Curso de Python

2 DAW DAM

Departamento de Informática
IES Inca Garcilaso
Rafael Ernesto Escobar Zurita

Índice

1 Introducción

Python es uno de los lenguajes de programación mas utilizado en el mundo. Según el índice TIOBE Python está a la cabeza del ranking como podemos ver en la siguiente imagen:

Sep 2025	Sep 2024	Change	Programming Language	Ratings	Change
1	1		Python	25.98%	+5.81%
2	2		C++	8.80%	-1.94%
3	4	↑	C	8.65%	-0.24%
4	3	↓	Java	8.35%	-1.09%
5	5		C#	6.38%	+0.30%
6	6		JavaScript	3.22%	-0.70%
7	7		Visual Basic	2.84%	+0.14%
8	8		Go	2.32%	-0.03%
9	11	↑	Delphi/Object Pascal	2.26%	+0.49%
10	27	⇈	Perl	2.03%	+1.33%

Python es también usado para fines muy diversos como son los siguientes:

- **Desarrollo Web**: Existen frameworks como Django, Pyramid, Flask o Bottle que permiten desarrollar páginas web a todos los niveles.
- **Ciencia y Educación**: Debido a su sintaxis tan sencilla, es una herramienta perfecta para enseñar conceptos de programación a todos los niveles. En lo relativo a ciencia y cálculo numérico, existen gran cantidad de librerías como SciPy o Pandas.
- **Desarrollo de Interfaces Gráficos**: Gran cantidad de los programas que utilizamos tienen un interfaz gráfico que facilita su uso. Python también puede ser usado para desarrollar GUIs con librerías como Kivy o pyqt.
- **Desarrollo Software**: También es usado como soporte para desarrolladores, como para testing.
- **Machine Learning**: En los último años ha crecido el número de implementaciones en Python de librerías de aprendizaje automático como Keras, TensorFlow, PyTorch o sklearn.
- **Visualización de Datos**: Existen varias librerías muy usadas para mostrar datos en gráficas, como matplotlib, seaborn o plotly.
- **Finanzas y Trading**: Gracias a librerías como QuantLib o qtpylib y a su facilidad de uso, es cada vez más usado en estos sectores.

Características de Python

Como cualquier otro lenguaje, Python tiene una serie de características que lo hacen diferente al resto. Las explicamos a continuación:

- Es un lenguaje interpretado, no compilado.
- Usa tipado dinámico, lo que significa que una variable puede tomar valores de distinto tipo.
- Es fuertemente tipado, lo que significa que el tipo no cambia de manera repentina. Para que se produzca un cambio de tipo tiene que hacer una conversión explícita.
- Es multiplataforma, ya que un código escrito en macOS funciona en Windows o Linux y vice versa.

Tal vez algunos de estos conceptos puedan resultarte extraños si estás empezando en el mundo de la programación. El siguiente código pretende ilustrar algunas de las características de Python.

Algunas cosas curiosidad que en otros lenguajes no pasan. La función acepta un parámetro entrada pero no se especifica su tipo. La x almacena primero una cadena, luego un float y luego un integer. La función funcion() es llamada con un int, pero su valor se divide entre 2 y el resultado es convertido automáticamente en un float.

```
def funcion(entrada):
    return entrada/2
x = "Hola"
x = 7.0
x = int(x)
x = funcion(x)
print(x)
print(type(x))
# 3.5
# <class 'float'>
```

2 Instalación de Python

La forma mas recomendable es la instalación desde la página oficial de Python, aunque muchas distribuciones Linux ya vienen con la versión 3 instalada.

La pagina oficial es la siguiente: https://www.python.org/downloads/ donde elegiremos el SO donde vamos a realizar la instalación.

Una vez instalado podemos comprobar la versión con *python3 -V*, y nos mostrará la versión instalada.

3 Sintaxis en Python

El termino sintaxis hace referencia al **conjunto de reglas que definen como se tiene que escribir el código en un determinado lenguaje de programación**. Es decir, hace referencia a la forma en la que debemos escribir las instrucciones para que el ordenador, o más bien lenguaje de programación, nos entienda.

En la mayoría de lenguajes existe una sintaxis común, como por ejemplo el uso de = para asignar un dato a una variable, o el uso de **{}** para designar bloques de código, pero Python tiene ciertas particularidades.

Lo veremos a continuación en detalle, pero Python no soporta el uso de **$** ni hace falta terminar las líneas con **;** como en otros lenguajes, y tampoco hay que usar **{}** en estructuras de control como en el if.

Comentarios

Los comentarios son bloques de texto usados para comentar el código. Es decir, para ofrecer a otros programadores o a nuestro yo futuro información relevante acerca del código que está escrito. A efectos prácticos, para Python es como si no existieran, ya que no son código propiamente dicho, solo anotaciones.

Los comentarios se inician con # y todo lo que vaya después en la misma línea será considerado un comentario.

```
# Esto es un comentario
```

Al igual que en otros lenguajes de programación, podemos también comentar varias líneas de código. Para ello es necesario hacer uso de triples comillas bien sean simples ''' o dobles """. Es necesario usarlas para abrir el bloque del comentario y para cerrarlo.

```
''' Esto es un comentario
de varias líneas de código '''
```

Indentación y bloques de código

En Python los bloques de código se representan con indentación, y aunque hay un poco de debate con respecto a usar tabulador o espacios, la norma general es usar **cuatro espacios**.

En el siguiente código tenemos un **condicional if**. Justo después tenemos un **print()** indentado con cuatro espacios. Por lo tanto, todo lo que tenga esa indentación pertenecerá al bloque del if.

```
if True:
    print("True")
```

Esto es muy importante ya que el código anterior y el siguiente no son lo mismo. De hecho el siguiente código daría un error ya que el if no contiene ningún bloque de código, y eso es algo que no se puede hacer en Python.

```
iif True:
print("True")
```

Por otro lado, a diferencia de en otros lenguajes de programación, no es necesario utilizar ; para terminar cada línea.

```
# Otros lenguajes como C
# requieren de ; al final de cada línea
x = 10;
```

Sin embargo en **Python no es necesario**, basta con un salto de línea.

```
x = 5
y = 10
```

Pero se puede usar el punto y coma ; para tener dos sentencias en la misma línea.

```
x = 5; y = 10
```

Múltiples líneas

En algunas situaciones se puede dar el caso de que queramos tener una sola instrucción en varias línea de código. Uno de los motivos principales podría ser que fuera **demasiado larga**, y de hecho en la especificación PEP8 se recomienda que las líneas **no excedan los 79 caracteres**.

Haciendo uso de \ **se puede romper el código en varias líneas**, lo que en determinados casos hace que el código sea mucho más legible.

```
x = 1 + 2 + 3 + 4 +\
    5 + 6 + 7 + 8
```

Si por lo contrario estamos dentro de un bloque rodeado con paréntesis (), bastaría con saltar a la siguiente línea.

```
x = (1 + 2 + 3 + 4 +
     5 + 6 + 7 + 8)
```

Se puede hacer lo mismo para llamadas a funciones

```
def funcion(a, b, c):
    return a+b+c

d = funcion(10,
23,
3)
```

Creando variables

Anteriormente ya hemos visto como crear una variable y asignarle un valor con el uso de =. Existen también otras formas de hacerlo de una manera un poco más sofisticada.

Podemos por ejemplo asignar el mismo valor a diferentes variables con el siguiente código.

```
x = y = z = 10
```

O también podemos asignar varios valores separados por coma.

```
x, y = 4, 2
x, y, z = 1, 2, 3
```

```
x = 10
y = "Nombre"
z = 3.9
```

Nombrando variables

Puedes nombrar a tus variables como quieras, pero es importante saber que las **mayúsculas y minúsculas son importantes.** Las variables **x** y **X** son distintas.

Por otro lado existen ciertas normas a la hora de nombrar variables:

- El nombre no puede empezar por un número
- No se permite el uso de guiones -
- Tampoco se permite el uso de espacios.

Se muestran unos ejemplos de nombres de variables válidos y no válidos.

```
# Válido
_variable = 10
vari_able = 20
variable10 = 30
variable = 60
variaBle = 10

# No válido
2variable = 10
var-iable = 10
var iable = 10
```

Asignar múltiples valores
Se pueden asignar múltiples variables en la misma línea.

```
x, y, z = 10, 20, 30
```

Imprimir variables

Una variable puede ser impresa por pantalla usando print()

```
x = 10
y = "Nombre"

print(x)
print(y)
```

Palabras reservadas en Python

Son palabras que no podemos utilizar para nombrar variables ni funciones, ya que las reserva internamente para su funcionamiento.

```
import keyword
print(keyword.kwlist)

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

Lectura de datos desde teclado

```
# Ejemplo 1: Leer una cadena (texto)
nombre = input("Introduce tu nombre: ")
print("Hola, " + nombre + "!")

# Ejemplo 2: Leer un número y convertirlo
edad_str = input("Introduce tu edad: ")
edad_int = int(edad_str) # Convertir el string a entero
print("Tienes", edad_int, "años.")

# Ejemplo 3: Leer un número decimal
altura_str = input("Introduce tu altura (ej. 1.75): ")
altura_float = float(altura_str) # Convertir el string a número flotante
print("Tu altura es", altura_float, "metros.")
```

Entrada y salida de Datos

```
#Entrada
cadena = input("Introduce una cadena: ")

 #Salida print(cadena)
#Como formatear texto y variables en un print
nombre = "Marcos"
apellidos = "Rivera Gavilán"
correo = "riveragavilanmarcos@gmail.com"

print("Hola me llamo " + nombre + " " + apellidos + " y mi correo es " + correo)
#El + concatena sin espacios

print(f"Hola me llamo {nombre} {apellidos} y mi correo es {correo}")
# Al estar dentro de una cadena ponemos los espacios normalmente

print("Hola me llamo {} {} y mi correo es {}".format(nombre, apellidos, correo))
# Al estar dentro de una cadena ponemos los espacios normalmente
```

4 Tipos de datos y operadores

4.1 Tipos de datos

4.1.1 Enteros

En otros lenguajes de programación, los int tenían un valor máximo que pueden representar. Dependiendo de la longitud de palabra o wordsize de la arquitectura del ordenador, existen unos números mínimos y máximos que era posible representar. Si por ejemplo se usan enteros de 32 bits el rango a representar es de -2^31 a 2^31–1, es decir, -2.147.483.648 a 2.147.483.647. Con 64 bits, el rango es de -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807. Una gran ventaja de Python es que ya no nos tenemos que preocupar de esto, ya que por debajo se encarga de asignar más o menos memoria al número, y podemos representar prácticamente cualquier número.

```
i = 12
print(i)        #12
print(type(i)) #<class 'int'>
```

Convertir a int

```
b = int(1.6)
print(b) #1
```

4.1.2 Booleanos

Al igual que en otros lenguajes de programación, en Python existe el tipo bool o booleano. Es un tipo de dato que permite almacenar dos valores **True** o **False**.

```
x = True
y = False
```

Un valor booleano también puede ser el resultado de evaluar una expresión. Ciertos operadores como el mayor que, menor que o igual que devuelven un valor bool.

```
print(1 > 0)  #True
print(1 <= 0) #False
print(9 == 9) #True
```

4.1.3 Float

El tipo numérico `float` permite representar un número positivo o negativo con decimales, es decir, números reales. Si vienes de otros lenguajes, tal vez conozcas el tipo `doble`, lo que significa que tiene el doble de precisión que un `float`. En Python las cosas son algo distintas, y los `float` son en realidad `double`.
Para saber más: Los valores float son almacenados de una forma muy particular, denominada representación en coma flotante. En el estándar IEEE 754 se explica con detalle.
Por lo tanto si declaramos una variable y le asignamos un valor decimal, por defecto la variable será de tipo `float`.

```
f = 0.10093
print(f)      #0.10093
print(type(f)) #<class 'float'>
```

Conversión a float
También se puede declarar usando la notación científica con `e` y el exponente. El siguiente ejemplo sería lo mismo que decir `1.93` multiplicado por diez elevado a `-3`.

```
f = 1.93e-3
```

También podemos convertir otro tipo a float haciendo uso de `float()`. Podemos ver como `True` es en realidad tratado como `1`, y al convertirlo a float, como `1.0`.

```
a = float(True)
b = float(1)
print(a, type(a)) #1.0 <class 'float'>
print(b, type(b)) #1.0 <class 'float'>
```

Rango representable
Alguna curiosidad es que los `float` no tienen precisión infinita. Podemos ver en el siguiente ejemplo como en realidad `f` se almacena como si fuera `1`, ya que no es posible representar tanta precisión decimal.

