# Tutorial de Python

## Guía básica de Python en español

Este tutorial de Python es una **introducción a los principales aspectos y características del lenguaje**. El tutorial sirve de guía de inicio para dar los primeros pasos con Python, familiarizarse con su sintaxis, conocer los principales tipos de datos y entender algunos conceptos clave.

## ¿Para quién es este tutorial Python?

En principio, **el tutorial es apto para todos los públicos**. Si nunca has programado en Python, te servirá para aprender lo básico del lenguaje. Por el contrario, si ya lo conoces y/o has desarrollado alguna aplicación con él, te será útil para reforzar conceptos y descubrir cosas nuevas que quizá desconozcas.

## Python es el lenguaje preferido de los programadores



## Tutorial Python para aprender el lenguaje desde cero

El tutorial de Python está dividido en diferentes de módulos. Cada uno de ellos contiene una serie tutoriales que cubren un aspecto específico del lenguaje.

### - Módulo 1 -

#### Características básicas del lenguaje

Este primer módulo del tutorial sobre Python es un recorrido a las características básicas del lenguaje: El intérprete, el sangrado, comentarios, convenciones de nombres, palabras reservadas, los tipos básicos del lenguaje, variables, operadores, …

* [Introducción a Python](https://j2logo.com/python/tutorial/introduccion-a-python/)
* [Variables](https://j2logo.com/python/tutorial/variables-python/)
* [Tipos de datos básicos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/)
* [Operadores](https://j2logo.com/python/tutorial/operadores-en-python/)

### - Módulo 2 -

#### Estructuras de control de flujo

El segundo módulo del tutorial sobre Python cubre las estructuras de control de flujo. Estas estructuras son instrucciones clave en cualquier lenguaje de programación. Su importancia radica en que permiten alterar la ejecución secuencial de un programa.

* [Sentencia if](https://j2logo.com/python/tutorial/python-if-sentencia/)
* [Bucle while](https://j2logo.com/python/tutorial/python-while-bucle/)
* [for en Python](https://j2logo.com/bucle-for-en-python/)

### - Módulo 3 -

#### Tipos de datos complejos

Además de los tipos de datos básicos (bool, int, float, complex y str), Python implementa un conjunto de tipos de datos complejos. Estos tipos de datos permiten agrupar, en una misma variable, elementos relacionados. También nos permiten representar secuencias de elementos, mapas o conjuntos. En este tercer módulo del tutorial de Python repasaremos los tipos de datos complejos más importantes.

* [Tipo list](https://j2logo.com/python/tutorial/tipo-list-python/)
* [Tipo tuple](https://j2logo.com/python/tutorial/tipo-tuple-python/)
* [Tipo range](https://j2logo.com/python/tutorial/tipo-range-python/)
* [Tipo set](https://j2logo.com/python/tutorial/tipo-set-python/)
* [Tipo dict](https://j2logo.com/python/tutorial/tipo-dict-python/)
* [Tipo str](https://j2logo.com/python/tutorial/tipo-str-python/)

### - Módulo 4 -

#### División de la lógica de un programa

Python, como otros lenguajes de programación, ofrece mecanismos para dividir y estructurar un programa a medida que este crece en número de líneas de código y complejidad. Estos mecanismos son las funciones, espacios de nombres, módulos, paquetes y clases.

* [Funciones](https://j2logo.com/python/tutorial/funciones-en-python/)
* [Espacios de nombres, módulos y paquetes](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/)
* [Programación orientada a objetos en Python](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/)

### - Otros artículos y tutoriales de interés -

Recopilación de artículos y tutoriales de interés del blog que complementan al tutorial de Python.

* [Entrada de datos por teclado](https://j2logo.com/entrada-datos-por-consola/)
* [Leer un fichero](https://j2logo.com/leer-fichero-linea-a-linea-python/)
* [Escribir en un fichero](https://j2logo.com/como-escribir-en-un-fichero-en-python/)

# Introducción a Python

En este tutorial de introducción a Python vas a descubrir las características principales del lenguaje, aquello que lo hace único y que te hará falta conocer para implementar cualquier programa en este lenguaje.

## **Índice**

* [Características del lenguaje Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-caracteristicas)
* [El intérprete de Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-interprete)
* [Operadores, expresiones y sentencias en Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-expresiones-sentencias)
* [Bloques de código (Indentación)](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-bloques)
* [Comentarios en Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-comentarios)
* [Convenciones de nombres en Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-convenciones)
* [Palabras reservadas de Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-palabras-reservadas)
* [Constantes en Python](https://j2logo.com/python/tutorial/introduccion-a-python/#intro-constantes)

## **Introducción a Python: Características del lenguaje**

Python es un lenguaje de programación de alto nivel cuya máxima es la legibilidad del código. Las principales características de Python son las siguientes:

* **Es multiparadigma**, ya que soporta la programación imperativa, programación orientada a objetos y funcional.
* **Es multiplataforma**: Se puede encontrar un intérprete de Python para los principales sistemas operativos: *Windows*, *Linux* y *Mac OS*. Además, se puede reutilizar el mismo código en cada una de las plataformas.
* **Es dinámicamente tipado**: Es decir, el tipo de las variables se decide en tiempo de ejecución.
* **Es fuertemente tipado:** No se puede usar una variable en un contexto fuera de su tipo. Si se quisiera, habría que hacer una conversión de tipos.
* **Es interpretado**: El código no se compila a lenguaje máquina.

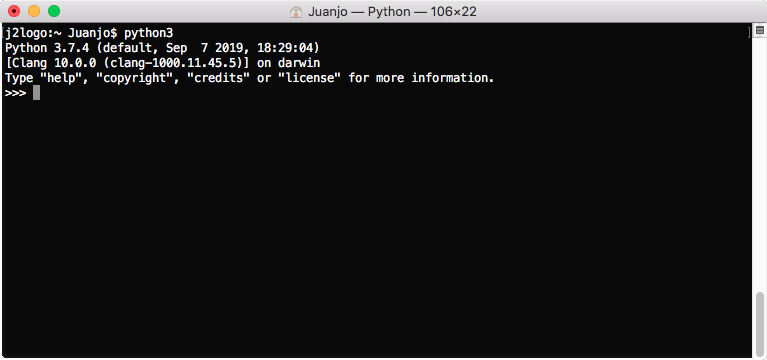
❗️ El hecho de que Python sea **interpretado** quiere decir que **hace falta un intérprete** que permita ejecutar un programa o script escrito en Python sin necesidad de compilarlo.

## **El intérprete de Python**

Cuando instalas Python correctamente (en cualquier sistema operativo) ocurren, entre otras, dos cosas: se añade el comando python (o python3, en caso de que instales la versión 3.x de Python) al path y se instala el intérprete de Python correspondiente.

### **Cómo saber qué versión de Python tengo instalada**

Si ya has instalado Python, abre una consola o terminal y ejecuta el comando python3. Este comando lanzará el intérprete de Python correspondiente. Debes ver algo similar a esta imagen:



Si te fijas bien, en la primera línea podemos ver la versión del intérprete de Python que tenemos instalado en nuestro ordenador. En mi caso es la versión 3.7.4.

En el intérprete de Python podemos escribir expresiones e instrucciones que este interpretará y ejecutará.

Puedes probar, por ejemplo, a escribir 2 + 3. El resultado debe ser el siguiente:

>>> 2 + 35

O ejecutar la instrucción print('¡Hola mundo!'):

>>> print('¡Hola mundo!')

¡Hola mundo!

Para salir del intérprete basta con ejecutar la instrucción quit().

No obstante, aunque esta forma de escribir código puede ser útil para aprender y en casos muy puntuales, no es la habitual a la hora de escribir un programa o script en Python.

### **Primer programa en Python**

Normalmente, los programas en Python se escriben en archivos con la extensión .py. Estos archivos se pasan al intérprete de Python para que los interprete y ejecute.

Vamos a verlo con un ejemplo. Crea con un editor de texto un fichero llamado suma.py con el siguiente contenido:

suma = 2 + 3

print(suma)

A continuación abre un terminal, sitúate en el directorio en el que creaste el archivo suma.py y ejecuta lo siguiente:

python3 suma.py

En el terminal verás que aparece el número 5 como resultado de ejecutar el programa anterior. ¿Qué ha ocurrido aquí? Básicamente que el intérprete de Python ha leído y ejecutado las líneas de código que hemos escrito en el fichero suma.py.

Esta es la manera más común de crear y ejecutar programas en Python.

## **Operadores, expresiones y sentencias en Python**

En este tutorial de introducción a Python quiero que aprendas la diferencia entre operador, expresión y sentencia, ya que son las formas básicas que componen la estructura de cualquier programa.

### **Operador**

Un **operador** es un carácter o conjunto de caracteres que actúa sobre una, dos o más **variables** y/o **literales** para llevar a cabo una **operación** con un **resultado** determinado.

Ejemplos de operadores comunes son los operadores aritméticos + (suma), - (resta) o \* (producto), aunque en Python existen otros operadores.

### **Expresión**

Una expresión es una unidad de código que devuelve un valor y está formada por una combinación de operandos (variables y literales) y operadores. Los siguientes son ejemplos de expresiones (cada línea es una expresión diferente):

5 + 2 # Suma del número 5 y el número 2

a < 10 # Compara si el valor de la variable a es menor que 10

b is None # Compara si la identidad de la variable b es None

3 \* (200 - c) # Resta a 200 el valor de c y lo multiplica por 3

### **Sentencia**

Por su parte, una sentencia o declaración es una instrucción que define una acción. Una sentencia puede estar formada por una o varias expresiones, aunque no siempre es así.

En definitiva, las sentencias son las instrucciones que componen nuestro programa y determinan su comportamiento.

Ejemplos de sentencias son la asignación = o las instrucciones if, if ... else ..., for o while entre otras.

❗️ Una sentencia está delimitada por el carácter Enter (\n).

#### **Sentencias de más de una línea**

Normalmente, las sentencias ocupan una sola línea. Por ejemplo:

a = 2 + 3 # Asigna a la variable <a> el resultado de 2 + 3

Sin embargo, aquellas sentencias que son muy largas pueden ocupar más de una línea ([la guía de estilo PEP 8](https://www.python.org/dev/peps/pep-0008/), recomienda una longitud de línea máxima de 72 caracteres).

Para dividir una sentencia en varias líneas se utiliza el carácter \. Por ejemplo:

a = 2 + 3 + 5 + \

7 + 9 + 4 + \

6

Además de la separación explícita (la que se realiza con el carácter \), en Python la continuación de línea es implícita siempre y cuando la expresión vaya dentro de los caracteres (), [] y {}.

Por ejemplo, podemos inicializar una lista del siguiente modo:

a = [1, 2, 7,

3, 8, 4,

9]

No te preocupes si no sabes lo que es una lista o no entiendes lo que hace el ejemplo anterior. Lo importante es que comprendas que lo anterior es una sentencia multi-línea ya que está comprendida entre los caracteres [].

## **Bloques de código (Indentación)**

Lo último que veremos sobre sentencias en esta introducción a Python es cómo se pueden agrupar en bloques de código.

Un bloque de código es un grupo de sentencias relacionadas bien delimitadas. A diferencia de otros lenguajes como JAVA o C, en los que se usan los caracteres {} para definir un bloque de código, en Python se usa la indentación o sangrado.

El sangrado o indentación consiste en mover un bloque de texto hacia la derecha insertando espacios o tabuladores al principio de la línea, dejando un margen a la izquierda.

👉🏻 Esta es una de las principales características de Python.

Un bloque comienza con un nuevo sangrado y acaba con la primera línea cuyo sangrado sea menor. De nuevo, la guía de estilo de Python recomienda usar los espacios en lugar de las tabulaciones para realizar el sangrado. Yo suelo utilizar 4 espacios.

❗️ Configura tu IDE de desarrollo para que use los espacios en lugar de los tabuladores para el sangrado. Establece el número de espacios a 4 ó 2.

Veamos todo esto con un ejemplo:

def suma\_numeros(numeros): # Bloque 1

suma = 0 # Bloque 2

**for** n **in** numeros: # Bloque 2

suma += n # Bloque 3

print(suma) # Bloque 3

**return** suma # Bloque 2

Como te decía en la sección anterior, no hace falta todavía que entiendas lo que hace el ejemplo. Simplemente debes comprender que en la línea 1 se define la función suma\_numeros. El cuerpo de esta función está definido por el grupo de sentencias que pertenecen al bloque 2 y 3. A su vez, la sentencia for define las acciones a realizar dentro de la misma en el conjunto de sentencias que pertenecen al bloque 3.

## **Comentarios en Python**

Como cualquier otro lenguaje de programación, Python permite escribir comentarios en el código. Los comentarios son útiles para explicar por qué estamos programando algo de un modo concreto o añadir indicaciones. Te aseguro que son de utilidad cuando se retoma un programa o aplicación en el futuro

Los comentarios son ignorados por el intérprete de Python. Solo tienen sentido para los programadores.

Para añadir un comentario a tu código simplemente comienza una línea con el carácter #:

# Esta línea es un comentario

a = 5

# Resultado de multiplicar a por 2

print(a \* 2)

### **Comentarios de varias líneas**

Para escribir comentarios que ocupan varias líneas, simplemente escribe cada una de las líneas anteponiendo el carácter #:

# Este comentario ocupa

# 2 líneas

También puedes escribir un comentario en varias líneas si lo encierras entre tres comillas simples ''' o dobles """

a = 2

'''Este comentario

también ocupa 2 líneas'''

print(a)

Sin embargo, personalmente no me gusta este modo de definir un comentario en varias líneas porque también es la forma de definir un string en varias líneas.

### **Docstrings**

Los docstrings son un tipo de comentarios especiales que se usan para documentar un módulo, función, clase o método. En realidad son la primera sentencia de cada uno de ellos y se encierran entre tres comillas simples o dobles.

Los docstrings son utilizados para generar la documentación de un programa. Además, suelen utilizarlos los entornos de desarrollo para mostrar la documentación al programador de forma fácil e intuitiva.

Veámoslo con un ejemplo:

def suma(a, b):

"""Esta función devuelve la suma de los parámetros a y b"

**return** a + b

## **Convenciones de nombres en Python**

A la hora de nombrar una variable, una función, un módulo, una clase, etc. en Python, siempre se siguen las siguientes reglas y recomendaciones:

* Un identificador puede ser cualquier combinación de letras (mayúsculas y minúsculas), números y el carácter guión bajo (\_).
* Un identificador no puede comenzar por un número.
* A excepción de los nombres de clases, es una convención que todos los identificadores se escriban en minúsculas, separando las palabras con el guión bajo. Ejemplos: contador, suma\_enteros.
* Es una convención que los nombres de clases sigan la notación *Camel Case*, es decir, todas las letras en minúscula a excepción del primer carácter de cada palabra, que se escribe en mayúscula. Ejemplos: Coche, VehiculoMotorizado.
* No se pueden usar como identificadores las palabras reservadas.
* Como recomendación, usa identificadores que sean expresivos. Por ejemplo, contador es mejor que simplemente c.
* Python diferencia entre mayúsculas y minúsculas, de manera que variable\_1 y Variable\_1 son dos identificadores totalmente diferentes.

## **Palabras reservadas de Python**

Python tiene una serie de palabras clave **reservadas**, por tanto, **no pueden usarse como nombres de variables, funciones, etc.**

Estas palabras clave se utilizan para definir la sintaxis y estructura del lenguaje Python.

La lista de palabras reservadas es la siguiente:

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

## **Constantes en Python**

Terminamos esta introducción a Python señalando que, a diferencia de otros lenguajes, **en Python no existen las constantes**.

Entendemos como *constante* una variable que una vez asignado un valor, este no se puede modificar. Es decir, que a la variable no se le puede asignar ningún otro valor una vez asignado el primero.

Se puede simular este comportamiento, siempre desde el punto de vista del programador y atendiendo a convenciones propias, pero no podemos cambiar la naturaleza mutable de las variables.

No obstante, sí que es cierto que el propio Python define una serie de valores constantes en su propio namespace. Los más importantes son:

* **False:** El valor *false* del tipo bool.
* **True:** El valor *true* del tipo bool.
* **None:** El valor del tipo NoneType. Generalmente None se utiliza para representar la ausencia de valor de una variable.

# Variables en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

Este tutorial es muy, muy importante. En él aprenderás qué son las variables en Python, cómo asignarles un valor y cómo modificarlas.

## **Índice**

* [Variables en Python](https://j2logo.com/python/tutorial/variables-python/#variables-intro)
* [Asignar un valor a una variable en Python](https://j2logo.com/python/tutorial/variables-python/#variables-asignacion)
* [Modificar el valor de una variable en Python](https://j2logo.com/python/tutorial/variables-python/#variables-modificar-valor)
* [Literales](https://j2logo.com/python/tutorial/variables-python/#variables-literales)
* [Asignar un valor a múltiples variables](https://j2logo.com/python/tutorial/variables-python/#variables-asignacion-multiple)
* [Asignar múltiples valores a múltiples variables](https://j2logo.com/python/tutorial/variables-python/#variables-asignar-multiples-valores)
* [Variables y memoria](https://j2logo.com/python/tutorial/variables-python/#variables-memoria)

## **Variables en Python**

Las variables son uno de los dos componentes básicos de cualquier programa.

En su esencia, un programa está compuesto por datos e instrucciones que manipulan esos datos. Normalmente, los datos se almacenan en memoria (memoria RAM) para que podamos acceder a ellos.

Entonces, ¿qué es una variable? Una variable es una forma de identificar, de forma sencilla, un dato que se encuentra almacenado en la memoria del ordenador. Imagina que una variable es un contenedor en el que se almacena un dato, el cuál, puede cambiar durante el flujo del programa. Una variable nos permite acceder fácilmente a dicho dato para ser manipulado y transformado.

Por ejemplo, supongamos que queremos mostrar el resultado de sumar 1 + 2. Para mostrar el resultado, debemos indicarle al programa dónde se encuentra dicho dato en memoria y, para ello, hacemos uso de una variable:

# Guardamos en la variable suma el resultado de 1 + 2

suma = 1 + 2

# Accedemos al resultado de 1 + 2 a través de la variable suma

print(suma)

## **Asignar un valor a una variable en Python**

Tal y como hemos visto en el ejemplo anterior, para asignar un valor (un dato) a una variable se utiliza el operador de asignación =.

En la operación de asignación se ven involucradas tres partes:

* El operador de asignación =
* Un identificador o nombre de variable, a la izquierda del operador
* Un literal, una expresión, una llamada a una función o una combinación de todos ellos a la derecha del operador de asignación

Ejemplos:

# Asigna a la variable <a> el valor 1

a = 1

# Asigna a la variable <a> el resultado de la expresión 3 \* 4

a = 3 \* 4

# Asigna a la variable <a> la cadena de caracteres 'Pythonista'

a = 'Pythonista'

Cuando asignamos un valor a una variable por primera vez, se dice que en ese lugar *se define* e *inicializa* la variable. En un script o programa escrito en Python, podemos definir variables en cualquier lugar del mismo. Sin embargo, es una buena práctica definir las variables que vayamos a utilizar al principio.

Si intentamos usar una variable que no ha sido definida/inicializada previamente, el intérprete nos mostrará un error:

>>> print(a)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

NameError: name 'a' is not defined

Para terminar esta sección de variables en Python, vamos a introducir el concepto *tipo de dato*. Al asignar un valor a una variable, dicho valor pertenece a un conjunto de valores conocido como *tipo de dato*. Un tipo de dato define una serie de características sobre esos datos y las variables que los contienen como, por ejemplo, las operaciones que se pueden realizar con ellos. En Python, los [tipos de datos básicos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/) son los numéricos (enteros, reales y complejos), los boolenaos (True, False) y las cadenas de caracteres.

Veamos un ejemplo:

a = 1 # La variable a es de tipo entero (int)

b = 'Hola' # La variable b es de tipo cadena de caracteres

A diferencia de otros lenguajes, en Python no es necesario indicar el tipo de dato cuando se define una variable. Además, en cualquier momento, una variable que es de un tipo puede convertirse en una variable de otro tipo cualquiera:

a = 1 # a es de tipo entero

a = 'Hola' # Ahora a es de tipo cadena de caracteres

Se dice que Python (el intérprete) infiere el tipo de dato al que pertenece una variable en tiempo de ejecución, es decir, es cuando se ejecuta el programa, cuando conoce su tipo de dato y qué operaciones pueden llevarse a cabo con él.

## **Modificar el valor de una variable en Python**

Para modificar el valor de una variable en Python, basta con asignarle un nuevo valor en cualquier momento y lugar después de la definición.

Por ejemplo:

>>> a = 1

>>> print(a)

1

>>> b = 'Hola'

>>> a = 3

>>> print(a)

3

Como vemos en el ejemplo anterior, hemos modificado el dato de la variable a. En la línea 1 le hemos asignado el valor 1. Más tarde, en la línea 5, hemos actualizado su valor a 3.

## **Literales**

Tal y como te he mencionado en un apartado anterior, a una variable se le puede asignar un literal, una expresión, una llamada a una función o una combinación de todos ellos.

Ahora bien, ¿qué es un literal? Ya hemos visto ejemplos de literales en los apartados anteriores. Un literal no es más que un dato en crudo que se puede asignar directamente a una variable o formar parte de una expresión.

En el siguiente ejemplo se asigna a la variable a el literal 1:

a = 1

El literal 1 representa el valor numérico del número entero 1.

Existen varios tipos de literales:

* Numéricos (Enteros, reales y complejos): 1, 3, 3.14, -1, 3.14j, …
* Cadenas de caracteres: 'Hola', 'Me gusta Python', …
* Booleanos: True y False
* El literal especial None. None significa ausencia de valor y es muy utilizado en programación.

## **Asignar un valor a múltiples variables**

Ahora vas a descubrir cómo asignar un mismo valor a múltiples variables a la vez. Si tienes que definir varias variables con un mismo dato, puedes usar la siguiente estructura:

>>> a = b = c = 1 # Inicializa a, b y c con el valor 1

>>> print(a)

1

>>> print(b)

1

>>> print(c)

1

## **Asignar múltiples valores a múltiples variables**

También es posible inicializar varias variables con un valor diferente cada una del siguiente modo:

>>> a, b, c = 1, 2, 3

>>> print(a)

1

>>> print(b)

2

>>> print(c)

3

No obstante, te desaconsejo que lo hagas así porque el código es menos legible.

## **Variables y memoria**

Finalmente, para terminar esta sección, quiero que te quede realmente claro qué es esto de una variable y cómo asigna Python las direcciones de memoria a los datos y a las variables.

Para empezar, te diré que **en Python todo es un objeto**. Sí, todavía no hemos visto el apartado de programación orientada a objetos con Python, pero ten presente que en Python todo es un objeto.

Entonces, ¿qué ocurre cuando a la variable a le asigno el valor 1?

a = 1

Realmente, la variable a hace referencia al objeto que representa al número entero con valor 1. Si ahora creas una nueva variable b y le asignas también el valor 1, b está haciendo referencia al mismo objeto que la variable a. En definitiva, a y b hacen referencia al mismo objeto y, por tanto, están asociadas a la misma dirección de memoria. Esto no es así en otros lenguajes, pero esto es Python, jaja. En otros lenguajes a y b estarían asociadas a direcciones de memoria diferentes.

Para que lo entiendas mejor te voy a presentar la función id() que viene con Python. La función id() devuelve un identificador único del objeto que se le pasa como parámetro. Este identificador es un entero el cuál se garantiza que es único e inmutable durante toda la vida del objeto en memoria.

❗️ En la implementación de Python *CPython* (la más común), este identificador es realmente la dirección en memoria del objeto.

Veamos todo lo anterior con un ejemplo:

# a y b hacen referencia al objeto que representa al entero 1

# Referencian a la misma dirección de memoria

>>> a = 1

>>> b = 1

>>> id(1)

4299329328

>>> id(a)

4299329328

>>> id(b)

4299329328

# b es asignado con el objeto que representa el entero 2

# a y b referencian a diferentes direcciones de memoria

# a mantiene la referencia al entero 1

>>> b = 2

>>> id(a)

4299329328

>>> id(b)

4299329360

# a es asignada con el valor de b

# a y b referencian al mismo objeto y, por tanto,

# a la misma dirección de memoria

>>> a = b

>>> id(a)

4299329360

>>> a

2

# Tipos de datos básicos de Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



SEste tutorial se centra en describir los tipos de datos básicos de Python: qué es un tipo de dato, cuáles son los tipos de datos básicos y cuáles son sus características y propiedades principales.

## **Índice**

* [Qué son los tipos de datos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-que-es)
* [Tipos de datos básicos de Python](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-basicos)
* [Tipos numéricos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-numericos)
* [Tipo booleano](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-booleanos)
* [Tipo cadena de caracteres](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-cadenas)
* [Otros tipos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-otros)
* [Conocer el tipo de una variable](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-variable)
* [Conversión de tipos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/#tipos-datos-conversion)

## **Qué son los tipos de datos**

En cualquier lenguaje de programación de alto nivel se manejan tipos de datos. Los tipos de datos definen un conjunto de valores que tienen una serie de características y propiedades determinadas.

