

# Python para principiantes

Nivelación



# Python para principiantes



A través de la lectura de este fastbook adquirirás los conocimientos básicos para empezar a programar con Python. Tras leer esta premisa, es posible que esperes encontrarte entonces un manual teórico sobre este lenguaje de programación. Nada más lejos de mi intención. Por mi experiencia te diré, que solo se aprende a programar programando, por eso, en estas páginas me centraré en enseñarte los comandos que más se usan en la práctica y cómo utilizarlos.

Los cuatro **objetivos** que me marco (y te marco) para tu aprendizaje son los siguientes:

- Las principales características de Python.
- Cómo manejar variables y estructuras de datos.
- Cómo crear flujos de ejecución.
- Cómo crear funciones.

*Autor: Ángel Delgado*

Introducción a Python

Tipos básicos

Estructuras de datos

Estructuras de control de flujo

Funciones

Resumen

# Introducción a Python



Qualentum Lab

Las principales **propiedades** que definen a Python como lenguaje de programación son estas dos:

1

**Es un lenguaje de programación de alto nivel.** Esto quiere decir que busca la sencillez frente a la eficiencia. De ahí que Python sea considerado uno de los lenguajes más sencillos y legibles, motivos también por los que se ha convertido en un lenguaje tan popular.

2

**Su paradigma de programación está orientado a objetos.** El concepto ‘paradigma’ hace referencia al estilo con el que está pensado para ser usado. En el caso de Python, la programación está orientada a objetos y busca que el código esté estructurado de manera que sea fácilmente *reutilizable*.

Más allá de su filosofía a la hora de definir el lenguaje, en lo relativo a su ejecución, Python destaca por estas cuatro **características de su intérprete**:



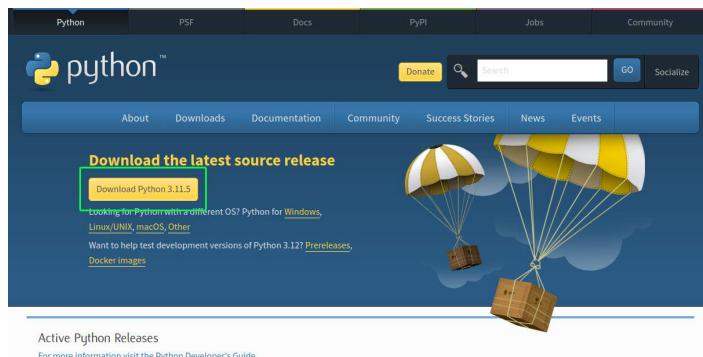
**Lenguaje de scripting.** Esto quiere decir que el intérprete de Python no requiere compilar el código y posteriormente ejecutarlo, sino que directamente se ejecuta el código a partir de un script de Python. Esto hace que las aplicaciones en Python sean más fácilmente portables entre sistemas.

- Monoproceso.** Python no está pensado para hacer procesamiento en paralelo, y, aunque sí permite la programación concurrente, tiene la característica de que solo puede haber un proceso usando Python en cada momento. A dicha característica de Python se le llama *GIL*.
- Tipado dinámico.** Los *tipos* sirven para definir las características de cada variable en términos de memoria y procesamiento. El tipado dinámico hace que las variables puedan cambiar su tipo durante la ejecución, lo cual hace que Python sea menos eficiente pero más sencillo de usar.
- Interfaz a aplicaciones en C.** Otro de los motivos de su popularidad es que, aunque es poco eficiente, Python es capaz de usar librerías creadas en C (que es un lenguaje de programación muy eficiente).

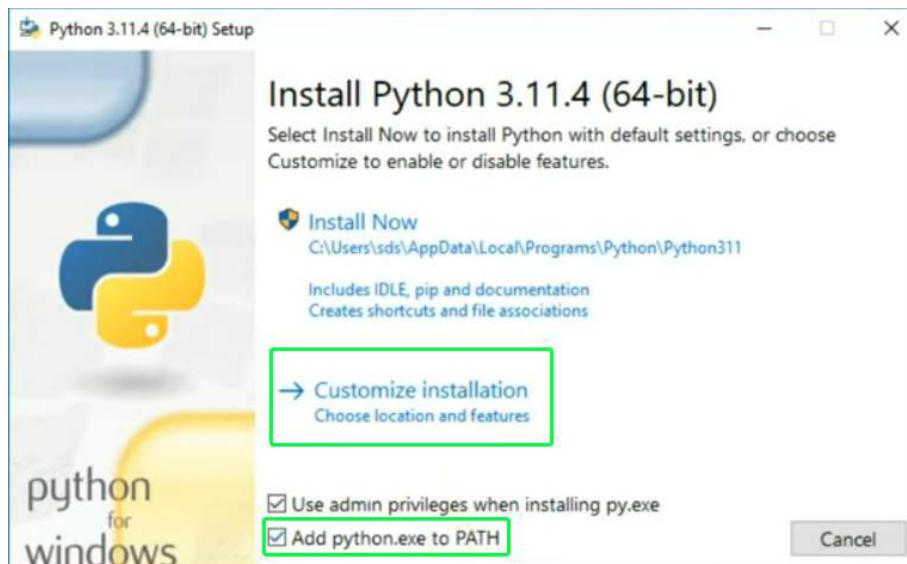
Ahora estudiemos cómo es la instalación de recursos.

