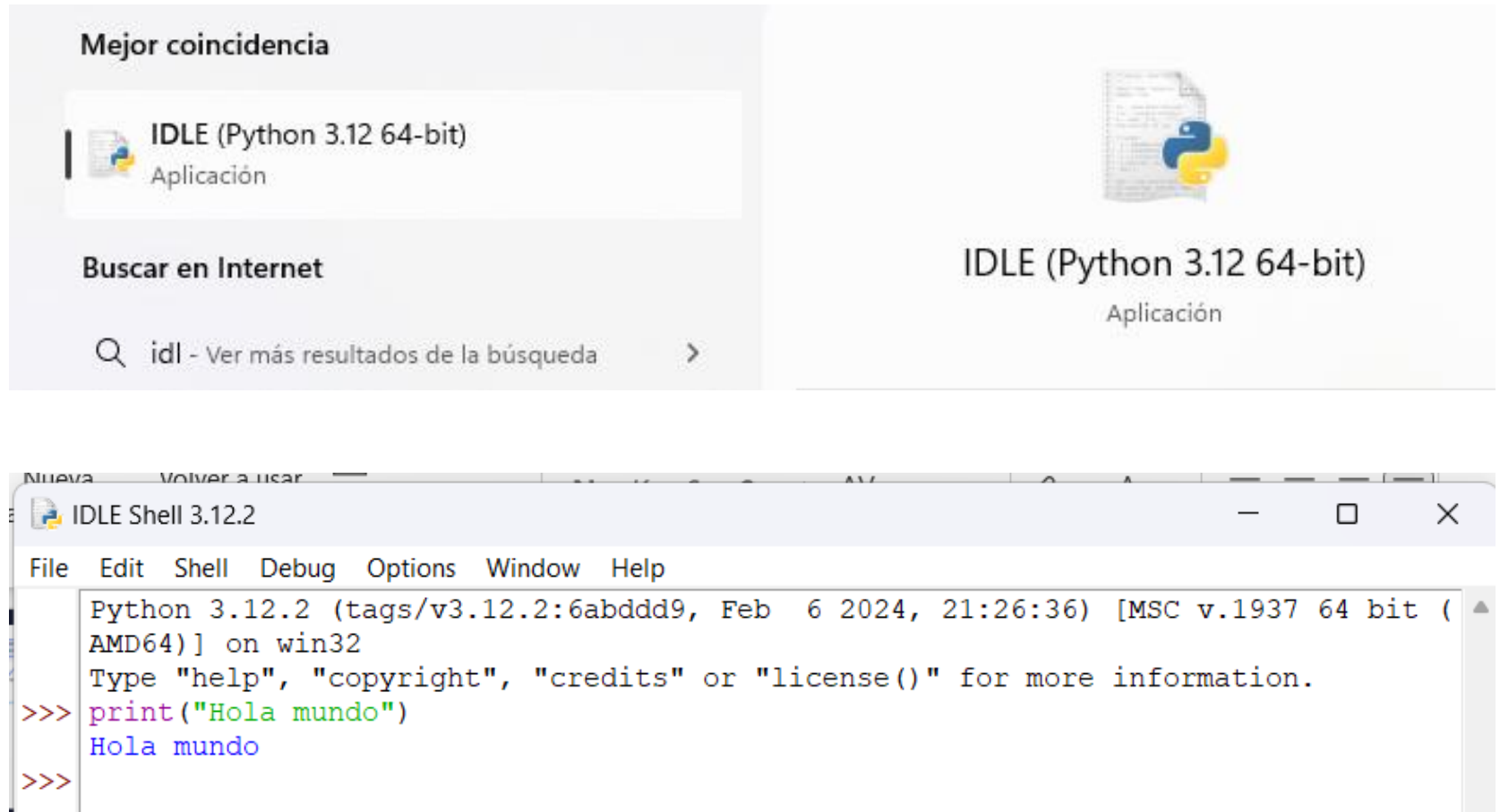
Next

Cancel

Por defecto, lo instala únicamente para el usuario actual.



Una vez instalado ya podremos ejecutar código Python directamente en Idle





También podemos abrir directamente Python en una terminal ejecutando el comando **python** que abre un intérprete interactivo.

```
PS C:\Users\victor> python
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hola mundo")
Hola mundo
>>> |
```

Los programas en Python tienen extensión **.py** y se pueden ejecutar invocando el programa python.



The screenshot displays the Visual Studio Code interface. The Explorer pane on the left shows a project named 'PYTHON' containing a file 'ejemplo.py'. The main editor window shows the code for 'ejemplo.py' with a single line: `1 print("Hola mundo")`. Below the editor, the TERMINAL pane is active, showing a PowerShell session. The command `python .\ejemplo.py` has been executed, and the output 'Hola mundo' is displayed. The status bar at the bottom indicates the file is at line 1, column 20, with 4 spaces, using UTF-8 encoding, and the Python 3.12.2 64-bit interpreter is selected.

```
File Edit Selection ... python
EXPLORER
PYTHON
ejemplo.py

ejemplo.py
1 print("Hola mundo")

PROBLEMS TERMINAL ... powershell
PS C:\proyectos\SGE\python> python .\ejemplo.py
Hola mundo
PS C:\proyectos\SGE\python>

OUTLINE
TIMELINE

Ln 1, Col 20 Spaces: 4 UTF-8 CRLF Python 3.12.2 64-bit Go Live Prettier
```

Si estamos en un sistema Linux, el ejecutable es **python3**

```
(vgonzalez@DESKTOP-J1QR201)~  
$ python3 --version  
Python 3.11.5
```

```
(vgonzalez@DESKTOP-J1QR201)~  
$ echo print\(\"Hola mundo\"\) > ejemplo.py  
  
(vgonzalez@DESKTOP-J1QR201)~  
$ python3 ejemplo.py  
Hola mundo
```

En Linux también podemos hacer el script ejecutable mediante el **shebang** (<https://realpython.com/python-shebang/>). Esto es un tipo especial de comentario que se pone en la primera línea del fichero y le indica al sistema dónde encontrar el intérprete para el resto del fichero.

Hay que acordarse también de asignarle permisos de ejecución.

```
GNU nano 7.2
#!/usr/bin/python3
print("Hola mundo")

(vgonzalez@DESKTOP-J1QR201)-
$ nano ejemplo.py

(vgonzalez@DESKTOP-J1QR201)-[~]
$ chmod u+x ejemplo.py

(vgonzalez@DESKTOP-J1QR201)-[~]
$ ./ejemplo.py
Hola mundo
```

<https://docs.python.org/es/3/tutorial/>



# 3

## CONTROL DE FLUJO DEL PROGRAMA



CONDICIONAL IF

La sentencia **if** evalúa una expresión y en función de su valor ejecuta un bloque de código.

Python determina los bloques de código mediante el **indentado**.

```
saludar = True
if (saludar):
    nombre = "Victor"
    print( f"Hola {nombre}" )
print("Adios")
```

```
Hola Victor
Adios
```

Observa que al final de la línea del **if** se pone el carácter dos puntos

Estas dos líneas tienen el mismo indentado, por lo que forman parte del mismo bloque de código

Esta línea está al mismo nivel que el **if**, por lo que ya está fuera del bloque del **if** y, por tanto, se ejecuta siempre

Como en otros lenguajes de programación, la sentencia **if** se puede combinar con **else** para indicar otro bloque de código que se ejecutará si la expresión se evalúa como falsa.

```
num = input("Dime un número: ")
if ( int(num)%2 == 0 ):
    print("El número es par")
else:
    print("El número es impar")
```

```
Dime un número: 7
El número es impar
```

Si queremos varios **if anidados** podemos utilizar la sentencia **elif**.

```
num = input("Dime un número: ")
if ( not num.isdigit() ):
    print("El valor que has introducido no es un número")
elif ( int(num)%2 == 0 ):
    print("El número es par")
else:
    print("El número es impar")
```

Dime un número: 7

El número es impar

Una alternativa al condicional **if** es el uso del **operador ternario**, que es una expresión condicional que devuelve uno de dos valores según sea el resultado de una condición.

```
edad = input("Dime tu edad: ")
mensaje = "Eres mayor de edad" if int(edad) >= 18 else "Eres menor de edad"
print(mensaje)
```

```
Dime tu edad: 21
Eres mayor de edad
```

También es posible anidar el operador ternario para condiciones más complejas.

```
edad = input("Dime tu edad: ")
mensaje = "Adulto" if int(edad) >= 18 else "Adolescente" if int(edad) >= 13 else "Niño"
print(mensaje)
```

```
Dime tu edad: 14
Adolescente
```

Si tenemos varios operadores ternarios anidados se puede hacer uso de los paréntesis para repartirlos en varias líneas y así obtener un código más legible.

```
edad = int( input("Dime tu edad: "))

mensaje = (
    "Adulto" if edad >= 18 else
    "Adolescente" if edad >= 13 else
    "Niño"
)

print(mensaje)
```

```
Dime tu edad: 17
Adolescente
```



SENTENCIA FOR

La **sentencia for** difiere de lo que estamos acostumbrados en otros lenguajes de programación, ya que no tiene una variable que va iterando sobre diversos valores numéricos hasta alcanzar un valor determinado, sino que **itera sobre los elementos de cualquier iterable** (por ejemplo, una lista o una cadena).



```
1  for ciclo in ['DAM', 'DAW', 'ASIR']:  
2      print(ciclo)  
3  
4  # DAM  
5  # DAW  
6  # ASIR
```

Si queremos iterar sobre una secuencia de valores nuḿricos es muy ́til la funci3n **range()**, que genera progresiones nuḿricas.



```
1  for a in range(3):  # Desde 0 hasta 3 (exc)
2      print(a)
3  # 0
4  # 1
5  # 2
6  for b in range(2, 4):  # Desde 2 hasta 4 (exc)
7      print(b)
8  # 2
9  # 3
10 for c in range(1, 9, 3):  # Desde 1 hasta 9 saltando de 3 en 3
11     print(c)
12 # 1
13 # 4
14 # 7
```

Si queremos iterar en sentido decreciente simplemente tendremos que indicar un valor negativo para el salto.

```
for i in range(5,1,-1):  
    print(i)
```

```
5  
4  
3  
2
```

**SENTENCIA WHILE**

La sentencia **while** ejecuta un bloque de código mientras la expresión indicada sea evaluada como *True*.



```
1  a = 1
2  while (a<4):
3      a+=1
4      print(a)
5  else:
6      print("Hemos llegado al 4")
7  # 2
8  # 3
9  # 4
```



```
1  a = 1
2  while (a<4):
3      a+=1
4      print(a)
5  # 2
6  # 3
7  # 4
```

Algo diferente de otros lenguajes es que se puede utilizar la sentencia **else** junto con el *while*.