```
f = 0.99999999999999999
print(f)     #1.0
print(1 == f) #True
```

Los `float` a diferencia de los `int` tienen unos valores mínimo y máximos que pueden representar. La mínima precisión es `2.2250738585072014e-308` y la máxima `1.7976931348623157e+308`, pero si no nos crees, lo puedes verificar tu mismo.

```
import sys
print(sys.float_info.min) #2.2250738585072014e-308
print(sys.float_info.max) #1.7976931348623157e+308
```

De hecho si intentas asignar a una variable un valor mayor que el `max`, lo que ocurre es que esa variable toma el valor `inf` o infinito.

```
f = 1.7976931348623157e+310
print(f) #inf
```

Si quieres representar una variable que tenga un valor muy alto, también puedes crear directamente una variable que contenga ese valor `inf`.

```
f = float('inf')
print(f) #inf
```

Precisión del float

Desafortunadamente, los ordenadores no pueden representar cualquier número, y menos aún si este es uno irracional. Debido a esto, suceden cosas curiosas como las siguientes.
Dividir $\frac{1}{3}$ debería resultar en 0.3 periódico, pero para Python es imposible representar tal número.

```
print("{:.20f}".format(1.0/3.0))
# 0.33333333333333331483
```

Por otro lado, la siguiente operación debería tener como resultado cero, pero como puedes ver esto no es así.

```
print(0.1 + 0.1 + 0.1 - 0.3)
# 5.551115123125783e-17
```

4.1.4 Cadenas o strings

Las cadenas en Python o `strings` son un tipo inmutable que permite almacenar secuencias de caracteres. Para crear una, es necesario incluir el texto entre comillas dobles `"`. Puedes obtener más ayuda con el comando `help(str)`.

```
s = "Esto es una cadena"
print(s)      #Esto es una cadena
print(type(s)) #<class 'str'>
```

También es valido declarar las cadenas con comillas simples simples `'`.

```
s = 'Esto es otra cadena'
print(s)       #Esto es otra cadena
print(type(s))  #<class 'str'>
```

Las cadenas no están limitadas en tamaño, por lo que el único límite es la memoria de tu ordenador. Una cadena puede estar también vacía.

```
s = ''
```

Una situación que muchas veces se puede dar, es cuando queremos introducir una comilla, bien sea simple `'` o doble `"` dentro de una cadena. Si lo hacemos de la siguiente forma tendríamos un error, ya que Python no sabe muy bien donde empieza y termina.

```
#s = "Esto es una comilla doble " de ejemplo" # Error!
```

Para resolver este problema debemos recurrir a las secuencias de escape. En Python hay varias, pero las analizaremos con más detalle en otro capítulo. Por ahora, la más importante es `\"`, que nos permite incrustar comillas dentro de una cadena.

```
s = "Esto es una comilla doble \" de ejemplo"
print(s) #Esto es una comilla doble " de ejemplo
```

También podemos incluir un salto de línea dentro de una cadena, lo que significa que lo que esté después del salto, se imprimirá en una nueva línea.

```
s = "Primer linea\nSegunda linea"
print(s)
#Primer linea
#Segunda linea
```

También podemos usar `\` acompañado de un número, lo que imprimirá el carácter asociado. En este caso imprimimos el carácter `110` que se corresponde con la `H`.

```
print("\110\110") #HH
```

Para saber más: Te recomendamos que busques información sobre ASCI y Unicode. Ambos son conceptos muy útiles a la hora de entender los strings.

Se puede definir una cadena que ocupe varias líneas usando triple `"""` comilla. Puede ser muy útil si tenemos textos muy largo que no queremos tener en una sola línea.
Existe también otra forma de declarar cadenas llamado `raw strings`. Usando como prefijo `r`, la cadena ignora todos las secuencias de escape, por lo que la salida es diferente a la anterior.

```
print(r"\110\110") #\110\110
print("""La siguiente
cadena ocupa
varias lineas""")
```

Formateo de cadenas
Tal vez queramos declarar una cadena que contenga variables en su interior, como números o incluso otras cadenas. Una forma de hacerlo sería concatenando la cadena que queremos con otra usando el operador `+`. Nótese que `str()` convierte en `string` lo que se pasa como parámetro.

```
x = 5
s = "El número es: " + str(x)
print(s) #El número es: 5
```

Otra forma es usando %. Por un lado tenemos `%s` que indica el tipo que se quiere imprimir, y por otro a la derecha del % tenemos la variable a imprimir. Para imprimir una cadena se usaría `%s` o `%f` para un valor en coma flotante.

Para saber más: En el siguiente enlace puedes encontrar más información sobre el uso de %.

```
x = 5
s = "El número es: %d" % x
print(s) #El número es: 5
```

```
s = "Los números son %d y %d." % (5, 10)
print(s) #Los números son 5 y 10.
```

Una forma un poco más moderna de realizar lo mismo, es haciendo uso de `format()`.

```
s = "Los números son {} y {}".format(5, 10)
print(s) #Los números son 5 y 10
```

Es posible también darle nombre a cada elemento, y `format()` se encargará de reemplazar todo.

```
s = "Los números son {a} y {b}".format(a=5, b=10)
print(s) #Los números son 5 y 10
```

Por si no fueran pocas ya, existe una tercera forma de hacerlo introducida en la versión `3.6` de Python. Reciben el nombre de cadenas literales o `f-strings`. Esta nueva característica, permite incrustar expresiones dentro de cadenas.

```
a = 5; b = 10
s = f"Los números son {a} y {b}"
print(s) #Los números son 5 y 10
```

Puedes incluso hacer operaciones dentro de la creación del `string`.

```
a = 5; b = 10
s = f"a + b = {a+b}"
print(s) #a + b = 15
```

Puedes incluso llamar a una función dentro.

```
def funcion():
    return 20
s = f"El resultado de la función es {funcion()}"
print(s) #El resultado de la funcion es 20
```

Ejemplos `string`

Para entender mejor la clase `string`, vamos a ver unos ejemplos de como se comportan. Podemos sumar dos strings con el operador `+`.

```
s1 = "Parte 1"
s2 = "Parte 2"
print(s1 + " " + s2) #Parte 1 Parte 2
```

Se puede multiplicar un `string` por un `int`. Su resultado es replicarlo tantas veces como el valor del entero.

```
s = "Hola "
print(s*3) #Hola Hola Hola
```

Podemos ver si una cadena esta contenida en otra con `in`.

```
print("mola" in "Python mola") #True
```

Con `chr()` and `ord()` podemos convertir entre carácter y su valor numérico que lo representa y viceversa. El segundo sólo función con caracteres, es decir, un `string` con un solo elemento.

```
print(chr(8364)) #€
print(ord("€"))  #110
```

La longitud de una cadena viene determinada por su número de caracteres, y se puede consultar con la función `len()`.

```
print(len("Esta es mi cadena"))
```

Como hemos visto al principio, se puede convertir a `string` otras clases, como `int` o `float`.

```
x = str(10.4)
print(x)      #10.4
print(type(x)) #<class 'str'>
```

También se pueden indexar las cadenas, como si de una lista se tratase.

```
x = "abcde"
print(x[0])  #a
print(x[-1]) #e
```

Del mismo modo, se pueden crear cadenas más pequeñas partiendo de una grande, usando indicando el primer elemento y el último que queremos tomar menos uno.

```
x = "abcde"
print(x[0:2])
```

Si no se indica ningún valor a la derecha de los `:` se llega hasta el final.

```
x = "abcde"
print(x[2:])
```

Es posible también crear subcadenas que contengan elementos salteados y no contiguos añadiendo un tercer elemento entre `[]`. Indica los elementos que se saltan. En el siguiente ejemplo se toman elementos del `0` al `5` de dos en dos.

```
x = "abcde"
print(x[0:5:2]) #ace
```

Tampoco es necesario saber el tamaño de la cadena, y el segundo valor se podría omitir. El siguiente ejemplo es igual al anterior.

```
x = "abcde"
print(x[0::2]) #ace
```

Algunos de los métodos de la clase `string`.

- `capitalize()`

El método `capitalize()` se aplica sobre una cadena y la devuelve con su primera letra en mayúscula.

```
s = "mi cadena"
print(s.capitalize()) #Mi cadena
```

- `lower()`

El método `lower()` convierte todos los caracteres alfabéticos en minúscula.

```
s = "MI CADENA"
print(s.lower()) #mi cadena
```

- `swapcase()`

El método `swapcase()` convierte los caracteres alfabéticos con mayúsculas en minúsculas y viceversa.

```
s = "mI cAdEnA"
print(s.swapcase()) #Mi CaDeNa
```

- `upper()`

El método `upper()` convierte todos los caracteres alfabéticos en mayúsculas.

```
s = "mi cadena"
print(s.upper())
```

- `count(<sub>[, <start>[, <end>]])`

El método `count()` permite contar las veces que otra cadena se encuentra dentro de la primera. Permite también dos parámetros opcionales que indican donde empezar y acabar de buscar.

```
s = "el bello cuello "
print(s.count("llo")) #2
```

- `isalnum()`

El método `isalnum()` devuelve `True` si la cadena esta formada únicamente por caracteres alfanuméricos, `False` de lo contrario. Caracteres como `@` o `&` no son alfanumericos.

```
s = "correo@dominio.com"
print(s.isalnum())
```

- `isalpha()`

El método `isalpha()` devuelve `True` si todos los caracteres son alfabéticos, `False` de lo contrario.

```
s = "abcdefg"
print(s.isalpha())
```

- `strip([<chars>])`

El método `strip()` elimina a la izquierda y derecha el carácter que se le introduce. Si se llama sin parámetros elimina los espacios. Muy útil para limpiar cadenas.

```
s = "  abc  "
print(s.strip()) #abc
```

- `zfill(<width>)`

El método `zfill()` rellena la cadena con ceros a la izquierda hasta llegar a la longitud pasada como parámetro.

```
s = "123"
print(s.zfill(5)) #00123
```

- `join(<iterable>)`

El método `join()` devuelve la primera cadena unida a cada uno de los elementos de la lista que se le pasa como parámetro.

```
s = " y ".join(["1", "2", "3"])
print(s) #1 y 2 y 3
```

- `split(sep=None, maxsplit=-1)`

El método `split()` divide una cadena en subcadenas y las devuelve almacenadas en una lista. La división es realizada de acuerdo a el primer parámetro, y el segundo parámetro indica el número máximo de divisiones a realizar.

```
s = "Python,Java,C"
print(s.split(",")) #['Python', 'Java', 'C']
```

4.1.5 Listas

Las listas en Python son un tipo de dato que permite almacenar datos de cualquier tipo. Son mutables y dinámicas, lo cual es la principal diferencia con los sets y las tuplas.

Crear listas Python

Las listas en Python son uno de los tipos o estructuras de datos más versátiles del lenguaje, ya que permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea. Si vienes de otros lenguajes de programación, se podría decir que son similares a los arrays.

```
lista = [1, 2, 3, 4]
```

También se puede crear usando `list`y pasando un objeto iterable.

```
lista = list("1234")
```

Una lista sea crea con`[]`separando sus elementos con comas`,`. Una gran ventaja es que pueden almacenar tipos de datos distintos.

```
lista = [1, "Hola", 3.67, [1, 2, 3]]
```

Algunas propiedades de las listas:

- Son **ordenadas**, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos **arbitrarios**
- Pueden ser **indexadas** con `[i]`.
- Se pueden **anidar**, es decir, meter una dentro de la otra.
- Son **mutables**, ya que sus elementos pueden ser modificados.
- Son **dinámicas**, ya que se pueden añadir o eliminar elementos.

Acceder y modificar listas

Si tenemos una lista `a` con 3 elementos almacenados en ella, podemos acceder a los mismos usando corchetes y un índice, que va desde `0` a `n-1` siendo `n` el tamaño de la lista.

```
a = [90, "Python", 3.87]
print(a[0]) #90
print(a[1]) #Python
print(a[2]) #3.87
```

Se puede también acceder al último elemento usando el índice `[-1]`.

```
a = [90, "Python", 3.87]
print(a[-1]) #3.87
```

De la misma manera, al igual que `[-1]` es el último elemento, podemos acceder a `[-2]` que será el penúltimo.

```
print(a[-2]) #Python
```

Y si queremos modificar un elemento de la lista, basta con asignar con el operador `=` el nuevo valor.

```
a[2] = 1
print(a) #[90, 'Python', 1]
```

Un elemento puede ser eliminado con diferentes métodos como veremos a continuación, o con `del` y la lista con el índice a eliminar.

```
l = [1, 2, 3, 4, 5]
del l[1]
print(l) #[1, 3, 4, 5]
```

También podemos tener **listas anidadas**, es decir, una lista dentro de otra. Incluso podemos tener una lista dentro de otra lista y a su vez dentro de otra lista. Para acceder a sus elementos sólo tenemos que usar `[]` tantas veces como niveles de anidado tengamos.