Pensemos por un momento cuando éramos jóvenes 🙄 y estábamos en el colegio en clase de matemáticas. Seguro que diste alguna clase en la que te enseñaban los distintos conjuntos de números. Los naturales (1, 2, 3, 4, …), los enteros (…, -2, -1, 0, 1, 2, …), los reales (… -1.1, -0.3, 2.1, …), etc. Pues bien, en programación (y por supuesto en Python), cada uno de esos conjuntos sería lo que llamamos tipo de datos.

En Python, **todo valor que pueda ser asignado a una variable tiene asociado un tipo de dato**. Como ya te he mencionado alguna vez, en Python todo es un objeto. Así que los tipos de datos serían las clases (donde se definen las propiedades y qué se puede hacer con ellas) y las variables serían las instancias (objetos) de los tipos de datos. No te preocupes si no entiendes qué es una clase o un objeto, lo veremos en otro tutorial.

En definitiva, **un tipo de dato establece qué valores puede tomar una variable y qué operaciones se pueden realizar sobre la misma**.

## **Tipos de datos básicos de Python**

¿Qué tipos de datos trae consigo Python?

En Python podemos encontrar distintos tipos de datos con diferentes características y clasificaciones. En este tutorial repasaremos los tipos de datos básicos, aunque te introduciré otros tipos de datos que veremos en tutoriales posteriores.

Los tipos de datos básicos de Python son los *booleanos*, los *numéricos* (enteros, punto flotante y complejos) y las *cadenas de caracteres*.

Python también define otros tipos de datos, entre los que se encuentran:

* Secuencias: Los tipos [*list*](https://j2logo.com/python/tutorial/tipo-list-python/), [*tuple*](https://j2logo.com/python/tutorial/tipo-tuple-python/) y [*range*](https://j2logo.com/python/tutorial/tipo-range-python/)
* Mapas: El tipo [*dic*t](https://j2logo.com/python/tutorial/tipo-dict-python/)
* Conjuntos: El tipo [*set*](https://j2logo.com/python/tutorial/tipo-set-python/)
* Iteradores
* Clases
* Instancias
* Excepciones

A su vez, los tipos anteriores se pueden agrupar de diferente manera. Por ejemplo: el tipo *cadena de caracteres* es una *secuencia inmutable*; las *listas*, *tuplas* o *diccionarios*, entre otros, son *contenedores* y *colecciones*, etc. Pero esto no lo veremos aquí.

En fin, no te agobies con tanto tipo ni tanto concepto nuevo. Tómatelo con calma que estás aprendiendo. Comencemos por lo fácil revisando los tipos de datos básicos de Python.

## **Tipos numéricos**

Python define tres tipos de datos numéricos básicos: *enteros*, *números de punto flotante* (simularía el conjunto de los números reales, pero ya veremos que no es así del todo) y los números *complejos*.

### **Números enteros**

El tipo de los números *enteros* es int. Este tipo de dato comprende el conjunto de todos los números enteros, pero como dicho conjunto es infinito, en Python el conjunto está limitado realmente por la capacidad de la memoria disponible. No hay un límite de representación impuesto por el lenguaje.

Pero tranquilidad, que para el 99% de los programas que desarrolles tendrás suficiente con el subconjunto que puedes representar.

Un número de tipo int se crea a partir de un literal que represente un número entero o bien como resultado de una expresión o una llamada a una función.

Ejemplos:

>>> a = -1 # a es de tipo int y su valor es -1

>>> b = a + 2 # b es de tipo int y su valor es 1

>>> print(b)

1

También podemos representar los números enteros en formato *binario*, *octal o hexadecimal*.

Los números *octales* se crean anteponiendo 0o a una secuencia de dígitos octales (del 0 al 7).

Para crear un número entero en *hexadecimal*, hay que anteponer 0x a una secuencia de dígitos en hexadecimal (del 0 al 9 y de la A la F).

En cuanto a los números en *binario*, se antepone 0b a una secuencia de dígitos en binario (0 y 1).

>>> diez = 10

>>> diez\_binario = *0b1010*

>>> diez\_octal = 0o12

>>> diez\_hex = *0xa*

>>> print(diez)

10

>>> print(diez\_binario)

10

>>> print(diez\_octal)

10

>>> print(diez\_hex)

10

### **Números de punto flotante**

Bueno, bueno, bueno, entramos en tema caliente y no sé muy bien cómo explicar esto para que sea fácil de entender.

¿Recuerdas que te dije que los números de *punto flotante* representaban, *más o menos*, al conjunto de los números reales?

Vamos a hacer un experimento que te va a dejar a cuadros. Abre un terminal y ejecuta el comando python3 para lanzar el intérprete de Python. A continuación introduce la expresión 1.1 + 2.2 y mira cuál es el resultado.

>>> 1.1 + 2.2

3.3000000000000003

😳 ¿En serio? 😳 ¿Qué es ese 3 del final?

#### **Representación de los números de punto flotante**

Tenemos que repasar un poco de teoría que voy a tratar de simplificar porque la explicación completa da para un artículo entero.

Al igual que ocurre con los números enteros, los números reales son infinitos y, por tanto, es imposible representar todo el conjunto de números reales con un ordenador.

**Para representar** el mayor número posible de **los números reales** con las limitaciones de memoria (tamaños de palabra de 32 y 64 bits), **se adaptó la notación científica de representación de números reales al sistema binario** (que es el sistema que se utiliza en programación para representar los datos e instrucciones).

En esta notación científica, los números se representan así:

|  |  |
| --- | --- |
| Número | Notación científica |
| 101,1 | 1,011 \* 102 |
| 0,032 | 3,2 \* 10-2 |

Vaya tela, ¿no? Pero es una muy buena solución que ha llegado hasta nuestros días.

El caso es que la suma de la representación en punto flotante en binario del número 1,1 y de la representación en punto flotante en binario del número 2,2, dan como resultado 3,3000000000000003

Pero hay más casos, como por ejemplo la representación del número 1/3. En algún momento, el ordenador tiene que truncar el número periódico resultante.

La explicación final es que los números de punto flotante se representan en el hardware del ordenador como fracciones de base 2 (binarias). Y el problema está en que la mayoría de las fracciones decimales no se pueden representar de forma exacta como fracciones binarias porque tienen infinitos números decimales. Una consecuencia es que, en general, los números decimales de punto flotante que usas en tus aplicaciones son una aproximación de los números binarios de punto flotante realmente almacenados en la máquina.

#### **Números de Punto flotante en Python**

Pues una vez vista esta simplificada introducción a los *números de punto flotante*, te diré que este tipo de datos en Python es float.

Puedes usar el tipo float sin problemas para representar cualquier número real (siempre teniendo en cuenta que es una aproximación lo más precisa posible). Por tanto para longitudes, pesos, frecuencias, …, en los que prácticamente es lo mismo 3,3 que 3,3000000000000003 el tipo float es el más apropiado.

Cuando un número float vaya a ser usado por una persona, en lugar de por el ordenador, puedes darle formato al número de la siguiente manera:

>>> real = 1.1 + 2.2 # real es un float

>>> print(real)

3.3000000000000003 # Representación aproximada de 3.3

>>> print(f'{real:.2f}')

3.30 # real mostrando únicamente 2 cifras decimales

Al igual que los números enteros, un float se crea a partir de un literal, o bien como resultado de una expresión o una función.

>>> un\_real = 1.1 # El literal debe incluir el carácter .

>>> otro\_real = 1/2 # El resultado de 1/2 es un float

>>> not\_cient = 1.23E3 # float con notación científica (1230.0)

Y para terminar esta sección, te adelanto que, si por cualquier motivo sí que necesitas una mayor precisión a la hora de trabajar con los números reales, Python tiene otros tipos de datos, como *Decimal*.

El tipo *Decimal* es ideal a la hora de trabajar, por ejemplo, con dinero o tipos de interés. Este tipo de dato trunca la parte decimal del número para ser más preciso, pero no es el objetivo de este tutorial hablar sobre el tipo de dato *Decimal*.

### **Números complejos**

El último tipo de dato numérico básico que tiene Python es el de los números *complejos*, complex.

Los números complejos tienen una parte *real* y otra *imaginaria* y cada una de ellas se representa como un float.

Para crear un número complejo, se sigue la siguiente estructura <parte\_real>+<parte\_imaginaria>j. Y se puede acceder a la parte real e imaginaria a través de los atributos real e imag:

>>> complejo = 1+2j

>>> complejo.real

1.0

>>> complejo.imag

2.0

### **Aritmética de los tipos numéricos**

Con todos los tipos numéricos se pueden aplicar las operaciones de la aritmética: suma, resta, producto, división, …

En Python está permitido realizar una operación aritmética con números de distinto tipo. En este caso, el tipo numérico «más pequeño» se convierte al del tipo «más grande», de manera que el tipo del resultado siempre es el del tipo *mayor*. Entendemos que el tipo int es menor que el tipo float que a su vez es menor que el tipo complex.

Por tanto, es posible, por ejemplo, sumar un int y un float:

>>> 1 + 2.0

3.0

>>> 2+3j + 5.7

(7.7+3j)

## **Tipo booleano**

En Python la clase que representa los valores booleanos es bool. Esta clase solo se puede instanciar con dos valores/objetos: True para representar verdadero y False para representar falso.

Una particularidad del lenguaje es que cualquier objeto puede ser usado en un contexto donde se requiera comprobar si algo es *verdadero* o *falso*. Por tanto, cualquier objeto se puede usar en la condición de un if o un while (son estructuras de control que veremos en tutoriales posteriores) o como operando de una operación booleana.

Por defecto, **cualquier objeto es considerado como verdadero con dos excepciones**:

* Que implemente el método \_\_bool\_\_() y este devuelva False.
* Que impleménte el método \_\_len\_\_() y este devuelva 0.

Además, los siguientes objetos/instancias también son consideradas falsas:

* None
* False
* El valor cero de cualquier tipo numérico: 0, 0.0, 0j, …
* Secuencias y colecciones vacías (veremos estos tipos en otros tutoriales): '', (), [], {}, set(), range(0)

## **Tipo cadena de caracteres**

Una vez que hemos acabado con los números, es el turno de las letras 😜

Otro tipo básico de Python, e imprescindible, son las secuencias o cadenas de caracteres. Este tipo es conocido como *string* aunque su clase verdadera es str.

Formalmente, un *string* es una secuencia **inmutable** de caracteres en formato [**Unicode**](https://es.wikipedia.org/wiki/Unicode).

Para crear un string, simplemente tienes que encerrar entre comillas simples '' o dobles "" una secuencia de caracteres.

Se puede usar indistintamente comillas simples o dobles, con una particularidad. Si en la cadena de caracteres se necesita usar una comilla simple, tienes dos opciones: usar comillas dobles para encerrar el string, o bien, usar comillas simples pero anteponer el carácter \ a la comilla simple del interior de la cadena. El caso contrario es similar.

Veamos todo esto con un ejemplo:

>>> hola = 'Hola "Pythonista"'

>>> hola\_2 = 'Hola \'Pythonista\''

>>> hola\_3 = "Hola 'Pythonista'"

>>> print(hola)

Hola "Pythonista"

>>> print(hola\_2)

Hola 'Pythonista'

>>> print(hola\_3)

Hola 'Pythonista'

A diferencia de otros lenguajes, en Python no existe el tipo «carácter». No obstante, se puede simular con un string de un solo carácter:

>>> caracter\_a = 'a'

>>> print(caracter\_a)

a

Revisa este tutorial si quieres conocer más sobre la clase [str de Python](https://j2logo.com/python/tutorial/tipo-str-python/).

## **Otros tipos**

Hasta aquí hemos repasado los tipos de datos básicos de Python, sin embargo, el lenguaje ofrece muchos tipos más. Te hago aquí un avance de los más importantes, aunque los veremos en detalle en otros tutoriales.

Además de los tipos básicos, otros tipos fundamentales de Python son las secuencias (list y tuple), los conjuntos (set) y los mapas (dict).

Todos ellos son tipos compuestos y se utilizan para agrupar juntos varios valores.

* Las [listas](https://j2logo.com/python/tutorial/tipo-list-python/) son secuencias mutables de valores.
* Las [tuplas](https://j2logo.com/python/tutorial/tipo-tuple-python/) son secuencias inmutables de valores.
* Los [conjuntos](https://j2logo.com/python/tutorial/tipo-set-python/) se utilizan para representar conjuntos únicos de elementos, es decir, en un conjunto no pueden existir dos objetos iguales.
* Los [diccionarios](https://j2logo.com/python/tutorial/tipo-dict-python/) son tipos especiales de contenedores en los que se puede acceder a sus elementos a partir de una clave única.

>>> lista = [1, 2, 3, 8, 9]

>>> tupla = (1, 4, 8, 0, 5)

>>> conjunto = set([1, 3, 1, 4])

>>> diccionario = {'a': 1, 'b': 3, 'z': 8}

>>> print(lista)

[1, 2, 3, 8, 9]

>>> print(tupla)

(1, 4, 8, 0, 5)

>>> print(conjunto)

{1, 3, 4}

>>> print(diccionario)

{'a': 1, 'b': 3, 'z': 8}

## **Conocer el tipo de una variable**

Ahora te voy a presentar dos funciones para que puedas jugar con todo lo que hemos visto en este tutorial. Son type() e isinstance():

* type() recibe como parámetro un objeto y te devuelve el tipo del mismo.
* isinstance() recibe dos parámetros: un objeto y un tipo. Devuelve True si el objeto es del tipo que se pasa como parámetro y False en caso contrario.

>>> type(3)

<**class** 'int'>

>>> type(2.78)

<**class** 'float'>

>>> type('Hola')

<**class** 'str'>

>>> isinstance(3, float)

**False**

>>> isinstance(3, int)

**True**

>>> isinstance(3, bool)

**False**

>>> isinstance(**False**, bool)

True

## **Conversión de tipos**

Lo último que veremos en este tutorial sobre tipos de datos es la conversión de tipos.

¿Esto qué significa?

Imagina que tienes una variable edad de tipo *string* cuyo valor es '25'. Se podría decir que edad, aunque realmente es una cadena de caracteres, contiene un número. Sin embargo, si intentas sumar 10 a edad, el intérprete te dará un error porque edad es de tipo str y 10 un tipo numérico.

>>> edad = '25'

>>> edad = edad + 10

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

TypeError: can only concatenate str (not "int") to str

¿Cómo puedo tratar la variable edad como un número? Convirtiéndola a un tipo numérico, por ejemplo, al tipo int.

Para ello, Python ofrece las siguientes funciones:

* str(): Devuelve la representación en cadena de caracteres del objeto que se pasa como parámetro.
* int(): Devuelve un int a partir de un número o secuencia de caracteres.
* float(): Devuelve un float a partir de un número o secuencia de caracteres.
* complex(): Devuelve un complex a partir de un número o secuencia de caracteres.

Si a las funciones anteriores se les pasa como parámetro un valor inválido, el intérprete mostrará un error.

>>> edad = int(edad) + 10 # Convierte edad a int

>>> edad # edad es un int

35

>>> edad = str(edad) # Convierte edad a str

>>> edad # edad es un str (se muestran las '')

'35'

>>> float('18.66') # Convierte un str a float

18.66

>>> float('hola') # Convierte un str a float (pero no es válido)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

ValueError: could not convert string to float: 'hola'

# Operadores en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*operadores*](https://j2logo.com/tag/operadores/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)

En el [tutorial de Introducción](https://j2logo.com/python/tutorial/introduccion-a-python/) ya hablamos sobre los operadores en Python. Como te indiqué, los operadores son símbolos reservados por el propio lenguaje que se utilizan para llevar a cabo operaciones sobre uno, dos o más elementos llamados operandos. Los operandos pueden ser variables, literales, el valor devuelto por una expresión o el valor devuelto por una función.

El ejemplo más típico que siempre viene a la mente es el operador suma, +, que se utiliza para obtener la suma aritmética de dos valores:

>>> 9 + 1 # 9 y 1 son los operandos

10 # 10 es el resultado

En este tutorial entraremos más en detalle sobre los distintos tipos de operadores disponibles en Python.

## **Índice – Todos los operadores en Python**

* [Operador de concatenación de cadenas de caracteres](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-cadenas)
* [Operadores lógicos o booleanos](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-logicos)
* [Operadores de comparación](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-comparacion)
* [Operadores aritméticos](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-aritmeticos)
* [Operadores a nivel de bits](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-bits)
* [Operadores de asignación](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-asignacion)
* [Operadores de pertenencia](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-pertenencia)
* [Operadores de identidad](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-identidad)
* [Prioridad de los operadores en Python](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-prioridad)

## **Operador de concatenación de cadenas de caracteres**

Una de las operaciones más básicas cuando se trabaja con cadenas de caracteres es la concatenación. Esto consiste en unir dos cadenas en una sola, siendo el resultado un nuevo *string*.

La forma más simple de concatenar dos cadenas en Python es utilizando el operador de concatenación +:

>>> hola = 'Hola'

>>> python = 'Pythonista'

>>> hola\_python = hola + ' ' + python # concatenamos 3 strings

>>> print(hola\_python)

Hola Pythonista

En esta entrada puedes conocer otras formas de [concatenar en Python más avanzadas](https://j2logo.com/como-concatenar-y-formatear-strings/).

## **Operadores lógicos o booleanos**

A la hora de operar con valores booleanos, tenemos a nuestra disposición los operadores and, or y not.

❗️ **IMPORTANTE:** Las operaciones and, or y not realmente no devuelven True o False, sino que devuelven uno de los operandos como veremos en el cuadro de abajo.

A continuación te muestro cómo funcionan los operadores booleanos (en orden de preferencia ascendente):

|  |  |  |
| --- | --- | --- |
| Operación | Resultado | Descripción |
| a or b | Si a se evalúa a falso, entonces devuelve b, si no devuelve a | Solo se evalúa el segundo operando si el primero es falso |
| a and b | Si a se evalúa a falso, entonces devuelve a, si no devuelve b | Solo se evalúa el segundo operando si el primero es verdadero |
| not a | Si a se evalúa a falso, entonces devuelve True, si no devuelve False | Tiene menos prioridad que otros operadores no booleanos |

Ejemplos:

>>> x = **True**

>>> y = **False**

>>> x or y

**True**

>>> x and y

**False**

>>> not x

**False**

>>> x = 0

>>> y = 10

>>> x or y

10

>>> x and y

0

>>> not x

True

## **Operadores de comparación**

Los operadores de comparación se utilizan, como su nombre indica, para comparar dos o más valores. El resultado de estos operadores siempre es True o False.

|  |  |
| --- | --- |
| Operador | Descripción |
| > | Mayor que. True si el operando de la izquierda es estrictamente mayor que el de la derecha; False en caso contrario. |
| >= | Mayor o igual que. True si el operando de la izquierda es mayor o igual que el de la derecha; False en caso contrario. |
| < | Menor que. True si el operando de la izquierda es estrictamente menor que el de la derecha; False en caso contrario. |
| <= | Menor o igual que. True si el operando de la izquierda es menor o igual que el de la derecha; False en caso contrario. |
| == | Igual. True si el operando de la izquierda es igual que el de la derecha; False en caso contrario. |
| != | Distinto. True si los operandos son distintos; False en caso contrario. |

Ejemplos:

>>> x = 9

>>> y = 1

>>> x < y

**False**

>>> x > y

**True**

>>> x == y

False

### **Consideraciones sobre los operadores de comparación**

Los objetos de diferentes tipos, excepto los tipos numéricos, nunca se comparan igual. El operador == siempre está definido, pero para algunos tipos de objetos (por ejemplo, objetos de clase) es equivalente a [is](https://j2logo.com/python/tutorial/operadores-en-python/#operadores-identidad).

Las instancias no idénticas de una clase normalmente se comparan como no iguales a menos que la clase defina el método \_\_eq\_\_().

Las instancias de una clase no se pueden ordenar con respecto a otras instancias de la misma clase u otros tipos de objeto, a menos que la clase defina los métodos \_\_lt\_\_(), \_\_gt\_\_().

Los operadores de comparación se pueden concatenar. Ejemplo:

# Las comparaciones siguientes son idénticas

>>> x = 9

>>> 1 < x and x < 20

**True**

>>> 1 < x < 20

True

## **Operadores aritméticos en Python**

En cuanto a los operadores aritméticos, estos permiten realizar las diferentes operaciones aritméticas del álgebra: suma, resta, producto, división, … Estos operadores Python son de los más utilizados. El listado completo es el siguiente:

|  |  |
| --- | --- |
| Operador | Descripción |
| + | Suma dos operandos. |
| – | Resta al operando de la izquierda el valor del operando de la derecha. Utilizado sobre un único operando, le cambia el signo. |
| \* | Producto/Multiplicación de dos operandos. |
| / | Divide el operando de la izquierda por el de la derecha (el resultado siempre es un float). |
| % | Operador módulo. Obtiene el resto de dividir el operando de la izquierda por el de la derecha. |
| // | Obtiene el cociente entero de dividir el operando de la izquierda por el de la derecha. |
| \*\* | Potencia. El resultado es el operando de la izquierda elevado a la potencia del operando de la derecha. |

>>> x = 7

>>> y = 2

>>> x + y # Suma

9

>>> x - y # Resta

5

>>> x \* y # Producto

14

>>> x / y # División

3.5

>>> x % y # Resto

1

>>> x // y # Cociente

3

>>> x \*\* y # Potencia

49

## **Operadores a nivel de bits**

Los operadores a nivel de bits actúan sobre los operandos como si fueran una cadena de dígitos binarios. Como su nombre indica, actúan sobre los operandos bit a bit. Son los siguientes:

|  |  |
| --- | --- |
| Operación | Descripción |
| x | y | or bit a bit de x e y. |
| x ^ y | or exclusivo bit a bit de x e y. |
| x & y | and bit a bit de x e y. |
| x << n | Desplaza x n bits a la izquierda. |
| x >> n | Desplaza x n bits a la derecha. |
| ~x | not x. Obtiene los bits de x invertidos. |

Supongamos que tenemos el entero 2 (en bits es 00010) y el entero 7 (00111). El resultado de aplicar las operaciones anteriores es:

>>> x = 2

>>> y = 7

>>> x | y

7

>>> x ^ y

5

>>> x & y

2

>>> x << 1

4

>>> x >> 1

1

>>> ~x

-3

## **Operadores de asignación**

El operador de asignación se utiliza para asignar un valor a una variable. Como te he mencionado en otras secciones, este operador es el signo =.

Además del operador de asignación, existen otros operadores de asignación compuestos que realizan una operación básica sobre la variable a la que se le asigna el valor.

Por ejemplo, x += 1 es lo mismo que x = x + 1. Los operadores compuestos realizan la operación que hay antes del signo igual, tomando como operandos la propia variable y el valor a la derecha del signo igual.

A continuación, aparece la lista de todos los operadores de asignación compuestos:

|  |  |  |
| --- | --- | --- |
| Operador | Ejemplo | Equivalencia |
| += | x += 2 | x = x + 2 |
| -= | x -= 2 | x = x – 2 |
| \*= | x \*= 2 | x = x \* 2 |
| /= | x /= 2 | x = x / 2 |
| %= | x %= 2 | x = x % 2 |
| //= | x //= 2 | x = x // 2 |
| \*\*= | x \*\*= 2 | x = x \*\* 2 |
| &= | x &= 2 | x = x & 2 |
| |= | x |= 2 | x = x | 2 |
| ^= | x ^= 2 | x = x ^ 2 |
| >>= | x >>= 2 | x = x >> 2 |
| <<= | x <<= 2 | x = x << 2 |

## **Operadores de pertenencia**

Los operadores de pertenencia se utilizan para comprobar si un valor o variable se encuentran en una secuencia (list, tuple, dict, set o str).

Todavía no hemos visto estos tipos, pero son operadores muy utilizados.

|  |  |
| --- | --- |
| Operador | Descripción |
| in | Devuelve True si el valor se encuentra en una secuencia; False en caso contrario. |
| not in | Devuelve True si el valor no se encuentra en una secuencia; False en caso contrario. |

A continuación vemos unos ejemplos que son muy intuitivos:

>>> lista = [1, 3, 2, 7, 9, 8, 6]

>>> 4 **in** lista

**False**

>>> 3 **in** lista

**True**

>>> 4 not **in** lista

True

## **Operadores de identidad**

Por último, los operadores de identidad se utilizan para comprobar si dos variables son, o no, el mismo objeto.

|  |  |
| --- | --- |
| Operador | Descripción |
| is | Devuelve True si ambos operandos hacen referencia al mismo objeto; False en caso contrario. |
| is not | Devuelve True si ambos operandos no hacen referencia al mismo objeto; False en caso contrario. |

❗️ **Recuerda:** Para conocer la identidad de un objeto se usa la función id().