El contenido del *else* se ejecutará una vez que la condición deje de cumplirse

**SENTENCIA MATCH**



La sentencia **match** es el equivalente a las sentencias *switch* o *case* de otros lenguajes de programación.

En el siguiente ejemplo puedes ver su sintaxis.



```
1 a = input("Dime qué ciclo estás estudiando: ")
2 match a:
3     case 'DAM':
4         print('Desarrollo de Aplicaciones Multiplataforma')
5     case 'DAW':
6         print('Desarrollo de Aplicaciones Web')
7     case 'ASIR':
8         print("Administración de Sistemas Informáticos y Redes")
9     case _:
10        print("No lo conozco")
```

El carácter \_ sirve como comodín para indicar la opción por defecto

Se pueden combinar varios literales en un mismo patrón utilizando el operador |



```
1 a = input("Dime qué ciclo estás estudiando: ")
2 match a:
3     case 'DAM' | 'DAW':
4         print('Es un ciclo de desarrollo')
5     case 'ASIR':
6         print('Es un ciclo de sistema')
7     case _:
8         print("No lo conozco")
```

# 4

## TIPOS DE DATOS



ENTEROS

Los **tipos enteros** permiten almacenar cualquier valor no decimal, tanto positivo como negativo.

Al contrario que en otros lenguajes de programación, Python no tiene límite en cuanto al tamaño de los números

```
a = 238203840293849028390482903849028394028390482093842390842
print( type(a) )

<class 'int'>
```

De lo anterior puedes deducir que Python asigna diferente tamaño en memoria según el número que almacene la variable.

Puedes ver el tamaño reservado para una variable en bits con la función **sys.getsizeof()**

```
import sys
x = 5**10000
y = 10
print(sys.getsizeof(x), type(x))
print(sys.getsizeof(y), type(y))
```

```
3120 <class 'int'>
```

```
28 <class 'int'>
```

Los números que nosotros indicamos están en base 10, pero tenemos la posibilidad de introducir los números en diferentes bases precediéndolos del prefijo correspondiente:

- **0o** para números en base octal
- **0x** para números en base hexadecimal
- **0b** para números en base dos



```
1 print(0o1234)      # 668
2
3 print(0x1AF5)      # 6901
4
5 print(0b1010)      # 10
```

Ten en cuenta que esto solo afecta a la forma en que escribimos nosotros los números, no tiene ningún efecto en la forma en que se almacenan.

Si queremos convertir cualquier otro tipo de datos a entero lo podemos hacer con la función **int()**

```
print( int( 3.141592 ) )  
print( int( "0123" ) )  
print( int( True ) )  
print( int( "0xAB", 16 ) )  
print( int( "010101", 2 ) )
```

3

123

1

171

21



BOOLEANOS

Las variables de tipo **booleano** permiten almacenar uno de dos valores: *True* o *False*. Se puede asignar su valor directamente o almacenarlo como resultado de una operación.

```
a = True
print( a )
b = 6 > 7
print( b )
```

True

False

Podemos convertir cualquier otro tipo de datos a booleano con la función **bool()**

```
print(bool(10))      # True
print(bool(-10))     # True
print(bool("Hola"))  # True
print(bool(0.1))     # True
print(bool([]))      # False
```

```
True
True
True
True
False
```

**FLOTANTES**

Permiten representar números positivos o negativos con decimales.

Al contrario que otros lenguajes de programación, que distinguen entre float y double, en Python no hay esa distinción, siendo todos los números decimales de tipo **float**.

```
pi = 3.141592
r = 7
print( pi * r**2 )

153.938008
```

También podemos representar los números decimales mediante la **notación científica**.

```
a = 1.85e-3  
print( a )
```

```
0.00185
```

Si queremos convertir otro tipo de datos a flotante debemos utilizar la función **float()**

```
a = float( 5 )  
b = float( "3.14" )  
print(a)  
print(b)
```

```
5.0
```

```
3.14
```

Al almacenar internamente los valores flotantes utilizando IEEE754, la precisión no es infinita.

```
f = 0.9999999999999999
print(f)
print(1 == f)
```

```
1.0
True
```

```
a = 0.3
b = 0.1
print(a-b)
```

```
0.19999999999999998
```

Al contrario que los enteros, los valores flotantes tienen un valor máximo y uno mínimo.

```
import sys
print( "Mínimo: " + str(sys.float_info.min) )
print( "Máximo: " + str(sys.float_info.max) )
```

Mínimo: 2.2250738585072014e-308

Máximo: 1.7976931348623157e+308



# CADENAS DE TEXTO

Al igual que otros lenguajes, las cadenas de texto en Python se pueden rodear de **comillas simples** o **cadenas dobles**. Esto es útil cuando queremos incluir uno de estos tipos de comillas dentro de la cadena

Alternativamente, también podemos usar el **símbolo de escape** (`\`) si quiero que ignore unas comillas dentro de otras.

```
1 print( "That's all" )
2 print( 'That\' all' )
```


El símbolo de escape también se puede utilizar para mostrar caracteres especiales, por ejemplo, `\n` para salto de línea.

Las **cadenas sin formato** se indican precediendo las primeras comillas con el carácter `r`. Esto hace que se trate el símbolo `\` como un carácter normal.

```
1 print( "Hola \n mundo" )    # Imprime el salto de línea
2 print( r"Hola \n mundo" )  # Imprime literalmente la cadena
```

Otro tipo de cadenas son las **cadenas multilínea**, representadas por las triples comillas. Estas se pueden prolongar más allá de una línea, respetando los saltos de línea.

Si queremos que se ignore un salto de línea deberemos usar el símbolo de escape al final de la línea.



```
1 print("""\
2 Uso:
3     -h: ayuda
4     -v: modo verbose
5 """)
```

En la versión 3.7 apareció otro tipo de cadenas llamadas **cadenas F**.

Se caracterizan porque están precedidas por el carácter **f** (o **F**) y permiten incluir en su interior una expresión rodeada de llaves que se evaluará en tiempo de ejecución.



```
1 name = "Victor"
2 print( f'Hola {name}' ) # Hola Victor
```



```
1 names = ['Victor', 'Pepe', 'Antonio']
2 for name in names:
3     print( f'Hola {name}' )
4
5 # Hola Victor
6 # Hola Pepe
7 # Hola Antonio
```

En el ejemplo anterior solo había una variable, pero se puede incluir cualquier **expresión** entre las llaves.

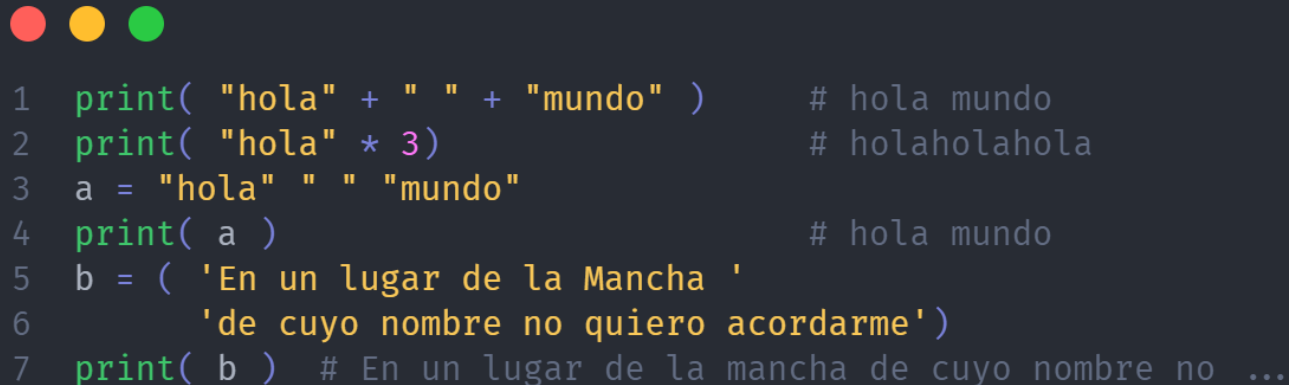


```
1 num = input("Dime un número: ")
2 print( f"El cuadrado del número es { int(num)**2 }" )
```

Observa en el ejemplo anterior que utilizamos la función **input()** para leer un dato por teclado y también usamos la función **int()** para convertir el dato introducido por el usuario a entero ya que, por defecto, aunque introduzcamos un número lo tratará como una cadena.


Las cadenas se pueden **concatenar** con el símbolo + y repetir con el símbolo \*

Dos cadenas literales una al lado de otra, se concatenan automáticamente, este es útil cuando se quiere dividir cadenas largas.



```
1 print( "hola" + " " + "mundo" )      # hola mundo
2 print( "hola" * 3 )                  # holaholahola
3 a = "hola" " " "mundo"
4 print( a )                           # hola mundo
5 b = ( 'En un lugar de la Mancha '
6       'de cuyo nombre no quiero acordarme' )
7 print( b ) # En un lugar de la mancha de cuyo nombre no ...
```

Las cadenas de texto en Python se pueden **indexar**, accediendo a caracteres como si fuera un array, teniendo en cuenta que el índice 0 corresponde al primer carácter. También se pueden utilizar índices negativos para indicar que se comienza desde el final de la cadena.



```
1  cad = "Hola mundo"
2  print( cad[0] )      # H
3  print( cad[2] )      # l
4  print( cad[-2] )     # d
```

Otra posibilidad con las cadenas es el denominado **slicing**, que permite extraer subcadenas de una cadena indicando los índices del primer (incluido) y último elemento de la subcadena (excluido) separados por el carácter dos puntos.