- **Instalación de Python**

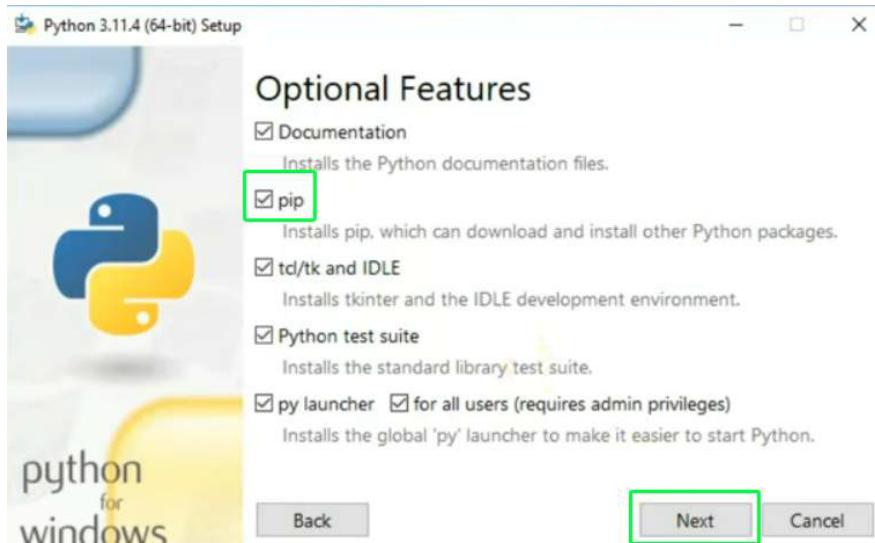
Python se puede descargar de manera gratuita desde la página web oficial <https://www.python.org/downloads/>. Solo es necesario elegir la versión y el sistema operativo. En nuestro caso, elegiremos Windows y la última versión estable actualmente, la 3.11. Clicando en el botón de descarga se iniciará la descarga del instalador.



Después de haber acabado la descarga, clicando en el instalador se abrirá la siguiente ventana. En Windows es importante **seleccionar la opción de añadir python.exe al path**. Una vez elegido, seleccionamos la instalación personalizada.

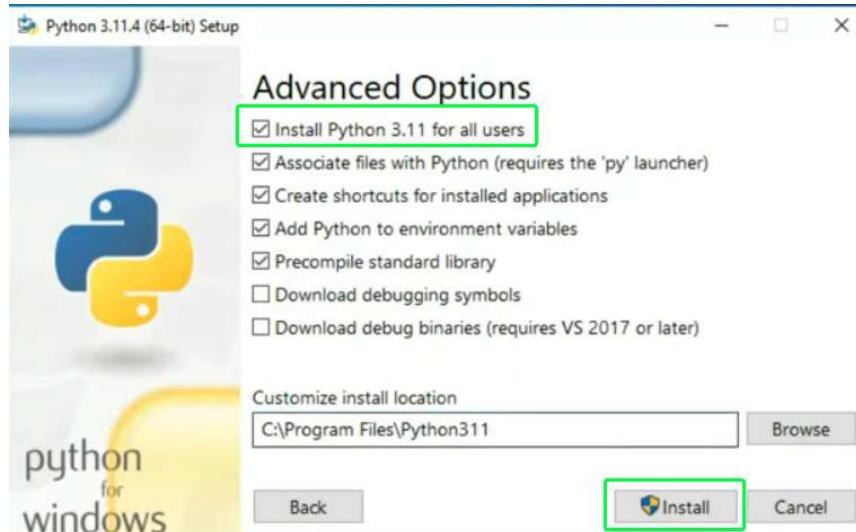


En las características adicionales, seleccionamos la opción de instalar pip. **Pip es el gestor de librerías de Python** que nos va a permitir en próximos cursos descargar librerías. Después de seleccionarlo, le damos a siguiente.



Finalmente, seleccionamos que se instale para todos los usuarios e iniciamos el proceso de instalación. Con esto ya estaría instalado Python.

Para comprobarlo basta con buscar ‘Python’ en el buscador de programas y/o en una terminal de CMD.