>>> x = 4

>>> y = 2

>>> lista = [1, 5]

>>> x is lista

**False**

>>> x is y

**False**

>>> x is 4

True

## **Prioridad de los operadores en Python**

Al igual que ocurre en las matemáticas, los operadores en Python tienen un orden de prioridad. Este orden es el siguiente, de menos prioritario a más prioritario: asignación; operadores booleanos; operadores de comparación, identidad y pertenencia; a nivel de bits y finalmente los aritméticos (con el mismo orden de prioridad que en las matemáticas).

Este orden de prioridad se puede alterar con el uso de los paréntesis ():

>>> x = 5

>>> y = 2

>>> z = x + 3 \* y # El producto tiene prioridad sobre la suma

>>> z

11

>>> z = (x + 3) \* y # Los paréntesis tienen prioridad

>>> z

16

# Python if – Sentencia if de control de flujo

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*controlflujo*](https://j2logo.com/tag/controlflujo/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

En Python if es una de las principales sentencias de control de flujo, junto a [*while*](https://j2logo.com/python/tutorial/python-while-bucle/) y [*for*](https://j2logo.com/bucle-for-en-python/).

En este tutorial te mostraré en detalle la sentencia if de control de flujo en Python, desde la estructura más básica hasta las diferentes alternativas.

## **Índice**

* [Python if – Sentencia básica](https://j2logo.com/python/tutorial/python-if-sentencia/#python-if-basico)
* [Sentencia if … else](https://j2logo.com/python/tutorial/python-if-sentencia/#python-if-else)
* [if … elif … else](https://j2logo.com/python/tutorial/python-if-sentencia/#python-if-elif)
* [Sentencias if anidadas](https://j2logo.com/python/tutorial/python-if-sentencia/#python-if-anidadas)

## **Python if – Sentencia básica**

En Python, la sentencia if se utiliza para ejecutar un bloque de código si, y solo si, se cumple una determinada condición. Por tanto, if es usado para la toma de decisiones.

La estructura básica de esta sentencia if es la siguiente:

if condición:

bloque de código

Es decir, solo si condición se evalúa a True, se ejecutarán las sentencias que forman parte de bloque de código. En caso de que se evalúe a False no se ejecutará ninguna sentencia perteneciente a bloque de código.

Aquí, condición puede ser un literal, el valor de una variable, el resultado de una expresión o el valor devuelto por una función.

En las expresiones es muy común usar [los operadores booleanos y de comparación](https://j2logo.com/python/tutorial/operadores-en-python/).

❗️**IMPORTANTE:** El cuerpo del bloque está indicado con un sangrado mayor. Dicho bloque termina cuando se encuentre la primera línea con un sangrado menor.

Veamos un ejemplo:

x = 17

**if** x < 20:

print('x es menor que 20')

En el código anterior la variable x toma el valor 17. En la línea 2, la condición de la sentencia if evalúa si x es menor que 20. Como el valor devuelto por la expresión es True, se ejecuta el bloque del if, mostrando por pantalla la cadena x es menor que 20.

Veamos otro ejemplo:

valores = [1, 3, 4, 8]

**if** 5 **in** valores:

print('está en valores')

print('fin')

En este caso tenemos una lista de valores. El if comprueba si el número 5 se encuentra entre estos valores. Como la expresión devuelve como resultado False, porque 5 no está entre los valores, el código anterior simplemente mostrará por pantalla la cadena fin.

Si repetimos el ejemplo anterior, modificando la condición a esta otra 3 in valores, el resultado del programa sería:

está en valores

fin

❗️ **RECUERDA:** En principio, en Python todos los objetos/instancias se evalúan a True a excepción de None, False, 0 de todos los tipos numéricos y secuencias/colecciones vacías, que se evalúan a False.

## **Sentencia if … else**

Hay ocasiones en que la sentencia if básica no es suficiente y es necesario ejecutar un conjunto de instrucciones o sentencias cuando la condición se evalúa a False.

Para ello se utiliza la estructura if ... else... Esta es estructura es como sigue:

if condición:

bloque de código (cuando condición se evalúa a **True**)

**else**:

bloque de código 2 (cuando condición se evalúa a **False**)

Imagina que estás implementado un programa en el que necesitas realizar la división de dos números. Si divides un número entre 0, el programa lanzará un error y terminará. Para evitar esto, podemos añadir una sentencia if ... else... como en el ejemplo siguiente:

resultado = None

x = 10

y = 2

**if** y > 0:

resultado = x / y

**else**:

resultado = f'No se puede dividir {x} entre {y}'

print(resultado)

El resultado del script anterior es 5.0. Si ahora actualizamos el valor de y a 0, el resultado sería este otro:

No se puede dividir 10 entre 0

## **if … elif … else**

También es posible que te encuentres situaciones en que una decisión dependa de más de una condición.

En estos casos se usa una sentencia if compuesta, cuya estructura es como se indica a continuación:

if cond1:

bloque cond1 (sentencias si se evalúa la cond1 a **True**)

elif cond2:

bloque cond2 (sentencias si cond1 es **False** pero cond2 es **True**)

...

**else**:

bloque **else** (sentencias si todas las condiciones se evalúan a **False**)

Es decir, en esta ocasión, se comprueba la condición cond1. Si se evalúa a True, se ejecuta el bloque bloque cond1 y después el flujo continúa después del bloque if.

Sin embargo, si cond1 se evalúa a False, se comprueba la condición cond2. Si esta se evalúa a True, se ejecuta el bloque de sentencias bloque cond2. En caso contrario, pasaría a comprobar la condición del siguiente elif y así sucesivamente hasta que llegue al else, que se ejecuta si ninguna de las condiciones anteriores se evaluaron a True.

Veamos un ejemplo. Imagina que queremos comprobar si un número es menor que 0, igual a 0 o mayor que 0 y en cada situación debemos ejecutar un código diferente. Podemos hacer uso de la estructura if … elif … else anterior:

x = 28

**if** x < 0:

print(f'{x} es menor que 0')

elif x > 0:

print(f'{x} es mayor que 0')

**else**:

print('x es 0')

En el ejemplo anterior x toma el valor 28. La primera condición de la sentencia if se evalúa a False y pasa a la siguiente (a la del bloque elif). En esta ocasión, la expresión se evalúa a True, mostrándose por pantalla la cadena 28 es mayor que 0.

## **Sentencias if anidadas**

Para terminar este tutorial, simplemente me gustaría añadir que en cualquiera de los bloques de sentencias anteriores se puede volver a incluir una sentencia if, o if … else … o if … elif … else …

Por ejemplo, podemos simular el caso de la última sección de la siguiente manera:

x = 28

**if** x < 0:

print(f'{x} es menor que 0')

**else**:

**if** x > 0:

print(f'{x} es mayor que 0')

**else**:

print('x es 0')

# While Python – Bucle while en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*controlflujo*](https://j2logo.com/tag/controlflujo/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

La sentencia o bucle while en Python es una sentencia de control de flujo que se utiliza para ejecutar un bloque de instrucciones de forma continuada mientras se cumpla una condición determinada.

En este tutorial te explicaré en detalle la sentencia while en Python y te mostraré ejemplos de uso común.

## **Índice**

* [La sentencia while en Python](https://j2logo.com/python/tutorial/python-while-bucle/#while-python)
* [Bucle while en Python – Ejemplo típico](https://j2logo.com/python/tutorial/python-while-bucle/#while-ejemplo)
* [Bucle while … else …](https://j2logo.com/python/tutorial/python-while-bucle/#while-else)

## **La sentencia while en Python**

Como te decía, el uso principal de la sentencia *while* es ejecutar repetidamente un bloque de código mientras se cumpla una condición.

La estructura de esta sentencia *while* es la siguiente:

while condición:

bloque de código

Es decir, mientras condición se evalúe a True, se ejecutarán las instrucciones y sentencias de bloque de código.

Aquí, condición puede ser un literal, el valor de una variable, el resultado de una expresión o el valor devuelto por una función.

❗️ **IMPORTANTE:** El cuerpo del bloque while está indicado con un sangrado mayor. Dicho bloque termina cuando se encuentre la primera línea con un sangrado menor.

Veamos un ejemplo:

numero = 0

print('Tabla del 3')

**while** numero <= 10:

print(f'{numero \* 3}')

numero += 1

print('Fin')

En el script anterior se inicializa una variable numero con el valor 0.

Seguidamente se muestra un mensaje y en la *línea 3* aparece una sentencia *while*. En esta sentencia se comprueba si el valor de la variable numero es menor o igual a 10.

Como se evalúa a True, se muestra por pantalla el resultado de multiplicar numero por 3 y después se incrementa el valor de numero en 1.

A continuación, se debería ejecutar el código de la *línea 6*. Sin embargo, como hemos definido un bucle *while*, el flujo de ejecución del programa no continúa por la *línea 6*, sino que vuelve a la *línea 3* y de nuevo se evalúa si la expresión numero <= 10 sigue siendo True. En esta ocasión el valor de numero es 1, por lo que la expresión da como resultado True y vuelven a ejecutarse las líneas de la sentencia *while*.

Esto será así hasta que numero sea igual a 11. En esa ocasión la expresión de comparación se evaluará a False y el flujo continuará, ahora sí, por la *línea 6*.

El resultado del script anterior es el siguiente (la tabla de multiplicar del número 3):

Tabla del 3

0

3

6

9

12

15

18

21

24

27

30

Fin

## **Bucle while en Python – Ejemplo típico**

La sentencia *while* la puedes usar en multitud de ocasiones. A continuación te mostraré un escenario típico de uso de bucle *while*: Comprobar si existe un elemento en una secuencia.

Imagina que tienes la siguiente lista de valores [5, 1, 9, 2, 7, 4] y quieres saber si el número 2 está contenido en dicha lista. La estructura típica de bucle while para ello es como sigue:

valores = [5, 1, 9, 2, 7, 4]

encontrado = **False**

indice = 0

longitud = len(valores)

**while** not encontrado and indice < longitud:

valor = valores[indice]

**if** valor == 2:

encontrado = **True**

**else**:

indice += 1

**if** encontrado:

print(f'El número 2 ha sido encontrado en el índice {indice}')

**else**:

print('El número 2 no se encuentra en la lista de valores')

Como puedes observar, en el ejemplo, se utilizan 3 variables de control:

* encontrado: Indica si el número 2 ha sido encontrado en la lista.
* indice: Contiene el índice del elemento de la lista valores que va a ser evaluado.
* longitud: Indica el número de elementos de la lista valores.

En esta ocasión, la condición de continuación del bucle *while* es que no se haya encontrado el número 2 y que el índice del elemento a evaluar sea menor que la longitud de la lista (es posible acceder a un elemento de una lista a partir del índice de su posición, comenzando por 0).

Por tanto, el bucle finaliza bien cuando se haya encontrado el elemento, bien cuando se haya evaluado el último elemento de la lista. Si se ha encontrado el número 2, se muestra un mensaje indicando el índice en el que está. En caso contrario, se muestra un mensaje indicando que el número 2 no se encuentra en la lista.

❗️ **NOTA:** El anterior es solo un ejemplo para que veas cómo funciona la sentencia *while*. En realidad, en Python se puede usar el operador in para comprobar de forma más eficiente si un elemento está contenido en una secuencia.

## **Bucle while … else …**

Al igual que sucede con el [bucle for](https://j2logo.com/bucle-for-en-python/), podemos alterar el flujo de ejecución del bucle *while* con las sentencias break y continue:

* break se utiliza para finalizar y salir el bucle, por ejemplo, si se cumple alguna condición.
* Por su parte, continue salta al siguiente paso de la iteración, ignorando todas las sentencias que le siguen y que forman parte del bucle.

Veamos el ejemplo de la sección anterior modificado, añadiendo la sentencia break:

valores = [5, 1, 9, 2, 7, 4]

encontrado = **False**

indice = 0

longitud = len(valores)

**while** indice < longitud:

valor = valores[indice]

**if** valor == 2:

encontrado = **True**

break

**else**:

indice += 1

**if** encontrado:

print(f'El elemento 2 ha sido encontrado en el índice {indice}')

**else**:

print('El elemento 2 no se encuentra en la lista de valores')

Lo que he hecho ha sido eliminar de la condición de continuación del bucle la comprobación de la variable encontrado. Además, he añadido la sentencia break si el número 2 se encuentra en la lista.

Por otro lado, al bucle *while* le podemos añadir la sentencia opcional else. El bloque de código del else se ejecutará siempre y cuando la condición de la sentencia while se evalúe a False y no se haya ejecutado una sentencia break.

Continuando con el ejemplo anterior, lo podemos modificar del siguiente modo:

valores = [5, 1, 9, 2, 7, 4]

indice = 0

longitud = len(valores)

**while** indice < longitud:

valor = valores[indice]

**if** valor == 2:

print(f'El elemento 2 ha sido encontrado en el índice {indice}')

break

**else**:

indice += 1

**else**:

print('El elemento 2 no se encuentra en la lista de valores')

En esta ocasión se ha eliminado completamente el uso de la variable de control encontrado.

# for en Python – El bucle for en Python: estructura y ejemplos

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*controlflujo*](https://j2logo.com/tag/controlflujo/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*python3*](https://j2logo.com/tag/python3/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

Una de las principales tareas a la hora de aprender un nuevo lenguaje de programación es conocer las estructuras de control que ofrece el propio lenguaje. El **bucle for en Python** es una de las estructuras que más utilizarás en tus programas.

A continuación veremos por qué el bucle for en Python es importante, cuándo usarlo y cómo.

## **Índice**

* [El bucle for en python](https://j2logo.com/bucle-for-en-python/#for-en-python)
* [Qué es un iterable](https://j2logo.com/bucle-for-en-python/#for-iterable)
* [Bucle for en diccionarios](https://j2logo.com/bucle-for-en-python/#for-diccionarios)
* [Python for y la clase range](https://j2logo.com/bucle-for-en-python/#for-range)
* [for \_ in](https://j2logo.com/bucle-for-en-python/#for-guionbajo)
* [Bucle for, break y continue](https://j2logo.com/bucle-for-en-python/#for-break-continue)
* [for … else](https://j2logo.com/bucle-for-en-python/#for-else)

## **El bucle for en python**

El bucle for se utiliza para recorrer los elementos de un objeto *iterable* (lista, tupla, conjunto, diccionario, …) y ejecutar un bloque de código. En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable, sobre el cuál se pueden aplicar una serie de operaciones.

Su sintaxis es la siguiente:

**for** <elem> **in** <iterable>:

<Tu código>

Aquí, elem es la variable que toma el valor del elemento dentro del iterador en cada paso del bucle. Este finaliza su ejecución cuando se recorren todos los elementos.

Como te indicaba, es muy frecuente usar el bucle for para iterar sobre los elementos de listas, tuplas o diccionarios.

### **Ejemplo de cómo usar el bucle for**

Imaginemos que tenemos una lista de números y queremos mostrarlos por consola. El código podría ser así:

nums = [4, 78, 9, 84]

**for** n **in** nums:

print(n)

4

78

9

84

## **Qué es un iterable**

Veamos con más detalle el concepto de iterable mencionado en el apartado anterior. Un *iterable* es un objeto que se puede iterar sobre él, es decir, que permite recorrer sus elementos uno a uno. Para ser más técnico, un objeto iterable es aquél que puede pasarse como parámetro de la función iter().

Esta función devuelve un iterador basado en el objeto iterable que se pasa como parámetro.

Finalmente, un *iterador* es un objeto que define un mecanismo para recorrer los elementos del *iterable* asociado.

Te lo enseño mejor con un ejemplo:

>>> nums = [4, 78, 9, 84]

>>> it = iter(nums)

>>> next(it)

4

>>> next(it)

78

>>> next(it)

9

>>> next(it)

84

>>> next(it)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

StopIteration

Como puedes observar, un *iterador* recorre los elementos de un *iterable* solo hacia delante. Cada vez que se llama a la función next() se recupera el siguiente valor del *iterador*.

En Python, los tipos principales list, tuple, dict, set o string entre otros, son iterables, por lo que podrán ser usados en el bucle for.

## **Bucle for en diccionarios**

Un caso es especial de bucle for se da al recorrer los elementos de un diccionario. Dado que un diccionario está compuesto por pares clave/valor, hay distintas formas de iterar sobre ellas.

1 – Recorrer las claves del diccionario.

valores = {'A': 4, 'E': 3, 'I': 1, 'O': 0}

**for** k **in** valores:

print(k)

A

E

I

O

2 – Iterar sobre los valores del diccionario

valores = {'A': 4, 'E': 3, 'I': 1, 'O': 0}

**for** v **in** valores.values():

print(v)

4

3

1

0

3 – Iterar a la vez sobre la clave y el valor de cada uno de los elementos del diccionario.

valores = {'A': 4, 'E': 3, 'I': 1, 'O': 0}

**for** k, v **in** valores.items():

print('k=', k, ', v=', v)

k=A, v=4

k=E, v=3

k=I, v=1

k=O, v=0

## **Python for y la clase range**

¿Cómo implementamos y/o simulamos en Python el bucle for basado en una secuencia numérica? Para estos casos, Python pone a nuestra disposición la clase range (en Python 2 era una función). El constructor de esta clase, range(max), devuelve un iterable cuyos valores van desde 0 hasta max - 1.

**for** i **in** range(11):

print(i)

0

1

2

3

...

10

El tipo de datos range se puede invocar con uno, dos e incluso tres parámetros:

* range(max): Un iterable de números enteros consecutivos que empieza en 0 y acaba en max - 1
* range(min, max): Un iterable de números enteros consecutivos que empieza en min y acaba en max - 1
* range(min, max, step): Un iterable de números enteros consecutivos que empieza en min acaba en max - 1 y los valores se van incrementando de step en step. Este último caso simula el bucle for con variable de control.

Por ejemplo, para mostrar por pantalla los números pares del 0 al 10 podríamos usar la función range del siguiente modo:

**for** num **in** range(0, 11, 2):

print(num)

0

2

4

6

8

10

## **for \_ in <iterable>**

En ocasiones, es posible que el elemento actual del iterable que se está recorriendo sea irrelevante. En estos casos puedes usar la variable \_ para indicar esta situación. No obstante, yo no te lo recomiendo ya que también se suele usar en librerías de traducciones de texto. Aún así, te muestro un caso de uso. Imagínate que queremos implementar una función que devuelva la longitud de una lista. El código podría ser como el que te muestro a continuación:

**def** longitud(mi\_lista):

cont = 0

**for** \_ **in** mi\_lista:

cont += 1

**return** cont

## **Modificando la iteración del bucle for: break y continue**

Por último, vamos a ver que es posible alterar la iteración de un bucle for en Python. Para ello, nos valdremos de las sentencias break y continue. Pero, ¿qué hacen estas sentencias?

* break se utiliza para finalizar y salir el bucle, por ejemplo, si se cumple alguna condición.
* Por su parte, continue salta al siguiente paso de la iteración, ignorando todas las sentencias que le siguen y que forman parte del bucle.

Un ejemplo es la mejor manera de entenderlo.

### **Uso de break. Encontrar un elemento en una colección**

coleccion = [2, 4, 5, 7, 8, 9, 3, 4]

**for** e **in** coleccion:

**if** e == 7:

**break**

print(e)

El código anterior mostrará los números 2, 4 y 5.

### **Uso de continue. Imprimir solo los números pares de una colección**

coleccion = [2, 4, 5, 7, 8, 9, 3, 4]

**for** e **in** coleccion:

**if** e % 2 != 0:

**continue**

print(e)

En este caso, el código anterior mostrará los números 2, 4, 8 y 4.

## **for … else**

En relación al apartado anterior, Python ofrece una estructura adicional de bucle for cuya estructura es la siguiente:

**for** e **in** iterable:

# Tu código aquí

**else**:

# Este código siempre se ejecuta si no

# se ejecutó la sentencia break en el bloque for

Es decir, el código del bloque else se ejecutará siempre y cuando no se haya ejecutado la sentencia break dentro del bloque del for.  
Veamos un ejemplo:

numeros = [1, 2, 4, 3, 5, 8, 6]

**for** n **in** numeros:

**if** n == 3:

**break**

**else**:

print('No se encontró el número 3')

Como en el ejemplo anterior la secuencia numeros contiene al número 3, la instrucción print nunca se ejecutará.

## **Conclusión**

En este tutorial hemos repasado los aspectos más importantes del bucle for en Python. Esta estructura de control es clave para trabajar colecciones.

También hemos visto el concepto de *iterable* e *iterador* en Python. Son los objetos en los que se basa la operativa del bucle for.

Por último, se ha explicado cómo simular en Python un bucle for basado en rango numérico y variable de control y cómo alterar la iteración con las sentencias break y continue.

# list python – Listas en Python: El tipo list y operaciones más comunes

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*list*](https://j2logo.com/tag/list/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

La clase list en Python es una de las más utilizadas por su naturaleza, dinamismo, fácil manejo y potencia. Tengo que decir que el tipo lista de Python me sorprende gratamente y es uno de los grandes aciertos del lenguaje. Descubre en este tutorial qué es el tipo list y cuáles son sus operaciones más comunes para sacarle el máximo provecho.

## **Índice**

* [Qué es una lista](https://j2logo.com/python/tutorial/tipo-list-python/#list-clase)
* [Cómo acceder a los elementos de una lista en Python](https://j2logo.com/python/tutorial/tipo-list-python/#list-elemento)
* [for list Python – Recorrer una lista](https://j2logo.com/python/tutorial/tipo-list-python/#list-for)
* [Añadir elementos a una lista en Python](https://j2logo.com/python/tutorial/tipo-list-python/#list-add)
* [Modificar elementos de una lista](https://j2logo.com/python/tutorial/tipo-list-python/#list-update)
* [Eliminar un elemento de una lista en Python](https://j2logo.com/python/tutorial/tipo-list-python/#list-del)
* [Longitud (len) de una lista en Python](https://j2logo.com/python/tutorial/tipo-list-python/#list-len)
* [Cómo saber si un elemento está en una lista en Python](https://j2logo.com/python/tutorial/tipo-list-python/#list-in)
* [sort list Python – Ordenar una lista en Python](https://j2logo.com/python/tutorial/tipo-list-python/#list-sort)
* [Listado de métodos de la clase list](https://j2logo.com/python/tutorial/tipo-list-python/#list-methods)

## **Qué es una lista**

Las listas en Python son un tipo **contenedor**, compuesto, que **se usan para almacenar conjuntos de elementos relacionados** del mismo tipo o de tipos distintos.

Junto a las clases [*tuple*](https://j2logo.com/python/tutorial/tipo-tuple-python/), [*range*](https://j2logo.com/python/tutorial/tipo-range-python/) y [*str*](https://j2logo.com/python/tutorial/tipo-str-python/), **son uno de los tipos de secuencia en Python**, con la particularidad de que son *mutables*. Esto último quiere decir que su contenido se puede modificar después de haber sido creada.

Para crear una lista en Python, simplemente hay que encerrar una secuencia de elementos separados por comas entre paréntesis cuadrados [].

Por ejemplo, para crear una lista con los números del 1 al 10 se haría del siguiente modo:

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

Como te decía, las listas pueden almacenar elementos de distinto tipo. La siguiente lista también es válida:

>>> elementos = [3, 'a', 8, 7.2, 'hola']

Incluso pueden contener otros elementos compuestos, como objetos u otras listas:

>>> lista = [1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola']

Las listas también se pueden crear usando el constructor de la clase, list(iterable). En este caso, el constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto *iterable* puede ser o una secuencia, un contenedor que soporte la iteración o un objeto iterador.

Por ejemplo, el tipo *str* también es un tipo secuencia. Si pasamos un string al constructor list() creará una lista cuyos elementos son cada uno de los caracteres de la cadena:

>>> vocales = list('aeiou')

>>> vocales

['a', 'e', 'i', 'o', 'u']

Termino esta sección mostrando dos alternativas de crear una lista vacía:

>>> lista\_1 = [] # Opción 1

>>> lista\_2 = list() # Opción 2

## **Cómo acceder a los elementos de una lista en Python**

Para acceder a un elemento de una lista se utilizan los índices. **Un índice es un número entero que indica la posición de un elemento en una lista**. El primer elemento de una lista siempre comienza en el índice 0.

Por ejemplo, en una lista con 4 elementos, los índices de cada uno de los ítems serían 0, 1, 2 y 3.