```
x = [1, 2, 3, ['p', 'q', [5, 6, 7]]]
print(x[3][0])    #p
print(x[3][2][0]) #5
print(x[3][2][2]) #7
```

También es posible crear sublistas más pequeñas de una más grande. Para ello debemos de usar `:` entre corchetes, indicando a la izquierda el valor de inicio, y a la izquierda el valor final que no está incluido. Por lo tanto `[0:2]` creará una lista con los elementos `[0]` y `[1]` de la original.

```
l = [1, 2, 3, 4, 5, 6]
print(l[0:2]) #[1, 2]
print(l[2:6]) #[3, 4, 5, 6]
```

Y de la misma manera podemos modificar múltiples valores de la lista a la vez usando `:`.

```
l = [1, 2, 3, 4, 5, 6]
l[0:3] = [0, 0, 0]
print(l) #[0, 0, 0, 4, 5, 6]
```

Hay ciertos operadores como el `+` que pueden ser usados sobre las listas.

```
l = [1, 2, 3]
l += [4, 5]
print(l) #[1, 2, 3, 4, 5]
```

Y una funcionalidad muy interesante es que se puede asignar una lista con `n` elementos a `n` variables.

```
l = [1, 2, 3]
x, y, z = l
print(x, y, z) #1 2 3
```

Iterar listas

En Python es muy fácil iterar una lista, mucho más que en otros lenguajes de programación.

```
lista = [5, 9, 10]
for l in lista:
    print(l)
#5
#9
#10
```

Si necesitamos un índice acompañado con la lista, que tome valores desde `0` hasta `n-1`, se puede hacer de la siguiente manera.

```
lista = [5, 9, 10]
for index, l in enumerate(lista):
    print(index, l)
#0 5
```

```
#1 9
#2 10
```

O si tenemos dos listas y las queremos iterar a la vez, también es posible hacerlo.

```
lista1 = [5, 9, 10]
lista2 = ["Jazz", "Rock", "Djent"]
for l1, l2 in zip(lista1, lista2):
    print(l1, l2)
#5 Jazz
#9 Rock
#10 Djent
```

Y por supuesto, también se pueden iterar las listas usando los índices como hemos visto al principio, y haciendo uso de `len()`, que nos devuelve la longitud de la lista.

```
lista1 = [5, 9, 10]
for i in range(0, len(lista)):
    print(lista1[i])
#5
#9
#10
```

Métodos Listas

`append(<obj>)`

El método `append()` añade un elemento al final de la lista.

```
l = [1, 2]
l.append(3)
print(l) #[1, 2, 3]
```

`extend(<iterable>)`

El método `extend()` permite añadir una lista a la lista inicial.

```
l = [1, 2]
l.extend([3, 4])
print(l) #[1, 2, 3, 4]
```

`insert(<index>, <obj>)`

El método `insert()` añade un elemento en una posición o índice determinado.

```
l = [1, 3]
```

```
l.insert(1, 2)
print(l) #[1, 2, 3]
```

`remove(<obj>)`
El método `remove()` recibe como argumento un objeto y lo borra de la lista.

```
l = [1, 2, 3]
l.remove(3)
print(l) #[1, 2]
```

`pop(index=-1)`
El método `pop()` elimina por defecto el último elemento de la lista, pero si se pasa como parámetro un `índice` permite borrar elementos diferentes al último.

```
l = [1, 2, 3]
l.pop()
print(l) #[1, 2]
```

`reverse()`
El método `reverse()` inverte el órden de la lista.

```
l = [1, 2, 3]
l.reverse()
print(l) #[3, 2, 1]
```

`sort()`
El método `sort()` ordena los elementos de menos a mayor por defecto.

```
l = [3, 1, 2]
l.sort()
print(l) #[1, 2, 3]
```

Y también permite ordenar de mayor a menor si se pasa como parámetro `reverse=True`.

```
l = [3, 1, 2]
l.sort(reverse=True)
print(l) #[3, 2, 1]
```

`index(<obj>[,index])`
El método `index()` recibe como parámetro un objeto y devuelve el índice de su primera aparición. Como hemos visto en otras ocasiones, el índice del primer elemento es el `0`.

```
l = ["Periphery", "Intervals", "Monuments"]
print(l.index("Intervals"))
```

También permite introducir un parámetro opcional que representa el índice desde el que comenzar la búsqueda del objeto. Es como si ignorara todo lo que hay antes de ese índice para la búsqueda, en este caso el `4`.

```
l = [1, 1, 1, 1, 2, 1, 4, 5]
print(l.index(1, 4)) #5
```

4.1.6 Set

Los sets en Python son una estructura de datos usada para almacenar elementos de una manera similar a las listas, pero con ciertas diferencias.

Crear set Python

Los `set` en Python son un tipo que permite almacenar varios elementos y acceder a ellos de una forma muy **similar a las listas** pero con ciertas diferencias:

- Los elementos de un set son **único**, lo que significa que no puede haber elementos duplicados.
- Los set son **desordenados**, lo que significa que no mantienen el orden de cuando son declarados.
- Sus elementos deben ser **inmutables**.

Para entender mejor los `sets`, es necesario entender ciertos conceptos matemáticos como la **teoría de conjuntos**.

Para **crear** un set en Python se puede hacer con `set()` y pasando como entrada cualquier tipo iterable, como puede ser una lista. Se puede ver como a pesar de pasar elementos duplicados como dos `8` y en un orden determinado, al imprimir el set no conserva ese orden y los duplicados se han eliminado.

```
s = set([5, 4, 6, 8, 8, 1])
print(s)      #{1, 4, 5, 6, 8}
print(type(s)) #<class 'set'>
```

Se puede hacer lo mismo haciendo uso de `{}` y sin usar la palabra `set()` como se muestra a continuación.

```
s = {5, 4, 6, 8, 8, 1}
print(s)      #{1, 4, 5, 6, 8}
print(type(s)) #<class 'set'>
```

Operaciones con sets

A diferencia de las listas, en los set no podemos modificar un elemento a través de su índice. Si lo intentamos, tendremos un `TypeError`.

```
s = set([5, 6, 7, 8])
#s[0] = 3 #Error! TypeError
```

Los elementos de un `set` deben ser **inmutables**, por lo que un elemento de un `set` no puede ser ni un diccionario ni una lista. Si lo intentamos tendremos un `TypeError`

```
lista = ["Perú", "Argentina"]
#s = set(["México", "España", lista]) #Error! TypeError
```

Los `sets` se pueden iterar de la misma forma que las listas.

```
s = set([5, 6, 7, 8])
for ss in s:
    print(ss) #8, 5, 6, 7
```

Con la función `len()` podemos saber la longitud total del `set`. Como ya hemos indicado, los duplicados son eliminados.

```
s = set([1, 2, 2, 3, 4])
print(len(s)) #4
```

También podemos saber si un elemento está presente en un set con el operador `in`. Se el valor existe en el set, se devolverá `True`.

```
s = set(["Guitarra", "Bajo"])
print("Guitarra" in s) #True
```

Los `sets` tienen además diferentes funcionalidades, que se pueden aplicar en forma de operador o de método. Por ejemplo, el operador `|` nos permite realizar la **unión** de dos sets, lo que equivale a juntarlos. El equivalente es el método `union()` que vemos a continuación.

```
s1 = set([1, 2, 3])
s2 = set([3, 4, 5])
print(s1 | s2) #{1, 2, 3, 4, 5}
```

Métodos sets

`s.add(<elem>)`

El método `add()` permite añadir un elemento al `set`.

```
l = set([1, 2])
l.add(3)
print(l) #{1, 2, 3}
```

`s.remove(<elem>)`

El método `remove()` elimina el elemento que se pasa como parámetro. Si no se encuentra, se lanza la excepción `KeyError`.

```
s = set([1, 2])
s.remove(2)
print(s) #{1}
```

`s.discard(<elem>)`

El método `discard()` es muy parecido al `remove()`, borra el elemento que se pasa como parámetro, y si no se encuentra no hace nada.

```
s = set([1, 2])
s.discard(3)
```

```
print(s) #{1, 2}
```

`s.pop()`

El método `pop()` elimina un elemento aleatorio del `set`.

```
s = set([1, 2])
s.pop()
print(s) #{2}
```

`s.clear()`

El método `clear()` elimina todos los elementos de `set`.

```
s = set([1, 2])
s.clear()
print(s) #set()
```

Otros

Los `sets` cuentan con una gran cantidad de métodos que permiten realizar operaciones con dos o más, como la **unión** o la **intersección**.

Podemos calcular la **unión** entre dos sets usando el método `union()`. Esta operación representa la "mezcla" de ambos sets. Nótese que el método puede ser llamado con más parámetros de entrada, y su resultado será la unión de todos los sets.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1.union(s2)) #{1, 2, 3, 4, 5}
```

También podemos calcular la **intersección** entre dos o más set. Su resultado serán aquellos elementos que pertenecen a ambos sets.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1.intersection(s2)) #{3}
```

Los `set` en Python tiene gran cantidad de métodos, por lo que lo dejaremos para otro capítulo, pero aquí os dejamos con un listado de ellos:

- `s1.union(s2[, s3 ...])`
- `s1.intersection(s2[, s3 ...])`
- `s1.difference(s2[, s3 ...])`
- `s1.symmetric_difference(s2)`
- `s1.isdisjoint(s2)`
- `s1.issubset(s2)`
- `s1.issuperset(s2)`
- `s1.update(s2[, s3 ...])`
- `s1.intersection_update(s2[, s3 ...])`
- `s1.difference_update(s2[, s3 ...])`
- `s1.symmetric_difference_update(s2)`

4.1.7 Tupla

Las tuplas en Python son un tipo o estructura de datos que permite almacenar datos de una manera muy parecida a las listas, con la salvedad de que son inmutables.

Crear tupla Python

Las tuplas en Python o `tuples` son muy similares a las listas, pero con dos diferencias. Son **inmutables**, lo que significa que no pueden ser modificadas una vez declaradas, y en vez de inicializarse con corchetes se hace con `()`. Dependiendo de lo que queramos hacer, **las tuplas pueden ser más rápidas**.

```
tupla = (1, 2, 3)
print(tupla) #(1, 2, 3)
```

También pueden declararse sin `()`, separando por `,` todos sus elementos.

```
tupla = 1, 2, 3
print(type(tupla)) #<class 'tuple'>
print(tupla)       #(1, 2, 3)
```

Operaciones con tuplas

Como hemos comentado, las tuplas son tipos **inmutables**, lo que significa que una vez asignado su valor, no puede ser modificado. Si se intenta, tendremos un `TypeError`.

```
tupla = (1, 2, 3)
#tupla[0] = 5 # Error! TypeError
```

Al igual que las listas, las tuplas también pueden ser anidadas.

```
tupla = 1, 2, ('a', 'b'), 3
print(tupla)       #(1, 2, ('a', 'b'), 3)
print(tupla[2][0]) #a
```

Y también es posible convertir una lista en tupla haciendo uso de al función `tuple()`.

```
lista = [1, 2, 3]
tupla = tuple(lista)
print(type(tupla)) #<class 'tuple'>
print(tupla)       #(1, 2, 3)
```

Se puede **iterar** una tupla de la misma forma que se hacía con las listas.

```
tupla = [1, 2, 3]
for t in tupla:
    print(t) #1, 2, 3
```

Y se puede también asignar el valor de una tupla con n elementos a n variables.

```
l = (1, 2, 3)
x, y, z = l
print(x, y, z) #1 2 3
```

Aunque tal vez no tenga mucho sentido a nivel práctico, es posible crear una tupla de un solo elemento. Para ello debes usar `,` antes del paréntesis, porque de lo contrario `(2)` sería interpretado como `int`.

```
tupla = (2,)
print(type(tupla)) #<class 'tuple'>
```

Métodos tuplas

`count(<obj>)`

El método `count()` cuenta el número de veces que el objeto pasado como parámetro se ha encontrado en la lista.

```
l = [1, 1, 1, 3, 5]
print(l.count(1)) #3
```

`index(<obj>[,index])`

El método `index()` busca el objeto que se le pasa como parámetro y devuelve el índice en el que se ha encontrado.

```
l = [7, 7, 7, 3, 5]
print(l.index(5)) #4
```

En el caso de no encontrarse, se devuelve un `ValueError`.

```
l = [7, 7, 7, 3, 5]
#print(l.index(35)) #Error! ValueError
```

El método `index()` también acepta un segundo parámetro opcional, que indica a partir de que índice empezar a buscar el objeto.