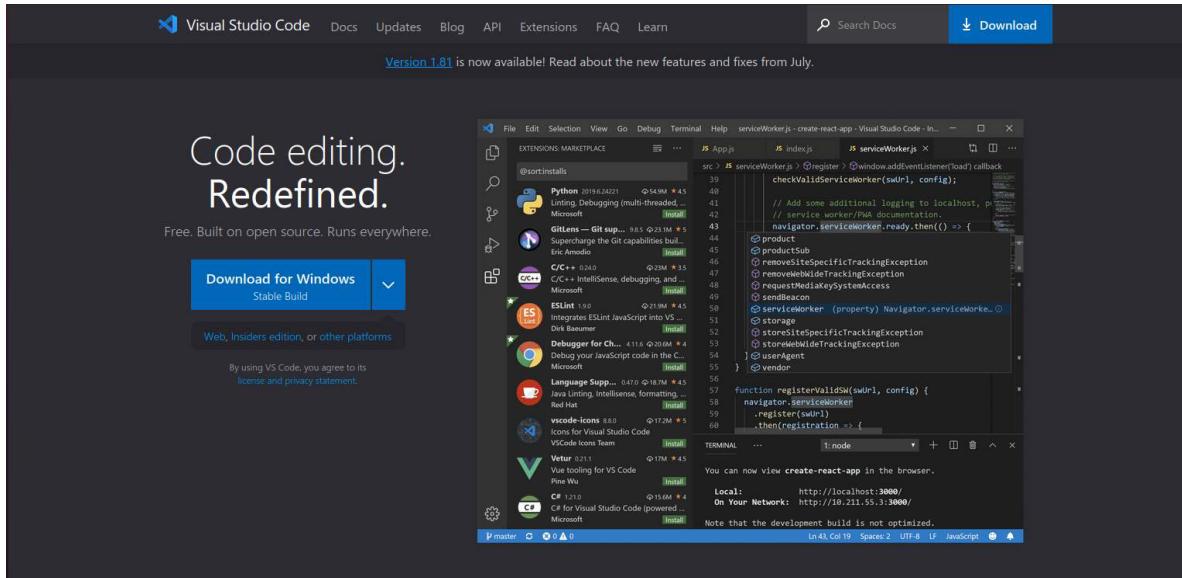


### • Instalación del IDE

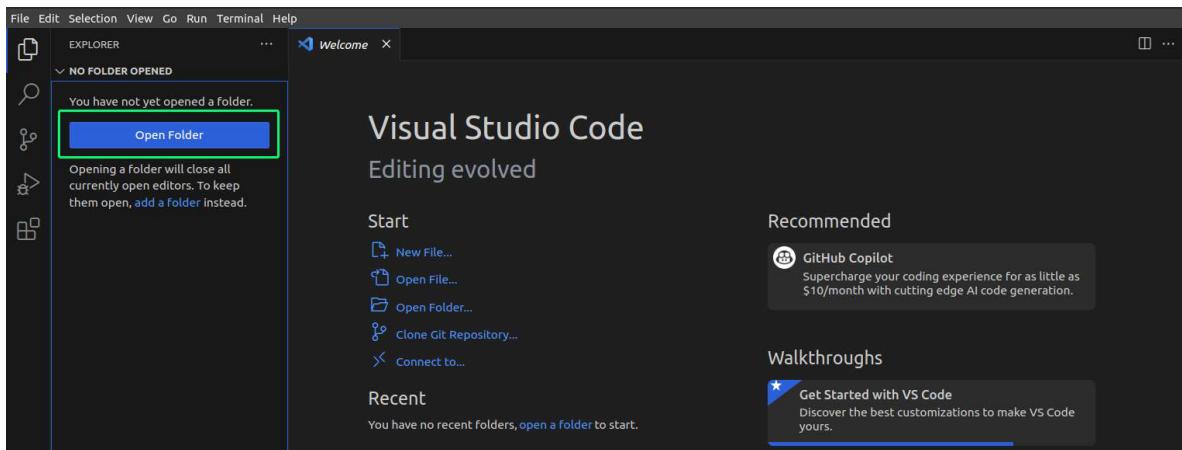
El IDE (*integrated development environment*) es como se denomina a los programas que se utilizan para programar. Diferentes lenguajes suelen usar diferentes IDE, por ejemplo, el más popular en Python es **Pycharm**.

Sin embargo, nosotros vamos a optar por **Vscode**, ya que es un IDE de propósito general y que funciona bastante bien con Python.

Podemos descargar gratis **Vscode** desde la página oficial <https://code.visualstudio.com/>



Al pulsar el botón de descarga se descargará un instalador. Una vez se haya descargado, clicando sobre él podemos iniciar el proceso de instalación. En este caso, podemos seguir la instalación por defecto. Una vez que se haya instalado, al abrirlo veremos una ventana como la que muestra la imagen.



Si hacemos clic en el botón indicado, podemos abrir una carpeta para empezar a trabajar (si no tenemos ninguna, podemos crearla). Una vez la hemos seleccionado, volvemos a la ventana principal de Vscode.

Para empezar a trabajar con Python, tenemos que instalar la extensión de Python de Vscode. Para ello, clicamos en el ícono de la izquierda de las extensiones. Escribimos el nombre de la extensión (vale con escribir Python) y, finalmente, clicamos en el botón de instalar.



Con todo esto ya tendríamos configurado Python para su uso.



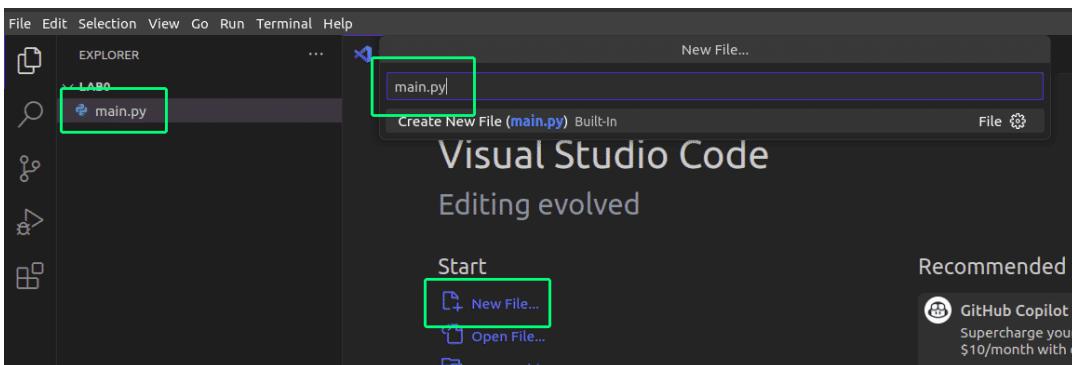
En caso de duda se puede ver un tutorial con más detalle de cómo configurar Vscode para Python [aquí](#).

## Hello world

Una vez que tenemos instalado Python y hemos entendido sus características básicas, en esta sección vamos a ver cómo ejecutar “hello world”.

### Creamos un script

Para ello tenemos que crear un script de Python, el cual debe tener la extensión .py (por ejemplo, main.py).



### Importar librerías

Las librerías nos permiten añadir funcionalidad a Python. Python viene con las librerías de la librería standard de Python, no obstante, **podemos instalar más con pip**. Para poder usar una librería tenemos que usar el comando *import*.