Admite índices negativos y, si se omite uno de los dos, se entiende que es desde el principio de la cadena o hasta el final de la misma.



```
1 cad = "Hola mundo"
2 print( cad[0:2] )      # Ha
3 print( cad[3:8] )      # a mun
4 print( cad[:4] )        # Hola
5 print( cad[-5:-1])     # mund
6 print( cad[8:])         # do
```



Opcionalmente se puede poner un tercer elemento al hacer slicing poniendo otros dos puntos. Este tercer elemento representará el **salto**, es decir, alternará los elementos que coge de la cadena.

```
print("abcdefghijkl"[0:6:2])
```

```
ace
```

También se puede poner un salto negativo para indicar que la cadena se recorrerá en sentido inverso.

```
print("abcdefghijkl"[8:0:-2])  
print("abcdefghijkl"[::-1])
```

```
igec
```

```
kjihgfedcba
```

Una característica de las cadenas en Python es que son **inmutables**, es decir, no se pueden modificar, por lo que si intentamos asignar un valor a una posición indexada nos dará error.

```
1 cad = "DAM"
2 cad[2] = 'W'
```

```
PS C:\proyectos\SGE\python> python .\ejemplo.py
Traceback (most recent call last):
  File "C:\proyectos\SGE\python\ejemplo.py", line 2, in <module>
    cad[2] = 'W'
    ~~~^^^
TypeError: 'str' object does not support item assignment
PS C:\proyectos\SGE\python>
```

Si queremos modificar una cadena necesitaríamos crear una nueva cadena y asignarla a la misma variable, tal como se ve en el siguiente ejemplo



```
1 cad = "DAM"
2 #cad[2] = 'W'          ERROR
3 cad = cad[:2] + 'W'
4 print( cad )          # DAW
```

Al igual que todo en Python, las cadenas de texto son objetos, y, como tales, tienen una serie de métodos asociados.

Podemos saber cuál es el nombre del objeto al que pertenece una variable utilizando la función integrada `type()`

Si le pasamos una cadena veremos que el nombre de la clase es **str**.



```
1 print(type("hola"))
```



```
1 dir(str)
```

Para ver todos los métodos que pertenecen a la clase **str** utilizamos la función integrada `dir()`

```
>>> type("a")
<class 'str'>
>>> dir(str)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__l
en__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__s
etattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandt
abs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removep
refix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith
', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> []
```

Si queremos saber qué hace exactamente alguno de estos métodos tenemos la función integrada `help()`, que recibe como parámetro el nombre de un método.

```
>>> help(str.swapcase)
Help on method_descriptor:

swapcase(self, /) unbound builtins.str method
    Convert uppercase characters to lowercase and lowercase characters to uppercase.
```

Observa que esto lo podemos aplicar a cualquier objeto.

```
>>> dir( list )
['_add_', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir( float )
['_abs_', '__add__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattr__', '__getformat__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__int__', '__le__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

Además, hay una serie de funciones que no pertenecen a ninguna clase (o más bien pertenecen a un objeto especial llamado `__builtins__`)

Estas funciones pueden ser invocadas sin necesidad de indicar el objeto delante.



```
1  dir( __builtins__ )
```

```
>>> dir( __builtins__ )
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'Conne
ctionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup',
'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'Lc
okupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError',
'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'PythonFinalizationError', 'Re
cursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'Synta
xError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'Unicc
deDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warr
ing', 'WindowsError', 'ZeroDivisionError', '_', '_IncompleteInputError', '__build_class__', '__debug__', '__doc__', '__import__', '
__loader__', '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 't
ytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divm
od', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help',
'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'se
tattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

- **str.capitalize():** devuelve la cadena con todas las letras en minúsculas salvo la primera letra

```
"HOLA MUNDO".capitalize()
```

```
'Hola mundo'
```

- **str.title():** devuelve la cadena con todas las letras en minúsculas salvo la primera letra de cada palabra.

```
"HOLA MUNDO".title()
```

```
'Hola Mundo'
```

- **str.upper():** devuelve la cadena con todas las letras en minúsculas salvo la primera letra

```
"Hola mundo".upper()
```

```
'HOLA MUNDO'
```

- **str.lower():** devuelve la cadena con todas las letras en minúsculas salvo la primera letra de cada palabra.

```
: "Hola Mundo".lower()
```

```
: 'hola mundo'
```



- **str.center(width, [fillchar]):** centra el texto en una cadena de la longitud indicada y rellenando con el carácter indicado

```
"DAM".center(9, '*')
```

```
'***DAM***'
```

- **str.count(sub[, start[, end]]):** devuelve el número de ocurrencias de la subcadena *sub* en la cadena entre las posiciones indicadas

```
"Desarrollo de Aplicaciones Multiplataforma".count('a')
```

```
5
```

- **`str.find(sub[, start[, end]])`**: devuelve la posición de la primera ocurrencia de la cadena *sub* en la cadena entre las posiciones indicadas.

```
"Desarrollo de Aplicaciones Multiplataforma".find('de')
```

```
11
```

- **`str.isalnum()` | `str.isalpha()` | `str.isdigit()`**: devuelve *true* si todos los caracteres de la cadena son alfanuméricos, alfabéticos o numéricos respectivamente.

```
"DAM2".isalpha()
```

```
False
```

- **str.join(iterable):** devuelve la concatenación de las cadenas en el iterable, usa como separador la cadena *str*.

```
"-".join( ["a", "b", "c"] )
```

```
'a-b-c'
```

- **str.replace(old, new[, count]):** devuelve la cadena reemplazando todas las ocurrencias de la cadena *old* por la cadena *new* un máximo de *count* veces.

```
"Desarrollo de Aplicaciones Multiplataforma".replace('a', 'X')
```

```
'DesXrrollo de AplicXciones MultiplXtXformX'
```

- **str.split(sep, maxsplit):** divide la cadena usando como separador el valor *sep* y devuelve una lista con todas las subcadenas obtenidas.

```
"DAM ASIR DAW".split(" ")
```

```
['DAM', 'ASIR', 'DAW']
```

- **str.strip([chars]):** devuelve la cadena con los caracteres indicados eliminados, tanto al principio como al final

```
"  DAM  ".strip(" ")
```

```
'DAM'
```

LISTAS

Python tiene varios tipos de datos compuestos que agrupan varios valores. El más común es la **lista**, que es una lista de valores separados por comas entre corchetes.

Los elementos de la lista pueden ser de diferentes tipos de datos.

Las listas se pueden **indexar** y **segmentar** al igual que las cadenas.



```
1 letras = [ 'a', 'b', 'c', 'd', 'e', 'f' ]
2 print( letras[1] )      # b
3 print( letras[2:4] )    # ['c', 'd']
```

También se pueden concatenar con el símbolo +



```
1 letras = [ 'a', 'b', 'c' ]
2 print( letras + ['y', 'z'] )    # ['a', 'b', 'c', 'y', 'z']
```

Al contrario que las cadenas, las listas son **mutables**, por lo que podemos cambiar el valor de sus elementos



```
1 letras = [ 'a', 'b', 'c' ]  
2 letras[1] = 'y'  
3 print( letras ) # [ 'a', 'y', 'c' ]
```

Una cosa importante cuando manipulamos listas en Python es **que las variables se almacenan por referencia.**

Observa el siguiente ejemplo:



```
1 a = [ 'a', 'b', 'c' ]
2 b = a
3 a[2] = 'x'
4 print(a)      # ['a', 'b', 'x']
5 print(b)      # ['a', 'b', 'x']
```


Si te fijas, tenemos una lista en la variable `a` y la copiamos a la variable `b`, pero si modificamos un elemento de la lista `a` también se modificará en la lista `b`.



Esto se debe a que Python almacena las variables **por referencia**, esto quiere decir que cuando yo creo una lista lo que hace en realidad es que: se crea la lista, se almacena en una posición de memoria y en la variable se **almacena la dirección a esa posición de memoria**.

De esta forma, cuando en nuestro ejemplo hemos hecho `b=a` lo que estamos haciendo es que `b` contenga una copia del contenido de `a`, que es la dirección en que se almacena la lista y no la propia lista. Así, tendremos dos variables que apuntan a la misma dirección de memoria que almacena un único array.

La duda que surge entonces es: ¿cómo hacemos si queremos copiar la lista creando una nueva? La solución es mediante el *slicing* como se ve en el siguiente ejemplo:



```
1  a = [ 'a', 'b', 'c' ]
2  b = a[:]
3  a[2] = 'x'
4  print(a)      # ['a', 'b', 'x']
5  print(b)      # ['a', 'b', 'c']
```

Al hacer el *slicing* obtenemos una nueva copia de la lista del segmento indicado (en este caso toda la lista), y es lo que asignamos a la variable *b*.

Al igual que con las cadenas, las listas también disponen de una serie de métodos:

- **list.append(item):** agrega un elemento al final de la lista

```
a = [ 'a', 'b', 'c' ]  
a.append('x')  
print(a)
```

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

- **list.extend(list2):** extiende una lista agregando todos los elementos de la segunda lista al final.

```
a = [ 'a', 'b', 'c' ]  
a.extend(['y', 'z'])  
print(a)
```

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

- **list.insert(index, item):** inserta un elemento en la posición indicada de la lista

```
a = [ 'a', 'b', 'c' ]  
a.insert(2, 'x')  
print(a)
```

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

- **list.remove(item):** elimina la primera ocurrencia del elemento indicado como parámetro en la lista.

```
a = [ 'a', 'b', 'c' ]  
a.remove('b')  
print(a)
```

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

- **list.pop(index):** elimina y devuelve el elemento que se encuentra en la posición indicada. Si no se indica parámetro devuelve el último elemento

```
lista = ['a', 'b', 'c']  
elem = a.pop()  
print(lista)  
print(elem)
```

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

- **list.index(item):** devuelve la posición de la primera ocurrencia del elemento indicado como parámetro

```
lista = ['a', 'b', 'c', 'a', 'b']  
pos = lista.index('b')  
print(pos)
```

```
1
```

- **list.count(item):** devuelve el número de ocurrencias del elemento pasado como parámetro en la lista.

```
lista = ['a', 'b', 'c', 'a', 'b']  
num = lista.count('b')  
print(num)
```

2

- **list.sort():** ordena los elementos de la lista

```
lista = [ 's', 'b', 'a', 'm', 'e' ]  
lista.sort()  
print(lista)
```

```
['a', 'b', 'e', 'm', 's']
```


- **list.reverse()**: invierte el orden de los elementos de la lista

```
lista = [ 's', 'b', 'a', 'm', 'e' ]  
lista.reverse()  
print(lista)  
  
['e', 'm', 'a', 'b', 's']
```

Observa que, dado que las listas son **mutables**, todos los métodos anteriores modifican el propio array.