```
l = [7, 7, 7, 3, 5]
print(l.index(7, 2)) #2
```

4.1.8 Diccionario

Los diccionarios en Python son una estructura de datos que permite almacenar su contenido en forma de llave y valor.

Crear diccionario Python

Un diccionario en Python es una colección de elementos, donde cada uno tiene una llave `key` y un valor `value`. Los diccionarios se pueden crear con paréntesis `{}` separando con una coma cada par `key: value`. En el siguiente ejemplo tenemos tres `keys` que son el nombre, la edad y el documento.

```
d1 = {
  "Nombre": "Sara",
  "Edad": 27,
  "Documento": 1003882
}
print(d1)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

Otra forma equivalente de crear un diccionario en Python es usando `dict()` e introduciendo los pares `key: value` entre paréntesis.

```
d2 = dict([
      ('Nombre', 'Sara'),
      ('Edad', 27),
      ('Documento', 1003882),
])
print(d2)
#{'Nombre': 'Sara', 'Edad': '27', 'Documento': '1003882'}
```

También es posible usar el constructor `dict()` para crear un diccionario.

```
d3 = dict(Nombre='Sara',
          Edad=27,
          Documento=1003882)
print(d3)
#{'Nombre': 'Sara', 'Edad': 27, 'Documento': 1003882}
```

Algunas propiedades de los diccionario en Python son las siguientes:

- Son **dinámicos**, pueden crecer o decrecer, se pueden añadir o eliminar elementos.
- Son **indexados**, los elementos del diccionario son accesibles a través del `key`.
- Y son **anidados**, un diccionario puede contener a otro diccionario en su campo `value`.

Acceder y modificar elementos

Se puede acceder a sus elementos con `[]` o también con la función `get()`

```
print(d1['Nombre'])     #Sara
print(d1.get('Nombre')) #Sara
```

Para modificar un elemento basta con usar `[]` con el nombre del `key` y asignar el valor que queremos.

```
d1['Nombre'] = "Laura"
print(d1)
#{'Nombre': Laura', 'Edad': 27, 'Documento': 1003882}
```

Si el `key` al que accedemos no existe, se añade automáticamente.

```
d1['Direccion'] = "Calle 123"
print(d1)
#{'Nombre': 'Laura', 'Edad': 27, 'Documento': 1003882, 'Direccion':
'Calle 123'}
```

Iterar diccionario

Los diccionarios se pueden iterar de manera muy similar a las listas u otras estructuras de datos. Para imprimir los `key`.

```
# Imprime los key del diccionario
for x in d1:
    print(x)
#Nombre
#Edad
#Documento
#Direccion
```

Se puede imprimir también solo el `value`.

```
# Impre los value del diccionario
for x in d1:
    print(d1[x])
#Laura
#27
#1003882
#Calle 123
```

O si queremos imprimir el `key` y el `value` a la vez.

```
# Imprime los key y value del diccionario
for x, y in d1.items():
    print(x, y)
#Nombre Laura
#Edad 27
#Documento 1003882
#Direccion Calle 123
```

Diccionarios anidados

Los diccionarios en Python pueden contener uno dentro de otro. Podemos ver como los valores anidado uno y dos del diccionario `d` contienen a su vez otro diccionario.

```
anidado1 = {"a": 1, "b": 2}
anidado2 = {"a": 1, "b": 2}
d = {
  "anidado1" : anidado1,
  "anidado2" : anidado2
}
print(d)
#{'anidado1': {'a': 1, 'b': 2}, 'anidado2': {'a': 1, 'b': 2}}
```

Métodos diccionarios Python

`clear()`

El método `clear()` elimina todo el contenido del diccionario.

```
d = {'a': 1, 'b': 2}
d.clear()
print(d) #{}
```

`get(<key>[,<default>])`

El método `get()` nos permite consultar el `value` para un `key` determinado. El segundo parámetro es opcional, y en el caso de proporcionarlo es el valor a devolver si no se encuentra la `key`.

```
d = {'a': 1, 'b': 2}
print(d.get('a')) #1
print(d.get('z', 'No encontrado')) #No encontrado
```

`items()`

El método `items()` devuelve una lista con los `keys` y `values` del diccionario. Si se convierte en `list` se puede indexar como si de una lista normal se tratase, siendo los primeros elementos las `key` y los segundos los `value`.

```
d = {'a': 1, 'b': 2}
it = d.items()
print(it)          #dict_items([('a', 1), ('b', 2)])
print(list(it))     #[('a', 1), ('b', 2)]
print(list(it)[0][0]) #a
```

`keys()`

El método `keys()` devuelve una lista con todas las `keys` del diccionario.

```
d = {'a': 1, 'b': 2}
k = d.keys()
print(k)      #dict_keys(['a', 'b'])
print(list(k)) #['a', 'b']
```

`values()`

El método `values()` devuelve una lista con todos los `values` o valores del diccionario.

```
d = {'a': 1, 'b': 2}
print(list(d.values())) #[1, 2]
```

`pop(<key>[,<default>])`

El método `pop()` busca y elimina la `key` que se pasa como parámetro y devuelve su valor asociado. Daría un error si se intenta eliminar una `key` que no existe.

```
d = {'a': 1, 'b': 2}
d.pop('a')
print(d) #{'b': 2}
```

También se puede pasar un segundo parámetro que es el valor a devolver si la `key` no se ha encontrado. En este caso si no se encuentra no habría error.

```
d = {'a': 1, 'b': 2}
d.pop('c', -1)
print(d) #{'a': 1, 'b': 2}
```

`popitem()`

El método `popitem()` elimina de manera aleatoria un elemento del diccionario.

```
d = {'a': 1, 'b': 2}
d.popitem()
print(d)
```

```
#{'a': 1}
```

update(<obj>)

El método `update()` se llama sobre un diccionario y tiene como entrada otro diccionario. Los `value` son actualizados y si alguna `key` del nuevo diccionario no esta, es añadida.

```
d1 = {'a': 1, 'b': 2}
d2 = {'a': 0, 'd': 400}

d1.update(d2)

print(d1)

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

4.2 Tipos de operadores

4.2.1 De asignación

Anteriormente hemos visto los operadores aritméticos, que usaban dos números para calcular una operación aritmética (como suma o resta) y devolver su resultado. En este caso, los operadores de asignación o *assignment operators* nos **permiten realizar una operación y almacenar su resultado en la variable inicial**. Podemos ver como realmente el único operador nuevo es el =. El resto son abreviaciones de otros operadores que habíamos visto con anterioridad. Ponemos un ejemplo con x=7

Operador	Ejemplo	Equivalente
=	x=7	x=7
+=	x+=2	x=x+2 = 7
-=	x-=2	x=x-2 = 5
*=	x*=2	x=x*2 = 14
/=	x/=2	x=x/2 = 3.5
%=	x%=2	x=x%2 = 1
//=	x//=2	x=x//2 = 3
**=	x**=2	x=x**2 = 49
&=	x&=2	x=x&2 = 2
|=	x|=2	x=x|2 = 7
^=	x^=2	x=x^2 = 5
>>=	x>>=2	x=x>>2 = 1
&lt&lt=	x&lt&lt=2	x=x&lt&lt2 = 28

```
a=7; b=2
print("Operadores de asignación")
x=a; x+=b;  print("x+=", x)  # 9
x=a; x-=b;  print("x-=", x)  # 5
x=a; x*=b;  print("x*=", x)  # 14
x=a; x/=b;  print("x/=", x)  # 3.5
x=a; x%=b;  print("x%=", x)  # 1
x=a; x//=b; print("x//=", x) # 3
x=a; x**=b; print("x**=", x) # 49
x=a; x&=b;  print("x&=", x)  # 2
x=a; x|=b;  print("x|=", x)  # 7
x=a; x^=b; print("x^=", x)   # 5
```

```
x=a; x>>=b; print("x>>=", x) # 1
x=a; x<<=b; print("x<<=", x) # 28
```

Operador =
El operador `=` prácticamente no necesita explicación, simplemente asigna a la variable de la izquierda el contenido que le ponemos a la derecha. Ponemos en negrita variable porque si hacemos algo del tipo `3=5` tendremos un error. Como siempre, nunca te fíes de nada y experimenta con ello.

```
x=2      # Uso correcto del operador =
print(x)  # 2
#3=5     # Daría error, 3 no es una variable
```

Tal vez pienses que el operador `=` es trivial y apenas merezca explicación, pero es importante explorar los límites del lenguaje. Si sabes lo que es un **puntero**, o una **referencia** tal vez el ejemplo siguiente tenga sentido para tí. Vamos a ver, si todo lo que hemos visto anteriormente es cierto, `a=[1, 2, 3]` asigna `[1, 2, 3]` a `a`, por lo que si no tocamos `a`, el valor de `a` deberá ser siempre `[1, 2, 3]`. Bueno, pues en el siguiente ejemplo vemos como eso no es así, el valor de `a` ha cambiado. Se podría decir que no es lo mismo `x=3` con un número que `x=[1, 2, 3]` con una lista. No te preocupes si no lo has seguido, en otros capítulos lo explicaremos mejor.

```
a = [1, 2, 3]
b = a
b += [4]
print(a)
# [1, 2, 3, 4]
```

Operador +=
Como podemos ver, todos los operadores de asignación no son más que atajos para escribir otros operadores de manera más corta, y asignar su resultado a la variable inicial. El operador `+=` en `x+=1` es equivalente a `x=x+1`. Sabiendo esto, sería justo preguntarse ¿realmente merece la pena crear un operador nuevo que hace algo que ya podemos hacer pero de manera mas corta? Bien, la pregunta no es fácil de responder y en cierto modo viene heredado de lenguajes como C que en los años 1970s introdujeron esto.

```
x=5     # Ejemplo de como incrementar
x+=1    # en una unidad x
print(x)
# 6
```

Para saber más: Aunque se podría decir que el operador x+=1 es igual que x=x+1, no es del todo cierto. De hecho el operador que Python invoca por debajo es "__iadd__" en el primer caso frente a "__add__" para el segundo. A efectos prácticos, se podría considerar lo mismo, pero aquí puedes leer más sobre esto
Se puede jugar un poco con el operador `+=` y aplicarlo sobre variables que no necesariamente son números. Como vimos en otros capítulos, se podría emplear sobre una lista.

```
x=[1,2,3] # En este caso la x es una lista
x+=[4,5]  # Se aplica el operador sobre otra lista
print(x)  # Y el resultado de la unión de ambas
# [1, 2, 3, 5, 6]
```

Es muy importante, que si `x` es una lista, no podemos aplicar el operador `+=` con un elemento que no sea una lista, como por ejemplo, un número. El siguiente código daría error porque el operador no esta definido para un elemento lista y otro entero.

```
x=[1,2,3] #
#x+=3    # ERROR! TypeError
```

Operador -=

El operador `-=` es equivalente a restar y asignar el resultado a la variable inicial. Es decir, `x-=1` es equivalente a `x=x-1`. Si vienes de otros lenguajes de programación, tal vez conozcas la forma `x--`, pero en Python no existe. El operador es muy usado para decrementar el valor de una variable.

```
i = 5
i -= 1
print(i) # 4
```

Y algo que nunca se haría en la realidad, pero nos permite explorar los límites del lenguaje, sería restar -1, lo que equivale a sumar uno. Pero de verdad, no hagas esto, en serio.

```
i = 0
i-=-1    # Aumenta el contador
print(i) # 1
```

Operador *=

El operador `*=` equivale a multiplicar una variable por otra y almacenar el resultado en la primera, es decir `x*=2` equivale a `x=x*2`. Hasta ahora hemos usado todos los operadores de asignación con una variable y un número, pero es totalmente correcto hacerlo con dos variables.

```
a=10; b=2 # Inicializamos a 10 y 20
a*=b      # Usando dos variables
print(a)  # 20
```

Operador /=

El operador `/=` equivale a dividir una variable por otra y almacenar el resultado en la primera, es decir, `x/=2` equivale a `x=x/2`. Acuérdate que en otros capítulos vimos como `5/3` en versiones antiguas de Python, podía causar problemas ya que el resultado no era un numero entero. En el siguiente ejemplo podemos ver como Python hace el trabajo por nosotros, y cambia el tipo de la variable `x` de lo que inicialmente era `int` a un `float` con el objetivo de que el nuevo valor pueda ser almacenado.

```
x = 10
print(type(x)) # <class 'int'>
x/=3
print(type(x)) # <class 'float'>
```

Operador %=
El operador %= equivale a hacer el módulo de la división de dos variables y almacenar su resultado en la primera.

```
x = 3
x%=2
print(x) # 1
```

Una curiosidad a tener en cuenta, es que el operador módulo tiene diferentes comportamientos en Python del que tiene en otros lenguajes como C cuando se usan números negativos tanto de dividendo como de divisor. Así que ten cuidado si haces cosas como las siguientes.