A screenshot of the Visual Studio Code interface showing a Python script named "main.py". The code in the editor is:

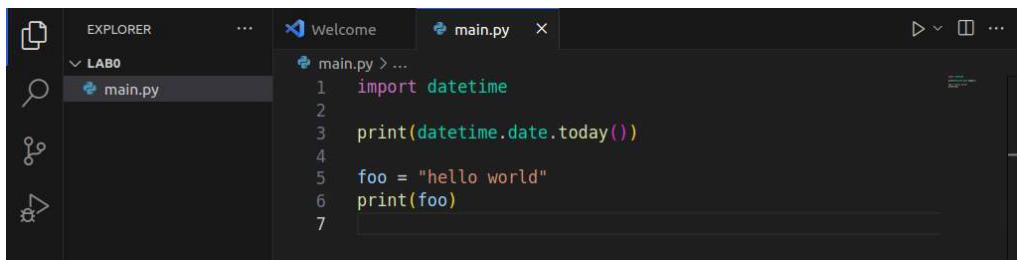
```
1 import datetime
2
3 print(datetime.date.today())
```

A green box highlights the "main.py" file in the Explorer sidebar and the code in the editor.

En nuestro caso hemos importado la librería `datetime`, que nos va a permitir ver la fecha usando la función `.date.now()` y la función `print` para mostrarlo por pantalla.

## Asignación de variables

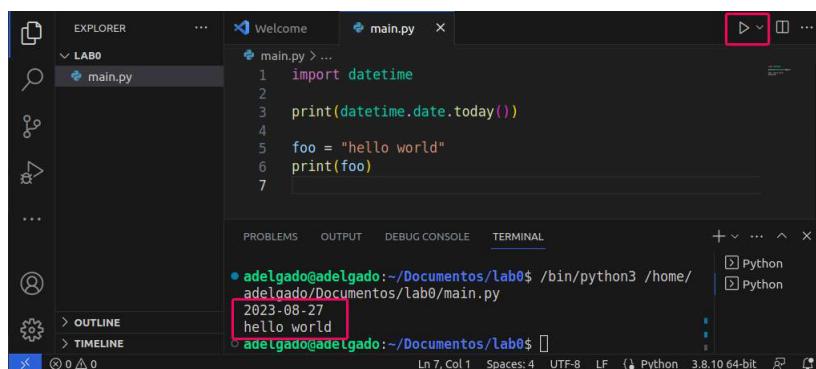
Dentro del script `main.py` vamos a crear una variable con el texto “hello world”. En Python la asignación de variables se hace usando el símbolo de igual (`=`). Vamos a asignar este texto a una variable llamada `foo` y a mostrar su valor usando el comando `print`.



```
import datetime
print(datetime.date.today())
foo = "hello world"
print(foo)
```

## Ejecución del script

Para ejecutar el script basta con usar el botón de Vscode para ejecutarlo. Podemos ver en la imagen que en la parte de abajo de Vscode se abre un terminal con los resultados de la ejecución.



```
import datetime
print(datetime.date.today())
foo = "hello world"
print(foo)
```

Terminal output:

```
adelgado@adelgado:~/Documentos/lab0$ /bin/python3 /home/adelgado/Documentos/lab0/main.py
2023-08-27
hello world
adelgado@adelgado:~/Documentos/lab0$
```

Como ya podemos intuir, Python permite ejecutar directamente los scripts de Python sin tener que compilar el código.

# Tipos básicos



Qualentum Lab

En este apartado, estudiaremos las **variables** (numéricas, *strings* y booleanas) y los **operadores** de uso frecuente para que puedas ir familiarizándote con ellos.

## Variables de carácter numérico

Las variables numéricas sirven para representar datos de tipo numérico. Es común que en los lenguajes de programación existan diferentes tipos de variables numéricas según la precisión y el número de bites con el que se quiera almacenar. Sin embargo, en Python **existen únicamente dos tipos de valores numéricos**: los enteros (*int*) y los decimales (*float*).

```
>>> type(6)
<class 'int'>
```

```
>>> type(3.14)
<class 'float'>
```

- **Operadores**

Las operaciones matemáticas básicas de Python son las siguientes:

Operación	Sintaxis	Ejemplo	Resultado
Suma	+	5 + 2	7
Resta	-	5 - 2	3
Producto	*	5 * 2	10
División	/	5 / 2	2.5
Exponente	**	5 ** 2	25
Resto	%	5 % 2	1
División entera	//	5 // 2	2

Además de estas operaciones, la librería standard de Python posee la librería *math*. Esta librería contiene muchas más funciones comunes de aritmética. Para poder usar la librería *math*, es necesario importarla usando la sentencia *import*. Veámoslo con este ejemplo:

```
>>> import math
>>> math.cos(1)
0.5403023058681398
```



Encuentra aquí el [recurso](#).

Para aprender más sobre las funcionalidades de la librería Standard de Python, puedes consultarla en su [página web](#).

## String

Este tipo de variable sirve para representar **caracteres y cadenas de caracteres**. A diferencia de otros lenguajes, Python no diferencia entre ambos. Las cadenas de caracteres se crean usando las comillas dobles o simples.

```
>>> type("esto es un string en python")
<class 'str'>

>>> foo = """con triple comillas
>>>   podemos crear
>>>   strings multilínea"""
>>>
```

- **Operadores**

Algunas de las funciones más comunes a la hora de trabajar con strings son las siguientes:

Operación	Sintaxis	Ejemplo	Resultado
concatenar	+	"aaaa" + "bbb"	"aaaabbb"
substring	[ : ]	"0123456"[2:5]	"2345"
sustituir	.replace()	"aabb".replace("a","c")	"ccbba"
dividir	.split()	"aaa-bbb-ccc".split("-")	["aaa", "bbb", "ccc"]
upper	.upper()	"aaa".upper()	"AAA"

Además de las funciones básicas que vienen precargadas con el intérprete, es común también usar las **librerías re** para expresiones regulares y la **string** para *encoding* (codificación) y puntuación.