Las listas son consideradas **iterables**, lo que quiere decir que podemos recorrer cada uno de sus elementos en un bucle.

Observa cómo es la sintaxis para hacerlo.



```
1  lista = [ 'a', 'b', 'c', 'd' ]
2  for elem in lista:
3      print(elem)
```

Es importante que te fijas en el carácter dos puntos al final de la línea para indicar que comienza un bloque de código y en que el bloque de código está tabulado con respecto a la línea que lo contiene.



Si queremos iterar sobre una lista numérica disponemos de la función **range()** para generarla.

```
for num in range(4):  
    print(num)
```

```
0  
1  
2  
3
```

Según el número de parámetros puede variar su comportamiento:

- 1 parámetro: iterará desde el 0 hasta el valor pasado como parámetro
- 2 parámetros: iterará desde el primer valor hasta el segundo
- 3 parámetros: utilizará el tercer parámetro como salto

El parámetro salto permite obtener listas con elementos a intervalos

```
for num in range(0, 6, 2):  
    print(num)
```

0  
2  
4

Con un salto negativo podremos obtener listas descendentes

```
for num in range(0, -5, -1):  
    print(num)
```

0  
-1  
-2  
-3  
-4

Una funcionalidad interesante que tenemos con listas es el denominado **desempaquetado** (*unpacking*), que consiste en extraer valores de una secuencia (lista, tupla, conjunto, ...) y asignarlos directamente a variables individuales.

```
num = [ 345, 75, 392 ]  
a, b, c = num  
print(a)  
print(b)  
print(c)
```

345

75

392

El número de variables tiene que ser igual que el número de elementos en la secuencia, pero también se puede usar el **desempaquetado extendido**, que permite asignar múltiples valores restantes a una lista utilizando el operador `*`.

```
num = [ 345, 75, 392 ]  
a, *b = num  
print(a)  
print(b)
```

345

[75, 392]

Observa que el operador `*` lo podemos poner en cualquiera de las variables. En este caso, empieza a rellenar las variables por la izquierda y por la derecha y, lo que quede, lo almacena en la lista.

```
num = [ 345, 75, 392, 127 ]  
a, *b, c = num  
print(a)  
print(b)  
print(c)
```

345

[75, 392]

127

La **comprensión de listas** es un constructor de Python que permite generar listas fácilmente con una única línea de código.

Observa el siguiente ejemplo:

```
cuadrados = [i**2 for i in range(5)]  
print(cuadrados)
```

```
[0, 1, 4, 9, 16]
```

El funcionamiento del código anterior es el siguiente:

- La variable *i* iterará entre los valores 0 y 4
- Para cada uno de los valores se evaluará la expresión *i\*\*2*
- El valor devuelto por esta expresión se añadirá a la lista devuelta

Otro ejemplo:

```
lista = [ 'asir', 'dam', 'daw' ]  
lista2 = [ c.upper() for c in lista ]  
print(lista2)
```

```
['ASIR', 'DAM', 'DAW']
```

Observa que lo que va después del **in** puede ser cualquier iterable, por ejemplo, también una cadena.

```
lista = [ c for c in "python" ]  
print(lista)
```

```
['p', 'y', 't', 'h', 'o', 'n']
```

En los ejemplos anteriores, para cada elemento del iterable generamos un elemento en la nueva lista, pero se pueden aplicar **condicionales** para no tener en cuenta todos los elementos.

```
frase = "El perro de san roque no tiene rabo"  
erres = [ i for i in frase if i == 'r' ]  
print(erres)  
  
['r', 'r', 'r', 'r']
```

En este caso solo se evalúa la expresión si se cumple la condición.



# TUPLAS

Las **tuplas** son un tipo de datos muy similar a las listas, con la excepción de que son **inmutables**.

Para declarar una tupla, lo haremos igual que con las listas, pero usando paréntesis en lugar de corchetes.

```
ciclos = ( "DAM", "DAW", "ASIR" )  
print( type(ciclos) )  
  
<class 'tuple'>
```

**Similitudes** entre tuplas y listas:

- Ambas pueden contener datos de diferentes tipos
- Son estructuras de datos ordenadas
- Son estructuras de datos secuenciales sobre los que se puede iterar
- Ambas permiten acceder a sus elementos directamente mediante su índice indicándolo entre corchetes.

**Diferencias** entre tuplas y listas:

- Las tuplas son **inmutables**, mientras que las listas son mutables. Esto es útil cuando queremos definir un dato que no pueda ser cambiado.
- Una vez definidas, las tuplas tienen una **longitud fija** y las listas tienen una longitud dinámica.
- Las tuplas usan **menos memoria** y tienen un **acceso más rápido** que las listas

Las tuplas tienen muchos menos métodos que las listas, en concreto, solo tienen los métodos **count()** e **index()**

```
letters = ('a', 'a', 'b', 'c', 'a', 'c')
print( letters.count('c') )      # Número de elementos de la tupla
print( letters.index('c') )      # Índice de la primera ocurrencia
```

2

3

Podemos realizar conversiones entre tuplas y listas mediante las funciones **tuple()** y **list()**

```
print( tuple(['a', 'b', 'c']) )
print( list((1, 2, 3)) )
```

('a', 'b', 'c')

[1, 2, 3]

DICCIONARIOS

Los **diccionarios** son estructuras de datos que nos permiten almacenar datos en forma de clave valor.

Es equivalente a lo que en otros lenguajes de programación se conoce como arrays asociativos, tablas hash o mapas.

Se declaran mediante una serie de pares **clave: valor**, separados por comas y rodeado todo del símbolo de llaves.

```
user = {  
    "nombre": "Victor",  
    "pass": "paso",  
}
```

También se pueden crear mediante **dict()** pasándole una lista de tuplas de la siguiente forma:

```
user = dict([
    ("nombre", "victor"),
    ("pass", "paso")
])
print(user)
```

```
{'nombre': 'victor', 'pass': 'paso'}
```

Y una tercera forma:

```
user2 = dict( nombre='victor',
              passwd='paso' )
print(user2)
```

```
{'nombre': 'victor', 'passwd': 'paso'}
```

Se puede acceder a los elementos de un diccionario mediante el nombre de la clave usando los corchetes o bien con la función **get()**

```
user = dict( nombre='victor',  
             passwd='paso' )  
print( user['nombre'] )  
print( user.get('passwd') )
```

```
victor  
paso
```

Como son elementos **mutables**, se puede cambiar el valor de una clave simplemente asignándole otro valor. Si la clave no existiera la crea.

```
user['passwd'] = 'P@ssw0rd'  
user['isAdmin'] = True  
print(user)
```

```
{'nombre': 'victor', 'passwd': 'P@ssw0rd', 'isAdmin': True}
```



Se puede iterar sobre los elementos de un diccionario de forma similar a otros iterables. Observa que lo que obtenemos son las claves de cada elemento.

```
for data in user:  
    print(data)
```

```
nombre  
passwd  
isAdmin
```

Si lo que queremos son los valores

```
for data in user:  
    print(user[data])
```

```
victor  
P@ssw0rd  
True
```

Si al iterar queremos obtener ambos valores, podemos utilizar el método **ítems()**, que en cada iteración devolverá una tupla que contiene la clave y el valor correspondiente.

```
for data in user.items():  
    print(data)
```

```
('nombre', 'victor')  
('passwd', 'P@ssw0rd')  
('isAdmin', True)
```

Los diccionarios también disponen de una serie de métodos para operar con ellos:

- **dict.clear():** borra todos los registros de un diccionario

```
a = { "nombre": "Victor", "apellido": "González"}
a.clear()
print(a)

{}
```

- **dict.get(key[, default]):** devuelve el valor de una clave, si no la encuentra y hay segundo parámetro devolverá este.

```
a = { "nombre": "Victor", "apellido": "González"}
print( a.get("nombre") )
print( a.get("nickname", "unknown") )

Victor
unknown
```

- `dict.items()`: devuelve un iterable que contiene pares clave valor del diccionario. Se puede convertir a lista para iterar sobre estos pares.

```
a = { "nombre": "Victor", "apellido": "González"}  
print(a.items())  
print(type(a.items()))  
print(list(a.items()))
```

```
dict_items([('nombre', 'Victor'), ('apellido', 'González')])  
<class 'dict_items'>  
[('nombre', 'Victor'), ('apellido', 'González')]
```

- **dict.keys():** devuelve un iterable con todas las claves del diccionario

```
a = { "nombre": "Victor", "apellido": "González"}  
print(a.keys())  
for b in a.keys():  
    print(b)
```

```
dict_keys(['nombre', 'apellido'])  
nombre  
apellido
```

- **dict.values():** devuelve un iterable con todos los valores del diccionario

```
a = { "nombre": "Victor", "apellido": "González"}  
print(a.values())
```

```
dict_values(['Victor', 'González'])
```

- `dict.pop(key[, default])`: busca y elimina el elemento cuya clave se pasa como parámetro, devolviendo su valor asociado. Da error si la clave no existe, salvo que se indique un segundo parámetro que es el valor que devolverá en caso de que la clave no exista.

```
a = { "nombre": "Victor", "apellido": "González"}
a.pop("apellido")
print(a)
a.pop("nickname", "unknown")

{'nombre': 'Victor'}
'unknown'
```

- **dict.update()**: se llama sobre un diccionario y tiene como entrada otro diccionario. Los valores de las claves ya existentes son actualizadas y, si hay alguna clave nueva, se añade al diccionario.

```
d1 = {'a': 1, 'b': 2}
d2 = {'a': 0, 'd': 400}
d1.update(d2)
print(d1)

{'a': 0, 'b': 2, 'd': 400}
```

De forma análoga a la compresión de listas, Python también permite la **compresión de diccionarios** para generar diccionarios, la única diferencia con respecto a las listas es que en este caso tienen que devolver dos valores (lo cual se indica separando ambos valores por el carácter dos puntos).

```
alumnos = {  
    "Andrea": ['SGE', 'DI', 'PSP', 'PMDM' ],  
    "Alonso": ['DI', 'SGE'],  
    "Evaristo": ['DI', 'PSP', 'PMDM']  
}  
  
alumnos_sge = {nombre: modulos for nombre, modulos in alumnos.items() if 'SGE' in modulos}  
print(alumnos_sge)  
  
{'Andrea': ['SGE', 'DI', 'PSP', 'PMDM'], 'Alonso': ['DI', 'SGE']}
```



### Otro ejemplo sin aplicar filtros

```
lista1 = ['nombre', 'edad', 'región']  
lista2 = ['Pelayo', 30, 'Asturias']  
  
mi_dict = {i:j for i,j in zip(lista1, lista2)}  
print(mi_dict)  
  
{'nombre': 'Pelayo', 'edad': 30, 'región': 'Asturias'}
```

Aquí estamos utilizando la función **zip()** que sirve para combinar varias listas generando un iterable en el que cada elemento es una tupla (observa cómo en el ejemplo estamos usando el desempaquetado para extraer los dos valores de la tupla en las variables *i* y *j*) con el enésimo elemento de cada lista.