```
print(-5%-3) # -2
print(5%-3)  # -1
print(-5%3)  #  1
print(5%3)   #  2
```

Operador //=
El operador `//=` realiza la operación cociente entre dos variables y almacena el resultado en la primera. El equivalente de `x//=2` sería `x=x//2`.

```
x=5     # El resultado es el cociente
x//=3   # de la división
print(x) # 1
```

Operador **=
El operador `**=` realiza la operación exponente del primer número elevado al segundo, y almacena el resultado en la primera variable. El equivalente de `x**=2` sería `x=x**2`.

```
x=5     # Eleva el número al cuadrado
x**=2   # y guarda el resultado en la misma
print(x) # 25
```

Otro ejemplo similar, sería empleando un exponente negativo, algo que es totalmente válido y equivale matemáticamente al inverso del número elevado al exponente en positivo. Dicho de otra forma, x−2 equivale a 1/x2.

```
x=5     # Elevar 5 a -2 equivale a dividir
x**=-2  # uno entre 25.
print(x) # 0.04
```

Operador &=
El operador `&=` realiza la comparación & bit a bit entre dos variables y almacena su resultado en la primera. El equivalente de `x&=1` sería `x=x&1`

```
a = 0b101010
a&= 0b111111
print(bin(a))
# 0b101010
```

Operador |=
El operador `|=` realiza el operador `|` elemento a elemento entre dos variables y almacena su resultado en la primera. El equivalente de `x|=2` sería `x=x|2`

```
a = 0b101010
a|= 0b111111
print(bin(a))
# 0b111111
```

Operador ^=
El operador `^=` realiza el operador `^` elemento a elemento entre dos variables y almacena su resultado en la primera. El equivalente de `x^=2` sería `x=x^2`

```
a = 0b101010
a^= 0b111111
print(bin(a))
# 0b10101
```

Operador »=
El operador `>>=` es similar al operador `>>` pero permite almacenar el resultado en la primera variable. Por lo tanto `x>>=3` sería equivalente a `x=x>>3`

```
x = 10
x>>=1
print(x) # 5
```

Es importante tener cuidado y saber el tipo de la variable `x` antes de aplicar este operador, ya que se podría dar el caso de que `x` fuera una variable tipo `float`. En ese caso, tendríamos un error porque el operador `>>` no esta definido para `float`.

```
x=10.0        # Si la x es float
print(type(x)) # <class 'float'>
#x>>=1        # ERROR! TypeError
```

Operador «=
Muy similar al anterior, `<<=` aplica el operador `<<` y almacena su contenido en la primera variable. El equivalente de `x<<=1` sería `x=x<<1`

```
x=10    # Inicializamos a 10
x<<=1   # Desplazamos 1 a la izquierda
print(x) # 20
```

Sería justo pensar que si `<<` realiza un desplazamiento de bits a la izquierda y `>>` lo realiza a la derecha, tal vez un desplazamiento `<<` una unidad, podría equivaler a `-1` desplazamiento a la derecha.

```
#x<<=-1 # ERROR! Python no define un desplazamiento negativo a la izquierda
#x>>=-1 # ERROR! Python no define un desplazamiento negativo a la derecha
```

4.2.2 Aritméticos

Los operadores aritméticos o *arithmetic operators* son los más comunes que nos podemos encontrar, y nos **permiten realizar operaciones aritméticas** sencillas, como pueden ser la suma, resta o exponente. A continuación, condensamos en la siguiente tabla todos ellos con un ejemplo, donde `x=10` y `y=3`.

Operador	Nombre	Ejemplo
+	Suma	x + y = 13
-	Resta	x - y = 7
*	Multiplicación	x * y = 30
/	División	x/y = 3.333
%	Módulo	x%y = 1
**	Exponente	x ** y = 1000
//	Cociente	3

```
x = 10; y = 3
print("Operadores aritméticos")
print("x+y =", x+y)   #13
print("x-y =", x-y)   #7
print("x*y =", x*y)   #30
print("x/y =", x/y)   #3.3333333333333335
print("x%y =", x%y)   #1
print("x**y =", x**y) #1000
```

```
print("x//y =", x//y) #3
```

Operador +

El operador + suma los números presentes a la izquierda y derecha del operador. Recalcamos lo de números porque no tendría sentido sumar dos cadenas de texto, o dos listas, pero en Python es posible hacer este tipo de cosas.

```
print(10 + 3) # 13
```

Es posible sumar también dos cadenas de texto, pero la suma no será aritmética, sino que se unirán ambas cadenas en una. También se pueden sumar dos listas, cuyo resultado es la unión de ambas.

```
print("2" + "2")      # 22

print([1, 3] + [6, 7]) # [1, 3, 6, 7]
```

Operador -

El operador - resta los números presentes a la izquierda y derecha del operador. A diferencia el operador + en este caso no podemos restar cadenas o listas.

```
print(10 - 3) #7
```

Operador *

El operador * multiplica los números presentes a la izquierda y derecha del operador.

```
print(10 * 3) #30
```

Como también pasaba con el operador + podemos hacer cosas “raras” con *. Explicar porque pasan estas cosas es un poquito más complejo, por lo que lo dejamos para otro capítulo, donde explicaremos como definir el comportamiento de determinados operadores para nuestras clases.

```
print("Hola" * 3) #HolaHolaHola
```

Operador /

El operador / divide los números presentes a la izquierda y derecha del operador. Un aspecto importante a tener en cuenta es que si realizamos una división cuyo resultado no es entero (es decimal) podríamos tener problemas. En Python 3 esto no supone un problema porque el mismo se encarga de convertir los números y el resultado que se muestra si es decimal.

```
print(10/3) #3.3333333333333335

print(1/2)  #0.5
```

Sin embargo, en Python 2, esto hubiera tenido un resultado diferente. El primer ejemplo `10/3=3` y el segundo `1/2=0`. El comportamiento realmente sería el de calcular el cociente y no la división.
Para saber más: Si quieres saber más acerca de este cambio del operador de división, puedes leer la la PEP238

Operador %

El operador % realiza la operación módulo entre los números presentes a la izquierda y la derecha. Se trata de calcular el resto de la división entera entre ambos números. Es decir, si dividimos 10 entre 3, el cociente sería 3 y el resto 1. Ese resto es lo que calcula el módulo.

```
print(10%3) # 1

print(10%2) # 0
```

Operador **
El operador `**` realiza el exponente del número a la izquierda elevado al número de la derecha.

```
print(10**3) #1000
print(2**2)  #4
```

Si ya has usado alguna vez Python, tal vez hayas oido hablar de la librería `math`. En esta librería también tenemos una función llamada `pow()` que es equivalente al operador `**`.

```
import math
print(math.pow(10, 3)) #1000.0
```

Para saber más: Puedes consultar más información de la librería math en la documentación oficial de Python

Operador //
Por último, el operador `//` calcula el cociente de la división entre los números que están a su izquierda y derecha.

```
print(10//3)  #3
print(10//10) #1
```

Tal vez te hayas dado cuenta que el operador cociente `//` está muy relacionado con el operador módulo %. Volviendo a las lecciones del colegio sobre la división, recordaremos que el Dividendo `D` es igual al divisor `d` multiplicado por el cociente `c` y sumado al resto `r`, es decir `D=d*c+r`. Se puede ver como en el siguiente ejemplo, `10//3` es el cociente y `10%3` es el resto. Al aplicar la fórmula, verificamos que efectivamente `10` era el dividendo.

```
D = 10 # Número que queremos dividir
d = 3  # Número entre el que queremos dividir
print(3 * (10//3) + 10%3) # 10
```

Orden de aplicación
En los ejemplos anteriores simplemente hemos aplicado un operador a dos números sin mezclarlos entre ellos. También es posible tener varios operadores en la misma línea de código, y en este caso es muy importante tener en cuenta las prioridades de cada operador y cual se aplica primero. Ante la duda siempre podemos usar paréntesis, ya que todo lo que está dentro de un paréntesis se evaluará conjuntamente, pero es importante saber las prioridades.
El orden de prioridad sería el siguiente para los operadores aritméticos, siendo el primero el de mayor prioridad:

- `()` Paréntesis
- `**` Exponente
- `-x` Negación
- `* / //` Multiplicación, División, Cociente, Módulo
- `+ -` Suma, Resta

```
print(10*(5+3)) # Con paréntesis se realiza primero la suma
# 80
print(10*5+3)   # Sin paréntesis se realiza primero la multiplicación
```

```
# 53
print(3*3+2/5+5%4) # Primero se multiplica y divide, después se suma
#10.4
print(-2**4)      # Primero se hace la potencia, después se aplica el signo
#-16
```

Para saber más: Si quieres saber más sobre el orden de prioridad de diferentes operadores, aquí tienes la documentación oficial de Python

4.2.3 Relacionales

Los operadores relacionales, o también llamados *comparison operators* nos permiten saber la **relación existente entre dos variables**. Se usan para saber si por ejemplo un número es mayor o menor que otro. Dado que estos operadores indican si se cumple o no una operación, el valor que devuelven es `True` o `False`. Veamos un ejemplo con `x=2` e `y=3`

Operador	Nombre	Ejemplo
==	Igual	x == y = False
!=	Distinto	x != y = True
>	Mayor	x > y = False
&lt	Menor	x &lt y = True
>=	Mayor o igual	x >= y = False
&lt=	Menor o igual	x &lt y = True

```
x=2; y=3
print("Operadores Relacionales")
print("x==y =", x==y) # False
print("x!=y =", x!=y) # True
print("x>y  =", x>y)  # False
print("x<y  =", x<y)  # True
print("x>=y =", x>=y) # False
print("x<=y =", x<=y) # True
```

Operador ==
El operador `==` permite comparar si las variables introducidas a su izquierda y derecha son iguales. Muy importante no confundir con `=`, que es el operador de asignación.

```
print(4==4)         # True
print(4==5)         # False
print(4==4.0)       # True
print(0==False)     # True
print("asd"=="asd")  # True
print("asd"=="asdf") # False
print(2=="2")       # False
print([1, 2, 3] == [1, 2, 3]) # True
```

Operador !=
El operador `!=` devuelve `True` si los elementos a comparar son iguales y `False` si estos son distintos. De hecho, una vez definido el operador `==`, no sería necesario ni explicar `!=` ya que hace exactamente lo contrario. Definido primero, definido el segundo. Es decir, si probamos con los mismos ejemplo que el apartado anterior, veremos como el resultado es el contrario, es decir `False` donde era `True` y viceversa.

```
print(4!=4)         # False
print(4!=5)         # True
print(4!=4.0)       # False
print(0!=False)     # False
print("asd"!="asd")  # False
print("asd"!="asdf") # True
print(2!="2")       # True
print([1, 2, 3] != [1, 2, 3]) # False
```

Operador >
El operador `>` devuelve `True` si el primer valor es mayor que el segundo y `False` de lo contrario.

```
print(5>3) # True
print(5>5) # False
```

Algo bastante curioso, es como Python trata al tipo booleano. Por ejemplo, podemos ver como `True` es igual a `1`, por lo que podemos comprar el tipo `True` como si de un número se tratase.

```
print(True==1)     # True
print(True>0.999)  # True
```

Para saber más: De hecho, el tipo bool en Python hereda de la clase int. Si quieres saber más acerca del tipo bool en Python puedes leer la PEP285
También se pueden comparar listas. Si los elementos de la lista son numéricos, se comparará elemento a elemento.

```
print([1, 2] > [10, 10]) # False
```

Operador <
El operador `<` devuelve `True` si el primer elemento es mayor que el segundo. Es totalmente válido aplicar operadores relacionales como `<` sobre cadenas de texto, pero el comportamiento es un tanto difícil de ver a simple vista. Por ejemplo `abc` es menor que `abd` y `A` es menor que `a`

```
print("abc" < "abd") # True

print("A"<"a")       # True
```

Para el caso de `A` y `a` la explicación es muy sencilla, ya que `Python` lo que en realidad está comparando es el valor entero `Unicode` que representa tal caracter. La función `ord()` nos da ese valor. Por lo tanto cuando hacemos `"A"<"a"` lo que en realidad hacemos es comprar dos números.

```
print(ord('A')) # 65

print(ord('a')) # 97
```

Para saber más: En el siguiente enlace tienes más información de como Python compara variables que no son números.

Operador >=
Similar a los anteriores, `>=` permite comparar si el primer elemento es mayor o igual que es segundo, devolviendo `True` en el caso de ser cierto.