- **f-strings**

Los f-string es una funcionalidad de Python que permite **inyectar variables** (o incluso el resultado de la ejecución de una sentencia) dentro de un string de Python. Para ello se añade la letra *f* al principio del string y el código que se quiera meter dentro del string se pone entre corchetes, "{ }".

```
>>> a, b = 3, 2 # Esto es una asignación doble, en python está permitido  
>>> print(f"El valor de a es {a} y el valor de b es {b}. Su suma es {a+b}")
```

El valor de a es 3 y el valor de b es 2. Su suma es 5

Los f-string son muy comunes en Python para hacer print de errores y monitorizar la aplicación, ya que permiten injectar fácilmente variables dentro de un texto. No obstante, hay que tener cuidado con ellas ya que son propensas a fallo.



**Si quieres investigar más sobre las cadenas de caracteres haz clic en [El libro de Python](#).**

## Bool

Este tipo de variables sirven para almacenar valores lógicos. En Python los valores lógicos se escriben con las palabras reservadas “True” y “False”.

- **Operadores**

Los operadores lógicos permiten hacer comparación o combinación de variables lógicas. Este tipo de operadores siguen las reglas lógicas convencionales.

Operación	Sintaxis	Ejemplo	Resultado
AND	and	True and False	False
OR	or	True or False	True
NOT	not	not True	False

Más allá de los operadores lógicos, existe una serie de operadores que actúan sobre otro tipo de variables, cuyo resultado es un valor lógico. Aquí te muestro algunos ejemplos:

Operación	Sintaxis	Ejemplo	Resultado
igualdad	==	5==2	False
identidad	is	5 is 2	False
mayor/menor	> / <	5 > 2	True
mayor/menor o igual	>= / <=	5 >= 2	True



Puedes consultar más información sobre las variables booleanas [aquí](#).

# Estructuras de datos



Para abordar cómo Python trabaja con las grandes bases de datos, primero debemos aprender qué son las listas y los diccionarios en este programa.

## Listas

Una lista de Python es una forma de almacenar un conjunto de elementos de manera ordenada. Las listas en Python se definen con corchetes “[ ]” y separando sus elementos por comas.

```
>>> lista = [1, 2, 3]
```

A diferencia de otros tipos similares en otros lenguajes de programación (como el array de C), las listas de Python cuentan con las siguientes características:

1

**Tipado flexible:** no tienen por qué contener elementos de un mismo tipo, pueden mezclar elementos de tipo *int* con elementos de tipo *string*.

2

**Tamaño dinámico:** tampoco es obligatorio que tengan espacio y/o tamaño predefinido. De hecho, puede cambiar su tamaño durante la ejecución del programa.

- **Operadores**

Operación	Sintaxis	Ejemplo	Resultado
concatenación	+	[1, 2, 3] + [4, 5]	[1, 2, 3, 4, 5]
consultar elemento	[ ]	[0, 11, 22][1]	11 *
cambiar elemento	[ ] =	[0, 11, 22][1] = 33	[0, 33, 22]
añadir elemento	.append()	[0, 1, 2].append(3)	[0, 1, 2, 3]
eliminar elemento	.pop()	[0, 11, 22].pop(1)	[0, 22] **
sublista	[ : ]	[0, 1, 2, 3, 4][1:4]	[1, 2, 3]
comprobar si está	in	2 in [1, 2, 3]	True

Echa un vistazo de nuevo a la tabla, a las filas destacadas en color, porque es necesario que recuerdes algunas consideraciones especiales.

- **Consideraciones**

**(\*) Numeración en Python: se empieza por cero**

Al consultar un elemento usando un índice, vemos que el índice "1" no se corresponde al primer elemento sino al segundo. Esto es porque, en Python, el convenio es que la numeración siempre empiece por 0, por tanto, el primer elemento es 0.

**(\*\*) Modificación in place**

La función `pop()`, realmente modifica la lista '*in place*'. Esto quiere decir que no devuelve la nueva lista sin el elemento, sino solamente el elemento eliminado, mientras que la variable con la lista se queda modificada.

```
>>> lista = [1,2,3]
>>> elemento = lista.pop(1)
>>> print(elemento)
2
>>> print(lista)
[1,3]
```

Este tipo de comportamiento, en el cual la función modifica la lista, sin devolver el resultado de la modificación, es muy común en funciones que operan sobre listas y otras estructuras de datos.



Dispones de más información [aquí](#).