>>> lista = ['a', 'b', 'd', 'i', 'j']

>>> lista[0] # Primer elemento de la lista. Índice 0

'a'

>>> lista[3] # Cuarto elemento de la lista. Índice 3

'i'

Si se intenta acceder a un índice que está fuera del rango de la lista, el intérprete lanzará la excepción IndexError. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción TypeError:

>>> lista = [1, 2, 3] # Los índices válidos son 0, 1 y 2

>>> lista[8]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

IndexError: list index out **of** range

>>> lista[1.0]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

TypeError: list indices must be integers or slices, not float

Como hemos visto, las listas pueden contener otros elementos de tipo secuencia de forma anidada. Por ejemplo, una lista que uno de sus ítems es otra lista. Del mismo modo, se puede acceder a los elementos de estos tipos usando índices compuestos o anidados:

>>> lista = ['a', ['d', 'b'], 'z']

>>> lista[1][1] # lista[1] hace referencia a la lista anidada

'b'

### **Acceso a los elementos usando un índice negativo**

En Python está permitido usar índices negativos para acceder a los elementos de una secuencia. En este caso, el índice -1 hace referencia al último elemento de la secuencia, el -2 al penúltimo y así, sucesivamente:

>>> vocales = ['a', 'e', 'i', 'o', 'u']

>>> vocales[-1]

'u'

>>> vocales[-4]

'e'

### **Acceso a un subconjunto de elementos**

También es posible acceder a un subconjunto de elementos de una lista utilizando rangos en los índices. Esto es usando el operador [:]:

>>> vocales = ['a', 'e', 'i', 'o', 'u']

>>> vocales[2:3] # Elementos desde el índice 2 hasta el índice 3-1

['i']

>>> vocales[2:4] # Elementos desde el 2 hasta el índice 4-1

['i', 'o']

>>> vocales[:] # Todos los elementos

['a', 'e', 'i', 'o', 'u']

>>> vocales[1:] # Elementos desde el índice 1

['e', 'i', 'o', 'u']

>>> vocales[:3] # Elementos hasta el índice 3-1

['a', 'e', 'i']

También es posible acceder a los elementos de una lista indicando un paso con el operador [::]:

>>> letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']

>>> letras[::2] # Acceso a los elementos de 2 en 2

['a', 'c', 'e', 'g', 'i', 'k']

>>> letras[1:5:2] # Elementos del índice 1 al 4 de 2 en 2

['b', 'd']

>>> letras[1:6:3] # Elementos del índice 1 al 5 de 3 en 3

['b', 'e']

## **for list Python – Recorrer una lista**

Ya hemos visto que se puede usar el bucle [for en Python](https://j2logo.com/bucle-for-en-python/) para recorrer los elementos de una secuencia. En nuestro caso, para recorrer una lista en Python utilizaríamos la siguiente estructura:

>>> colores = ['azul', 'blanco', 'negro']

>>> **for** color **in** colores:

print(color)

azul

blanco

negro

## **Añadir elementos a una lista en Python**

Tal y como te he adelantado, las listas son secuencias mutables, es decir, sus elementos pueden ser modificados (se pueden añadir nuevos ítems, actualizar o eliminar).

Para añadir un nuevo elemento a una lista se utiliza el método append() y para añadir varios elementos, el método extend():

>>> vocales = ['a']

>>> vocales.append('e') # Añade un elemento

>>> vocales

['a', 'e']

>>> vocales.extend(['i', 'o', 'u']) # Añade un grupo de elementos

>>> vocales

['a', 'e', 'i', 'o', 'u']

También es posible utilizar el operador de concatenación + para unir dos listas en una sola. El resultado es una nueva lista con los elementos de ambas:

>>> lista\_1 = [1, 2, 3]

>>> lista\_2 = [4, 5, 6]

>>> nueva\_lista = lista\_1 + lista\_2

>>> nueva\_lista

[1, 2, 3, 4, 5, 6]

Por otro lado, el operador \* repite el contenido de una lista n veces:

>>> numeros = [1, 2, 3]

>>> numeros \*= 3

>>> numeros

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

Y para terminar esta sección, indicarte que también es posible añadir un elemento en una posición concreta de una lista con el método insert(índice, elemento). Los elementos cuyo índice sea mayor a índice se desplazan una posición a la derecha:

>>> vocales = ['a', 'e', 'u']

>>> vocales.insert(2, 'i')

>>> vocales

['a', 'e', 'i', 'u']

## **Modificar elementos de una lista**

Es posible modificar un elemento de una lista en Python con el operador de asignación =. Para ello, lo único que necesitas conocer es el índice del elemento que quieres modificar o el rango de índices:

>>> vocales = ['o', 'o', 'o', 'o', 'u']

# Actualiza el elemento del índice 0

>>> vocales[0] = 'a'

>>> vocales

['a', 'o', 'o', 'o', 'u']

# Actualiza los elementos entre las posiciones 1 y 2

>>> vocales[1:3] = ['e', 'i']

>>> vocales

['a', 'e', 'i', 'o', 'u']

## **Eliminar un elemento de una lista en Python**

En Python se puede eliminar un elemento de una lista de varias formas.

Con la sentencia del se puede eliminar un elemento a partir de su índice:

# Elimina el elemento del índice 1

>>> vocales = ['a', 'e', 'i', 'o', 'u']

>>> del vocales[1]

>>> vocales

['a', 'i', 'o', 'u']

# Elimina los elementos con índices 2 y 3

>>> vocales = ['a', 'e', 'i', 'o', 'u']

>>> del vocales[2:4]

>>> vocales

['a', 'e', 'u']

# Elimina todos los elementos

>>> del vocales[:]

>>> vocales

[]

Además de la sentencia del, podemos usar los métodos remove() y pop([i]). remove() elimina la primera ocurrencia que se encuentre del elemento en una lista. Por su parte, pop([i]) obtiene el elemento cuyo índice sea igual a i y lo elimina de la lista. Si no se especifica ningún índice, recupera y elimina el último elemento.

>>> letras = ['a', 'b', 'k', 'a', 'v']

# Elimina la primera ocurrencia del carácter a

>>> letras.remove('a')

>>> letras

['b', 'k', 'a', 'v']

# Obtiene y elimina el último elemento

>>> letras.pop()

'v'

>>> letras

['b', 'k', 'a']

Finalmente, es posible eliminar todos los elementos de una lista a través del método clear():

>>> letras = ['a', 'b', 'c']

>>> letras.clear()

>>> letras

[]

El código anterior sería equivalente a del letras[:].

## **Longitud (len) de una lista en Python**

Como cualquier tipo secuencia, para conocer la longitud de una lista en Python se hace uso de la función len(). Esta función devuelve el número de elementos de una lista:

>>> vocales = ['a', 'e', 'i', 'o', 'u']

>>> len(vocales)

5

## **Cómo saber si un elemento está en una lista en Python**

Ya adelanté en el tutorial sobre [operadores en Python](https://j2logo.com/python/tutorial/operadores-en-python/), que para saber si un elemento está contenido en una lista, se utiliza el operador de pertenencia in:

>>> vocales = ['a', 'e', 'i', 'o', 'u']

>>> **if** 'a' **in** vocales:

... print('Sí')

...

Sí

>>> **if** 'b' not **in** vocales:

... print('No')

...

No

## **sort list Python – Ordenar una lista en Python**

Las listas son secuencias ordenadas. Esto quiere decir que sus elementos siempre se devuelven en el mismo orden en que fueron añadidos.

No obstante, es posible ordenar los elementos de una lista con el método sort(). El método sort() ordena los elementos de la lista utilizando únicamente el operador < y modifica la lista actual (no se obtiene una nueva lista):

# Lista desordenada de números enteros

>>> numeros = [3, 2, 6, 1, 7, 4]

# Identidad del objeto numeros

>>> id(numeros)

4475439216

# Se llama al método sort() para ordenar los elementos de la lista

>>> numeros.sort()

>>> numeros

[1, 2, 3, 4, 6, 7]

# Se comprueba que la identidad del objeto numeros es la misma

>>> id(numeros)

4475439216

## **Listado de métodos de la clase list**

Termino este tutorial mostrando la lista completa de métodos de la clase list. Algunos de ellos ya se han mencionado en las secciones anteriores.

|  |  |
| --- | --- |
| Método | Descripción |
| append() | Añade un nuevo elemento al final de la lista. |
| extend() | Añade un grupo de elementos (iterables) al final de la lista. |
| insert(indice, elemento) | Inserta un elemento en una posición concreta de la lista. |
| remove(elemento) | Elimina la primera ocurrencia del elemento en la lista. |
| pop([i]) | Obtiene y elimina el elemento de la lista en la posición i. Si no se especifica, obtiene y elimina el último elemento. |
| clear() | Borra todos los elementos de la lista. |
| index(elemento) | Obtiene el índice de la primera ocurrencia del elemento en la lista. Si el elemento no se encuentra, se lanza la excepción ValueError. |
| count(elemento) | Devuelve el número de ocurrencias del elemento en la lista. |
| sort() | Ordena los elementos de la lista utilizando el operador <. |
| reverse() | Obtiene los elementos de la lista en orden inverso. |
| copy() | Devuelve una copia poco profunda de la lista. |

# El tipo tuple y operaciones más comunes

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tuple*](https://j2logo.com/tag/tuple/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

La clase tuple en Python es uno de los tipos de datos más singulares del lenguaje. Pertenece a la categoría de tipos secuenciales y, a diferencia del tipo [*list*](https://j2logo.com/python/tutorial/tipo-list-python/), **las tuplas son un tipo de secuencia inmutable**. Esto quiere decir que una tupla no puede ser modificada (no se pueden añadir ni eliminar elementos a una tupla).

Este tutorial describe al detalle qué es el tipo tuple y cuáles son sus operaciones más comunes. ¿Quieres conocerlas?

## **Índice**

* [Qué es una tupla](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-clase)
* [Cómo acceder a los elementos de una tupla en Python](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-elemento)
* [for tuple Python – Recorrer una tupla](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-for)
* [Modificar una tupla en Python](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-update)
* [Longitud (len) de una tupla en Python](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-len)
* [Cómo saber si un elemento está en una tupla en Python](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-in)
* [Listado de métodos de la clase tuple en Python](https://j2logo.com/python/tutorial/tipo-tuple-python/#tuple-methods)

## **Qué es una tupla**

La clase tuple en Python es un tipo **contenedor**, compuesto, que **en un principio se pensó para almacenar grupos de elementos heterogéneos**, aunque también puede contener elementos homogéneos.

Junto a las clases *list* y *range*, **es uno de los tipos de secuencia en Python**, con la particularidad de que son *inmutables*. Esto último quiere decir que su contenido **NO** se puede modificar después de haber sido creada.

En general, para crear una tupla en Python simplemente hay que definir una secuencia de elementos separados por comas.

Por ejemplo, para crear una tupla con los números del 1 al 5 se haría del siguiente modo:

>>> numeros = 1, 2, 3, 4, 5

Como te indicaba, la clase *tuple* también puede almacenar elementos de distinto tipo:

>>> elementos = 3, 'a', 8, 7.2, 'hola'

Incluso pueden contener otros elementos compuestos y objetos, como listas, otras tuplas, etc.:

>>> tup = 1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola'

A continuación te indico las diferentes formas que existen de crear una tupla en Python:

* Para crear una tupla vacía, usa paréntesis () o el constructor de la clase tuple() sin parámetros.
* Para crear una tupla con un único elemento: elem, o (elem, ). Observa que siempre se añade una coma.
* Para crear una tupla de varios elementos, sepáralos con comas: a, b, c o (a, b, c).
* Las tuplas también se pueden crear usando el constructor de la clase, tuple(iterable). En este caso, el constructor crea una tupla cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto *iterable* puede ser una secuencia, un contenedor que soporte la iteración o un objeto iterador.

❗️**IMPORTANTE:** El hecho que determina que una secuencia de elementos sea una tupla es la coma , no los paréntesis. Los paréntesis son opcionales y solo se necesitan para crear una tupla vacía o para evitar ambigüedades.

# Aquí, a, b y c no son una tupla, sino tres argumentos con

# los que se llama a la función "una\_funcion"

>>> una\_funcion(a, b, c)

# Aquí, a, b y c son tres elementos de una tupla. Esta tupla,

# es el único argumento con el que se invoca a la

# función "una\_funcion"

>>> una\_funcion((a, b, c))

La forma de crear una tupla sin paréntesis es conocida como *tuple packing* (algo así como empaquetado de tuplas).

## **Cómo acceder a los elementos de una tupla en Python**

Para acceder a un elemento de una tupla se utilizan los índices. **Un índice es un número entero que indica la posición de un elemento en una tupla**. El primer elemento de una tupla siempre comienza en el índice 0.

Por ejemplo, en una tupla con 3 elementos, los índices de cada uno de los ítems serían 0, 1 y 2.

>>> tupla = ('a', 'b', 'd')

>>> tupla[0] # Primer elemento de la tupla. Índice 0

'a'

>>> tupla[1] # Segundo elemento de la tupla. Índice 1

'b'

Si se intenta acceder a un índice que está fuera del rango de la tupla, el intérprete lanzará la excepción IndexError. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción TypeError:

>>> tupla = 1, 2, 3 # Los índices válidos son 0, 1 y 2

>>> tupla[8]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

IndexError: tuple index out **of** range

>>> tupla[1.0]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

TypeError: tuple indices must be integers or slices, not float

### **Acceso a los elementos usando un índice negativo**

Al igual que ocurre con las listas (y todos los tipos secuenciales), está permitido usar índices negativos para acceder a los elementos de una tupla. En este caso, el índice -1 hace referencia al último elemento de la secuencia, el -2 al penúltimo y así, sucesivamente:

>>> bebidas = ('agua', 'café', 'batido', 'sorbete')

>>> bebidas[-1]

'sorbete'

>>> bebidas[-3]

'café'

### **Acceso a un subconjunto de elementos**

También es posible acceder a un subconjunto de elementos de una tupla utilizando el operador [:]:

>>> vocales = 'a', 'e', 'i', 'o', 'u'

>>> vocales[2:3] # Elementos desde el índice 2 hasta el índice 3-1

('i',)

>>> vocales[2:4] # Elementos desde el 2 hasta el índice 4-1

('i', 'o')

>>> vocales[:] # Todos los elementos

('a', 'e', 'i', 'o', 'u')

>>> vocales[1:] # Elementos desde el índice 1

('e', 'i', 'o', 'u')

>>> vocales[:3] # Elementos hasta el índice 3-1

('a', 'e', 'i')

O indicando un salto entre los elementos con el operador [::]:

>>> pares = 2, 4, 6, 8, 10, 12, 14

>>> pares[::2] # Acceso a los elementos de 2 en 2

(2, 6, 10, 14)

>>> pares[1:5:2] # Elementos del índice 1 al 4 de 2 en 2

(4, 8)

>>> pares[1:6:3] # Elementos del índice 1 al 5 de 3 en 3

(4, 10)

### **tuple unpacking**

En este último apartado de esta sección vamos a ver el concepto conocido como *tuple unpacking* (desempaquetado de una tupla). Realmente el *unpacking* se puede aplicar sobre cualquier objeto de tipo secuencia, aunque se usa mayoritariamente con las tuplas, y consiste en lo siguiente:

>>> bebidas = 'agua', 'café', 'batido'

>>> a, b, c = bebidas

>>> a

'agua'

>>> b

'café'

>>> c

'batido'

Como puedes apreciar, es un tipo de asignación múltiple. Requiere que haya tantas variables a la izquierda del operador de asignación = como elementos haya en la secuencia.

## **for tuple Python – Recorrer una tupla**

El bucle [for en Python](https://j2logo.com/bucle-for-en-python/) es una de las estructuras ideales para iterar sobre los elementos de una secuencia. Para recorrer una tupla en Python utiliza la siguiente estructura:

>>> colores = 'azul', 'blanco', 'negro'

>>> **for** color **in** colores:

print(color)

azul

blanco

negro

## **Modificar una tupla en Python**

Como te he indicado, las tuplas son objetos inmutables. No obstante, las tuplas pueden contener objetos u otros elementos de tipo secuencia, por ejemplo, una lista. Estos objetos, si son mutables, sí se pueden modificar:

>>> tupla = (1, ['a', 'b'], 'hola', 8.2)

>>> tupla[1].append('c') # tupla[1] hace referencia a la lista

>>> tupla

(1, ['a', 'b', 'c'], 'hola', 8.2)

## **Longitud (len) de una tupla en Python**

Como cualquier tipo secuencia, para conocer la longitud de una tupla en Python se hace uso de la función len(). Esta función devuelve el número de elementos de una tupla:

>>> vocales = ('a', 'e', 'i', 'o', 'u')

>>> len(vocales)

5

## **Cómo saber si un elemento está en una tupla en Python**

Ya adelanté en el tutorial sobre [operadores en Python](https://j2logo.com/python/tutorial/operadores-en-python/), que para saber si un elemento está contenido en una tupla, se utiliza el operador de pertenencia in:

>>> colores = 'azul', 'blanco', 'negro'

>>> **if** 'azul' **in** colores:

... print('Sí')

...

Sí

>>> **if** 'verde' not **in** colores:

... print('No')

...

No

## **Listado de métodos de la clase tuple en Python**

Para terminar, se muestran los métodos de la clase tuple en Python, que son los métodos definidos para cualquier tipo secuencial:

|  |  |
| --- | --- |
| Método | Descripción |
| index(elemento) | Obtiene el índice de la primera ocurrencia del elemento en la tupla. Si el elemento no se encuentra, se lanza la excepción ValueError. |
| count(elemento) | Devuelve el número de ocurrencias del elemento en la tupla. |

Con esto damos por finalizado el tutorial sobre la clase *tuple* en Python. Recuerda que son un tipo secuencial que no se puede modificar. Úsalas cuando no quieras que los elementos de una lista se deban modificar o para almacenar las claves de un diccionario.

# range python – El tipo range en Python: Secuencias numéricas

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*range*](https://j2logo.com/tag/range/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

*range* en Python es un tipo que se utiliza para representar una secuencia inmutable de números. Uno de sus principales usos es junto a la sentencia *for*, para definir un bucle sobre el que se itera un número determinado de veces.

Descubre en este tutorial la clase *range* y sus principales usos y operaciones.

## **Índice**

* [La clase range en Python](https://j2logo.com/python/tutorial/tipo-range-python/#range-intro)
* [Recorrer una secuencia numérica](https://j2logo.com/python/tutorial/tipo-range-python/#range-for)
* [Ventajas de usar range en Python](https://j2logo.com/python/tutorial/tipo-range-python/#range-adv)
* [Operaciones de la clase range](https://j2logo.com/python/tutorial/tipo-range-python/#range-operaciones)

## **La clase range en Python**

Como te adelantaba en la introducción, *range* es un tipo de dato que representa una secuencia de números inmutable.

Para crear un objeto de tipo range, se pueden usar dos constructores :

* range(fin): Crea una secuencia numérica que va desde 0 hasta fin - 1.
* range(inicio, fin, [paso]): Crea una secuencia numérica que va desde inicio hasta fin - 1. Si además se indica el parámetro paso, la secuencia genera los números de paso en paso.

Veámoslo con un ejemplo:

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(5, 10))

[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))

[0, 3, 6, 9]

>>> list(range(0, -10, -1))

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

>>> list(range(5, -5, -2))

[5, 3, 1, -1, -3]

¿Por qué estoy utilizando en los ejemplos la clase range como argumento de list? range, además de un tipo *secuencial* es un tipo *iterable* con una particularidad: a diferencia de los tipos [list](https://j2logo.com/python/tutorial/tipo-list-python/) o [tuple](https://j2logo.com/python/tutorial/tipo-tuple-python/), range **calcula los ítems al vuelo, cuando los necesita**. Como *list* acepta un objeto *iterable* como parámetro, por eso paso un objeto *range* al constructor de *list*, para que se muestre por pantalla la secuencia completa que se genera con *range*.

❗️**NOTA:** Si se llama al constructor range() con parámetros incorrectos, se obtendrá un objeto vacío.

# Nunca se puede ir de 0 a 10 de -1 en -1

>>> list(range(0, 10, -1))

[]

## **Recorrer una secuencia numérica**

En la introducción te adelanté que uno de los principales usos de range() es utilizarlo en bucles for. Aquí puedes ver todos los detalles del uso de [for y range en Python](https://j2logo.com/bucle-for-en-python/#for-range).

>>> **for** i **in** range(1, 11):

... print(i)

...

1

2

3

4

5

6

7

8

9

10

## **Ventajas de usar range en Python**

La principal ventaja de usar *range* sobre *list* o *tuple* es que es un iterable que genera los elementos solo en el momento en que realmente los necesita. Esto implica que usa una cantidad de memoria mínima, por muy grande que sea el rango de números que represente.

Veamos una comparación de una lista que almacena los números del 0 al 100.000 y un rango del 0 al 100.000:

>>> import sys

>>> lista = list(range(0, 100000))

>>> rango = range(0, 100000)

>>> sys.getsizeof(lista)

900120

>>> sys.getsizeof(rango)

48

Como puedes apreciar, la lista ocupa casi *1 MB* en memoria frente a los *48 bytes* que ocupa el rango.

## **Operaciones de la clase range**

Al tratarse de un tipo secuencial, *range* implementa las operaciones básicas de ese tipo a excepción de la concatenación y la repetición:

>>> r = range(0,30, 3)

>>> r[2]

6

>>> r[-1]

27

>>> 13 **in** r

**False**

>>> 12 **in** r

**True**

>>> r.index(18)

6

Bueno, pues con esto, hemos llegado al final de este tutorial. Recuerda, usa la clase *range* para generar secuencias de números, especialmente si las necesitas en un bucle *for*.

# set python – Conjuntos en Python: El tipo set y operaciones más comunes

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*set*](https://j2logo.com/tag/set/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

El tipo set en Python es la clase utilizada por el lenguaje para representar los conjuntos. Un conjunto es una colección desordenada de elementos únicos, es decir, que no se repiten.

Este tutorial describe en detalle la clase set de Python, sus principales usos y operaciones. ¡No te lo pierdas!

## **Índice**

* [Qué es el tipo set en Python](https://j2logo.com/python/tutorial/tipo-set-python/#set-clase)
* [Cómo acceder a los elementos de un conjunto en Python](https://j2logo.com/python/tutorial/tipo-set-python/#set-elemento)
* [Añadir elementos a un conjunto (set) en Python](https://j2logo.com/python/tutorial/tipo-set-python/#set-add)
* [Eliminar un elemento de un conjunto en Python](https://j2logo.com/python/tutorial/tipo-set-python/#set-del)
* [Número de elementos (len) de un conjunto](https://j2logo.com/python/tutorial/tipo-set-python/#set-len)
* [Cómo saber si un elemento está en un conjunto](https://j2logo.com/python/tutorial/tipo-set-python/#set-in)
* [Operaciones sobre conjuntos en Python (set operations)](https://j2logo.com/python/tutorial/tipo-set-python/#set-op)
* [Métodos de la clase set en Python](https://j2logo.com/python/tutorial/tipo-set-python/#set-methods)

## **Qué es el tipo set en Python**

Al comienzo del tutorial adelantaba que el tipo set en Python es utilizado para trabajar con conjuntos de elementos. La principal característica de este tipo de datos es que es una colección cuyos elementos no guardan ningún orden y que además son únicos.

Estas características hacen que los principales usos de esta clase sean conocer si un elemento pertenece o no a una colección y eliminar duplicados de un tipo secuencial ([*list*](https://j2logo.com/python/tutorial/tipo-list-python/), [*tuple*](https://j2logo.com/python/tutorial/tipo-tuple-python/) o [*str*](https://j2logo.com/python/tutorial/tipo-str-python/)).

Además, esta clase también implementa las típicas operaciones matemáticas sobre conjuntos: *unión*, *intersección*, *diferencia*, …

Para crear un conjunto, basta con encerrar una serie de elementos entre llaves {}, o bien usar el constructor de la clase set() y pasarle como argumento un objeto *iterable* (como una *lista*, una *tupla*, una *cadena* …).

# Crea un conjunto con una serie de elementos entre llaves

# Los elementos repetidos se eliminan

>>> c = {1, 3, 2, 9, 3, 1}

>>> c

{1, 2, 3, 9}

# Crea un conjunto a partir de un string

# Los caracteres repetidos se eliminan

>>> a = set('Hola Pythonista')

>>> a

{'a', 'H', 'h', 'y', 'n', 's', 'P', 't', ' ', 'i', 'l', 'o'}

# Crea un conjunto a partir de una lista

# Los elementos repetidos de la lista se eliminan

>>> unicos = set([3, 5, 6, 1, 5])

>>> unicos

{1, 3, 5, 6}

Para crear un conjunto vacío, simplemente llama al constructor set() sin parámetros.

❗️ **IMPORTANTE:** {} NO crea un conjunto vacío, sino un *diccionario* vacío. Usa set() si quieres crear un conjunto sin elementos.

🎯 **NOTA:** Los elementos que se pueden añadir a un conjunto deben ser de tipo *hashable*. Un objeto es *hashable* si tiene un valor de *hash* que no cambia durante todo su ciclo de vida. En principio, los objetos que son instancias de clases definidas por el usuario son *hashables*. También lo son la mayoría de tipos inmutables definidos por Python.

### **set vs frozenset**

En realidad, en Python existen dos clases para representar conjuntos: set y frozenset. La principal diferencia es que *set* es mutable, por lo que después de ser creado, se pueden añadir y/o eliminar elementos del conjunto, como veremos en secciones posteriores. Por su parte, *frozenset* es inmutable y su contenido no puede ser modificado una vez que ha sido inicializado.