```
print(3>=3)          # True

print([3,4] >= [3,5]) # False
```

Operador <=
De la misma manera, `<=` devuelve `True` si el primer elemento es menor o igual que el segundo. Nos podemos encontrar con cosas interesantes debido a la precisión numérica existente al representar valores, como el siguiente ejemplo.

```
print(3<=2.9999999999999999)
```

4.2.4 Lógicos

Los operadores lógicos o *logical operators* nos **permiten trabajar con valores de tipo booleano**. Un valor booleano o `bool` es un tipo que solo puede tomar valores `True` o `False`. Por lo tanto, estos operadores nos permiten realizar diferentes operaciones con estos tipos, y su resultado será otro booleano. Por ejemplo, `True and True` usa el operador `and`, y su resultado será `True`. A continuación lo explicaremos mas en detalle.

Operador	Nombre	Ejemplo
and	Devuelve True si ambos elementos son True	True and True = True
or	Devuelve True si al menos un elemento es True	True or False = True
not	Devuelve el contrario, True si es Falso y viceversa	not True = False

Operador `and`

El operador `and` evalúa si el valor a la izquierda **y** el de la derecha son `True`, y en el caso de ser cierto, devuelve `True`. Si uno de los dos valores es `False`, el resultado será `False`. Es realmente un operador muy lógico e intuitivo que incluso usamos en la vida real. Si hace sol `y` es fin de semana, iré a la playa. Si ambas condiciones se cumplen, es decir que la variable `haceSol=True` y la variable `finDeSemana=True`, iré a la playa, o visto de otra forma `irALaPlaya=(haceSol and finDeSemana)`.

```
print(True and True)   # True
print(True and False)  # False
print(False and True)  # False
print(False and False) # False
```

Operador `or`

El operador `or` devuelve `True` cuando al menos uno de los elementos es igual a `True`. Es decir, evalúa si el valor a la izquierda **o** el de la derecha son `True`.

```
print(True or True)   # True
print(True or False)  # True
print(False or True)  # True
print(False or False) # False
```

Es importante notar que varios operadores pueden ser usados conjuntamente, y salvo que existan paréntesis que indiquen una cierta prioridad, el primer operador que se evaluará será el `and`. En el ejemplo que se muestra a continuación, podemos ver que tenemos dos operadores `and`. Para calcular el resultado final, tenemos que empezar por el `and` calculando el resultado en grupos de dos y arrastrando el resultado hacia el siguiente grupo. El resultado del primer grupo sería `True` ya que estamos evaluando `True and True`. Después, nos guardamos ese `True` y vamos a por el siguiente y último elemento, que es `False`. Por lo tanto hacemos `True and False` por lo que el resultado final es `False`

```
print(True and True and False)
```

```
#    |----------|
#         True  and  False
#       |-----------------|
#              False
```

También podemos mezclar los operadores. En el siguiente ejemplo empezamos por `False and True` que es `False`, después `False or True` que es `True` y por último `True or False` que es `True`. Es decir, el resultado final de toda esa expresión es `True`.

```
print(False and True or True or False)
#    False and True = False
#             Fase or True = True
#                   True or False = True
# True
```

Operador `not`

Y por último tenemos el operador `not`, que simplemente invierte `True` por `False` y `False` por `True`. También puedes usar varios `not` juntos y simplemente se irán aplicando uno tras otro. La verdad que es algo difícil de ver en la realidad, pero simplemente puedes contar el número de `not` y si es par el valor se quedará igual. Si por lo contrario es impar, el valor se invertirá.

```
print(not True)  # False
print(not False) # True
print(not not not not True) # True
```

Dado que estamos tratando con booleanos, hemos considerado que usar `True` y `False` es lo mejor y más claro, pero es totalmente válido emplear `1` y `0` respectivamente para representar ambos estados. Y por supuesto los resultados no varían

```
print(not 0) # True
print(not 1) # False
```

4.2.5 A nivel de bit

Los operadores a nivel de bit son operadores que **actúan sobre números enteros pero usando su representación binaria**. Si aún no sabes como se representa un número en forma binaria, a continuación lo explicamos.

Operador	Nombre
|	Or bit a bit
&	And bit a bit
~	Not bit a bit
^	Xor bit a bit
>>	Desplazamiento a la derecha
&lt&lt	Desplazamiento a la izquierda

Para entender los operadores de bit, es importante antes entender como se representa un número de manera binaria. Todos estamos acostumbrados a lo que se denomina representación `decimal`. Se llama así porque se usan diez números, del `0` al `9` para representar todos los valores. Nuestro decimal, es también posicional, ya que no es lo mismo un `9` en `19` que en `98`. En el primer caso, el valor es de simplemente `9`, pero en el segundo, en realidad ese 9 vale `90`.
Lo mismo pasa con la representación binaria pero con algunas diferencias. La primera diferencia es que sólo existen dos posibles números, el `0` y el `1`. La segunda diferencia es que a pesar de ser un sistema que también es posicional, los valores no son como en el caso decimal, donde el valor se multiplica por `1` en la primera posición, `10` en la segunda, `100` en la tercera, y así sucesivamente. En el caso binario, los valores son potencias de 2, es decir `1, 2, 4, 8, 16` o lo que es lo mismo `$$2^0, 2^1, 2^2, 2^3, 2^4$$`

```
# Sistema decimal
# 7264
# 7-> En realidad vale 7*1000 = 7000
#  2-> En relidad vale 2*100  = 200
#   6-> En reaidad vale 6*10  = 60
#    4 En realidad vale 4*1  = 4
#                    +---------
#         Sumando todo:      7264
```

Entonces por ejemplo el número en binario `11011` es en realidad el número `27` en decimal. Es posible convertir entre binario y decimal y viceversa. Para números pequeños se puede hacer mentalmente muy rápido, pero para números más grandes, os recomendamos hacer uso de alguna función en `Python`, como la función `bin()`

```
# Sistema binario
# 11011
```

```
# 1->  En realidad vale 1*16   = 16
#  1->  En realidad vale 1*8   = 8
#   0->  En realidad vale 1*4  = 0
#    1->  En realidad vale 1*2 = 2
#     1->  En realidad vle 1*1 = 1
#                    +--------
#       Sumando todo        27
```

Usando la función `bin()` podemos convertir un número `decimal` en `binario`. Podemos comprobar como el número anterior `11011` es efectivamente `27` en decimal. Fíjate que al imprimirlo con `Python`, se añade el prefijo `0b` antes del número. Esto es muy importante, y nos permite identificar que estamos ante un número binario.

```
print(bin(27))
# 0b11011
```

Ahora que ya sabemos como es la representación `binaria`, estamos ya en condiciones de continuar con los operadores a nivel de bit, que realizan operaciones sobre los bits de estos números binarios que acabamos de introducir.

Operador &

El operador & realiza la operación que vimos en otros capítulos `and`, pero por cada bit existente en la representación binaria de los dos números que le introducimos. Es decir, recorre ambos números en su representación binaria elemento a elemento, y hace una operación `and` con cada uno de ellos. En el siguiente ejemplo se estaría haciendo lo siguiente. Primer elemento de `a` con primer elemento de `b`, sería `1 and 1` por lo que el resultado es `1`. Ese valor se almacena en la salida. Continuamos con el segundo `1 and 0` que es `0`, tercero `0 and 1` que es `0` y por último `1 and 1` que es `1`. Por lo tanto el número resultante es `0b1001`, lo que representa el `9` en decimal.

```
a = 0b1101
b = 0b1011
print(bin(a & b))
#0b1001
```

Operador |

El operador `|` realiza la operación `or` elemento a elemento con cada uno de los bits de los números que introducimos. En el siguiente ejemplo podemos ver como el resultado es `1111` ya que siempre uno de los dos elementos es `1`. Se harían las siguientes comparaciones `1 or 1`, `1 or 0`, `0 or 1` y `1 or 1`.

```
a = 0b1101
b = 0b1011
print(bin(a | b))
# 0b1111
```

Operador ~

El operador `~` realiza la operación `not` sobre cada bit del número que le introducimos, es decir, invierte el valor de cada bit, poniendo los `0` a `1` y los `1` a `0`. El comportamiento en `Python` puede ser algo distinto del esperado. En el siguiente ejemplo, tenemos el número `40` que en binario es `101000`. Si hacemos `~101000` sería de esperar que como hemos dicho, se inviertan todos los bits y el resultado sea `010111`, pero en realidad el resultado es `101001`. Para entender porque pasa esto, te invitamos a leer más información sobre el complemento a uno y el complemento a dos.

```
a = 40
print(bin(a))
print(bin(~a))
0b101000
-0b101001
```

Para saber más: Para entender este operador mejor, es necesario saber que es el complemento a uno y a dos. Te dejamos este enlace con mas información.

Si vemos el resultado con números decimales, es equivalente a hacer `~a` sería `-a-1` como se puede ver en el siguiente ejemplo. En este caso, en vez de mostrar el valor binario mostramos el decimal, y se puede ver como efectivamente si `a=40`, tras aplicar el operador `~` el resultado es `-40-1`.

```
a = 40
print(a)
print(~a)
```

Operador ^

El operador `^` realiza la función `xor` con cada bit de las dos variables que se le proporciona. Anteriormente en otros capítulos hemos hablado de la `and` o `or`, que son operadores bastante usados y comunes. Tal vez `xor` sea menos común, y lo que hace es devolver `True` o `1` cuando hay al menos un valor `True` pero no los dos. Es decir, solo devuelve `1` para las combinaciones `1,0` y `0,1` y `0` para las demás.

```
x = 0b0110 ^ 0b1010
print(bin(x))
# 0 xor 1 = 1
# 1 xor 0 = 1
# 1 xor 1 = 0
# 0 xor 0 = 0
# 0b1100
```

Para saber más: Si quieres saber más sobre la puerta XOR, te dejamos un enlace donde se explica.

Operador >>

El operador `>>` desplaza todos los bit `n` unidades a la derecha. Por lo tanto es necesario proporcionar dos parámetros, donde el primer es el número que se desplazará o *shift* y el segundo es el número de posiciones. En el siguiente ejemplo tenemos `1000` en binario, por lo que si aplicamos un `>>2`, deberemos mover cada bit `2` unidades a la derecha. Lo que queda por la izquierda se rellena con ceros, y lo que sale por la derecha se descarta. Por lo tanto `1000>>2` será `0010`. Es importante notar que `Python` por defecto elimina los ceros a la izquierda, ya que igual que en el sistema decimal, son irrelevantes.

```
a=0b1000
print(bin(a>>2))
# 0b10
```

Operador <<

El operador `<<` es análogo al `>>` con la diferencia que en este caso el desplazamiento es realizado a la izquierda. Por lo tanto si tenemos `0001` y desplazamos `<<3`, el resultado será `1000`.

```
a=0b0001
print(bin(a<<3))
# 0b1000
```

Es importante que no nos dejemos engañar por el número de bits que ponemos a la entrada. Si por ejemplo desplazamos en `4` o en mas unidades nuestra variable `a` el número de bits que se nos mostrará también se incrementará. Con esto queremos destacar que aunque la entrada sean `4` bits, Python internamente rellena todo lo que está a la izquierda con ceros.

```
a=0b0001
print(bin(a<<10))
# 0b10000000000
```

4.2.6 De identidad

El operador de identidad o *identity operator* `is` nos indica si dos variables hacen referencia al mismo objeto. Esto implica que si dos variables distintas tienen el mismo `id()`, el resultado de aplicar el operador `is` sobre ellas será `True`.

Operador	Nombre
is	Devuelve True si hacen referencia a el mismo objeto
is not	Devuelve False si no hacen referencia a el mismo objeto

Operador `is`

El operador `is` comprueba si dos variables hacen referencia a el mismo objeto. En el siguiente ejemplo podemos ver como al aplicarse sobre `a` y `b` el resultado es `True`.

```
a = 10
b = 10

print(a is b) # True
```

Esto es debido a que Python reutiliza el mismo objeto que almacena el valor 10 para ambas variables. De hecho, si usamos la función `id()`, podemos ver que el objeto es el mismo.

```
print(id(a)) # 4397849536
print(id(b)) # 4397849536
```

Podemos ver como también, ambos valores son iguales con el operador relacional `==`, pero esto es una mera casualidad como veremos a continuación. Que dos variables tengan el mismo contenido, no implica necesariamente que hagan referencia a el mismo objeto.

```
print(a == b) # True
```

En el siguiente ejemplo, podemos ver como `a` y `b` almacenan el mismo valor, por lo que `==` nos indica `True`.

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b) # True
print(a is b) # False
```

Sin embargo, por como Python funciona por debajo, almacena el contenido en dos objetos diferentes. Al tratarse de objetos diferentes, esto hace que el operador `is` devuelva `False`.
A diferencia de antes, podemos ver como la función `id()` en este caso nos devuelve un valor diferente.

```
print(id(a)) # 4496626880
print(id(b)) # 4496626816
```

Esta diferencia puede resultar algo liosa, por lo que te recomendamos que leas más acerca de la mutabilidad en Python.
Para saber más: Si quieres saber más acerca del operador id() te dejamos este enlace a la documentación oficial.