SET

Los **set** son un tipo de datos de Python similares a las listas, pero que tienen las siguientes características:

- Los elementos del set son **únicos**. No hay elementos duplicados.
- Los sets son **desordenados**
- Sus **elementos** deben ser **inmutables**.

Los sets se crean con la función **set()** pasándole cualquier iterable como argumento.

```
a = set( [5, 3, 6, 3, 7, 4, 4, 4] )  
print(a)
```

```
{3, 4, 5, 6, 7}
```

Alternativamente, se pueden crear de forma análoga a una lista, pero utilizando llaves en lugar de corchetes.

```
s = {5, 4, 6, 8, 8, 1}
print(s)
print(type(s))

{1, 4, 5, 6, 8}
<class 'set'>
```

Los sets son **iterables**, por lo que podemos iterar sobre sus elementos con un for.

```
s = {5, 4, 6, 8, 8, 1}
for elem in s:
    print(elem)
```

```
1
4
5
6
8
```

Podemos saber el número de elementos que tiene con len()

```
s = {5, 4, 6, 8, 8, 1}
print( len(s))
```

```
5
```

Como con otros tipos de datos, podemos verificar la pertenencia con el operador **in**

```
s = {5, 4, 6, 8, 8, 1}
print( 2 in s )

False
```

Con el operador **|** podemos realizar la operación unión entre dos sets

```
a = {5, 4, 6, 8, 8, 1}
b = {5, 3, 6, 3, 2}
print( a | b )

{1, 2, 3, 4, 5, 6, 8}
```

Métodos de los sets.

- `set.add(elem)`: añade un elemento al set.
- `set.remove(elem)`: elimina el elemento del set. Si no lo encontrara lanza una excepción

- `set.discard(elem)`: similar a `remove()`, pero no hace nada si no encuentra el elemento.

- `set.pop()`: devuelve y elimina un elemento aleatorio del set.



- `set.clear()`: elimina todos los elementos del set

- `set.union(s1[, s2...])`: devuelve la unión de dos o más sets

- `set.intersection(s1[, s2...])`: devuelve la intersección de dos o más sets

- `set.issubset(s)`: indica si un set es subconjunto de otro

**MUTABILIDAD**

En Python, todos los tipos de datos son objetos, ya sean enteros, cadenas o incluso funciones.

```
def func():  
    return True  
print(type(func))  
print(type(6))  
  
<class 'function'>  
<class 'int'>
```

Teniendo esto en cuenta, ¿cuándo consideramos que dos objetos son iguales? ¿En qué se diferencia un objeto de otro?

Para responder a estas preguntas necesitamos comprender tres conceptos: identidad, tipo y valor.

- **Identidad:** nunca cambia e identifica de manera unívoca el objeto. El operador **is** nos permite saber si dos objetos son en realidad el mismo, es decir, si dos variables hacen referencia al mismo objeto.
- **Tipo:** nos indica la clase a que pertenece un tipo de datos, como *float* o *str*.
- **Valor:** todo objeto tiene unas características particulares. Si estas características pueden ser modificadas diremos que es un tipo **mutable**, en caso contrario es **inmutable**.

Podemos obtener el identificador de un objeto con la función **id()**

```
a=512
b=a
c=512
print(f'a: {a} - b: {b} - c: {c}')
print(f'id(a): {id(a)} - id(b): {id(b)} - id(c): {id(c)}')
```

a: 512 - b: 512 - c: 512

id(a): 140491828811056 - id(b): 140491828811056 - id(c): 140491828819312

Aunque son dos variables  
diferentes referencian al mismo  
objeto

Aunque tiene el mismo valor es  
un objeto diferente, luego tiene  
diferente identificador

Los tipos de datos cuyo valor se puede modificar y siguen manteniendo su identificador se consideran **mutables**, el resto de tipos son **inmutables**.

```
a=512
print(id(a))
a=1024
print(id(a))
```

```
140106676208976
140106676207760
```

Los enteros son inmutables, por lo que si cambio su valor realmente lo que hago es crear una nueva variable

```
a = [ 12 ]
print( id(a) )
a[0] = 128
print( id(a) )
```

```
140106676203392
140106676203392
```

Las listas son mutables, por lo que, si modifico su valor, el objeto lista sigue siendo el mismo.

El operador igual (==) me indica si dos objetos tienen el mismo valor, pero si quiero saber si dos objetos son el mismo necesito el operador **is**.

```
a = [127]
b = [127]
print( f"a==b: {a==b}" )
print( f"a is b: {a is b}" )
```

```
a==b: True
a is b: False
```



**Elementos mutables:**

- Listas
- Bytearray
- Memoryview
- Diccionarios
- Sets
- Clases definidas por el usuario

**Elementos inmutables:**

- Booleanos
- Complejos
- Enteros
- Float
- Frozenset
- Cadenas
- Tuplas
- Range
- Bytes

Los elementos **inmutables** son **más rápidos de acceder**, pero más lentos de modificar (ya que realmente estamos creando un objeto nuevo), así que hay que pensar bien qué uso vamos a dar a un objeto cuando creamos objetos que tienen ambas variantes (p.e. listas y tuplas)

# 5

## FUNCIONES



Las funciones en Python se crean con la palabra reservada **def**, seguida del nombre de la función y luego la lista de parámetros que admite entre paréntesis, finalizando con el carácter dos puntos.

El bloque de código del cuerpo de la función se distingue por el indentado.

El valor devuelto por la función se indica mediante la sentencia **return**



```
1  # Devuelve True si el número es par
2  def is_even(num):
3      return num%2 != 0
```

Se puede indicar un **valor por defecto** para los argumentos utilizando el carácter igual




```
1 def saluda( nombre='desconocido' ):  
2     print( 'Hola ' + nombre)
```

Cuando invocamos una función hay dos formas de tratar los parámetros: posicionales y por nombre.

### Argumentos posicionales

Es como estamos acostumbrados en cualquier lenguaje de programación, los parámetros se indican en la llamada en el mismo orden en que están en la declaración de la función.

```
def potencia(a, b):  
    return a**b  
  
potencia(3, 6)
```



729

## Argumentos por nombre

En este caso, al invocar la función, se indica el nombre de cada parámetro y su valor separados por el símbolo igual. De esta forma, los argumentos se pueden indicar en cualquier orden.

```
def potencia(base, exponente):  
    return base**exponente  
  
potencia(exponente=6, base=3)
```

729

También es posible recibir en la función un **número variable de argumentos con nombre**, es decir, los recibiríamos en un diccionario en lugar de en una tupla. Esto se hace con el doble **\*\*** y, por costumbre, se suele llamar **\*\*kwargs**

```
def set_data( **kwargs ):
    print(kwargs)

set_data( nombre="victor", apellidos="gonzález" )

{'nombre': 'victor', 'apellidos': 'gonzález'}
```

Fíjate que todos estos tipos de parámetros se pueden combinar

```
def ejemplo_funcion(a, b, *args, c=10, d=20, **kwargs):  
    # a y b son parámetros posicionales obligatorios  
    print(f"Posicionales obligatorios: a={a}, b={b}")  
    # args es una tupla que captura argumentos posicionales adicionales  
    print(f"Argumentos posicionales adicionales (args): {args}")  
    # c y d son parámetros por nombre (con valores por defecto)  
    print(f"Parámetros por nombre: c={c}, d={d}")  
    # kwargs es un diccionario que captura argumentos con nombre adicionales  
    print(f"Argumentos con nombre adicionales (kwargs): {kwargs}")  
  
# Llamada a la función  
ejemplo_funcion(1, 2, 3, 4, 5, c=30, e=40, f=50)
```

Posicionales obligatorios: a=1, b=2

Argumentos posicionales adicionales (args): (3, 4, 5)

Parámetros por nombre: c=30, d=20

Argumentos con nombre adicionales (kwargs): {'e': 40, 'f': 50}



Al igual que con casi todos los lenguajes de programación, la sentencia **return** sirve para salir de la función y también para devolver uno o varios parámetros.

Algo interesante en Python es que podemos **devolver varios valores** separándolos por comas. Esto hará que devuelva una tupla que podrá ser recogida en varias variables mediante el desempaqueado.

```
def suma_y_media(a, b, c):  
    suma = a+b+c  
    media = suma//3  
    return suma, media  
suma, media = suma_y_media(9, 6, 3)  
print(suma) # 18  
print(media) # 6.0  
valores = suma_y_media(7, 6, 8)  
print(valores)
```

18

6

(21, 7)

Al igual que con casi todos los lenguajes de programación, la sentencia **return** sirve para salir de la función y también para devolver uno o varios parámetros.