Para crear un conjunto de tipo *frozenset*, se usa el constructor de la clase frozenset():

>>> f = frozenset([3, 5, 6, 1, 5])

>>> f

frozenset({1, 3, 5, 6})

🎯 **NOTA:** El único modo en Python de tener un conjunto de conjuntos es utilizando objetos de tipo *frozenset* como elementos del propio conjunto.

## **Cómo acceder a los elementos de un conjunto en Python**

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos [list](https://j2logo.com/python/tutorial/tipo-list-python/) o [tuple](https://j2logo.com/python/tutorial/tipo-tuple-python/). Es por ello que no se puede acceder a los elementos a través de un índice.

Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle for:

>>> mi\_conjunto = {1, 3, 2, 9, 3, 1}

>>> **for** e **in** mi\_conjunto:

... print(e)

...

1

2

3

9

## **Añadir elementos a un conjunto (set) en Python**

Para añadir un elemento a un conjunto se utiliza el método add(). También existe el método update(), que puede tomar como argumento una *lista*, *tupla*, *string*, *conjunto* o cualquier objeto de tipo *iterable*.

>>> mi\_conjunto = {1, 3, 2, 9, 3, 1}

>>> mi\_conjunto

{1, 2, 3, 9}

# Añade el elemento 7 al conjunto

>>> mi\_conjunto.add(7)

>>> mi\_conjunto

{1, 2, 3, 7, 9}

# Añade los elementos 5, 3, 4 y 6 al conjunto

# Los elementos repetidos no se añaden al conjunto

>>> mi\_conjunto.update([5, 3, 4, 6])

>>> mi\_conjunto

{1, 2, 3, 4, 5, 6, 7, 9}

🎯 **NOTA:** add() y update() no añaden elementos que ya existen al conjunto.

## **Eliminar un elemento de un conjunto en Python**

La clase *set* ofrece cuatro métodos para eliminar elementos de un conjunto. Son: discard(), remove(), pop() y clear(). A continuación te explico qué hace cada uno de ellos.

discard(elemento) y remove(elemento) eliminan elemento del conjunto. La única diferencia es que si elemento no existe, discard() no hace nada mientras que remove() lanza la excepción KeyError.

pop() es un tanto peculiar. Este método devuelve un elemento aleatorio del conjunto y lo elimina del mismo. Si el conjunto está vacío, lanza la excepción KeyError.

Finalmente, clear() elimina todos los elementos contenidos en el conjunto.

>>> mi\_conjunto = {1, 3, 2, 9, 3, 1, 6, 4, 5}

>>> mi\_conjunto

{1, 2, 3, 4, 5, 6, 9}

# Elimina el elemento 1 con remove()

>>> mi\_conjunto.remove(1)

>>> mi\_conjunto

{2, 3, 4, 5, 6, 9}

# Elimina el elemento 4 con discard()

>>> mi\_conjunto.discard(4)

>>> mi\_conjunto

{2, 3, 5, 6, 9}

# Trata de eliminar el elemento 7 (no existe) con remove()

# Lanza la excepción KeyError

>>> mi\_conjunto.remove(7)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

KeyError: 7

# Trata de eliminar el elemento 7 (no existe) con discard()

# No hace nada

>>> mi\_conjunto.discard(7)

>>> mi\_conjunto

{2, 3, 5, 6, 9}

# Obtiene y elimina un elemento aleatorio con pop()

>>> mi\_conjunto.pop()

2

>>> mi\_conjunto

{3, 5, 6, 9}

# Elimina todos los elementos del conjunto

>>> mi\_conjunto.clear()

>>> mi\_conjunto

set()

## **Número de elementos (len) de un conjunto**

Como con cualquier otra colección, puedes usar la función len() para obtener el número de elementos contenidos en un conjunto:

>>> mi\_conjunto = set([1, 2, 5, 3, 1, 5])

>>> len(mi\_conjunto)

4

## **Cómo saber si un elemento está en un conjunto**

Con los conjuntos también se puede usar el operador de pertenencia in para comprobar si un elemento está contenido, o no, en un conjunto:

>>> mi\_conjunto = set([1, 2, 5, 3, 1, 5])

>>> print(1 **in** mi\_conjunto)

**True**

>>> print(6 **in** mi\_conjunto)

**False**

>>> print(2 not **in** mi\_conjunto)

False

## **Operaciones sobre conjuntos en Python (set operations)**

Tal y como te adelanté al comienzo de este tutorial, uno de los principales usos del tipo set es utilizarlo en operaciones del álgebra de conjuntos: *unión*, *intersección*, *diferencia*, *diferencia simétrica*, …

A continuación veremos cómo llevar a cabo estas operaciones en Python.

### **Unión de conjuntos en Python**

La unión de dos conjuntos A y B es el conjunto A ∪ B que contiene todos los elementos de A y de B.

En Python se utiliza el operador | para realizar la unión de dos o más conjuntos.

>>> a = {1, 2, 3, 4}

>>> b = {2, 4, 6, 8}

>>> a | b

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

### **Intersección de conjuntos en Python**

La intersección de dos conjuntos A y B es el conjunto A ∩ B que contiene todos los elementos comunes de A y B.

En Python se utiliza el operador & para realizar la intersección de dos o más conjuntos.

>>> a = {1, 2, 3, 4}

>>> b = {2, 4, 6, 8}

>>> a & b

{2, 4}

### **Diferencia de conjuntos en Python**

La diferencia entre dos conjuntos A y B es el conjunto A \ B que contiene todos los elementos de A que no pertenecen a B.

En Python se utiliza el operador - para realizar la diferencia de dos o más conjuntos.

>>> a = {1, 2, 3, 4}

>>> b = {2, 4, 6, 8}

>>> a - b

{1, 3}

### **Diferencia simétrica de conjuntos en Python**

La diferencia simétrica entre dos conjuntos A y B es el conjunto que contiene los elementos de A y B que no son comunes.

En Python se utiliza el operador ^ para realizar la diferencia simétrica de dos o más conjuntos.

>>> a = {1, 2, 3, 4}

>>> b = {2, 4, 6, 8}

a ^ b

{1, 3, 6, 8}

### **Inclusión de conjuntos en Python**

Dado un conjunto A, subcolección del conjunto B o igual a este, sus elementos son un subconjunto de B. Es decir, A es un subconjunto de B y B es un superconjunto de A.

En Python se utiliza el operador <= para comprobar si un conjunto A es subconjunto de B y el operador >= para comprobar si un conjunto A es superconjunto de B.

>>> a = {1, 2}

>>> b = {1, 2, 3, 4}

>>> a <= b

**True**

>>> a >= b

**False**

>>> b >= a

**True**

>>> a = {1, 2}

>>> b = {1, 2}

>>> a < b # Ojo al operador < sin el =

**False**

>>> a <= b

True

### **Conjuntos disjuntos en Python**

Dos conjuntos A y B son disjuntos si no tienen elementos en común, es decir, la intersección de A y B es el conjunto vacío.

En Python se utiliza el método isdisjoint() de la clase set para comprobar si un conjunto es disjunto de otro.

>>> a = {1, 2}

>>> b = {1, 2, 3, 4}

>>> a.isdisjoint(b)

**False**

>>> a = {1, 2}

>>> b = {3, 4}

>>> a.isdisjoint(b)

True

### **Igualdad de conjuntos en Python**

En Python dos conjuntos son iguales si y solo si todos los elementos de un conjunto están contenidos en el otro. Esto quiere decir que cada uno es un subconjunto del otro.

>>> a = {1, 2}

>>> b = {1, 2}

>>> id(a)

4475070656

>>> id(b)

4475072096

>>> a == b

True

## **Métodos de la clase set en Python**

Termino este tutorial listando los métodos principales de la clase *set* en Python:

|  |  |
| --- | --- |
| Método | Descripción |
| add(e) | Añade un elemento al conjunto. |
| clear() | Elimina todos los elementos del conjunto. |
| copy() | Devuelve una copia superficial del conjunto. |
| difference(iterable) | Devuelve la diferencia del conjunto con el iterable como un conjunto nuevo. |
| difference\_update(iterable) | Actualiza el conjunto tras realizar la diferencia con el iterable. |
| discard(e) | Elimina, si existe, el elemento del conjunto. |
| intersection(iterable) | Devuelve la intersección del conjunto con el iterable como un conjunto nuevo. |
| intersection\_update(iterable) | Actualiza el conjunto tras realizar la intersección con el iterable. |
| isdisjoint(iterable) | Devuelve True si dos conjuntos son disjuntos. |
| issubset(iterable) | Devuelve True si el conjunto es subconjunto del iterable. |
| issuperset(iterable) | Devuelve True si el conjunto es superconjunto del iterable. |
| pop() | Obtiene y elimina un elemento de forma aleatoria del conjunto. |
| remove(e) | Elimina el elemento del conjunto. Si no existe lanza un error. |
| symmetric\_difference(iterable) | Devuelve la diferencia simétrica del conjunto con el iterable como un conjunto nuevo. |
| symmetric\_difference\_update(iterable) | Actualiza el conjunto tras realizar la diferencia simétrica con el iterable. |
| union(iterable) | Devuelve la unión del conjunto con el iterable como un conjunto nuevo. |
| update(iterable) | Actualiza el conjunto tras realizar la unión con el iterable. |

❗️ **NOTA:** Los operadores |, &, … toman siempre como operandos objetos de tipo set. Sin embargo, sus respectivas versiones como métodos union(), intersection(), … toman como argumentos un *iterable* (*lista*, *tupla*, *conjunto*, etc.).

# dict python – Diccionarios en Python: El tipo diccionario y operaciones más comunes

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*dict*](https://j2logo.com/tag/dict/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

Yo diría que la clase dict en Python es, junto con [list](https://j2logo.com/python/tutorial/tipo-list-python/), una de las más importantes del lenguaje.

En este tutorial te describo con todo detalle la clase dict de Python para que le saques todo el partido y no haya diccionario que se te resista. ¡Vamos!

## **Índice**

* [Qué es el tipo dict en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-clase)
* [Cómo acceder a los elementos de un diccionario en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-elemento)
* [for dict Python – Recorrer un diccionario](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-for)
* [Añadir elementos a un diccionario en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-add)
* [Modificar elementos de un diccionario](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-update)
* [Eliminar un elemento de un diccionario en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-del)
* [Número de elementos (len) de un diccionario en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-len)
* [Comprobar si un elemento está en un diccionario en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-in)
* [Comparar si dos diccionarios son iguales](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-equal)
* [Diccionarios anidados en Python](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-nested)
* [Obtener una lista con las claves de un diccionario](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-keys)
* [Objetos vista de un diccionario](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-view)
* [Listado de métodos de la clase dict](https://j2logo.com/python/tutorial/tipo-dict-python/#dict-methods)

## **Qué es el tipo dict en Python**

La clase dict de Python es un tipo mapa que asocia claves a valores. A diferencia de los tipos secuenciales ([*list*](https://j2logo.com/python/tutorial/tipo-list-python/), [*tuple*](https://j2logo.com/python/tutorial/tipo-tuple-python/), [*range*](https://j2logo.com/python/tutorial/tipo-range-python/) o [*str*](https://j2logo.com/python/tutorial/tipo-str-python/)), que son indexados por un índice numérico, los diccionarios son indexados por claves. Estas claves siempre deben ser de un tipo inmutable, concretamente un tipo *hashable*.

🎯 **NOTA:** Un objeto es *hashable* si tiene un valor de *hash* que no cambia durante todo su ciclo de vida. En principio, los objetos que son instancias de clases definidas por el usuario son *hashables*. También lo son la mayoría de tipos inmutables definidos por Python (*int*, *float* o *str*).

Piensa siempre en un diccionario como un contenedor de pares *clave*: *valor*, en el que la clave puede ser de cualquier tipo hashable y es única en el diccionario que la contiene. Generalmente, se suelen usar como claves los tipos *int* y *str* aunque, como te he dicho, cualquier tipo *hashable* puede ser una clave.

**Las principales operaciones** que se suelen realizar con diccionarios **son almacenar un valor asociado a una clave y recuperar un valor a partir de una clave**. Esta es la esencia de los diccionarios y es aquí donde son realmente importantes. **En un diccionario, el acceso a un elemento a partir de una clave es una operación realmente rápida, eficaz y que consume pocos recursos** si lo comparamos con cómo lo haríamos con otros tipos de datos.

Otras características a resaltar de los diccionarios:

* **Es un tipo mutable**, es decir, su contenido se puede modificar después de haber sido creado.
* **Es un tipo ordenado**. Preserva el orden en que se insertan los pares *clave: valor*.

### **Cómo crear un diccionario**

En Python hay varias formas de crear un diccionario. Las veremos todas a continuación.

La más simple es encerrar una secuencia de pares *clave: valor* separados por comas entre llaves {}

>>> d = {1: 'hola', 89: 'Pythonista', 'a': 'b', 'c': 27}

En el diccionario anterior, los enteros 1 y 89 y las cadenas 'a' y 'c' son las claves. Como ves, se pueden mezclar claves y valores de distinto tipo sin problema.

Para crear un diccionario vacío, simplemente asigna a una variable el valor {}.

También se puede usar el constructor de la clase dict() de varias maneras:

* **Sin parámetros**. Esto creará un diccionario vacío.
* Con pares *clave: valor* encerrados entre llaves.
* **Con argumentos con nombre**. El nombre del argumento será la clave en el diccionario. En este caso, las claves solo pueden ser identificadores válidos y mantienen el orden en el que se indican. No se podría, por ejemplo, tener números enteros como claves.
* **Pasando un iterable**. En este caso, cada elemento del iterable debe ser también un iterable con solo dos elementos. El primero se toma como clave del diccionario y el segundo como valor. Si la clave aparece varias veces, el valor que prevalece es el último.

Veamos un ejemplo con todo lo anterior. Vamos a crear el mismo diccionario de todos los modos que te he explicado:

# 1. Pares clave: valor encerrados entre llaves

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

>>> d

{'uno': 1, 'dos': 2, 'tres': 3}

# 2. Argumentos con nombre

>>> d2 = dict(uno=1, dos=2, tres=3)

>>> d2

{'uno': 1, 'dos': 2, 'tres': 3}

# 3. Pares clave: valor encerrados entre llaves

>>> d3 = dict({'uno': 1, 'dos': 2, 'tres': 3})

>>> d3

{'uno': 1, 'dos': 2, 'tres': 3}

# 4. Iterable que contiene iterables con dos elementos

>>> d4 = dict([('uno', 1), ('dos', 2), ('tres', 3)])

>>> d4

{'uno': 1, 'dos': 2, 'tres': 3}

# 5. Diccionario vacío

>>> d5 = {}

>>> d5

{}

# 6. Diccionario vacío usando el constructor

>>> d6 = dict()

>>> d6

{}

## **Cómo acceder a los elementos de un diccionario en Python**

Acceder a un elemento de un diccionario es una de las principales operaciones por las que existe este tipo de dato. El acceso a un valor se realiza mediante indexación de la clave. Para ello, simplemente encierra entre corchetes la clave del elemento d[clave]. En caso de que la clave no exista, se lanzará la excepción KeyError.

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

>>> d['dos']

2

>>> d[4]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

KeyError: 4

La clase dict también ofrece el método get(clave[, valor por defecto]). Este método devuelve el valor correspondiente a la clave clave. En caso de que la clave no exista no lanza ningún error, sino que devuelve el segundo argumento valor por defecto. Si no se proporciona este argumento, se devuelve el valor None.

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

>>> d.get('uno')

1

# Devuelve 4 como valor por defecto si no encuentra la clave

>>> d.get('cuatro', 4)

4

# Devuelve None como valor por defecto si no encuentra la clave

>>> a = d.get('cuatro')

>>> a

>>> type(a)

<**class** 'NoneType'>

## **for dict Python – Recorrer un diccionario**

Hay varias formas de recorrer los elementos de un diccionario: recorrer solo las claves, solo los valores o recorrer a la vez las claves y los valores. Puedes ver aquí [cómo usar el bucle for para recorrer un diccionario](https://j2logo.com/bucle-for-en-python/#for-diccionarios).

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

>>> **for** e **in** d:

... print(e)

...

uno

dos

tres

# Recorrer las claves del diccionario

>>> **for** k **in** d.keys():

... print(k)

...

uno

dos

tres

# Recorrer los valores del diccionario

>>> **for** v **in** d.values():

... print(v)

...

1

2

3

# Recorrer los pares clave valor

>>> **for** i **in** d.items():

... print(i)

...

('uno', 1)

('dos', 2)

('tres', 3)

## **Añadir elementos a un diccionario en Python**

Como te decía, la clase *dict* es mutable, por lo que se pueden añadir, modificar y/o eliminar elementos después de haber creado un objeto de este tipo.

Para añadir un nuevo elemento a un diccionario existente, se usa el operador de asignación =. A la izquierda del operador aparece el objeto diccionario con la nueva clave entre corchetes [] y a la derecha el valor que se asocia a dicha clave.

>>> d = {'uno': 1, 'dos': 2}

>>> d

{'uno': 1, 'dos': 2}

# Añade un nuevo elemento al diccionario

>>> d['tres'] = 3

>>> d

{'uno': 1, 'dos': 2, 'tres': 3}

🎯 **NOTA:** Si la clave ya existe en el diccionario, se actualiza su valor.

También existe el método setdefault(clave[, valor]). Este método devuelve el valor de la clave si ya existe y, en caso contrario, le asigna el valor que se pasa como segundo argumento. Si no se especifica este segundo argumento, por defecto es None.

>>> d = {'uno': 1, 'dos': 2}

>>> d.setdefault('uno', 1.0)

1

>>> d.setdefault('tres', 3)

3

>>> d.setdefault('cuatro')

>>> d

{'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': None}

## **Modificar elementos de un diccionario**

En el apartado anterior hemos visto que para actualizar el valor asociado a una clave, simplemente se asigna un nuevo valor a dicha clave del diccionario.

>>> d = {'uno': 1, 'dos': 2}

>>> d

{'uno': 1, 'dos': 2}

>>> d['uno'] = 1.0

>>> d

{'uno': 1.0, 'dos': 2}

## **Eliminar un elemento de un diccionario en Python**

En Python existen diversos modos de eliminar un elemento de un diccionario. Son los siguientes:

* pop(clave [, valor por defecto]): Si la clave está en el diccionario, elimina el elemento y devuelve su valor; si no, devuelve el valor por defecto. Si no se proporciona el valor por defecto y la clave no está en el diccionario, se lanza la excepción KeyError.
* popitem(): Elimina el último par *clave: valor* del diccionario y lo devuelve. Si el diccionario está vacío se lanza la excepción KeyError. (**NOTA:** En versiones anteriores a Python 3.7, se elimina/devuelve un par aleatorio, no se garantiza que sea el último).
* del d[clave]: Elimina el par *clave: valor*. Si no existe la clave, se lanza la excepción KeyError.
* clear(): Borra todos los pares *clave: valor* del diccionario.

>>> d = {'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 4, 'cinco': 5}

# Elimina un elemento con pop()

>>> d.pop('uno')

1

>>> d

{'dos': 2, 'tres': 3, 'cuatro': 4, 'cinco': 5}

# Trata de eliminar una clave con pop() que no existe

>>> d.pop(6)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

KeyError: 6

# Elimina un elemento con popitem()

>>> d.popitem()

('cinco', 5)

>>> d

{'dos': 2, 'tres': 3, 'cuatro': 4}

# Elimina un elemento con del

>>> del d['tres']

>>> d

{'dos': 2, 'cuatro': 4}

# Trata de eliminar una clave con del que no existe

>>> del d['seis']

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

KeyError: 'seis'

# Borra todos los elementos del diccionario

>>> d.clear()

>>> d

{}

## **Número de elementos (len) de un diccionario en Python**

Al igual que sucede con otros tipos contenedores, se puede usar la función de Python len() para obtener el número de elementos de un diccionario.

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

>>> len(d)

3

## **Comprobar si un elemento está en un diccionario en Python**

Al operar con diccionarios, se puede usar el operador de pertenencia in para comprobar si una clave está contenida, o no, en un diccionario. Esto resulta útil, por ejemplo, para asegurarnos de que una clave existe antes de intentar eliminarla.

>>> print('uno' **in** d)

**True**

>>> print(1 **in** d)

**False**

>>> print(1 not **in** d)

**True**

# Intenta eliminar la clave 1 si existe

>>> **if** 1 **in** d:

... del d[1]

...

>>> d

{'uno': 1, 'dos': 2, 'tres': 3}

## **Comparar si dos diccionarios son iguales**

En Python se puede utilizar el operador de igualdad == para comparar si dos diccionarios son iguales. **Dos diccionarios son iguales si contienen el mismo conjunto de pares *clave: valor***, independientemente del orden que tengan.

Otro tipo de comparaciones entre diccionarios no están permitidas. Si se intenta, el intérprete lanzará la excepción TypeError.

>>> d1 = {'uno': 1, 'dos': 2}

>>> d2 = {'dos': 2, 'uno': 1}

>>> d3 = {'uno': 1}

>>> print(d1 == d2)

**True**

>>> print(d1 == d3)

**False**

>>> print(d1 > d2)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

TypeError: '>' not supported between instances **of** 'dict' and 'dict'

## **Diccionarios anidados en Python**

Un diccionario puede contener un valor de cualquier tipo, entre ellos, otro diccionario. Este hecho se conoce como diccionarios anidados.

Para acceder al valor de una de las claves de un diccionario interno, se usa el operador de indexación anidada [clave1][clave2]...

Veámoslo con un ejemplo:

>>> d = {'d1': {'k1': 1, 'k2': 2}, 'd2': {'k1': 3, 'k4': 4}}

>>> d['d1']['k1']

1

>>> d['d2']['k1']

3

>>> d['d2']['k4']

4

>>> d['d3']['k4']

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

KeyError: 'd3'

## **Obtener una lista con las claves de un diccionario**

En ocasiones, es necesario tener almacenado en una lista las claves de un diccionario. Para ello, simplemente pasa el diccionario como argumento del constructor list(). Esto devolverá las claves del diccionario en una lista.

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

>>> list(d)

['uno', 'dos', 'tres']

## **Objetos vista de un diccionario**

La clase *dict* implementa tres métodos muy particulares, dado que devuelven un tipo de dato, *iterable*, conocido como *objetos vista*. Estos objetos ofrecen una vista de las claves y valores contenidos en el diccionario y si el diccionario se modifica, dichos objetos se actualizan al instante.

Los métodos son los siguientes:

* keys(): Devuelve una vista de las claves del diccionario.
* values(): Devuelve una vista de los valores del diccionario.
* items(): Devuelve una vista de pares *(clave, valor)* del diccionario.

>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

# d.keys() es diferente a list(d), aunque ambos

# contengan las claves del diccionario

# d.keys() es de tipo dict\_keys y list(d) es de tipo list

>>> v = d.keys()

>>> type(v)

<**class** 'dict\_keys'>

>>> v

dict\_keys(['uno', 'dos', 'tres'])

>>> l = list(d)

>>> type(l)

<**class** 'list'>

>>> l

['uno', 'dos', 'tres']

>>> v = d.values()

>>> type(v)

<**class** 'dict\_values'>

>>> v

dict\_values([1, 2, 3])

>>> v = d.items()

>>> type(v)

<**class** 'dict\_items'>

>>> v

dict\_items([('uno', 1), ('dos', 2), ('tres', 3)])

## **Listado de métodos de la clase dict**

Termino este tutorial mostrándote el listado de los principales métodos de la clase *dict*. Algunos de ellos ya los hemos visto en las secciones anteriores:

|  |  |
| --- | --- |
| Método | Descripción |
| clear() | Elimina todos los elementos del diccionario. |
| copy() | Devuelve una copia poco profunda del diccionario. |
| get(clave[, valor]) | Devuelve el valor de la clave. Si no existe, devuelve el valor valor si se indica y si no, None. |
| items() | Devuelve una vista de los pares *clave: valor* del diccionario. |
| keys() | Devuelve una vista de las claves del diccionario. |
| pop(clave[, valor]) | Devuelve el valor del elemento cuya clave es clave y elimina el elemento del diccionario. Si la clave no se encuentra, devuelve valor si se proporciona. Si la clave no se encuentra y no se indica valor, lanza la excepción KeyError. |
| popitem() | Devuelve un par *(clave, valor)* aleatorio del diccionario. Si el diccionario está vacío, lanza la excepción KeyError. |
| setdefault(clave[, valor]) | Si la clave está en el diccionario, devuelve su valor. Si no lo está, inserta la clave con el valor valor y lo devuelve (si no se especifica valor, por defecto es None). |
| update(iterable) | Actualiza el diccionario con los pares *clave: valor* del iterable. |
| values() | Devuelve una vista de los valores del diccionario. |

# str python – Strings en Python: El tipo str y operaciones más comunes

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*string*](https://j2logo.com/tag/string/)*,* [*tiposdatos*](https://j2logo.com/tag/tiposdatos/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

En el tutorial sobre [tipos de datos básicos](https://j2logo.com/python/tutorial/tipos-de-datos-basicos-de-python/) ya indiqué que el tipo str en Python se utilizaba para representar cadenas de caracteres.