Operador `is not`
Una vez definido `is`, es trivial definir `is not` porque es exactamente lo contrario. Devuelve `True` cuando ambas variables no hacen referencia al mismo objeto.

```
# Python crea dos objetos diferentes, uno
# para cada lista. Las listas son mutables.
a = [1, 2, 3]
b = [1, 2, 3]

print(a is not b) # True
# Python reutiliza el objeto que almacena 5
# por lo que ambas variables apuntan a el mismo
a = 5
b = 5

print(a is not b) # False
```

Para saber más: Te dejamos un enlace muy interesante con más información sobre el is y is not.

4.2.7 Membresia

Los operadores de membresía o *membership operators* son operadores que nos **permiten saber si un elemento esta contenido en una secuencia**. Por ejemplo si un número está contenido en una lista de números.

Operador `in`
El operador `in` nos permite ver si un elemento esta contenido dentro de una secuencia, como podría ser una lista. En el siguiente ejemplo se ve un caso sencillo donde se verifica si `3` esta contenido en la lista `[1, 2, 3]`. Como efectivamente lo está, el resultado es `True`.

```
print(3 in [1, 2, 3])
# True
```

Vamos a complicar las cosas un poco y explorar los límites del operador. Que pasaría si intentásemos hacer algo como lo que se ve en el siguiente ejemplo. Podría ser lógico pensar que `3 in 3` sería `True`, porque realmente si que parece que el 3 esta contenido en el segundo 3. Pues no,

el siguiente código daría un error, diciendo que la clase `int` no es iterable. En otros capítulos exploraremos más acerca de esto. Por ahora nos basta con decir que el elemento a la derecha del `in` debe ser un objeto tipo lista

```
#print(3 in 3) # Error! TypeError
```

Vamos a darle una última vuelta de tuerca. Podríamos también ver si una lista está contenida en otra lista. En este caso, la lista de la derecha del `in` es una lista embebida dentro de otra lista. Como `[1, 2]` está dentro de la segunda lista, el resultado es `True`

```
print([1, 2] in [4, [1, 2], 7])

# True
```

Operador `not in`

Por último, el operador `not in` realiza lo contrario al operador `in`. Verifica que un elemento no está contenido en otra secuencia. En el siguiente ejemplo se puede ver como `3` no es parte de la secuencia, por lo que el resultado es `False`

```
print(3 not in [1, 2, 4, 5])

# True
```

La verdad que ambos operadores `in` y `not in` son muy útiles y nos ahorran mucho trabajo. Es importante tenerlo en cuenta, porque no otros lenguajes de programación no existen tales operadores, y debemos escribir código extra para obtener tal funcionalidad. Una forma de implementar nuestro operador `in` y `is not` con una función sería la siguiente. Simplemente iteramos la lista y si encontramos el elemento que estábamos buscando devolvemos `True`, de lo contrario `False`.

5 Sentencias Condicionales

Sentencia IF

Un ejemplo sería si tenemos dos valores `a` y `b` que queremos dividir. Antes de entrar en el bloque de código que divide `a/b`, sería importante verificar que `b` es distinto de cero, ya que la división por cero no está definida. Es aquí donde entran los condicionales `if`.

```
a = 4
b = 2
if b != 0:
    print(a/b)
```

En este ejemplo podemos ver como se puede usar un `if` en Python. Con el operador `!=` se comprueba que el número `b` sea distinto de cero, y si lo es, se ejecuta el código que está indentado. Por lo tanto un `if` tiene dos partes:

- La **condición** que se tiene que cumplir para que el bloque de código se ejecute, en nuestro caso `b!=0`.
- El **bloque de código** que se ejecutará si se cumple la condición anterior.

Es muy importante tener en cuenta que la sentencia `if` debe ir terminada por `:` y el bloque de código a ejecutar debe estar indentado. Si usas algún editor de código, seguramente la indentación se producirá automáticamente al presionar enter. Nótese que el bloque de código puede también contener más de una línea, es decir puede contener más de una instrucción.

```
if b != 0:
    c = a/b
    d = c + 1
    print(d)
```

Todo lo que vaya después del `if` y esté indentado, será parte del bloque de código que se ejecutará si la condición se cumple. Por lo tanto el segundo `print()` "Fuera if" será ejecutado siempre, ya que está fuera del bloque `if`.

```
if b != 0:
    c = a/b
    print("Dentro if")
print("Fuera if")
```

Existen otros operadores que se verán en otros capítulos, como el de comparar si un número es mayor que otro. Su uso es igual que el anterior.

```
if b > 0:
    print(a/b)
```

Se puede también combinar varias condiciones entre el `if` y los `:`. Por ejemplo, se puede requerir que un número sea mayor que 5 y además menor que 15. Tenemos en realidad tres operadores usados conjuntamente, que serán evaluados por separado hasta devolver el resultado final, que será `True` si la condición se cumple o `False` de lo contrario.

```
a = 10
if a > 5 and a < 15:
    print("Mayor que 5 y menos que 15")
```

Es muy importante tener en cuenta que a diferencia de en otros lenguajes, en Python no puede haber un bloque `if` vacío. El siguiente código daría un `SyntaxError`.

```
if a > 5:
```

Por lo tanto si tenemos un `if` sin contenido, tal vez porque sea una tarea pendiente que estamos dejando para implementar en un futuro, es necesario hacer uso de `pass` para evitar el error. Realmente `pass` no hace nada, simplemente es para tener contento al interprete de código.

```
if a > 5:
    pass
```

Algo no demasiado recomendable pero que es posible, es poner todo el bloque que va dentro del `if` en la misma línea, justo a continuación de los `:`. Si el bloque de código no es muy largo, puede ser útil para ahorrarse alguna línea de código.

```
if a > 5: print("Es > 5")
```

Si tu bloque de código tiene más de una línea, se pueden poner también en la misma línea separándolas con `;`.

```
if a > 5: print("Es > 5"); print("Dentro del if")
```

Uso de else y elif

Es posible que no solo queramos hacer algo si una determinada condición se cumple, sino que además queramos hacer algo de lo contrario. Es aquí donde entra la cláusula `else`. La parte del `if` se comporta de la manera que ya hemos explicado, con la diferencia que si esa condición no se cumple, se ejecutará el código presente dentro del `else`. Nótese que ambos bloque de código son excluyentes, se entra o en uno o en otro, pero nunca se ejecutarán los dos.

```
x = 5
if x == 5:
    print("Es 5")
else:
    print("No es 5")
```

Hasta ahora hemos visto como ejecutar un bloque de código si se cumple una instrucción, u otro si no se cumple, pero no es suficiente. En muchos casos, podemos tener varias condiciones diferentes y para cada una queremos un código distinto. Es aquí donde entra en juego el `elif`.

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
```

Con la cláusula `elif` podemos ejecutar tantos bloques de código distintos como queramos según la condición. Traducido al lenguaje natural, sería algo así como decir: si es igual a 5 haz esto, si es igual a 6 haz lo otro, si es igual a 7 haz lo otro.
Se puede usar también de manera conjunta todo, el `if` con el `elif` y un `else` al final. Es muy importante notar que `if` y `else` solamente puede haber uno, mientras que `elif` puede haber varios.

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
else:
    print("Es otro")
```

Si vienes de otros lenguajes de programación, sabrás que el `switch` es una forma alternativa de `elif`, sin embargo en Python esta cláusula no existe.
Operador ternario
El operador ternario o `ternary operator` es una herramienta muy potente que muchos lenguajes de programación tienen. En Python es un poco distinto a lo que sería en C, pero el concepto es el mismo. Se trata de una cláusula `if`, `else` que se define en una sola línea y puede ser usado por ejemplo, dentro de un `print()`.
Para saber más: El operador ternario fue propuesto en la PEP 308.

```
x = 5
print("Es 5" if x == 5 else "No es 5")
#Es 5
```

Existen tres partes en un operador ternario, que son exactamente iguales a los que había en un `if else`. Tenemos la condición a evaluar, el código que se ejecuta si se cumple, y el código que se ejecuta si no se cumple. En este caso, tenemos los tres en la misma línea.

```
# [código si se cumple] if [condición] else [código si no se cumple]
```

Es muy útil y permite ahorrarse algunas líneas de código, además de aumentar la rapidez a la que escribimos. Si por ejemplo tenemos una variable a la que queremos asignar un valor en función de una condición, se puede hacer de la siguiente manera. Siguiendo el ejemplo anterior, en el siguiente código intentamos dividir `a` entre `b`. Si `b` es diferente a cero, se realiza la división y se almacena en `c`, de lo contrario se almacena `-1`. Ese `-1` podría ser una forma de indicar que ha habido un error con la división.

```
a = 10
b = 5
c = a/b if b!=0 else -1
print(c)
#2
```

Ejemplos if

```
# Verifica si un número es par o impar
x = 6
if x % 2 == 0:
    print("Es par")
else:
    print("Es impar")
# Decrementa x en 1 unidad si es mayor que cero
x = 5
x -= 1 if x > 0 else x
print(x)
```

6 Sentencias Repetitivas

Bucle for

A continuación explicaremos el bucle `for` y sus particularidades en Python, que comparado con otros lenguajes de comparación, tiene ciertas diferencias.

El for es un tipo de bucle, parecido al while pero con ciertas diferencias. La principal es que el número de iteraciones de un for **esta definido** de antemano, mientras que en un while no. La diferencia principal con respecto al while es en la condición. Mientras que en el while la condición era evaluada en cada iteración para decidir si volver a ejecutar o no el código, en el `for` no existe tal condición, sino un `iterable` que define las veces que se ejecutará el código. En el siguiente ejemplo vemos un bucle `for` que se ejecuta `5` veces, y donde la `i` incrementa su valor "automáticamente" en 1 en cada iteración.

```
for i in range(0, 5):
    print(i)

# Salida:
# 0
# 1
# 2
# 3
# 4
```

En Python se puede iterar prácticamente todo, como por ejemplo una cadena. En el siguiente ejemplo vemos como la `i` va tomando los valores de cada letra. Mas adelante explicaremos que es esto de los **iterables** e **iteradores**.

```
for i in "Python":
    print(i)

# Salida:
# P
# y
# t
# h
# o
# n
```

Iterables e iteradores

Para entender al cien por cien los bucles for, y como Python fue diseñado como lenguaje de programación, es muy importante entender los conceptos de `iterables` e `iteradores`. Empecemos con un par de definiciones:

- Los **iterables** son aquellos objetos que como su nombre indica pueden ser iterados, lo que dicho de otra forma es, que puedan ser indexados. Si piensas en un array (o una `list` en Python), podemos indexarlo con `lista[1]` por ejemplo, por lo que sería un iterable.
- Los **iteradores** son objetos que hacen referencia a un elemento, y que tienen un método `next` que permite hacer referencia al siguiente.

Para saber más: Si quieres saber más sobre los iteradores te dejamos este enlace a la documentación oficial.

Ambos son conceptos un tanto abstractos y que pueden ser complicados de entender. Veamos unos ejemplos. Como hemos comentado, los **iterables** son objetos que pueden ser iterados o accedidos con un índice. Algunos ejemplos de iterables en Python son las listas, tuplas, cadenas o diccionarios. Sabiendo esto, lo primero que tenemos que tener claro es que en un `for`, lo que va después del `in` **deberá ser siempre un iterable**.

```
#for <variable> in <iterable>:
#    <Código>
```

Tiene bastante sentido, porque si queremos iterar una variable, esta variable debe ser **iterable**, todo muy lógico. Pero llegados a este punto, tal vez de preguntes ¿pero cómo se yo si algo es iterable o no?. Bien fácil, con la siguiente función `isinstance()` podemos saberlo. No te preocupes si no entiendes muy bien lo que estamos haciendo, fíjate solo en el resultado, `True` significa que es iterable y `False` que no lo es.

```
from collections import Iterable
lista = [1, 2, 3, 4]
cadena = "Python"
numero = 10
print(isinstance(lista, Iterable))  #True
print(isinstance(cadena, Iterable)) #True
print(isinstance(numero, Iterable)) #False
```

Por lo tanto las listas y las cadenas son iterables, pero `numero`, que es un entero no lo es. Es por eso por lo que no podemos hacer lo siguiente, ya que daría un error. De hecho el error sería `TypeError: int' object is not iterable`.

```
numero = 10
#for i in numero:
#    print(i)
```

Una vez entendidos los **iterables**, veamos los **iteradores**. Para entender los iteradores, es importante conocer la función `iter()` en Python. Dicha función puede ser llamada sobre un objeto que sea iterable, y nos devolverá un iterador como se ve en el siguiente ejemplo.

```
lista = [5, 6, 3, 2]
it = iter(lista)
print(it)       #<list_iterator object at 0x106243828>
print(type(it)) #<class 'list_iterator'>
```

Vemos que al imprimir `it` es un iterador, de la clase `list_iterator`. Esta variable iteradora, hace referencia a la lista original y nos permite acceder a sus elementos con la función `next()`. Cada vez que llamamos a `next()` sobre `it`, nos devuelve el siguiente elemento de la lista original. Por lo tanto, si queremos acceder al elemento 4, tendremos que llamar 4 veces a `next()`. Nótese que el iterador empieza apuntando fuera de la lista, y no hace referencia al primer elemento hasta que no se llama a `next()` por primera vez.

```
lista = [5, 6, 3, 2]
it = iter(lista)
print(next(it))
#    [5, 6, 3, 2]
#     ^
#     |
#    it
print(next(it))
#    [5, 6, 3, 2]
#       ^
#       |
#      it
print(next(it))
#    [5, 6, 3, 2]
#         ^
#         |
#        it
```

Para saber mas: Existen otros iteradores para diferentes clases:

- `str_iterator` para cadenas
- `list_iterator` para sets.
- `tuple_iterator` para tuplas.
- `set_iterator` para sets.
- `dict_keyiterator` para diccionarios.