## Diccionarios

Al igual que las listas, los diccionarios son otra estructura de datos, pero que se diferencian en que, en lugar de almacenar los datos de forma ordenada, los almacenan en **formato clave-valor**. Los diccionarios se definen usando **llaves**, **{ }**, **los dos puntos** para separar el par clave-valor y la **coma** para separar elementos. Echa un vistazo al ejemplo:

```
>>> diccionario = {"a": 1, "b": 2, "c": 3}
```

Las características de los diccionarios de Python son las siguientes:

- Tipado flexible:** los tipos tanto de los valores usados como clave y como valor pueden tener diferentes tipos dentro de un mismo diccionario. No tienen que compartir un mismo tipo.
- Claves hashables:** los tipos de los valores usados como clave pueden ser de cualquier tipo, siempre y cuando su tipo sea **hashable**. Esto quiere decir que deben tener un espacio de memoria asignado concreto. Por ejemplo, los datos de tipo *bool*, *int* y *str* son *hashables*, mientras que las listas no lo son.

```
>>> {"a": 12, True: 3.14, 3: "b"}  
{'a': 12, True: 3.14, 3: 'b'}  
  
>>> {1: 1, 2:2, [3]: 3}  
TypeError: unhashable type: 'list'
```

- **Operadores**

Estas son algunas de las principales operaciones de Python a la hora de operar con diccionarios:

Operación	Sintaxis	Ejemplo	Resultado
consultar valor	[ ]	{“a”:1, “b”:2} [“b”]	2
insertar valor	[ ] =	{“a”: 1} [“b”] = 2	{“a”:1, “b”:2}
eliminar valor	.pop()	{“a”:1, “b”:2}.pop(“b”)	{“a” : 1} *
consultar claves	.keys()	{“a”:1, “b”:2}.keys()	dict_keys([‘a’, ‘b’])
consultar valores	.values()	{"a":1, "b":2}.values()	dict_values([1, 2])
concatenación	.update()	{1:1, 2:2}.update({2:22,3:3})	{1:1, 2:22, 3:3}

**(\*) Modificación *in place***

La función `pop()`, realmente modifica el diccionario *in place*. Esto quiere decir que no devuelve la nueva lista sin el elemento, sino solamente el elemento eliminado, mientras que la variable con la lista se queda modificada.



Dispones de más información en [este enlace](#).

## Otras estructuras de datos

En la mayoría de los proyectos en Python las estructuras de datos más comunes son las listas y los diccionarios, pero además de estas existen otras estructuras. Te presento dos que debes conocer y su uso:

1

**Sets:** este tipo de estructuras de datos sirven para almacenar un conjunto de elementos, pero sin que tengan ninguna ordenación y sin que permitan los duplicados.

2

**Tuplas:** similares a las listas, salvo que las tuplas son inmutables (por ejemplo, no se pueden cambiar los elementos que la componen).

---

**Recuerda: más allá de las estructuras de datos que vienen precargadas en Python, existen otras que se pueden usar cargando la librería Standard Collections.**

# Estructuras de control de flujo



A continuación, vamos a revisar lo que se denomina ‘control condicional’. Estas estructuras nos permiten valorar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar. También estudiaremos qué son los bucles.

## If – else

Las estructuras de flujo de tipo *If* permiten evaluar si se ejecuta una sentencia de código u otra en base a una condición lógica que se haya definido. Para definirse **se usa la palabra reservada if, seguida de la condición lógica y de dos puntos**. La sentencia para ejecutar, si se cumple la condición, se escribe en un salto de línea e *indentado* (con 4 espacios).

```
if <condición>:  
    <sentencia a ejecutar>
```

Aquí un ejemplo para entenderlo mejor:

```
if foo == 1:      # Comprobamos que foo tiene el valor 1
    print("foo = 1") # Si lo anterior es True, se ejecuta este código
```

Hay veces que si la condición del *if* es *False*, queremos seguir evaluando otras condiciones a continuación. Esto se puede hacer usando el comando *elif* de manera análoga al *if*.

```
if foo == 1:
    print("foo = 1")

elif foo == 2:      # Si foo no es 1, comprobamos que si es 2
    print("foo = 2") # Si es 2, ejecutamos el siguiente fragmento de código
```

Otras veces, cuando definimos un flujo de sentencias *if* y *elif*, queremos que, si no se cumple ninguna de las condiciones, exista una condición que se ejecuta por defecto. Esto se puede hacer usando el comando *else*.

```
if foo == 1:
    print("foo = 1")

elif foo == 2:      # Si foo no es 1, comprobamos que si es 2
    print("foo = 2") # Si es 2, ejecutamos el siguiente fragmento de código

else:              # Si foo no cumple ninguna de las anteriores
    print("foo is not 1 or 2")
```



Amplía información sobre la función *if* [aquí](#).

- **Consideraciones y características**

Es importante que resaltemos algunas de las características y consideraciones importantes a la hora de hacer una estructura de control *if*. ¡Toma nota!

1

### **Los operadores lógicos**

Se pueden construir sentencias lógicas para los comandos *if* y *elif* usando operadores lógicos:

```
>>> if (foo>1 and foo<=3):  
>>>     print("foo must be 3 or 2")
```

2

### **If anidados**

Es muy común que además de evaluar una condición, quieras comprobar después si se cumple otra condición antes de ejecutar una sentencia. Esto se puede hacer usando *if* anidados dentro de otros, por ejemplo:

```
>>> if type(foo)==int:  
>>>     if foo: <= 3:  
>>>         print("foo is an integer lower than 3")
```

## 3

**Tipado dinámico de Python**

Los comandos *if*, *else* y *elif* necesitan evaluar una condición lógica cuando se ejecutan. Sin embargo, recordemos que Python tiene un tipado dinámico, lo cual quiere decir que una condición que no devuelva un resultado de tipo *bool*, no da error, sino que intenta convertir el resultado a un tipo *bool*.

```
>>> if "":          # Un string vacío se evalúa como False
>>>   print("False")

>>> elif "hola":    # Un string no vacío se evalúa como True
>>>   print("True")

>>> if 0:           # El valor numérico 0 se evalúa como False
>>>   print("False")
>>> elif 34:         # Pero cualquier otro se evalúa como True
>>>   print("True")

>>> if []:          # Una lista vacía se evalúa como False
>>>   print("False")

>>> elif [2,3]:     # Una lista no vacía se evalúa como True
>>>   print("True")
```

**Bucles for**

Los *bucles for* sirven para recorrer un conjunto de elementos de una estructura de datos iterable (por ejemplo, las estructuras de datos formadas por elementos, como las listas). De modo que por cada elemento se ejecutará una sentencia de código.