En este tutorial describiré en detalle la clase *str* y descubrirás las principales operaciones que se pueden realizar sobre un string en Python.

## **Índice**

* [Qué es el tipo str en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-clase)
* [Cómo crear una cadena en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-crear)
* [Cómo acceder a los caracteres de una cadena en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-elemento)
* [for str Python – Recorrer los caracteres de una cadena](https://j2logo.com/python/tutorial/tipo-str-python/#str-for)
* [Comprobar si un carácter está en una cadena en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-in)
* [Comprobar si dos strings son iguales en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-equal)
* [Longitud (len) de una cadena en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-len)
* [Escapar caracteres de una cadena en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-escapar)
* [Raw strings](https://j2logo.com/python/tutorial/tipo-str-python/#str-raw)
* [Representación de un objeto como cadena](https://j2logo.com/python/tutorial/tipo-str-python/#str-object)
* [Otras operaciones sobre cadenas en Python](https://j2logo.com/python/tutorial/tipo-str-python/#str-operaciones)

## **Qué es el tipo str en Python**

La clase *str* en Python se utiliza para representar texto, más conocido en el mundo de la programación como *string* o cadena de caracteres.

Poniéndome un poco más técnico, el tipo *str* es una **secuencia inmutable de caracteres** [**Unicode**](https://es.wikipedia.org/wiki/Unicode). Por tanto, al igual que [*list*](https://j2logo.com/python/tutorial/tipo-list-python/), [*tuple*](https://j2logo.com/python/tutorial/tipo-tuple-python/) o [*range*](https://j2logo.com/python/tutorial/tipo-range-python/), es un tipo secuencial y como es inmutable, un objeto de este tipo no se puede modificar después de haber sido creado.

❗️ **IMPORTANTE:** Cuidado cuando trabajes con texto procedente de ficheros u otras fuentes de datos. Fíjate en qué codificación está y haz las transformaciones necesarias si no quieres tener problemas. Por defecto, la codificación de un *string* en Python es *Unicode*, concretamente *UTF-8*.

## **Cómo crear una cadena en Python**

Crear una cadena de texto en Python es muy sencillo. Simplemente encierra una secuencia de caracteres entre comillas simples '' o dobles "".

>>> s = 'Hola Pythonista'

>>> s

'Hola Pythonista'

>>> type(s)

<**class** 'str'>

>>> s2 = "Me gusta Python"

>>> s2

'Me gusta Python'

>>> type(s2)

<**class** 'str'>

Si quieres o necesitas que un string ocupe más de una línea, entonces debes encerrar el texto entre tres comillas simples '''...''' o dobles """...""".

>>> s = '''

... Este string

... ocupa más

... de

... una línea'''

>>> s

'\nEste string\n ocupa más\n de\n una línea'

>>> print(s)

Este string

ocupa más

de

una línea

Como puedes observar, el uso de las tres comillas (simples o dobles) guarda el carácter de fin de línea. Esto se puede evitar añadiendo el carácter \ al final de cada línea.

>>> s = '''Este string \

... ocupa más \

... de \

... una línea'''

>>> s

'Este string ocupa más de una línea'

>>> print(s)

Este string ocupa más de una línea

❗️ **OJO:** No confundas un string multilínea con un *docstring*. Un *docstring* es un string multilínea que se utiliza para documentar un módulo, función, clase o método, entre otros.

Además, dos cadenas se pueden concatenar con el operador +, o incluso repetir, usando el operador \*. El resultado en ambos casos es un nuevo *string*.

>>> hola = 'hola'

>>> s = hola + ' Pythonista'

>>> s

'hola Pythonista'

>>> s2 = hola \* 3 + ' Pythonista'

>>> s2

'holaholahola Pythonista'

Y para terminar esta sección, simplemente resaltar que dos strings literales se pueden concatenar si aparecen juntos uno tras otro.

>>> s = 'Hola' ' Pythonista'

>>> s

'Hola Pythonista'

>>> s = ('Hola'

... ' Pythonista'

... ' ¿Te gusta Python?')

>>> s

'Hola Pythonista ¿Te gusta Python?'

## **Cómo acceder a los caracteres de una cadena en Python**

Como el resto de tipos secuencia, podemos acceder a los caracteres de una cadena a través de un índice numérico. **Un índice es un número entero que indica la posición de un carácter en la cadena.** Siempre el primer carácter de la secuencia tiene como índice 0.

>>> s = 'Hola Pythonista'

# Primer carácter de la cadena

>>> s[0]

'H'

# Sexto carácter de la cadena

>>> s[5]

'P'

# Tercer carácter de la cadena

>>> s[2]

'l'

Si se intenta acceder a un índice que está fuera del rango del string, el intérprete lanzará la excepción IndexError. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción TypeError

>>> s = 'Hola Pythonista'

>>> s[30]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

IndexError: string index out **of** range

>>> s[1.0]

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

TypeError: string indices must be integers

Los índices también puede ser negativos. En este caso, el índice -1 hace referencia al último carácter, -2 al penúltimo, y así, sucesivamente.

>>> s = 'Hola Pythonista'

>>> s[-1]

'a'

>>> s[-2]

't'

## **for str Python – Recorrer los caracteres de una cadena**

En Python es posible recorrer todos los caracteres de una cadena usando la sentencia *for*. Para ello, lo más fácil es seguir la siguiente plantilla:

>>> saludo = 'Hola'

>>> **for** c **in** saludo:

... print(c)

...

H

o

l

a

## **Comprobar si un carácter está en una cadena en Python**

Para comprobar si un carácter está contenido en una cadena, utiliza el operador de pertenencia in. Esto nos devolverá True si al menos hay una ocurrencia del carácter en el string o False en caso contrario.

>>> saludo = 'Hola'

>>> print('o' **in** saludo)

**True**

>>> print('p' **in** saludo)

**False**

>>> print('p' not **in** saludo)

True

## **Comprobar si dos strings son iguales en Python**

En Python, para comparar si dos cadenas de caracteres son iguales, se utiliza el operador de igualdad ==. **Dos strings son iguales si y solo si ambas cadenas contienen la misma secuencia de caracteres** (se distingue entre mayúsculas y minúsculas).

>>> s1 = 'hola'

>>> s2 = 'hola'

>>> print(s1 == s2)

**True**

>>> s3 = 'Hola'

>>> print(s1 == s3)

False

## **Longitud (len) de una cadena en Python**

Como con cualquier otro tipo contenedor o secuencia, para obtener la longitud de una cadena se debe utilizar la función de Python len(). A esta función se le pasa como argumento la cadena en cuestión y nos devolverá el número de caracteres que tiene.

>>> saludo = 'Hola'

>>> len(saludo)

4

## **Escapar caracteres de una cadena en Python**

Como un string está limitado por los caracteres '' o "", ¿qué ocurre si necesito usar el carácter ' o " dentro de una cadena?

Lo más fácil si tienes que usar el carácter ' en tu cadena, es encerarla entre comillas dobles. Por el contrario, si necesitas usar " dentro del string, enciérralo entre comillas simples.

>>> s = 'Dijo: "Hola Pythonista"'

>>> print(s)

Dijo: "Hola Pythonista"

>>> s = "Dijo: 'Hola Pythonista'"

>>> print(s)

Dijo: 'Hola Pythonista'

También puedes usar la combinación \' para mostrar una comilla simple o \" para mostrar una comilla doble, independientemente de si la cadena está encerrada entre comillas simples o dobles.

>>> s = 'Dijo: \'Hola Pythonista\''

>>> print(s)

Dijo: 'Hola Pythonista'

>>> s = "Dijo: \"Hola Pythonista\""

>>> print(s)

Dijo: "Hola Pythonista"

Además del carácter ' y ", hay otros caracteres especiales que para ser usados dentro de una cadena necesitan ser «escapados» con el carácter \. Son, entre otros, los siguientes: tabulador (\t), barra invertida (\\), retroceso (\b), nueva línea (\n) o retorno de carro (\r).

# Ejemplo para declarar una ruta en Windows

>>> s = 'C:\\Users\\Documents\\'

>>> print(s)

C:\Users\Documents\

# Nueva línea más tabulador

>>> s = 'Hola\n\tPythonista'

>>> print(s)

Hola

Pythonista

## **Raw strings**

En relación con la sección anterior, puede haber ocasiones en que se quiera usar el carácter \ pero sin ser utilizado como carácter de escape. Para ello, se puede hacer uso de las *raw strings*. Una cadena de este tipo comienza anteponiendo el carácter r a las comillas (simples o dobles).

# Aquí, \n es interpretado como nueva línea

>>> s = 'C:\python\noticias'

>>> print(s)

C:\python

oticias

# En una raw string no se interpreta el carácter \

>>> s = r'C:\python\noticias'

>>> print(s)

C:\python\noticias

## **Representación de un objeto como cadena**

Una singularidad de la clase *str* es que a su constructor se le puede pasar cualquier objeto. Al hacer esto, la función str() devuelve la representación en forma de cadena de caracteres del propio objeto (si se pasa un string devuelve el string en sí).

Normalmente, al llamar a la función str(objeto) lo que se hace internamente es llamar al método \_\_str\_\_() del objeto. Si este método no existe, entonces devuelve el resultado de invocar a repr(objeto).

## **Otras operaciones sobre cadenas en Python**

Para finalizar este tutorial, te dejo en esta sección una recopilación de otras entradas del blog en las que se muestran más operaciones y utilidades con cadenas de caracteres:

* [Cómo comprobar si una cadena está vacía](https://j2logo.com/comprobar-python-lista-cadena-diccionario-vacio/)
* [Concatenar strings en Python](https://j2logo.com/como-concatenar-y-formatear-strings/)
* [Dar formato a strings en Python](https://j2logo.com/como-concatenar-y-formatear-strings/)
* [replace() para reemplazar caracteres en una cadena](https://j2logo.com/string-replace-python/)
* [Eliminar los espacios en blanco (trim) con strip()](https://j2logo.com/eliminar-espacios-en-blanco-string-strip-python/)
* [Convertir una cadena a mayúsculas y minúsculas](https://j2logo.com/python/convertir-a-mayusculas-y-minusculas-en-python/)
* [Dividir string en Python en tokens](https://j2logo.com/python/split-dividir-un-string-en-python-en-tokens/)

# Funciones en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*funciones*](https://j2logo.com/tag/funciones/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

Las funciones en Python, y en cualquier lenguaje de programación, son estructuras esenciales de código. Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto.

Este tutorial es una guía que te muestra qué es una función, cuál es su estructura y cómo usarlas en tus aplicaciones.

## **Índice**

* [Qué son las funciones en Python](https://j2logo.com/python/tutorial/funciones-en-python/#function-quees)
* [Cómo definir una función en Python](https://j2logo.com/python/tutorial/funciones-en-python/#function-def)
* [Cómo usar o llamar a una función](https://j2logo.com/python/tutorial/funciones-en-python/#function-call)
* [Sentencia return](https://j2logo.com/python/tutorial/funciones-en-python/#function-return)
* [Parámetros de las funciones en Python](https://j2logo.com/python/tutorial/funciones-en-python/#function-params)
* [Ámbito y ciclo de vida de las variables](https://j2logo.com/python/tutorial/funciones-en-python/#function-vars)

## **Qué son las funciones en Python**

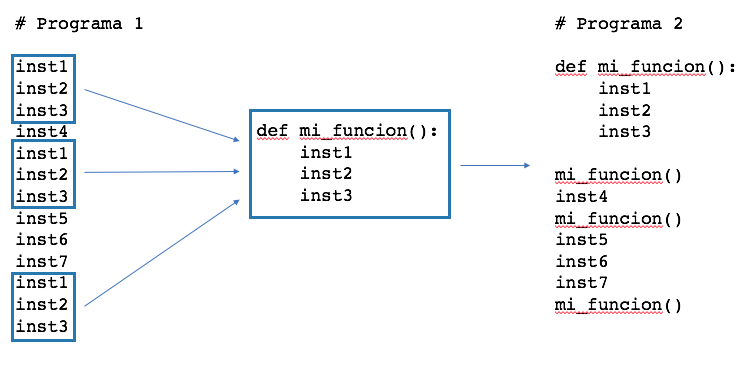
Como te decía en la introducción, las funciones en Python constituyen unidades lógicas de un programa y tienen un doble objetivo:

* Dividir y organizar el código en partes más sencillas.
* Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Python ya define de serie un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones. Algunas de ellas las has visto en tutoriales anteriores. Por ejemplo, la función len(), que obtiene el número de elementos de un objeto contenedor como una lista, una tupla, un diccionario o un conjunto. También hemos visto la función print(), que muestra por consola un texto.

Sin embargo, tú como programador, puedes definir tus propias funciones para estructurar el código de manera que sea más legible y para reutilizar aquellas partes que se repiten a lo largo de una aplicación. Esto es una tarea fundamental a medida que va creciendo el número de líneas de un programa.

La idea la puedes observar en la siguiente imagen:

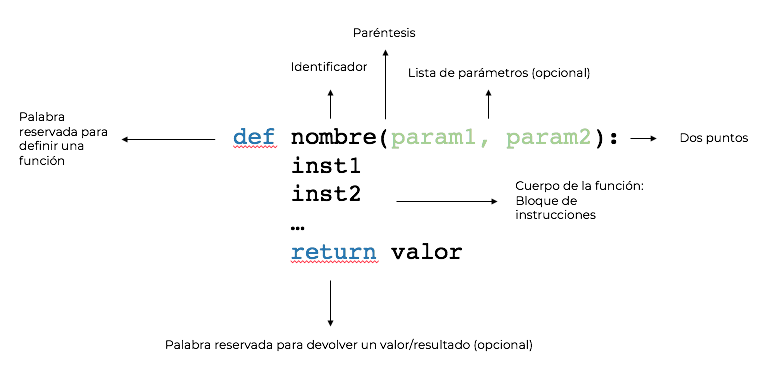


En principio, un programa es una secuencia ordenada de instrucciones que se ejecutan una a continuación de la otra. Sin embargo, cuando se utilizan funciones, puedes agrupar parte de esas instrucciones como una unidad más pequeña que ejecuta dichas instrucciones y suele devolver un resultado.

En el siguiente apartado te muestro cómo definir una función en Python.

## **Cómo definir una función en Python**

La siguiente imagen muestra el esquema de una función en Python:



Para definir una función en Python se utiliza la palabra reservada def. A continuación viene el nombre o identificador de la función que es el que se utiliza para invocarla. Después del nombre hay que incluir los paréntesis y una lista opcional de parámetros. Por último, la cabecera o definición de la función termina con dos puntos.

Tras los dos puntos se incluye el cuerpo de la función (con un sangrado mayor, generalmente cuatro espacios) que no es más que el conjunto de instrucciones que se encapsulan en dicha función y que le dan significado.

En último lugar y de manera opcional, se añade la instrucción con la palabra reservada return para devolver un resultado.

❗️ **NOTA:** Cuando la primera instrucción de una función es un string encerrado entre tres comillas simples ''' o dobles """, a dicha instrucción se le conoce como docstring. El *docstring* es una cadena que se utiliza para documentar la función, es decir, indicar qué hace dicha función.

## **Cómo usar o llamar a una función**

Para usar o invocar a una función, simplemente hay que escribir su nombre como si de una instrucción más se tratara. Eso sí, pasando los argumentos necesarios según los parámetros que defina la función.

Veámoslo con un ejemplo. Vamos a crear una función que muestra por pantalla el resultado de multiplicar un número por cinco:

def multiplica\_por\_5(numero):

print(f'{numero} \* 5 = {numero \* 5}')

print('Comienzo del programa')

multiplica\_por\_5(7)

print('Siguiente')

multiplica\_por\_5(113)

print('Fin')

La función multiplica\_por\_5() define un parámetro llamado numero que es el que se utiliza para multiplicar por 5. El resultado del programa anterior sería el siguiente:

Comienzo del programa

7 \* 5 = 35

Siguiente

113 \* 5 = 565

Fin

Como puedes observar, el programa comienza su ejecución en la *línea 4* y va ejecutando las instrucciones una a una de manera ordenada. Cuando se encuentra el nombre de la función multiplica\_por\_5(), el flujo de ejecución pasa a la primera instrucción de la función. Cuando se llega a la última instrucción de la función, el flujo del programa sigue por la instrucción que hay a continuación de la llamada de la función.

❗️ **IMPORTANTE:** Diferencia entre **parámetro** y **argumento**. La función multiplica\_por\_5() define un *parámetro* llamado numero. Sin embargo, cuando desde el código se invoca a la función, por ejemplo, multiplica\_por\_5(7), se dice que se llama a multiplica por cinco con el *argumento* 7.

## **Sentencia return**

Anteriormente te indicaba que cuando acaba la última instrucción de una función, el flujo del programa continúa por la instrucción que sigue a la llamada de dicha función. Hay una excepción: usar la sentencia return. return hace que termine la ejecución de la función cuando aparece y el programa continúa por su flujo normal.

Además, return se puede utilizar para devolver un valor.

La sentencia return es opcional, puede devolver, o no, un valor y es posible que aparezca más de una vez dentro de una misma función.

A continuación te muestro varios ejemplos:

### **return que no devuelve ningún valor**

La siguiente función muestra por pantalla el cuadrado de un número solo si este es par:

>>> **def** cuadrado\_de\_par(numero):

... **if** not numero % 2 == 0:

... **return**

... **else**:

... print(numero \*\* 2)

...

>>> cuadrado\_de\_par(8)

64

>>> cuadrado\_de\_par(3)

### **Varios return en una misma función**

La función es\_par() devuelve True si un número es par y False en caso contrario:

>>> **def** es\_par(numero):

... **if** numero % 2 == 0:

... **return** **True**

... **else**:

... **return** **False**

...

>>> es\_par(2)

**True**

>>> es\_par(5)

False

### **Devolver más de un valor con return en Python**

En Python, es posible devolver más de un valor con una sola sentencia return. Por defecto, con return se puede devolver una tupla de valores. Un ejemplo sería la siguiente función cuadrado\_y\_cubo() que devuelve el cuadrado y el cubo de un número:

>>> **def** cuadrado\_y\_cubo(numero):

... **return** numero \*\* 2, numero \*\* 3

...

>>> cuad, cubo = cuadrado\_y\_cubo(4)

>>> cuad

16

>>> cubo

64

Sin embargo, se puede usar otra técnica devolviendo los diferentes resultados/valores en una lista. Por ejemplo, la función tabla\_del() que se muestra a continuación hace esto:

>>> **def** tabla\_del(numero):

... resultados = []

... **for** i **in** range(11):

... resultados.append(numero \* i)

... **return** resultados

...

>>> res = tabla\_del(3)

>>> res

[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]

## **En Python una función siempre devuelve un valor**

Python, a diferencia de otros lenguajes de programación, no tiene procedimientos. Un procedimiento sería como una función pero que no devuelve ningún valor.

¿Por qué no tiene procedimientos si hemos vistos ejemplos de funciones que no retornan ningún valor? Porque Python, internamente, devuelve por defecto el valor None cuando en una función no aparece la sentencia return o esta no devuelve nada.

>>> **def** saludo(nombre):

... print(f'Hola {nombre}')

...

>>> print(saludo('j2logo'))

Hola j2logo

None

Como puedes ver en el ejemplo anterior, el print que envuelve a la función saludo() muestra None.

## **Parámetros de las funciones en Python**

Tal y como te he indicado, una función puede definir, opcionalmente, una secuencia de parámetros con los que invocarla. ¿Cómo se asignan en Python los valores a los parámetros? ¿Se puede modificar el valor de una variable dentro de una función?

Antes de contestar a estas dos preguntas, tenemos que conocer los conceptos de programación *paso por valor* y *paso por referencia*.

* **Paso por valor:** Un lenguaje de programación que utiliza paso por valor de los argumentos, lo que realmente hace es copiar el valor de las variables en los respectivos parámetros. Cualquier modificación del valor del parámetro, no afecta a la variable externa correspondiente.
* **Paso por referencia:** Un lenguaje de programación que utiliza paso por referencia, lo que realmente hace es copiar en los parámetros la dirección de memoria de las variables que se usan como argumento. Esto implica que realmente hagan referencia al mismo objeto/elemento y cualquier modificación del valor en el parámetro afectará a la variable externa correspondiente.

Muchos lenguajes de programación usan a la vez paso por valor y por referencia en función del tipo de la variable. Por ejemplo, paso por valor para los tipos simples: entero, float, … y paso por referencia para los objetos.

Sin embargo, en Python todo es un objeto. Ya vimos esto en el [tutorial sobre variables](https://j2logo.com/python/tutorial/variables-python/#variables-memoria). Entonces, ¿cómo se pasan los argumentos en Python, por valor o por referencia? Lo que ocurre en Python realmente es que se pasa por valor la referencia del objeto 😱 ¿Qué implicaciones tiene esto? Básicamente que si el tipo que se pasa como argumento es inmutable, cualquier modificación en el valor del parámetro no afectará a la variable externa pero, si es mutable (como una lista o diccionario), sí se verá afectado por las modificaciones. **Así que, ¡¡¡cuidado!!!** Para complementar esta explicación, no te pierdas la siguiente sección.

Una vez aclarado este tema, a continuación te dejo unos tutoriales muy, muy interesantes para que sigas profundizando sobre los tipos de parámetros en Python:

* [Tipos de parámetros](https://j2logo.com/tipo-parametros-funcion-python/)
* [\*args y \*\*kwargs. Número de parámetros indefinido](https://j2logo.com/args-y-kwargs-en-python/)

## **Ámbito y ciclo de vida de las variables**

En cualquier lenguaje de programación de alto nivel, toda variable está definida dentro de un **ámbito**. Esto es, los sitios en los que la variable tiene sentido y dónde se puede utilizar.

Los parámetros y variables definidos dentro de una función tienen un ámbito *local*, local a la propia función. Por tanto, estos parámetros y variables no pueden ser utilizados fuera de la función porque no serían reconocidos.

El *ciclo de vida* de una variable determina el tiempo en que una variable permanece en memoria. Una variable dentro de una función existe en memoria durante el tiempo en que está ejecutándose dicha función. Una vez que termina su ejecución, sus variables y parámetros desaparecen de memoria y, por tanto, no pueden ser referenciados.

>>> **def** saludo(nombre):

... x = 10

... print(f'Hola {nombre}')

...

>>> saludo('j2logo')

Hola j2logo

>>> print(x)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

NameError: name 'x' is not defined

Como ves, en el ejemplo anterior, al tratar de mostrar por pantalla el valor de la variable x, el intérprete mostrará un error.

El siguiente ejemplo es diferente:

>>> **def** muestra\_x():

... x = 10

... print(f'x vale {x}')

...

>>> x = 20

>>> muestra\_x()

x vale 10

>>> print(x)

20

Lo que sucede en este ejemplo es que dentro de la función muestra\_x() se está creando una nueva variable x que, precisamente, tiene el mismo nombre que la variable definida fuera de la función. Por tanto, x dentro de la función tiene el valor 10, pero una vez que la función termina, x hace referencia a la variable definida fuera, cuyo valor es 20.

Las variables definidas fuera de una función tienen un ámbito conocido como *global* y son visibles dentro de las funciones, dónde solo se puede **consultar** su valor.

>>> y = 20

>>> **def** muestra\_x():

... x = 10

... print(f'x vale {x}')

... print(f'y vale {y}')

...

>>> muestra\_x()

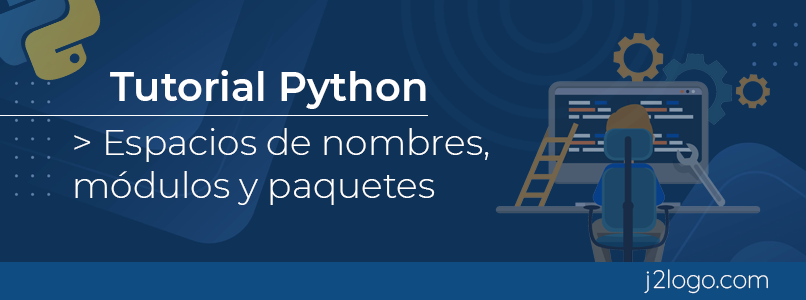
x vale 10

y vale 20

Para modificar dentro de una función una variable definida fuera de la misma, hay que usar la palabra reservada global para declarar la variable dentro de la función, pero esto lo veremos con más detenimiento en el [siguiente tutorial](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/). ¡No te lo pierdas!

# Espacios de nombres, módulos y paquetes en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

Los módulos y paquetes en Python son la forma de organizar los scripts y programas a medida que estos crecen en número de líneas de código. Por otro lado, un espacio de nombres define los límites en que se puede utilizar un nombre o identificador determinado.