Algo interesante en Python es que podemos **devolver varios valores** separándolos por comas. Esto hará que devuelva una tupla que podrá ser recogida en varias variables mediante el desempaquetado.

```
def suma_y_media(a, b, c):  
    suma = a+b+c  
    media = suma//3  
    return suma, media  
suma, media = suma_y_media(9, 6, 3)  
print(suma) # 18  
print(media) # 6.0  
valores = suma_y_media(7, 6, 8)  
print(valores)
```

18

6

(21, 7)

Un buen hábito de programación es **documentar** nuestro código, y Python contribuye a ello con el denominado **docstring**, que es una documentación que podemos poner en cada función al principio de la misma mediante triples comillas.

Esta documentación puede ser consultada posteriormente con la función **help()**

```
def suma_y_media(a, b, c):  
    """Obtiene la suma y la media de los parámetros que se le pasen"""  
    suma = a+b+c  
    media = suma//3  
    return suma, media  
help(suma_y_media)
```

Help on function suma\_y\_media in module \_\_main\_\_:

```
suma_y_media(a, b, c)  
    Obtiene la suma y la media de los parámetros que se le pasen
```

Otra funcionalidad que nos ayudará a documentar nuestro código son las **function annotation**, que permite añadir metadatos a las funciones indicando los tipos esperados tanto de entrada como de salida.

```
def suma(a:int, b:int) -> int:
    """Suma dos números"""
    return a + b
help(suma)
```

```
Help on function suma in module __main__:
```

```
suma(a: int, b: int) -> int
    Suma dos números
```

# 6

## PROGRAMACIÓN FUNCIONAL



Como dijimos al principio de esta unidad, Python también es un **lenguaje funcional**, lo que quiere decir que las funciones son valores de primera clase y, por tanto, pueden ser tratadas como cualquier otro objeto del lenguaje, por ejemplo, para pasarlas como parámetro a otra función.

```
def get_hello():  
    return "Hola mundo!!"
```

```
def get_bye():  
    return "Adios"
```

```
def print_message(f):  
    print( f() )
```

```
print_message( get_hello )  
print_message( get_bye )
```

```
Hola mundo!!  
Adios
```

**IMPORTANTE:** observa que cuando pasamos una función como parámetro lo hacemos sin los paréntesis. Si pusiéramos paréntesis estaríamos pasando el valor devuelto por la función, y no la propia función

Pero antes de ver con más detalles esto de la programación funcional, veamos un tipo especial de funciones en Python denominadas **funciones lambda** o **anónimas**.

Simplemente son una forma diferente de declarar funciones que es útil cuando tenemos funciones muy cortas cuyo código cabe en una línea.

Observa el ejemplo en el que se declaran dos funciones que son iguales:

```
def suma1(a, b):
```

```
    return a+b
```

```
suma2 = lambda a, b: a + b
```

```
print( suma1(5, 7) )
```

```
print( suma2(7, 3) )
```

```
12
```

```
10
```

Parámetros de  
la función

Cuerpo de la función, está implícito que la función  
devuelve el valor de esta expresión

Las funciones lambda no tienen  
nombre, en este ejemplo la  
asigno a una variable para  
poder utilizarla.

En el ejemplo anterior asignamos la función a una variable para usarla, pero si solo la vamos a usar una vez podemos hacerlo con esta sintaxis:

```
(lambda a, b: a + b)(2, 4)
```

6

Aunque lo más común probablemente sea crearlas directamente cuando se pasan como parámetro a otra función.

```
def apply_operation(func, data):  
    return [func(x) for x in data]  
  
result = apply_operation(lambda x: x * 2, [1, 2, 3, 4, 5])  
print(result)
```

[2, 4, 6, 8, 10]



Volviendo a la **programación funcional**, hay una serie de funciones en Python que son comunes en otros lenguajes funcionales y que serán muy útiles en muchas ocasiones para evitar el uso de bucles. Estas funciones son **map()**, **filter()** y **reduce()**

## Función map()

Esta función toma dos entradas\_\_

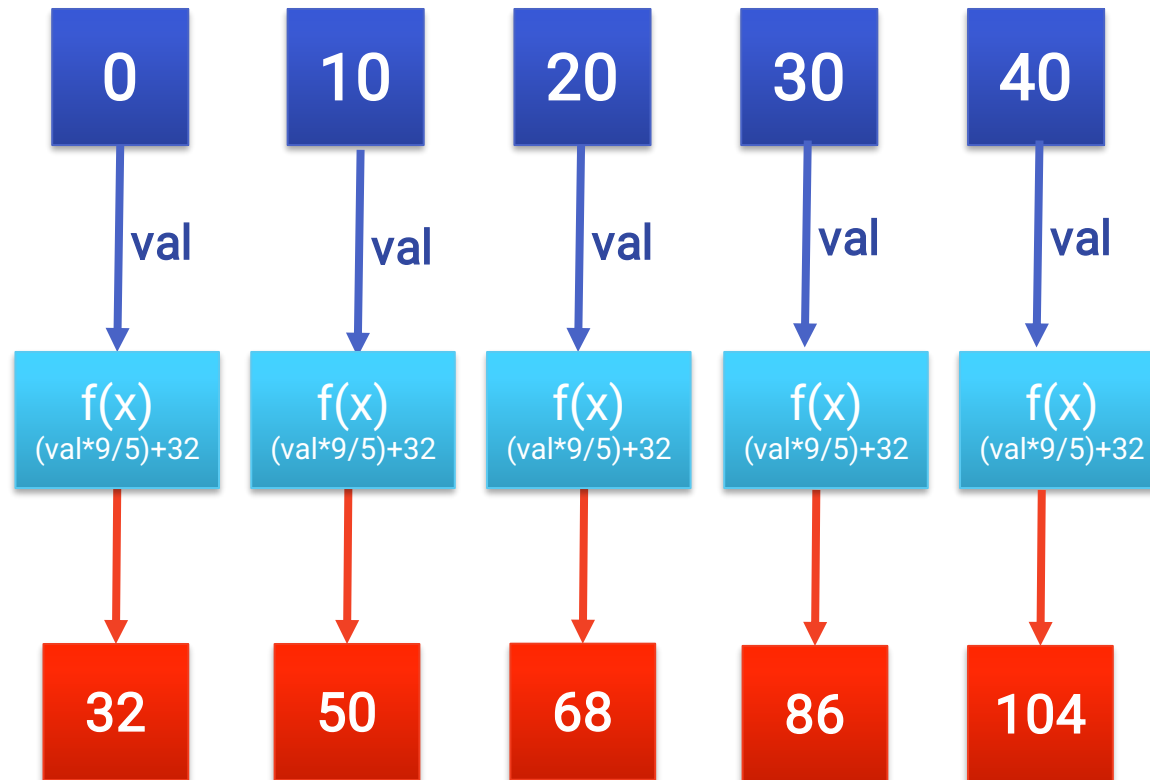
- Una lista o iterable
- Una función

Esta función aplicará la función pasada a cada uno de los elementos del iterable, devolviendo un iterable con los valores devueltos por la función.

```
celsius = [0, 10, 20, 30, 40]
fahrenheit = list(map(lambda x: (x * 9/5) + 32, celsius))

print(fahrenheit)

[32.0, 50.0, 68.0, 86.0, 104.0]
```



Observa que esta función se puede utilizar con cualquier tipo de iterable

```
# Precios en euros
precios_e = {
    "manzanas": 2.5,
    "naranjas": 3.0,
    "plátanos": 1.8
}

# Función para convertir euros a dólares
def to_dolars(euros):
    return round( euros * 1.1, 2 )

# Usamos map() para aplicar la conversión solo a los valores del diccionario
precios_d = dict(map(lambda item: (item[0], to_dolars(item[1])), precios_e.items()))

print(precios_d)

{'manzanas': 2.75, 'naranjas': 3.3, 'plátanos': 1.98}
```

## Función filter()

La función **filter()** también recibe como parámetros una función y un iterable, pero en este caso devuelve un iterable que contiene únicamente los elementos para los cuales la función pasada devuelva True.

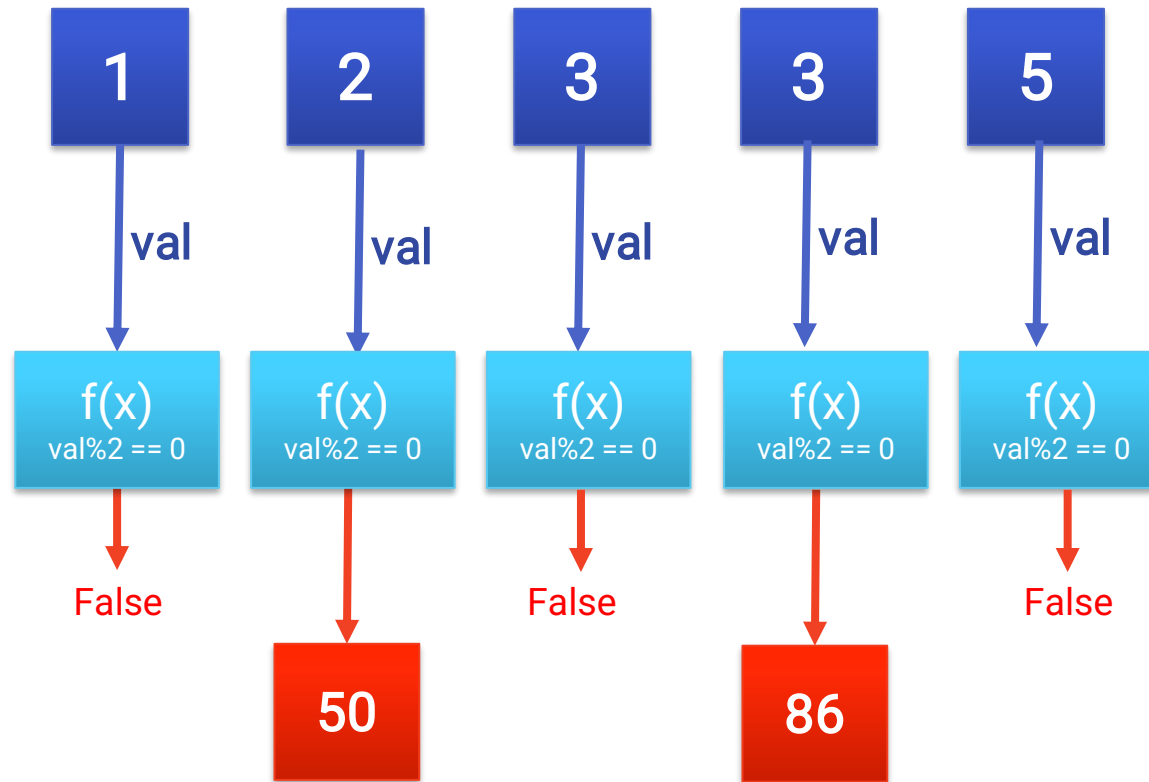
```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def es_par(numero):
    return numero % 2 == 0

numeros_pares = list(filter(es_par, numeros))

print(numeros_pares)

[2, 4, 6, 8, 10]
```