A nivel de sintaxis se definen con el comando `for`, seguido de un iterable y dos puntos. El *dejado e indentado* con 4 espacios se añade a la sentencia al ejecutar.

```
>>> for <element> in <iterable>:  
>>>   <sentencia de código>
```

Un ejemplo de un bucle `for` es el siguiente:

```
# Para cada elemento de la lista lo asignamos a la variable "i"  
>>> for i in [1,2,3]:  
>>>   print(f"Elemento: {i}") # Por cada iteración se ejecuta este código  
  
"Elemento: 1"  
"Elemento: 2"  
"Elemento: 3"
```

Como vemos, para cada elemento de la lista se hace una asignación de ese elemento a la **variable *i***. Luego, se ejecuta el código que hay debajo del `for`, dentro del bloque indentado.



Aprende más sobre el comando `for` [aquí](#).

- **Consideraciones y características**

Ahora vamos a ver, algunos condicionantes que debemos tener muy en cuenta a la hora de construir un bucle *for*.

### **¿Y si no queremos recorrer un iterable sino ejecutar un código N veces?**

En otros lenguajes es común que haya una forma de ejecutar un bucle *for* n-veces sin necesidad de recorrer un iterable. En Python no es así. La manera de hacer esto es usando la función *range*, que permite crear un iterable de N elementos dado un valor N.

```
>>> for i in range(100): # range me permite crear un iterable de 100 elementos  
>>> print(i)           # esto se ejecuta 100 veces  
  
1  
2  
...  
100
```

### **¿Se pueden recorrer dos iterables a la vez?**

Sí, esto se puede hacer usando la función *zip*. También puedes asignar cada elemento de la lista a una variable independiente.

```
>>> for elemento_lista_1, elemento_lista_2 in zip([1,2,3], [4,5,6]):  
>>> print(elemento_lista_1, elemento_lista_2)  
  
1 4  
2 5  
3 6
```

## ¿Cómo puedo recorrer los valores de un diccionario?

Los diccionarios, al ser un formato de clave-valor, si los recorres con un bucle `for` solo se recorren las *clave*. En el caso de que quieras también recorrer los valores, es necesario usar las funciones `values` o `items`.

```
>>> for clave in {"a": 1, "b": 2}:
>>>   print(clave)

a
b

>>> for valor in {"a": 1, "b": 2}.values():
>>>   print(valor)

1
2

>>> for clave, valor in {"a": 1, "b": 2}.items():
>>>   print(clave, valor)
a 1
b 2
```

## Bucles anidados

La función de los bucles como estructura de control es repetir la ejecución de un bloque de código múltiples veces y recorrer una estructura de datos. Sin embargo, hay veces que no es suficiente con un bucle y es necesario usar bucles anidados.

```
>>> dicc = {"a": [1,2, 3], "b": [11,22, 33]} #Queremos recorrer todos los elementos

>>> for clave, valor in dicc.items(): # Recorremos el diccionario
>>>   for elemento in valor:        # Para cada valor, recorremos la lista
>>>     print(clave, elemento)

a 1
a 2
a 3
b 11
b 22
b 33
```

- **List comprehension y dict comprehension**

Las *list comprehension* y *dict comprehension* es un tipo de construcción sintáctica que permite definir listas y diccionarios respectivamente sin tener que crearlos explícitamente a partir de un bucle *for*. Este tipo de sintaxis no es exclusivo de Python, pero sí es especialmente popular.

```
>>> [i for i in range(5)] # Ejemplo de list comprehension  
[0, 1, 2, 3, 4]  
  
>>> {i : i * 10 for i in range(5)} # Ejemplo de dict comprehension  
{0: 0, 1:10, 2:20, 3:30, 4:40}
```



Consulta [esta documentación](#) para saber más.

## Otras estructuras de control

Las sentencias *if* y *for* son las estructuras de control más usadas en la mayoría de los desarrollos. Sin embargo, también nos encontramos con otras estructuras de control. Te presento algunas de ellas.

1

## Switch

Es una estructura de control similar a una estructura *if*. Solo que, en lugar de evaluar condiciones lógicas, se ejecuta un código u otro en función del valor que tome una variable concreta definida en el *switch*.

A diferencia de otros lenguajes de programación, en Python no existe un comando específico para hacer una estructura *switch*, sin embargo, a partir de Python 3.10, la alternativa es el comando ***match***. Este comando es más poderoso que un *switch*.

```
>>> command = 'Hello, World!'

>>> match command:
...     case 'Hello, World!':
...         print('Hello to you too!')
...     case 'Goodbye, World!':
...         print('See you later')
...     case other:
...         print('No match found')

'Hello to you too!'
```



Amplía información [aquí](#).

2

## While

Al igual que un bucle *for*, repite la ejecución de una secuencia de código. La diferencia es que el bucle *for* lo repite para cada elemento de un iterable, mientras que el comando *while* lo repite de manera indefinida siempre que se cumpla una condición lógica.

```
>>> x = 3
>>> while x > 0:
>>>     x = x-1
>>>     print("Valor de x: ", x)
```

```
Valor de x: 3
Valor de x: 2
Valor de x: 1
```



Amplía información sobre el comando **while** [aquí](#).