No importa si ahora mismo no sabes qué significan estos conceptos o no entiendes a qué me refiero. Lo descubrirás a lo largo de este tutorial. ¿Te lo vas a perder?

## **Índice**

* [Nombres y espacios de nombres en Python](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-namespaces)
* [Ámbito de una variable en Python](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-scopes)
* [Qué es un módulo en Python](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-modules)
* [Cómo importar módulos en Python](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-import)
* [Ejecutar módulos como scripts](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-scripts)
* [Dónde y cómo busca Python los módulos](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-path)
* [La función dir()](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-dir)
* [Paquetes en Python](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-packages)
* [Importar definiciones de un paquete](https://j2logo.com/python/tutorial/espacios-de-nombres-modulos-y-paquetes/#python-import-definitions)

## **Nombres y espacios de nombres en Python**

### **Nombres**

Lo primero que debes tener claro antes de profundizar en los conceptos de este tutorial es qué es un *nombre*.

Como ya te he señalado en varias ocasiones a lo largo de los diferentes tutoriales, **en Python todo es un objeto**. El *número 2* es un objeto, el texto *‘Hola mundo’* es un objeto, las funciones son objetos, … Pues bien, **un *nombre* o *identificador* es la forma que existe en Python de referenciar a un objeto concreto. Equivaldría al concepto de variable**. En definitiva, una variable no es más que el nombre con el que nos referimos a un objeto que existe en memoria.

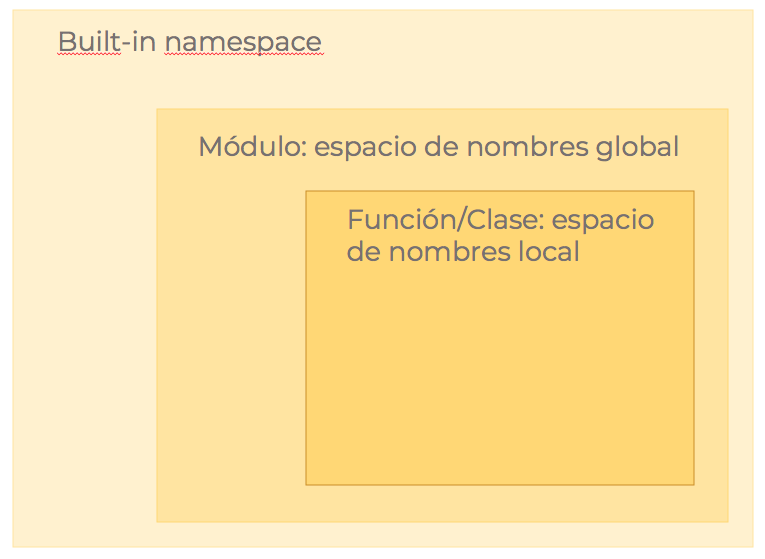
### **Espacios de nombres**

Una vez aclarado este término, paso a explicarte qué son los *espacios de nombres* en Python. **Un espacio de nombres es una colección aislada de nombres (o identificadores) que referencian a objetos.** Como veremos a continuación, en un mismo script o programa Python pueden coexistir varios espacios de nombres a la vez.

Cuando accedemos a un intérprete de Python o ejecutamos un programa, **todos los identificadores que define el lenguaje son añadidos a un espacio de nombres al que es posible acceder desde cualquier punto de un script**. Es por esto que las funciones como print() o len() están siempre accesibles. Este espacio de nombres es conocido como *espacio de nombres incorporado* (o *built-in namespace*)

Además, **cada módulo en Python crea su propio espacio de nombres *global***. Como te decía, los espacios de nombres están aislados. Esa es la razón por la que en diferentes módulos se pueden usar los mismos nombres y estos no interfieren entre sí.

A su vez, en un módulo existen funciones y clases. **Cuando se invoca a una función se crea un espacio de nombres *local* asociado a dicha función que contiene todos los nombres definidos dentro de la misma** (sucede algo similar para las clases).



## **Ámbito de una variable en Python**

**Un *ámbito* define los límites de un programa en los que un espacio de nombres puede ser accedido** sin utilizar un prefijo.

Como te he mostrado en el apartado anterior, en principio existen, como mínimo, tres ámbitos. Uno por cada espacio de nombres:

* Ámbito de la función actual, que tiene los nombres locales a la función.
* Ámbito a nivel de módulo, que tiene los nombres globales, los que se definen en el propio módulo.
* Ámbito incorporado, el más externo, que tiene los nombres que define Python.

Cuando desde dentro de una función se hace referencia a un nombre, este se busca en primer lugar en el espacio de nombres *local*, luego en el espacio de nombres *global* y finalmente en el espacio de nombres *incorporado*.

Si hay una función dentro de otra función, se anida un nuevo ámbito dentro del ámbito local.

### **local, nonlocal y global en Python**

Veamos mejor los conceptos anteriores con unos ejemplos:

def funcion\_a():

y = 2

**def** funcion\_b():

z = 3

print(z)

funcion\_b()

print(y)

x = 1

funcion\_a()

print(x)

En el programa de arriba tenemos una variable x que está definida en el espacio de nombres *global*, una variable y definida en el espacio de nombres *local* de la función funcion\_a y una variable z que está definida en el espacio de nombres *local* de la función funcion\_b.

Imagina por un momento que estamos dentro de la función funcion\_b. La variable z es *local* para nosotros (está en el ámbito local), y es *no local* y x es *global*. Esto quiere decir que podemos acceder y modificar la variable z pero solo podemos consultar el valor de x e y puesto que se encuentran en un ámbito diferente al nuestro.

Si dentro de la función funcion\_b asignamos un valor a una variable y, realmente estamos creando una nueva variable en nuestro espacio de nombres local. Esta variable es diferente a la variable *no local* y que está definida en la función funcion\_a. Lo mismo ocurriría con la variable *global* x.

Para poder modificar la variable x dentro de funcion\_b, debemos definir la variable como global. Y para modificar la variable y, hay que definirla como nonlocal.

Redefiniendo el programa del comienzo de este apartado de este modo

def funcion\_a():

x = 2

**def** funcion\_b():

x = 3

print(x)

funcion\_b()

print(x)

x = 1

funcion\_a()

print(x)

el resultado sería el siguiente:

3

2

1

Sin embargo, si definimos la variable x como global, la cosa cambia:

def funcion\_a():

global x

x = 2

**def** funcion\_b():

global x

x = 3

print(x)

funcion\_b()

print(x)

x = 1

funcion\_a()

print(x)

Ahora el resultado sería este otro:

3

3

3

## **Qué es un módulo en Python**

En Python, **un módulo no es más que un fichero que contiene instrucciones y definiciones** (variables, funciones, clases, …). El fichero debe tener extensión .py y su nombre se corresponde con el nombre del módulo.

❗️ **NOTA:** Dentro de un módulo, puedes acceder al nombre del mismo a través de la variable global \_\_name\_\_.

Los módulos tienen un doble propósito:

* Dividir un programa con muchas líneas de código en partes más pequeñas.
* Extraer un conjunto de definiciones que utilizas frecuentemente en tus programas para ser reutilizadas. Esto evita, por ejemplo, tener que estar copiando funciones de un programa a otro.

🎯 Es una buena práctica que un módulo solo contenga instrucciones y definiciones relacionadas entre sí.

¿Quieres crear tu primer módulo?

Sitúate en un directorio para un nuevo proyecto y crea en él un fichero llamado mis\_funciones.py con el siguiente contenido:

def saludo(nombre):

print(f'Hola {nombre}')

Ahora, desde una consola, sitúate en el directorio anterior y ejecuta el comando python3 para lanzar el intérprete de Python.

Una vez dentro del intérprete, ejecuta el siguiente comando:

>>> import mis\_funciones

Con el comando anterior estamos importando el módulo mis\_funciones en el intérprete. Prueba a llamar ahora a la función saludo() de este modo:

>>> mis\_funciones.saludo('j2logo')

Hola j2logo

## **Cómo importar módulos en Python**

Como has visto al final del apartado anterior, para usar las definiciones de un módulo en el intérprete o en otro módulo, primero hay que importarlo. Para ello, se usa la palabra reservada import. Una vez que un módulo ha sido importado, se puede acceder a sus definiciones a través del operador punto ..

🎯 Aunque puedes importar los módulos y sus definiciones dónde y cuando quieras, es una buena práctica que aparezcan al principio del módulo.

Ya sabes que un módulo puede contener instrucciones y definiciones. Normalmente, las instrucciones son utilizadas para inicializar el módulo y solo son ejecutadas la primera vez que aparece el nombre del módulo en una sentencia import.

Previamente te he enseñado una forma de importar un módulo en otro módulo para usar sus definiciones, sin embargo, existen otros modos de usarlas e importarlas.

### **from … import …**

Podemos importar uno o varios nombres de un módulo del siguiente modo:

from mis\_funciones import saludo, otra\_funcion

saludo('j2logo')

Esto nos permite acceder directamente a los nombres definidos en el módulo sin tener que utilizar el operador ..

### **from … import \***

from mis\_funciones import \*

saludo('j2logo')

Es similar al caso anterior, solo que importa todas las definiciones del módulo a excepción de los nombres que comienzan por guión bajo \_.

❗️ **IMPORTANTE:** No es una buena práctica importar así las definiciones de un módulo porque dificulta la lectura y los nombres importados pueden ocultar identificadores y nombres usados en el módulo en el que se importan.

### **from … import as**

Por último, podemos redefinir el nombre con el que una definición será usada dentro de un módulo utilizando la palabra reservada as:

>>> from mis\_funciones import saludo as hola

>>> hola('j2logo')

Hola j2logo

## **Ejecutar módulos como scripts**

Un módulo puede ser ejecutado como un script o como punto de entrada de un programa cuando se pasa directamente como parámetro al intérprete de Python:

>>> python mis\_funciones.py

Cuando esto ocurre, el código del módulo se ejecuta como si se hubiera importado, con la particularidad de que el nombre del módulo, \_\_name\_\_, es \_\_main\_\_.

Esto hace realmente interesante añadir al final del módulo las siguientes líneas de código, que solo se ejecutarán en caso de que dicho módulo se haya ejecutado como el *principal*:

if \_\_name\_\_ == '\_\_main\_\_':

hola('j2logo')

No se ejecutarán en caso de que el módulo se importe en otro módulo.

## **Dónde y cómo busca Python los módulos**

Python trae consigo un gran catálogo de módulos estándar con multitud de funciones y clases que puedes usar en tus aplicaciones. Este catálogo es conocido como la [Biblioteca de Referencia](https://docs.python.org/3/py-modindex.html).

Ahora bien, cuando importamos un módulo, ¿cómo sabe Python a qué módulo nos referimos? ¿Dónde busca Python los ficheros correspondientes a los módulos?

En primer lugar, Python busca el módulo en el catálogo de módulos estándar y si no lo encuentra, entonces busca el fichero en el listado de directorios definidos en la variable sys.path. Esta variable es inicializada con las siguientes rutas y localizaciones:

* El directorio en el que se encuentra el script *principal*.
* PYTHONPATH, una variable de entorno similar a PATH.
* Directorios de instalación por defecto de Python

## **La función dir()**

La función dir() devuelve una lista con todas las definiciones (variables, funciones, clases, …) contenidas en un módulo.

Por ejemplo, si ejecutamos dir() sobre el módulo mis\_funciones que creamos previamente, obtendríamos el siguiente resultado:

>>> dir(mis\_funciones)

['\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'saludo']

Fíjate en que al final aparece el nombre saludo referente a la función que hemos definido.

Si a dir() no se le pasa ningún argumento, entonces devuelve todas las definiciones del módulo actual.

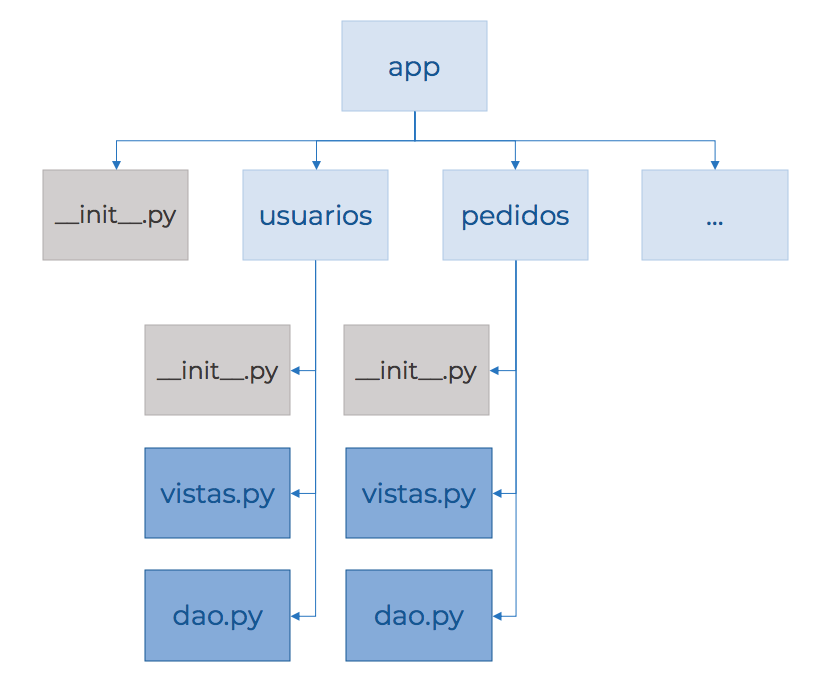
## **Paquetes en Python**

Del mismo modo en que agrupamos las funciones y demás definiciones en *módulos*, los *paquetes* en Python permiten organizar y estructurar de forma jerárquica los diferentes módulos que componen un programa. Además, los paquetes hacen posible que existan varios módulos con el mismo nombre y que no se produzcan errores.

**Un paquete es simplemente un directorio que contiene otros paquetes y módulos**. Además, en Python, para que un directorio sea considerado un paquete, este debe incluir un módulo llamado \_\_init\_\_.py. En la mayoría de ocasiones, el fichero \_\_init\_\_.py estará vacío, sin embargo, se puede utilizar para inicializar código relacionado con el paquete.

Al igual que sucede con los módulos, cuando se importa un paquete, Python busca a través de los directorios definidos en sys.path el directorio perteneciente a dicho paquete.

Para que lo veas todo de forma gráfica, te muestro los conceptos con una imagen. Imagina que estás haciendo una aplicación para gestionar pedidos. Una forma de organizar los diferentes módulos podría ser la siguiente:



## **Importar definiciones de un paquete**

Para importar módulos y definiciones de módulos que están contenidos en paquetes, se usa el operador .. Las referencias se hacen indicando el nombre completo del módulo, es decir, especificando los paquetes hasta llegar al módulo en cuestión separándolos con puntos.

Teniendo en cuenta el diagrama de la sección anterior, si en el módulo app.pedidos.vistas se quiere importar el módulo app.usuarios.dao, simplemente hay que añadir la siguiente sentencia:

# Módulo app.pedidos.vistas

import app.usuarios.dao

El único problema de hacerlo así, es que si, por ejemplo, dicho módulo define una función llamada guardar(), hay que especificar toda la jerarquía para invocar a esta función:

app.usuarios.dao.guardar(usuario)

Una forma mejor es importar el módulo. Esto se consigue de la siguiente manera:

# Módulo app.pedidos.vistas

from app.usuarios import dao

dao.guardar(usuario)

Incluso, se puede importar una definición de un módulo del siguiente modo:

# Módulo app.pedidos.vistas

from app.usuarios.dao import guardar

guardar(usuario)

Y para terminar este tutorial, vamos a ver que dentro de un módulo los paquetes se pueden referenciar de forma relativa.

Imagina ahora que desde el módulo app.pedidos.vistas quieres importar los módulos app.pedidos.dao y app.usuarios.vistas. Se podría hacer como hemos visto hasta ahora:

# Módulo app.pedidos.vistas

from app.pedidos import dao

from app.usuarios import vistas

O también se podría hacer así:

# Módulo app.pedidos.vistas

from . import dao # Un punto referencia al paquete actual

from ..usuarios import vistas # Dos puntos referencian al paquete padre

Bueno, hemos llegado al final de este tutorial. Ha sido intenso pero necesario para conocer mejor cómo funciona Python internamente y poder sacar más provecho al lenguaje. ¡Te espero en el siguiente, en el que veremos conceptos de programación orientada a objetos!

# Programación orientada a objetos (POO) en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*poo*](https://j2logo.com/tag/poo/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*tutorial python*](https://j2logo.com/tag/tutorial-python/)



Share on facebook

Share on twitter

Share on linkedin

De todos los tutoriales que hemos visto hasta ahora, diría que este es uno de los más importantes: **Programación Orientada a Objetos en Python**. Y es que, como te he mencionado en varias ocasiones, en Python todo es un objeto. Si dominas los conceptos que describo en este artículo, estarás un paso más cerca de ser un auténtico Pythonista.

Como sabrás, Python es un lenguaje multiparadigma: soporta la programación imperativa y funcional, pero también la programación orientada a objetos.

La verdad es que el tema da para mucho. Por eso, **este tutorial es un resumen de los conceptos clave de la programación orientada a objetos desde el punto de vista de Python**. ¡No te lo puedes perder!

## **Índice**

* [Python es un lenguaje orientado a objetos](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-python)
* [Clases y objetos en Python](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-clases-objetos)
* [Constructor de una clase en Python](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-init)
* [Atributos, atributos de datos y métodos](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-atributos)
* [Atributos de clase y atributos de instancia](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-atributos-clase-instancia)
* [Herencia en Python](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-herencia)
* [Herencia múltiple en Python](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-herencia-multiple)
* [Encapsulación: atributos privados](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-encapsulacion)
* [Polimorfismo](https://j2logo.com/python/tutorial/programacion-orientada-a-objetos/#poo-polimorfismo)

## **Python es un lenguaje orientado a objetos**

Sí, soy un pesado 😂 y por eso te lo vuelvo a decir: **En Python todo es un objeto**. Cuando creas una variable y le asignas un valor entero, ese valor es un objeto; una función es un objeto; las listas, tuplas, diccionarios, conjuntos, … son objetos; una cadena de caracteres es un objeto. Y así podría seguir indefinidamente.

Pero, ¿por qué es tan importante la programación orientada a objetos? Bien, este tipo de programación introduce un nuevo paradigma que nos permite encapsular y aislar datos y operaciones que se pueden realizar sobre dichos datos.

Sigue leyendo para que entiendas qué quiero decir.

## **Clases y objetos en Python**

Básicamente, una clase es una entidad que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).

Por su parte, un objeto es una concreción o instancia de una clase.

Tranqui, que lo vas a entender con el siguiente ejemplo.

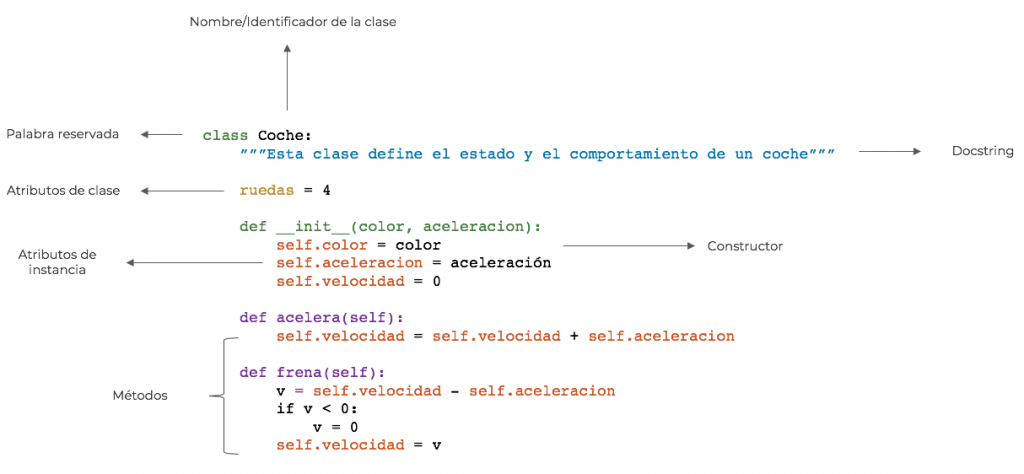
Seguro que si te digo que te imagines un coche, en tu mente comienzas a visualizar la carrocería, el color, las ruedas, el volante, si es diésel o gasolina, el color de la tapicería, si es manual o automático, si acelera o va marcha atrás, etc.

Pues todo lo que acabo de describir viene a ser una clase y cada uno de los de coches que has imaginado, serían objetos de dicha clase.

¿Cómo pasamos lo anterior a Python? Veámoslo.

Como te decía, una clase engloba datos y funcionalidad. Cada vez que se define una clase en Python, se crea a su vez un tipo nuevo (¿recuerdas? tipo int, float, str, list, tuple, … todos ellos están definidos en una clase).

Para definir una clase en Python se utiliza la palabra reservada class. El siguiente esquema visualiza los elementos principales que componen una clase. Todos ellos los iremos viendo con detenimiento en las siguientes secciones:



El esquema anterior define la clase Coche (es una versión muy, muy simplificada de lo que es un coche, jajaja, pero nos sirve de ejemplo). Dicha clase establece una serie datos, como ruedas, color, aceleración o velocidad y las operaciones acelera() y frena().

Cuando se crea una variable de tipo Coche, realmente se está instanciando un objeto de dicha clase. En el siguiente ejemplo se crean dos objetos de tipo Coche:

>>> c1 = Coche('rojo', 20)

>>> print(c1.color)

rojo

>>> print(c1.ruedas)

4

>>> c2 = Coche('azul', 30)

>>> print(c2.color)

azul

>>> print(c2.ruedas)

4

c1 y c2 son objetos, objetos cuya clase es Coche. Ambos objetos pueden acelerar y frenar, porque su clase define estas operaciones y tienen un color, porque la clase Coche también define este *dato*. Lo que ocurre es que c1 es de color *rojo*, mientras que c2 es de color *azul*.

¿Ves ya la diferencia?

🎯 **NOTA:** Es una convención utilizar la notación CamelCase para los nombres de las clases. Esto es, la primera letra de cada palabra del nombre está en mayúsculas y el resto de letras se mantienen en minúsculas.

## **Constructor de una clase en Python**

En la sección anterior me he adelantado un poco… Para crear un objeto de una clase determinada, es decir, *instanciar* una clase, se usa el nombre de la clase y a continuación se añaden paréntesis (como si se llamara a una función).

obj = MiClase()

El código anterior crea una nueva instancia de la clase MiClase y asigna dicho objeto a la variable obj. Esto crea un objeto *vacío*, sin estado.

Sin embargo, hay clases (como nuestra clase Coche) que deben o necesitan crear instancias de objetos con un estado inicial.

Esto se consigue implementando el método especial \_\_init\_\_(). Este método es conocido como el constructor de la clase y se invoca cada vez que se instancia un nuevo objeto.

El método \_\_init\_\_() establece un primer parámetro especial que se suele llamar self (veremos qué significa este nombre en la siguiente sección). Pero puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función.

En nuestro caso, el constructor de la clase coche es el siguiente:

def \_\_init\_\_(self, color, aceleracion):

self.color = color

self.aceleracion = aceleracion

self.velocidad = 0

Como puedes observar, además del parámetro self, define los parámetros color y aceleracion, que determinan el estado inicial de un objeto de tipo Coche.

En este caso, para instanciar un objeto de tipo coche, debemos pasar como argumentos el color y la aceleración como vimos en el ejemplo:

c1 = Coche('rojo', 20)

❗️ **IMPORTANTE:** A diferencia de otros lenguajes, en los que está permitido implementar más de un constructor, en Python solo se puede definir un método \_\_init\_\_().

## **Atributos, atributos de datos y métodos**

Una vez que sabemos qué es un objeto, tengo que decirte que la única operación que pueden realizar los objetos es referenciar a sus atributos por medio del operador .

Como habrás podido apreciar, un objeto tiene dos tipos de atributos: *atributos de datos* y *métodos*.

* Los atributos de datos definen el estado del objeto. En otros lenguajes son conocidos simplemente como atributos o miembros.
* Los métodos son las funciones definidas dentro de la clase.

Siguiendo con nuestro ejemplo de la clase Coche, vamos a crear el siguiente objeto:

>>> c1 = Coche('rojo', 20)

>>> print(c1.color)

rojo

>>> print(c1.velocidad)

0

>>> c1.acelera()

>>> print(c1.velocidad)

20

En la *línea 2* del código anterior, el objeto c1 está referenciando al atributo de dato color y en la *línea 4* al atributo velocidad. Sin embargo, en la *línea 6* se referencia al método acelera(). Llamar a este método tiene una implicación como puedes observar y es que modifica el estado del objeto, dado que se incrementa su velocidad. Este hecho lo puedes apreciar cuando se vuelve a referenciar al atributo velocidad en la *línea 7*.

### **Atributos de datos**

A diferencia de otros lenguajes, los atributos de datos no necesitan ser declarados previamente. Un objeto los crea del mismo modo en que se crean las variables en Python, es decir, cuando les asigna un valor por primera vez.