## Función reduce()

La función **reduce()** sirve para reducir todos los elementos del iterable a un único valor. Aquí cambia la estructura de la función pasada, ya que debe tener dos parámetros, uno el elemento correspondiente y otro el valor acumulado desde el elemento anterior.

```
from functools import reduce
```

Observa que esta función no está integrada, sino que hay que importarla del módulo functools

```
lista = [ 3, 4, 5, 6, 7 ]
```

```
multiplicacion = reduce(lambda acc, val: acc * val, lista, 1)
```

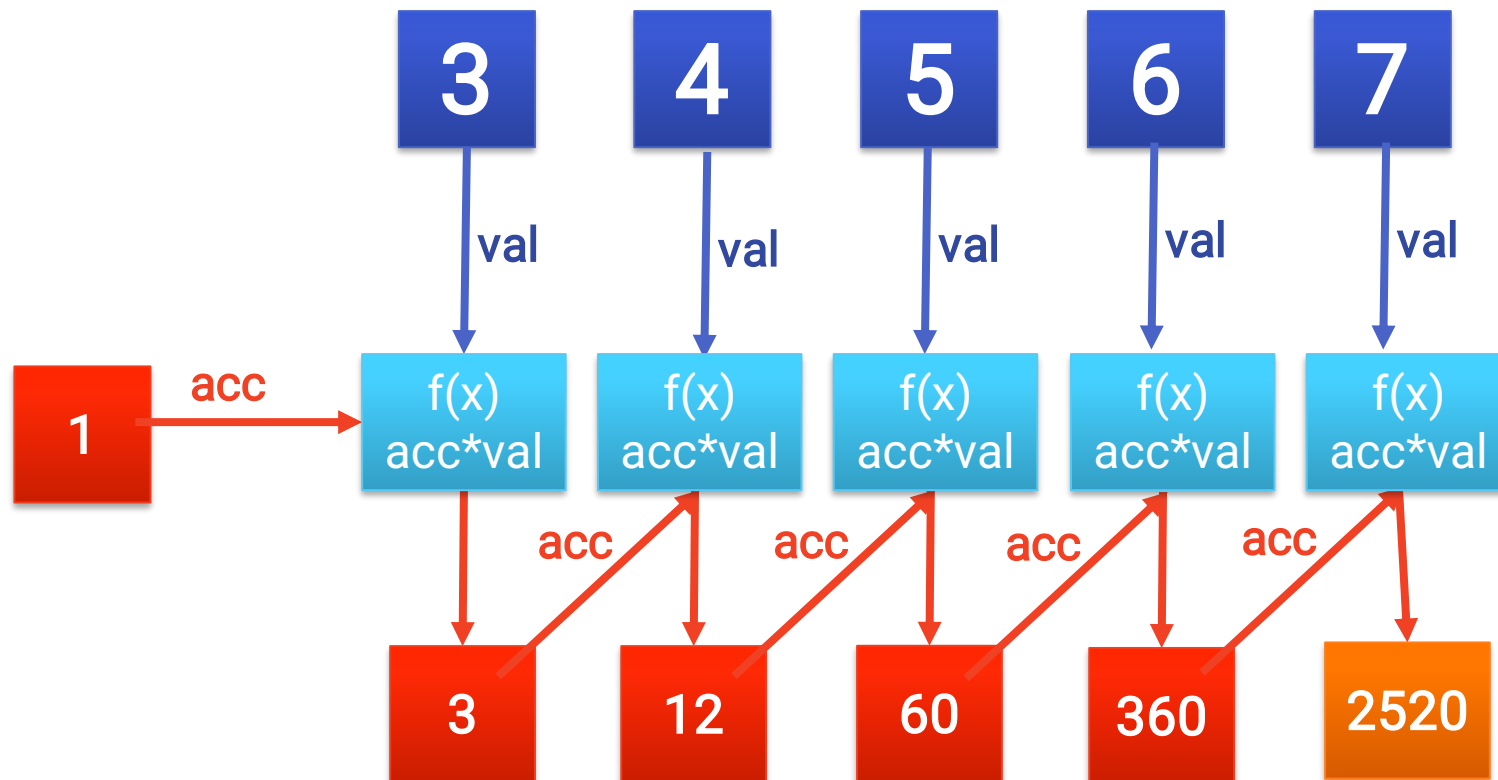
```
print(multiplicacion)
```

```
2520
```

Este es el valor que lleva acumulado en este elemento

Este es valor de cada uno de los elementos del iterable

Devuelve 3 valores: el resultado de la operación para cada elemento, el iterable y el valor del acumulador para el primer elemento



# 6

## TRABAJANDO CON FICHEROS EN PYTHON





Lo primero que debemos hacer al trabajar en Python con ficheros de texto es abrirlos con la función **open()**. Esta función devolverá un objeto de fichero que permitirá realizar las operaciones sobre el mismo.

Es importante recordar cerrar el fichero una vez hayamos finalizado para liberar recursos, para lo que utilizaremos el método **close()**

```
# Abrir y cerrar un fichero
fichero = open("mi_fichero.txt", "r") # "r" indica modo lectura
# Realizar operaciones con el fichero aquí
fichero.close() # Cerrarlo al finalizar
```

Alternativamente, podemos usar la palabra clave **with** para que el fichero se cierre automáticamente al finalizar el bloque de código dentro de dicha sentencia:

```
with open("mi_fichero.txt", "r") as fichero:
    # Operaciones de lectura o escritura
```

Al abrir el fichero, hay que indicar el modo en que lo abrimos, lo cual viene indicado por uno de los siguientes caracteres:

- **"r"**: Lectura. Da error si el fichero no existe.
- **"w"**: Escritura. Crea el fichero si no existe, o lo sobrescribe si existe.
- **"a"**: Escritura en modo adjuntar. Agrega datos al final del fichero.
- **"r+"**: Lectura y escritura sin sobrescribir. Da error si el fichero no existe.

## Lectura de ficheros

Existen varias formas de leer el contenido de un fichero

Leer todo el contenido del fichero

```
with open("mi_fichero.txt", "r") as fichero:  
    contenido = fichero.read()  
    print(contenido)
```

Hola, este es un nuevo contenido.  
Añadiendo otra línea.

## Leer por líneas

```
with open("mi_fichero.txt", "r") as fichero:
    lineas = fichero.readlines() # Devuelve una lista con todas las líneas
    for linea in lineas:
        print(linea.strip()) # Elimina saltos de línea
```

Hola, este es un nuevo contenido.  
Añadiendo otra línea.

## Leer línea por línea (iterando)

```
with open("mi_fichero.txt", "r") as fichero:
    for linea in fichero:
        print(linea.strip())
```

Hola, este es un nuevo contenido.  
Añadiendo otra línea.

## Escritura en ficheros

Para escribir en un fichero, se deben utilizar los modos **w** o **a**. Al abrir en modo **w**, se sobrescribe el contenido, en modo **a**, se añade al final

Nuevamente, tenemos varias opciones para escribir en un fichero

### Escribir texto

```
with open("mi_fichero.txt", "w") as fichero:  
    fichero.write("Hola, este es un nuevo contenido.\n")  
    fichero.write("Añadiendo otra línea.\n")
```

## Escribir una lista de líneas

```
lineas = ["Línea 1\n", "Línea 2\n", "Línea 3\n"]

with open("mi_fichero.txt", "w") as fichero:
    fichero.writelines(lineas) # Escribe todas las líneas de la lista
```

Cuando trabajamos con ficheros, es buena idea controlar las **excepciones** ante errores inesperados.

```
try:
    with open("mi_fichero.txt", "r") as fichero:
        contenido = fichero.read()
        print(contenido)
except FileNotFoundError:
    print("El fichero no existe.")
except IOError:
    print("Error al leer o escribir en el fichero.")
```

Hola, este es un nuevo contenido.  
Añadiendo otra línea.

# 7

## MÓDULOS

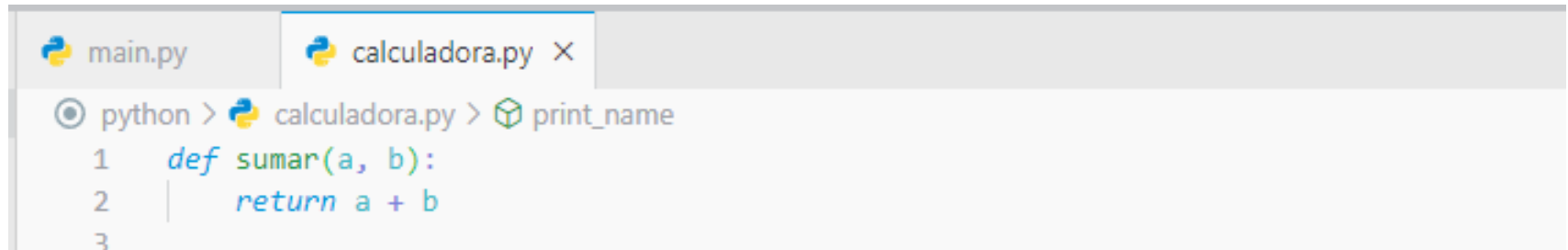




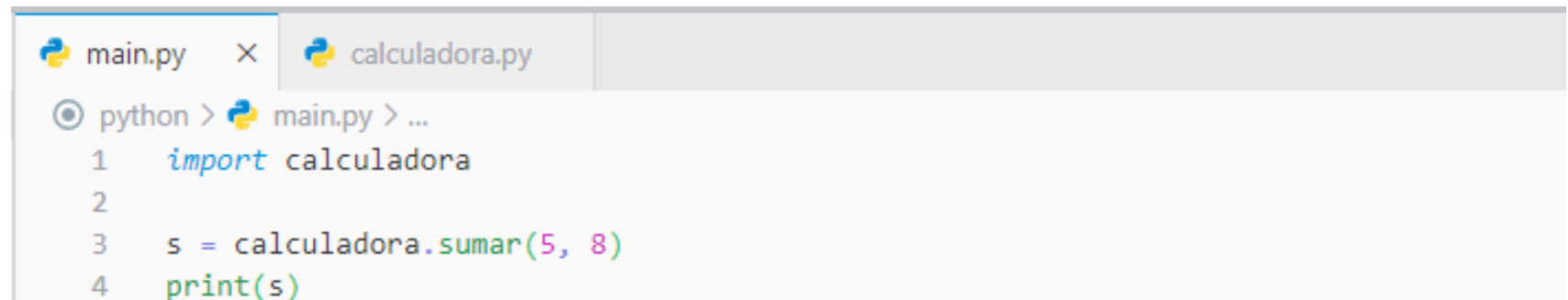
Un **módulo** es simplemente un archivo Python que contiene una serie de definiciones que pueden ser importadas a otros módulos.