3

### Try - Except

Esta estructura de control es similar a una estructura *if* (ejecuta un código u otro según una condición) solo que, en este caso, sirve para controlar errores en la ejecución del programa. Por ejemplo, supongamos que queremos escribir la división del número 10 entre una variable llamada 'value'.

```
>>> 10 / value
```

Aunque la sentencia es correcta, si el valor de *value* fuese 0 fallaría porque no se puede dividir un número entre 0. Del mismo modo, si *value* no tiene un valor numérico también fallaría. Si queremos que nuestro código no falle, sino que gestione ese error, podemos aplicar el comando **try-except** de la siguiente manera:

```
>>> try:  
>>>   10 / value  
>>> except ZeroDivisionError:  
>>>   print("Value no puede ser 0 porque si no da un error")  
>>> except TypeError:  
>>>   print("value tiene que ser de tipo int o float")
```



**Consulta información sobre el uso de `try-except` [aquí](#).**

# Funciones



Qualentum Lab

---

En lo que llevamos estudiando a lo largo del fastbook, todo el contenido ha ido dirigido a cómo crear estructuras de datos y de control, que permitirán definir un flujo de ejecución. Sin embargo, cuando una aplicación se hace más grande, es frecuente que haya partes que tengan una lógica similar y se repita el código. Para evitar esto surgen lo que denominamos 'funciones'.

---

**Las funciones son bloques de código con una lógica definida que se almacenan en una variable y se pueden ejecutar múltiples veces.**

## Definición y sintaxis

En Python una función se define usando el **comando def**, seguido del nombre de que se le quiera dar a la función y un (): (paréntesis con dos puntos). Después, en un salto de línea e indentado, se añade todo el código con la lógica que se quiere implementar en la función.

```
>>> def <nombre de la función>():
>>>   < código de la función >
```

Por ejemplo, una función que sume los 5 primeros números naturales podría definirse de la siguiente forma:

```
>>> def sum_numeros_naturales():
>>>   suma = sum([i+1 for i in range(5)])  # sum() suma los elementos de la lista
>>>   print(suma)
```

Una vez definida la función bajo el nombre '*sum\_numeros\_naturales*', podemos ejecutarla múltiples veces llamando a la función.

```
>>> sum_numeros_naturales()  # Así llamamos a la función para que se ejecute
15
>>> sum_numeros_naturales()  # Podemos llamarla de nuevo
15
```

## Return

El ejemplo anterior muestra el número 5 con la función *print*. No obstante, lo más común no es querer solo mostrar un valor, sino que el resultado de la función se puede almacenar en una variable. Para ello se puede añadir el comando *return* al final de la función.

```
>>> def sum_numeros_naturales():
>>>     suma = sum([i+1 for i in range(5)])
>>>     return suma             # Retornamos el valor de la variable suma

>>> resultado = sum_numeros_naturales()
>>> print(resultado)
15
```

## Argumentos

Ya hemos conseguido crear una función que es capaz de sumar los 5 primeros números naturales y devolver su valor. Sin embargo, esto es algo muy poco general. Seguramente en otras ejecuciones no queremos que sean los 5 primeros, sino los 10 o los 100...

Para conseguir que la función sea más general, podemos usar **variables de entrada** a la hora de ejecutar la función. Estas variables se definen **dentro de los paréntesis** a la hora de declarar la función.

```
>>> def sum_numeros_naturales(n_numeros):
>>>     suma = sum([i+1 for i in range(n_numeros)])
>>>     return suma

>>> sum_numeros_naturales(n_numeros=5)
15

>>> sum_numeros_naturales(n_numeros=2)
3
```



Amplía información sobre las funciones [aquí](#).

## Consideraciones

¿Y qué otros factores debemos tener en cuenta a la hora de definir las funciones? Te los detallo a continuación.

### Las variables locales

Los argumentos de entrada de una función son almacenados como variables durante la ejecución de la función, sin embargo, estas variables dejan de existir cuando la ejecución de la función se acaba.

En el caso de que el argumento tenga el mismo nombre que una variable fuera de la función, prevalece el valor del argumento dentro de la función. Cuando la función finaliza, recupera su valor inicial.

```
>>> n=5
>>> def foo(n):
...     print(n)
...
>>> foo(n=3)
3
>>> print(n)
5
```

### El alcance global

Si dentro de una función se llama a una variable que no existe dentro de la función, Python busca si existe esa variable fuera de la función. Si existe, usa ese valor (en general suele ser conveniente evitar usar variables que no existen dentro de la función).

```
>>> n = 5      # Definimos la variable n fuera de la función
>>> def foo():
...     print(n)  # La variable n no existe dentro de la función
...
>>> foo(n=3)
5
```

## Los valores por defecto

En ocasiones, queremos que, si no se pasa el valor de un argumento, la función tenga un valor por defecto. Esto se puede hacer asignando un valor al argumento de la función a la hora de crearla.

```
>>> def foo(n=5):
>>>     print(n)

>>> foo(n=3)
3

>>> foo()
5
```

Lección 6 de 6

# Resumen



Qualentum Lab

---

---

A lo largo de este tema, he querido acercarte a la programación en Python. Ya conoces las características básicas de Python, como lenguaje de programación e intérprete, y cómo podemos instalarlo, junto con un IDE para empezar a usarlo y crear scripts.

También has aprendido los fundamentos de la sintaxis Python para crear variables, las diferentes estructuras de datos, las estructuras de control y, finalmente, las funciones. ¿Cuál es el siguiente paso?

Seguro que sabes la respuesta: que practiques a diario para ir adquiriendo soltura. ¿No fue así como aprendiste a escribir?

**¡Enhorabuena! Fastbook superado**



[Qualentum.com](http://Qualentum.com)