El siguiente código es un ejemplo de ello:

>>> c1 = Coche('rojo', 20)

>>> c2 = Coche('azul', 10)

>>> print(c1.color)

rojo

>>> print(c2.color)

azul

>>> c1.marchas = 6

>>> print(c1.marchas)

6

>>> print(c2.marchas)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

AttributeError: 'Coche' object has no attribute 'marchas'

Los objetos c1 y c2 pueden referenciar al atributo color porque está definido en la clase Coche. Sin embargo, solo el objeto c1 puede referenciar al atributo marchas a partir de la *línea 7*, porque inicializa dicho atributo en esa línea. Si el objeto c2 intenta referenciar al mismo atributo, como no está definido en la clase y tampoco lo ha inicializado, el intérprete lanzará un error.

### **Métodos**

Como te explicaba al comienzo de esta sección, los métodos son las funciones que se definen dentro de una clase y que, por consiguiente, pueden ser referenciadas por los objetos de dicha clase. Sin embargo, realmente los métodos son algo más.

Si te has fijado bien, pero bien de verdad, habrás observado que las funciones acelera() y frena() definen un parámetro self.

def acelera(self):

self.velocidad = self.velocidad + self.aceleracion

No obstante, cuando se usan dichas funciones no se pasa ningún argumento. ¿Qué está pasando? Pues que acelera() está siendo utilizada como un método por los objetos de la clase Coche, de tal manera que cuando un objeto referencia a dicha función, realmente pasa su propia referencia como primer parámetro de la función.

🎯 **NOTA:** Por convención, se utiliza la palabra self para referenciar a la instancia actual en los métodos de una clase.

Sabiendo esto, podemos entender, por ejemplo, por qué todos los objetos de tipo Coche pueden referenciar a los atributos de datos velocidad o color. Son inicializados para cada objeto en el método \_\_init\_\_().

Del mismo modo, el siguiente ejemplo muestra dos formas diferentes y equivalentes de llamar al método acelera():

>>> c1 = Coche('rojo', 20)

>>> c2 = Coche('azul', 20)

>>> c1.acelera()

>>> Coche.acelera(c2)

>>> print(c1.velocidad)

20

>>> print(c2.velocidad)

20

Para la clase Coche, acelera() es una función. Sin embargo, para los objetos de la clase Coche, acelera() es un método.

>>> print(Coche.acelera)

<**function** Coche.acelera at *0x10c60b560*>

>>> print(c1.acelera)

<bound method Coche.acelera **of** <\_\_main\_\_.Coche object at *0x10c61efd0*>>

## **Atributos de clase y atributos de instancia**

Una clase puede definir dos tipos diferentes de atributos de datos: atributos de clase y atributos de instancia.

* Los atributos de clase son atributos compartidos por todas las instancias de esa clase.
* Los atributos de instancia, por el contrario, son únicos para cada uno de los objetos pertenecientes a dicha clase.

En el ejemplo de la clase Coche, ruedas se ha definido como un atributo de clase, mientras que color, aceleracion y velocidad son atributos de instancia.

Para referenciar a un atributo de clase se utiliza, generalmente, el nombre de la clase. Al modificar un atributo de este tipo, los cambios se verán reflejados en todas y cada una las instancias.

>>> c1 = Coche('rojo', 20)

>>> c2 = Coche('azul', 20)

>>> print(c1.color)

rojo

>>> print(c2.color)

azul

>>> print(c1.ruedas) # Atributo de clase

4

>>> print(c2.ruedas) # Atributo de clase

4

>>> Coche.ruedas = 6 # Atributo de clase

>>> print(c1.ruedas) # Atributo de clase

6

>>> print(c2.ruedas) # Atributo de clase

6

Si un objeto modifica un atributo de clase, lo que realmente hace es crear un atributo de instancia con el mismo nombre que el atributo de clase.

>>> c1 = Coche('rojo', 20)

>>> c2 = Coche('azul', 20)

>>> print(c1.color)

rojo

>>> print(c2.color)

azul

>>> c1.ruedas = 6 # Crea el atributo de instancia ruedas

>>> print(c1.ruedas)

6

>>> print(c2.ruedas)

4

>>> print(Coche.ruedas)

4

## **Herencia en Python**

En programación orientada a objetos, la herencia es la capacidad de reutilizar una clase extendiendo su funcionalidad. Una clase que hereda de otra puede añadir nuevos atributos, ocultarlos, añadir nuevos métodos o redefinirlos.

En Python, podemos indicar que una clase hereda de otra de la siguiente manera:

class CocheVolador(Coche):

ruedas = 6

**def** \_\_init\_\_(self, color, aceleracion, esta\_volando=**False**):

super().\_\_init\_\_(color, aceleracion)

self.esta\_volando = esta\_volando

**def** vuela(self):

self.esta\_volando = **True**

**def** aterriza(self):

self.esta\_volando = False

Como puedes observar, la clase CocheVolador hereda de la clase Coche. En Python, el nombre de la clase padre se indica entre paréntesis a continuación del nombre de la clase hija.

La clase CocheVolador redefine el atributo de clase ruedas, estableciendo su valor a 6 e implementa dos métodos nuevos: vuela() y aterriza().

Fíjate ahora en la primera línea del método \_\_init\_\_(). En ella aparece la función super(). Esta función devuelve un objeto temporal de la superclase que permite invocar a los métodos definidos en la misma. Lo que está ocurriendo es que se está redefiniendo el método \_\_init\_\_() de la clase hija usando la funcionalidad del método de la clase padre. Como la clase Coche es la que define los atributos color y aceleracion, estos se pasan al constructor de la clase padre y, a continuación, se crea el atributo de instancia esta\_volando solo para objetos de la clase CocheVolador.

Al utilizar la herencia, todos los atributos (atributos de datos y métodos) de la clase padre también pueden ser referenciados por objetos de las clases hijas. Al revés no ocurre lo mismo.

Veamos todo esto con un ejemplo:

>>> c = Coche('azul', 10)

>>> cv1 = CocheVolador('rojo', 60)

>>> print(cv1.color)

rojo

>>> print(cv1.esta\_volando)

**False**

>>> cv1.acelera()

>>> print(cv1.velocidad)

60

>>> print(CocheVolador.ruedas)

6

>>> print(c.esta\_volando)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

AttributeError: 'Coche' object has no attribute 'esta\_volando'

🎯 **NOTA:** Cuando no se indica, toda clase Python hereda implícitamente de la clase object, de tal modo que class MiClase es lo mismo que class MiClase(object).

### **Las funciones isinstance() e issubclass()**

Como ya vimos en otros tutoriales, la función incorporada type() devuelve el tipo o la clase a la que pertenece un objeto. En nuestro caso, si ejecutamos type() pasando como argumento un objeto de clase Coche o un objeto de clase CocheVolador obtendremos lo siguiente:

>>> c = Coche('rojo', 20)

>>> type(c)

<**class** 'objetos.Coche'>

>>> cv = CocheVolador('azul', 60)

>>> type(cv)

<**class** 'objetos.CocheVolador'>

Sin embargo, Python incorpora otras dos funciones que pueden ser de utilidad cuando se quiere conocer el tipo de una clase. Son: isinstance() e issubclass().

* isinstance(objeto, clase) devuelve True si objeto es de la clase clase o de una de sus clases hijas. Por tanto, un objeto de la clase CocheVolador es instancia de CocheVolador pero también lo es de Coche. Sin embargo, un objeto de la clase Coche nunca será instancia de la clase CocheVolador.
* issubclass(clase, claseinfo) comprueba la herencia de clases. Devuelve True en caso de que clase sea una subclase de claseinfo, False en caso contrario. claseinfo puede ser una clase o una tupla de clases.

>>> c = Coche('rojo', 20)

>>> cv = CocheVolador('azul', 60)

>>> isinstance(c, Coche)

**True**

isinstance(cv, Coche)

**True**

>>> isinstance(c, CocheVolador)

**False**

>>> isinstance(cv, CocheVolador)

**True**

>>> issubclass(CocheVolador, Coche)

**True**

>>> issubclass(Coche, CocheVolador)

False

## **Herencia múltiple en Python**

Python es un lenguaje de programación que permite herencia múltiple. Esto quiere decir que una clase puede heredar de más de una clase a la vez.

class A:

**def** print\_a(self):

print('a')

**class** B:

**def** print\_b(self):

print('b')

**class** C(A, B):

**def** print\_c(self):

print('c')

c = C()

c.print\_a()

c.print\_b()

c.print\_c()

El script anterior dará como resultado

a

b

c

## **Encapsulación: atributos privados**

E*ncapsulación* (o *encapsulamiento*), en programación orientada a objetos, hace referencia a la capacidad que tiene un objeto de ocultar su estado, de manera que sus datos solo se puedan modificar por medio de las operaciones (métodos) que ofrece.

Si vienes de otros lenguajes de programación, quizá te haya resultado raro que no haya mencionado nada sobre atributos públicos o privados.

Bien, por defecto, en Python, todos los atributos de una clase (atributos de datos y métodos) son públicos. Esto quiere decir que desde un código que use la clase, se puede acceder a todos los atributos y métodos de dicha clase.

No obstante, hay una forma de indicar en Python que un atributo, ya sea un dato o un método, es interno a una clase y no se debería utilizar fuera de ella. Algo así como los miembros privados de otros lenguajes. Esto es usando el carácter guión bajo \_atributo antes del nombre del atributo que queramos ocultar.

En cualquier caso, el atributo seguirá siendo accesible desde fuera de la clase, pero el programador está indicando que es privado y no debería utilizarse porque no se sabe qué consecuencias puede tener.

También es posible usar un doble guión bajo \_\_atributo. Esto hace que el identificador sea literalmente reemplazado por el texto \_Clase\_\_atributo, donde Clase es el nombre de la clase actual.

Un ejemplo nunca está de más.

class A:

**def** \_\_init\_\_(self):

self.\_contador = 0 # Este atributo es privado

**def** incrementa(self):

self.\_contador += 1

**def** cuenta(self):

**return** self.\_contador

**class** B(object):

**def** \_\_init\_\_(self):

self.\_\_contador = 0 # Este atributo es privado

**def** incrementa(self):

self.\_\_contador += 1

**def** cuenta(self):

**return** self.\_\_contador

En el ejemplo anterior, la clase A define el atributo privado \_contador. Un ejemplo de uso de la clase sería el siguiente:

>>> a = A()

>>> a.incrementa()

>>> a.incrementa()

>>> a.incrementa()

>>> print(a.cuenta())

3

>>> print(a.\_contador)

3

Como puedes observar, es posible acceder al atributo privado, aunque no se debiera.

En cambio, la clase B define el atributo privado \_\_contador anteponiendo un doble guión bajo. El resultado de hacer el mismo experimento cambia:

>>> b = B()

>>> b.incrementa()

>>> b.incrementa()

>>> print(b.cuenta())

2

>>> print(b.\_\_contador)

Traceback (most recent call last):

File "<input>", line 1, **in** <module>

AttributeError: 'B' object has no attribute '\_\_contador'

>>> print(b.\_B\_\_contador)

2

Si te fijas, no se puede acceder al atributo \_\_contador fuera de la clase. Este identificador se ha sustituido por \_B\_\_contador.

## **Polimorfismo**

Y llegamos al último concepto que veremos en este tutorial sobre programación orientada a objetos en Python.

*Polimorfismo* es la capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases.

Aunque este concepto te suene raro ahora mismo, lo vas a entender con un ejemplo. Imagina que tenemos las siguientes clases que representan animales:

class Perro:

**def** sonido(self):

print('Guauuuuu!!!')

**class** Gato:

**def** sonido(self):

print('Miaaauuuu!!!')

**class** Vaca:

**def** sonido(self):

print('Múuuuuuuu!!!')

Las tres clases implementan un método llamado sonido(). Ahora observa el siguiente script:

def a\_cantar(animales):

**for** animal **in** animales:

animal.sonido()

**if** \_\_name\_\_ == '\_\_main\_\_':

perro = Perro()

gato = Gato()

gato\_2 = Gato()

vaca = Vaca()

perro\_2 = Perro()

granja = [perro, gato, vaca, gato\_2, perro\_2]

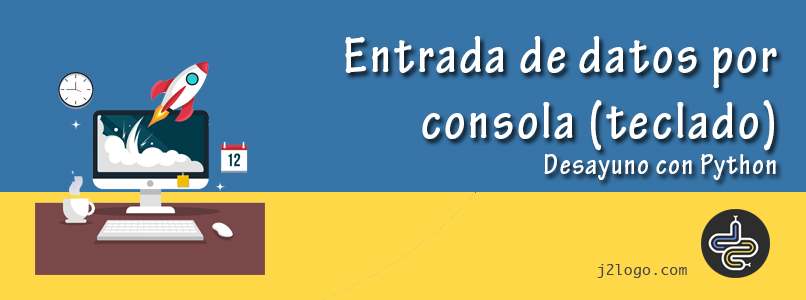
a\_cantar(granja)

En él se ha definido una función llamada a\_cantar(). La variable animal que se crea dentro del bucle for de la función es *polimórfica*, ya que en tiempo de ejecución hará referencia a objetos de las clases Perro, Gato y Vaca. Cuando se invoque al método sonido(), se llamará al método correspondiente de la clase a la que pertenezca cada uno de los animales.

Y bueno, esto ha sido todo en este tutorial sobre programación orientada a objetos en Python. Espero que lo hayas disfrutado y lo hayas entendido, porque es la base para ser un auténtico Pythonista 💪🏻🐍

# Entrada de datos por consola (teclado) en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*Desayuno con Python*](https://j2logo.com/tag/desayuno-con-python/)*,* [*python*](https://j2logo.com/tag/python/)*,* [*python2*](https://j2logo.com/tag/python2/)*,* [*python3*](https://j2logo.com/tag/python3/)



Share on facebook

Share on twitter

Share on linkedin

Cuando trabajamos con la consola o el terminal, es muy común solicitar al usuario la introducción de información a través del teclado. En este post de 🐍 *Desayunando con Python* vamos a ver **cómo realizar la entrada de datos por consola**, tanto en Python 3.x como en Python 2.x.

## **Entrada de datos por consola en Python 3: input()**

Para pedir al usuario que introduzca algún dato a través del teclado, usaremos el método input().

Este método, recibe como parámetro un str con el mensaje a mostrar al usuario:

>>> edad = input('¿Qué edad tienes?')

¿Qué edad tienes?

>?

En este momento, la consola queda a la espera de la entrada de datos por parte del usuario.

Al introducir cualquier texto, será asignado a la variable edad:

>>> edad = input('¿Qué edad tienes?')

¿Qué edad tienes?>? 28

>>> edad

'28'

❗️Ten en cuenta que **la entrada es convertida SIEMPRE a un objeto de tipo str UTF-8**. Por tanto, si lo que necesitamos es cualquier otro tipo, por ejemplo, un int, habrá que hacer la conversión correspondiente:

>>> edad = int (edad)

>>> edad

28

Como ves, ahora edad es un int (ya no se muestra con las comillas simples).

## **Cómo solicitar datos por consola en Python 2: input() y raw\_input()**

Sin embargo, en Python 2 las cosas eran un poco diferentes.

El método raw\_input() es el equivalente al método input() de Python 3: Lee una línea de la entrada estándar y la devuelve como un objeto str, esta vez en *ASCII.*

Por el contrario, **el método input() espera una expresión Python válida**. ¿Esto que quiere decir? Pues que le puedes pasar un número pero también una expresión del tipo 2 + 3 \* 5. input() se encarga de evaluar la entrada y procesarla correctamente. A continuación te muestro un ejemplo:

>>> edad = input('¿Qué edad tienes?')

¿Qué edad tienes?>? 28

>>> edad

28

>>> suma = input('¿Cuánto es 2 + 3?')

¿Cuánto es 2 + 3?>? 2 + 3

>>> suma

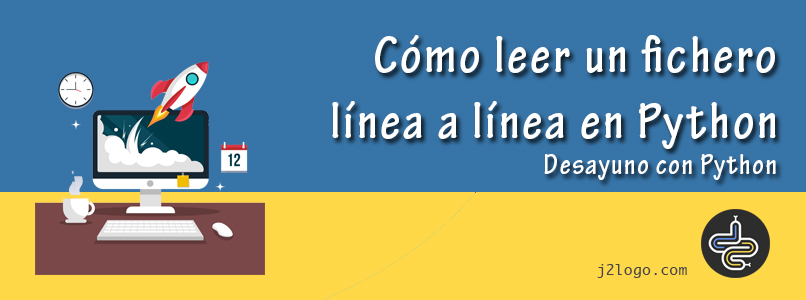
5

## **Conclusión**

Como ves, solicitar al usuario datos por consola en Python es muy simple. **Ten en cuenta las diferencias entre Python 2.x y Python 3.x,** especialmente si realizas una migración de versión. En este último caso, si necesitas que los datos no sean de tipo str asegúrate de llevar a cabo las conversiones y comprobaciones necesarias.

# Cómo leer un fichero línea a línea en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*Desayuno con Python*](https://j2logo.com/tag/desayuno-con-python/)*,* [*ficheros*](https://j2logo.com/tag/ficheros/)*,* [*python*](https://j2logo.com/tag/python/)



Share on facebook

Share on twitter

Share on linkedin

A la hora de programar siempre hay una serie de tareas repetitivas que me cuesta trabajo recordar. No sé si a ti te pasa lo mismo. Es por eso por lo que decidí crear la sección «Desayuno con Python» en el blog, para recopilar todos esos scripts sencillos pero útiles de uso cotidiano. Uno los scripts que no podía faltar en esta sección es cómo leer un fichero línea a línea en Python.

## **Leer un fichero línea a línea**

Leer las líneas de un fichero es una tarea muy común y muy sencilla. Básicamente, todo se reduce a dos líneas de código:

**with** open('ruta\_del\_fichero') **as** f:

**for** linea **in** f:

# Tu código aquí

❗ Ten en cuenta que la variable linea incluye el carácter \n al final.

## **Leer un fichero en python completo**

Un objeto de tipo file también ofrece un método llamado read(). Este método leerá el fichero en su totalidad y lo devolverá como una cadena de texto. Usa este método con precaución ya que puede consumir mucha memoria si el fichero es demasiado grande. Yo lo usaría solo en caso de que el fichero fuera muy pequeño.

**with** open('cloudbutton.py', 'r') **as** f:

contenido = f.read()

print(contenido)

## **¿Qué significa la palabra with al principio del bloque?**

A la hora de gestionar y manipular recursos, como puede ser un fichero, hay ciertos patrones que se suelen repetir. Para estos casos, Python nos ayuda a abstraernos del código repetitivo introduciendo lo que se conocen como «Manejadores de contexto» a través de la sentencia with.

En el caso de los ficheros, with nos asegura de que el fichero se cerrará correctamente después de ejecutarse el código en el interior del bloque, incluso si ocurre alguna excepción.

De manera que el siguiente código

**with** open('hola.txt', 'r') **as** f:

**for** linea **in** f:

...

sería equivalente a este

f = open('hola.txt', 'r')

**try**:

**for** linea **in** f:

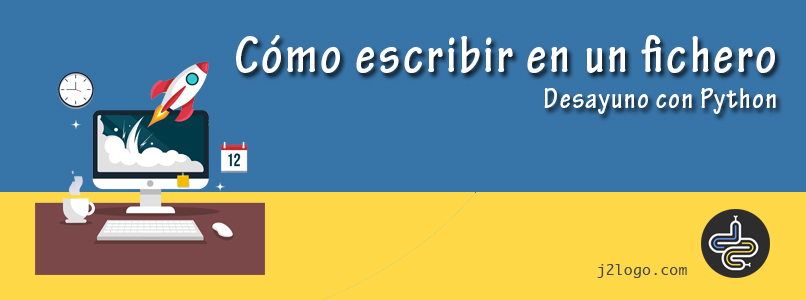
...

**finally**:

f.close()

# Cómo escribir en un fichero en Python

* *Categoría:* [*Tutoriales Python*](https://j2logo.com/category/blog/tutoriales-python/)
* [*básico*](https://j2logo.com/tag/basico/)*,* [*Desayuno con Python*](https://j2logo.com/tag/desayuno-con-python/)*,* [*ficheros*](https://j2logo.com/tag/ficheros/)*,* [*python*](https://j2logo.com/tag/python/)



Share on facebook

Share on twitter

Share on linkedin

Sí, antes o después cualquier programador se tiene que enfrentar a la tarea de escribir en un fichero. Y en Python, como en cualquier otro lenguaje, esto es posible. Por eso, este tutorial sobre cómo escribir en un fichero no podía faltar en la sección «Desayuno con Python».

## **El objeto File y sus tipos**

El objeto File en Python representa un fichero del sistema operativo. Los ficheros, sirven para guardar datos en disco que podrán ser leídos posteriormente (recuerda que ya vimos [cómo leer ficheros en Python](https://j2logo.com/leer-fichero-linea-a-linea-python/)).

Normalmente, existen dos tipos de ficheros: **ficheros de texto** y **ficheros binarios**. Un fichero de texto contiene caracteres que son legibles por el ser humano y están guardados con una codificación (ASCII, UTF-8, …). Por el contrario, un fichero binario está compuesto por un flujo de bytes y solo tienen sentido para los programas o aplicaciones para los que son creados. Un ejemplo de este tipo de archivos son las imágenes o la música.

Cuando se trabaja con un fichero de texto, hay que tener en cuenta que este se estructura como una secuencia de líneas. Cada una de estas líneas acaba con un carácter especial conocido como **EOL** (fin de línea). En función del sistema operativo, este carácter puede variar. Puede ser \n (Unix) o \r\n (Windows). No obstante, en Python, cuando escribimos o leemos el carácter \n en un fichero de texto, el propio lenguaje se encarga de convertir dicho carácter al correspondiente por el sistema operativo, por lo que es algo a prever si nuestro código se va a ejecutar en diferentes sistemas.

## **La función open() y sus modos**

En Python, para escribir en un fichero o simplemente leer su contenido utilizaremos la función predefinida open(). Al invocar a esta, se crea un objeto de tipo File.

Lo más común es llamar a la función open() con dos parámetros:

* El primero de ellos es la ruta del fichero (en la que está o donde se va a crear).
* El segundo es el modo en el que se abre el fichero: lectura, escritura, …

Por defecto, cuando se invoca a la función open(path, modo), el fichero **se abre en modo texto**. Si quisiéramos abrir un fichero en forma binaria, habría que añadir el carácter b al parámetro modo.

Los diferentes modos en los que se puede abrir un fichero son:

|  |  |
| --- | --- |
| r | Solo lectura. El fichero solo se puede leer. Es el modo por defecto si no se indica. |
| w | Solo escritura. En el fichero solo se puede escribir. Si ya existe el fichero, machaca su contenido. |
| a | Adición. En el fichero solo se puede escribir. Si ya existe el fichero, todo lo que se escriba se añadirá al final del mismo. |
| x | Como ‘w’ pero si existe el fichero lanza una excepción. |
| r+ | Lectura y escritura. El fichero se puede leer y escribir. |

Como te he indicado, todos estos modos abren el fichero en modo texto. Su versión correspondiente para abrir el fichero en modo binario sería rb, wb, ab, xb, rb+.

## **Chuleta para escribir en un fichero**

Bueno, ahora que ya conocemos los principales conceptos del objeto File y cómo trata Python a los ficheros, es hora de ver cómo escribir en un fichero en Python.

La estructura para abrir un fichero en modo escritura es la siguiente:

f = open('mi\_fichero', 'w')

**try**:

# Procesamiento para escribir en el fichero

**finally**:

f.close()

Aunque personalmente prefiero usar la sentencia with, ya que ella se encarga de cerrar el fichero y liberar sus recursos (incluso si ocurre cualquier error):

**with** open('mi\_fichero', 'w') **as** f:

# Procesamiento del fichero

El objeto f de los ejemplos anteriores representa un fichero y este pone a nuestra disposición el método write() para escribir cualquier contenido sobre el fichero.

### **Escribir un fichero de texto**

Con todo lo anterior, si quisiéramos escribir un fichero de texto haríamos lo siguiente:

**with** open('mi\_fichero', 'w') **as** f:

f.write('Hola mundo\n')

### **Escribir un fichero binario**

Para escribir un fichero binario, simplemente añadimos el carácter b al parámetro modo y escribimos bytes:

**with** open('mi\_fichero', 'wb') **as** f:

f.write(b'0x28')

## **Conclusión**

Bueno, este post es muy importante tenerlo en cuenta para cualquier pythonista. En él hemos repasado el objeto File de Python, los distintos tipos de ficheros que existen y cómo escribir en un fichero. Espero que te haya gustado.

Otros: <https://j2logo.com/como-escribir-en-un-fichero-en-python/>

# Tutorial de Flask en español: Desarrollando una aplicación web en Python

<https://j2logo.com/tutorial-flask-espanol/>