El nombre del archivo es el nombre del módulo con el sufijo .py agregado.

Dentro del módulo se puede acceder al propio nombre del módulo mediante la variable global **\_\_name\_\_**.



```
python > calculadora.py > print_name
1 def sumar(a, b):
2     return a + b
3
```



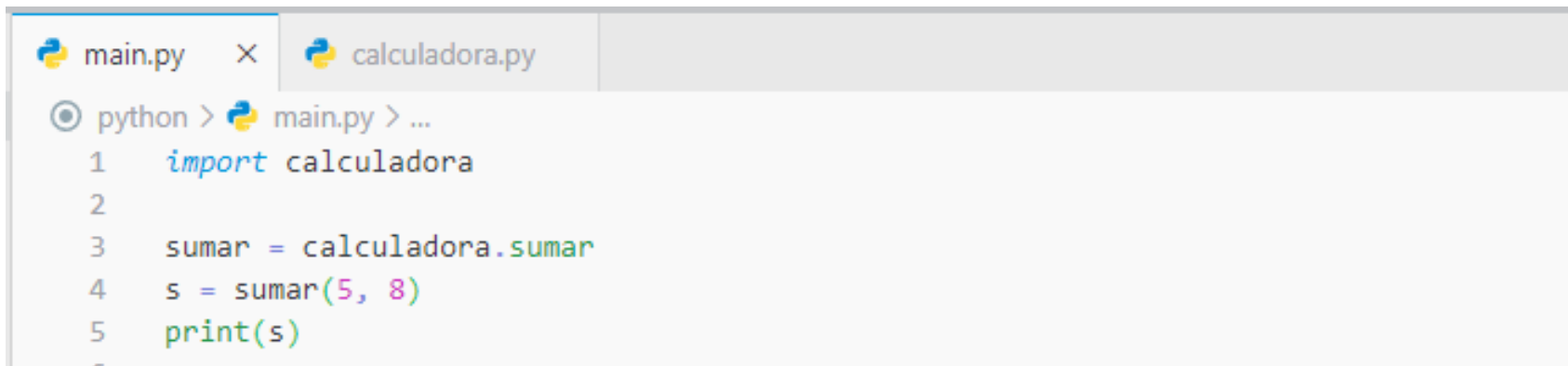
```
python > main.py > ...
1 import calculadora
2
3 s = calculadora.sumar(5, 8)
4 print(s)
```

Para usar las declaraciones de un módulo debemos utilizar la cláusula **import** *<nombre\_módulo>* en el módulo al que las queremos importar.

En el código anterior hay un módulo llamado *calculadora* que tiene operaciones matemáticas y lo importamos en el fichero *main.py*.

Observa que importar el módulo no añade los nombres de las funciones al **namespace** (espacio de nombres), solo el nombre del módulo, por ello, cuando queremos usar una función hay que indicar a qué módulo pertenece de la forma **calculadora.sumar()**

Si lo usáramos frecuentemente podríamos asignarles a un nombre local,



```
python > main.py > ...
1  import calculadora
2
3  sumar = calculadora.sumar
4  s = sumar(5, 8)
5  print(s)
```

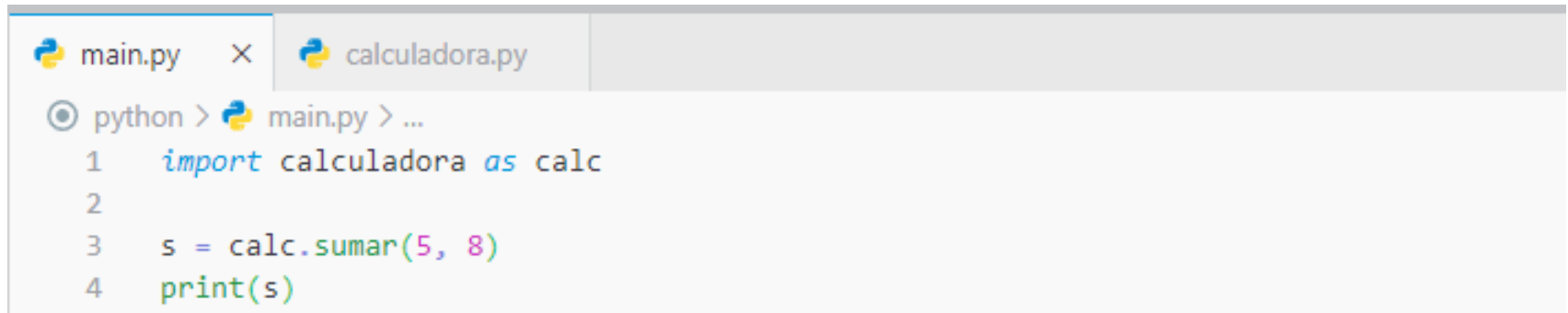
Una forma alternativa que permite importar los nombres de las funciones directamente al espacio de nombres del módulo que hace la importación es usando **from <mod> import <f1>,<f2>**



```
python > main.py > ...
1  from calculadora import sumar, restar
2
3  s = sumar(5, 8)
4  print(s)
```

También es posible importar todas las declaraciones que haya en el módulo mediante el símbolo asterisco, pero es altamente desaconsejado porque **genera código poco legible**, por lo que solo estaría recomendado en sesiones interactivas.

Por último, también podemos utilizar la palabra clave **as** en las importaciones, lo que nos permite renombrar tanto el módulo como las funciones que importemos.



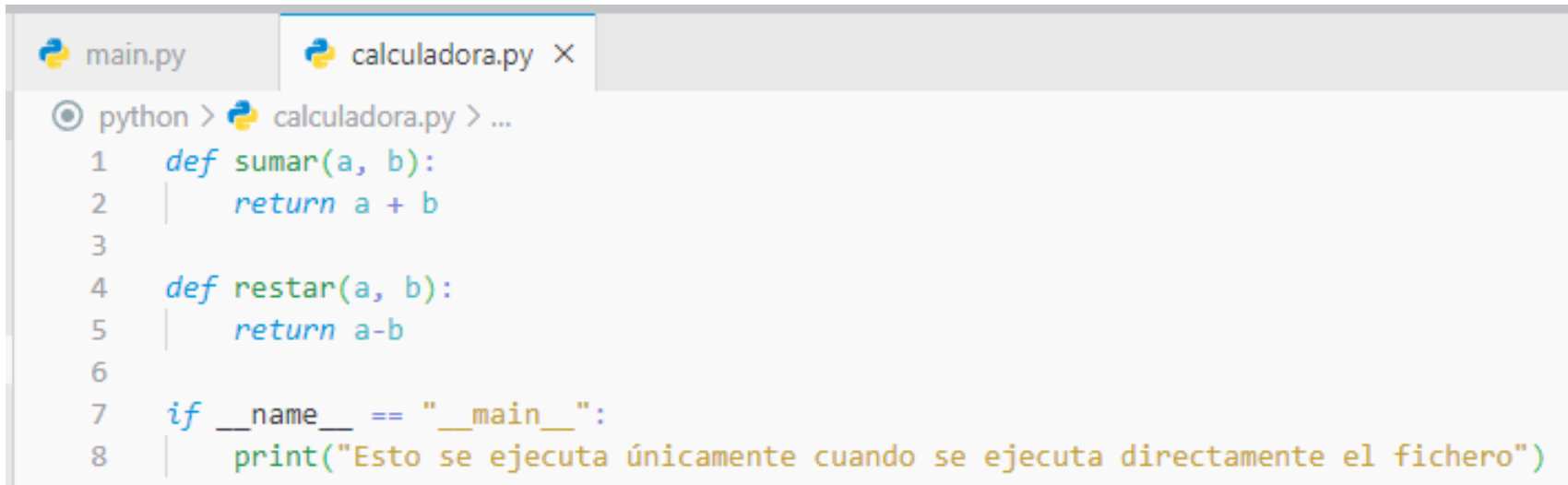
```
python > python main.py > ...
1  import calculadora as calc
2
3  s = calc.sumar(5, 8)
4  print(s)
```



```
python > python main.py
1  from calculadora import suma as s, resta as r
2
3  s = s(5, 8)
4  print(s)
```

Si hay código fuera de funciones en un módulo se ejecutará la primera vez que se importe el módulo.

Si queremos poner código en un módulo que solo se ejecute cuando ejecutemos directamente el archivo, pero no cuando se importe, podemos comprobar el valor de la variable `__name__`, que contendrá el nombre del módulo si ha sido importado, y el valor `__main__` si el módulo ha sido ejecutado directamente desde el intérprete.



```
python > calculadora.py > ...
1  def sumar(a, b):
2      return a + b
3
4  def restar(a, b):
5      return a-b
6
7  if __name__ == "__main__":
8      print("Esto se ejecuta únicamente cuando se ejecuta directamente el fichero")
```