Dado que el iterador hace referencia a nuestra lista, si llamamos más veces a `next()` que la longitud de la lista, se nos devolverá un error `StopIteration`. Lamentablemente no existe ninguna opción de volver al elemento anterior.

```
lista = [5, 6]
it = iter(lista)
print(next(it))
print(next(it))
#print(next(it)) # Error! StopIteration
```

Es perfectamente posible tener diferentes iteradores para la misma lista, y serán totalmente independientes. Tan solo dependerán de la lista, como es evidente, pero no entre ellos.

```
lista = [5, 6, 7]
it1 = iter(lista)
it2 = iter(lista)
print(next(it1)) #5
print(next(it1)) #6
print(next(it1)) #7
print(next(it2)) #5
```

For anidados

Es posible **anidar** los `for`, es decir, **meter uno dentro de otro**. Esto puede ser muy útil si queremos iterar algún objeto que en cada elemento, tiene a su vez otra clase iterable. Podemos tener por ejemplo, una lista de listas, una especie de matriz.

```
lista = [[56, 34, 1],
         [12, 4, 5],
         [9, 4, 3]]
```

Si iteramos usando sólo un `for`, estaremos realmente accediendo a la segunda lista, pero no a los elementos individuales.

```
for i in lista:
    print(i)
#[56, 34, 1]
#[12, 4, 5]
```

```
#[9, 4, 3]
```

Si queremos acceder a cada elemento individualmente, podemos anidar dos `for`. Uno de ellos se encargará de iterar las columnas y el otro las filas.

```
for i in lista:
    for j in i:
        print(j)
# Salida: 56,34,1,12,4,5,9,4,3
```

Ejemplos for

Iterando cadena al revés. Haciendo uso de `[::-1]` se puede iterar la lista desde el último al primer elemento.

```
texto = "Python"
for i in texto[::-1]:
    print(i) #n,o,h,t,y,P
```

Itera la cadena saltándose elementos. Con `[::2]` vamos tomando un elemento si y otro no.

```
texto = "Python"
for i in texto[::2]:
    print(i) #P,t,o
```

Un ejemplo de `for` usado con comprehensions lists.

```
print(sum(i for i in range(10)))

# Salida: 45
```

Uso del range

Uno de las iteraciones mas comunes que se realizan, es la de iterar un número entre por ejemplo 0 y `n`. Si ya programas, estoy seguro de que estas cansado de escribir esto, aunque sea en otro lenguaje. Pongamos que queremos iterar una variable `i` de 0 a 5. Haciendo uso de lo que hemos visto anteriormente, podríamos hacer lo siguiente.

```
for i in (0, 1, 2, 3, 4, 5):
    print(i) #0, 1, 2, 3, 4, 5
```

Se trata de una solución que cumple con nuestro requisito. El contenido después del `in` se trata de una clase que como ya hemos visto antes, es iterable, y es de hecho una tupla. Sin embargo, hay otras formas de hacer esto en Python, haciendo uso del `range()`.

```
for i in range(6):
    print(i) #0, 1, 2, 3, 4, 5
```

El `range()` genera una secuencia de números que van desde 0 por defecto hasta el número que se pasa como parámetro menos 1. En realidad, se pueden pasar hasta tres parámetros separados por coma, donde el primer es el inicio de la secuencia, el segundo el final y el tercero el salto que se desea entre números. Por defecto se empieza en 0 y el salto es de 1.

```
#range(inicio, fin, salto)
```

Por lo tanto, si llamamos a `range()` con `(5,20,2)`, se generarán números de 5 a 20 de dos en dos. Un truco es que el `range()` se puede convertir en `list`.

```
print(list(range(5, 20, 2)))
```

Y mezclándolo con el `for`, podemos hacer lo siguiente.

```
for i in range(5, 20, 2):
    print(i) #5,7,9,11,13,15,17,19
```

Se pueden generar también secuencias inversas, empezando por un número mayor y terminando en uno menor, pero para ello el salto deberá ser negativo.

```
for i in range (5, 0, -1):
    print(i) #5,4,3,2,1
```

While

El uso del `while` nos permite **ejecutar una sección de código repetidas veces**, de ahí su nombre. El código se ejecutará **mientras** una condición determinada se cumpla. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal. Llamaremos **iteración** a una ejecución completa del bloque de código.
Cabe destacar que existe dos tipos de bucles, los que tienen un número de iteraciones **no definidas**, y los que tienen un número de iteraciones **definidas**. El `while` estaría dentro del primer tipo. Mas adelante veremos los `for`, que se engloban en el segundo.

```
x = 5
while x > 0:
    x -=1
    print(x)

# Salida: 4,3,2,1,0
```

En el ejemplo anterior tenemos un caso sencillo de `while`. Tenemos una condición `x>0` y un bloque de código a ejecutar mientras dure esa condición `x-=1` y `print(x)`. Por lo tanto mientras que `x` sea mayor que 0, se ejecutará el código. Una vez se llega al final, se vuelve a empezar y si la condición se cumple, se ejecuta otra vez. En este caso se entra al bloque de código 5 veces, hasta que en la sexta, `x` vale cero y por lo tanto la condición ya no se cumple. Por lo tanto el `while` tiene dos partes:

- La **condición** que se tiene que cumplir para que se ejecute el código.
- El **bloque de código** que se ejecutará mientras la condición se cumpla.

Ten cuidado ya que un mal uso del `while` puede dar lugar a bucles infinitos y problemas. Cierto es que en algún caso tal vez nos interese tener un bucle infinito, pero salvo que estemos seguros de lo que estamos haciendo, hay que tener cuidado. Imaginemos que tenemos un bucle cuya condición siempre se cumple. Por ejemplo, si ponemos `True` en la condición del `while`, siempre que se evalúe esa expresión, el resultado será `True` y se ejecutará el bloque de código. Una vez llegado al final del bloque, se volverá a evaluar la condición, se cumplirá, y vuelta a empezar. No te recomiendo que ejecutes el siguiente código, pero puedes intentarlo.

```
# No ejecutes esto, en serio
while True:
    print("Bucle infinito")
```

Es posible tener un `while` en una sola línea, algo muy útil si el bloque que queremos ejecutar es corto. En el caso de tener mas de una sentencia, las debemos separar con `;`.

```
x = 5
while x > 0: x-=1; print(x)
```

También podemos usar otro tipo de operación dentro del `while`, como la que se muestra a continuación. En este caso tenemos una lista que mientras no este vacía, vamos eliminando su primer elemento.

```
x = ["Uno", "Dos", "Tres"]
while x:
    x.pop(0)
    print(x)
#['Dos', 'Tres']
#['Tres']
#[]
```

Else y while

Algo no muy corriente en otros lenguajes de programación pero si en Python, es el uso de la cláusula `else` al final del `while`. Podemos ver el ejemplo anterior mezclado con el `else`. La sección de código que se encuentra dentro del `else`, se ejecutará cuando el bucle termine, pero solo si lo hace “por razones naturales”. Es decir, si el bucle termina porque la condición se deja de cumplir, y no porque se ha hecho uso del `break`.

```
x = 5
while x > 0:
    x -=1
    print(x) #4,3,2,1,0
else:
    print("El bucle ha finalizado")
```

Podemos ver como si el bucle termina por el `break`, el `print()` no se ejecutará. Por lo tanto, se podría decir que si no hay realmente ninguna sentencia `break` dentro del bucle, tal vez no tenga mucho sentido el uso del `else`, ya que un bloque de código fuera del bucle cumplirá con la misma funcionalidad.

```
x = 5
while True:
    x -= 1
    print(x) #4, 3, 2, 1, 0
    if x == 0:
        break
else:
    # El print no se ejecuta
    print("Fin del bucle")
```

Bucles anidados

Ya hemos visto que los bucles `while` tienen una condición a evaluar y un bloque de código a ejecutar. Hemos visto ejemplos donde el bloque de código son operaciones sencillas como la resta `-`, pero podemos complicar un poco mas las cosas y meter otro bucle `while` dentro del primero. Es algo que resulta especialmente útil si por ejemplo queremos generar permutaciones de números, es decir, si queremos generar todas las combinaciones posibles. Imaginemos que queremos generar todas las combinaciones de de dos números hasta 2. Es decir, 0-0, 0-1, 0-2,… hasta 2-2.

```
# Permutación a generar
i = 0
j = 0
while i < 3:
    while j < 3:
        print(i,j)
        j += 1
    i += 1
    j = 0
```

Vamos a analizara el ejemplo paso por paso. El primer bucle genera números del 0 al 2, lo que corresponde a la variable `i`. Por otro lado el segundo bucle genera también número del 0 al 2, almacenados en la variable `j`. Al tener un bucle dentro de otro, lo que pasa es que por cada `i` se generan 3 `j`. Muy importante no olvidar que al finalizar el bucle de la `j`, debemos resetear `j=0` para que en la siguiente iteración la condición de entrada se cumpla.

Podemos complicar las cosas aún más y tener tres bucles anidados, generando combinaciones de 3 elementos con número 0, 1, 2. En este caso tendremos desde 0,0,0 hasta 2,2,2.

```
i, j, k = 0, 0, 0
while i < 3:
    while j < 3:
        while k < 3:
            print(i,j,k)
            k += 1
            j += 1
        k = 0
    i += 1
    j = 0
```

Ejemplos while

Árbol de navidad en Python. Imprime un árbol de navidad formado con `*` haciendo uso del `while` y de la multiplicación de un entero por una cadena, cuyo resultado en Python es replicar la cadena.

```
z = 7
x = 1
while z > 0:
    print(' ' * z + '*' * x + ' ' * z)
    x+=2
    z-=1
#      *
#     ***
#    *****
#   *******
#  *********
# ***********
```

Aunque esta no sea tal vez la mejor forma de iterar una cadena es un buen ejemplo para el uso del `while` e introducir el indexado de listas con `[]`, que veremos en otros capítulos.

```
text = "Python"
i = 0
while i < len(text):
    print(text[:i + 1])
    i += 1
```

```
# P
# Py
# Pyt
# Pyth
# Pytho
# Python
```

Sucesión de **Fibonacci** en Python. En matemáticas, la sucesión de *fibonacci* es una sucesión infinita de números naturales, donde el siguiente es calculado sumando los dos anteriores.

```
a, b = 0, 1
while b < 25:
    print(b)
    a, b = b, a + b
#1, 1, 2, 3, 5, 8, 13, 21
```

7 Funciones

Python tiene funciones nativas como `len()` para calcular la longitud de una lista, pero al igual que en otros lenguajes de programación, también podemos definir **nuestras propias funciones**. Para ello hacemos uso de `def`.

```
def nombre_funcion(argumentos):
    código
    return retorno
```

Cualquier función tendrá un **nombre**, unos **argumentos de entrada**, un **código** a ejecutar y unos **parámetros de salida**. Al igual que las funciones matemáticas, en programación nos permiten realizar diferentes operaciones con la entrada, para entregar una determinada salida que dependerá del código que escribamos dentro. Por lo tanto, es totalmente análogo al clásico `y=f(x)` de las matemáticas.

```
def f(x):
    return 2*x
y = f(3)
print(y) # 6
```

Algo que diferencia en cierto modo las funciones en el mundo de la programación, es que no sólo realizan una operación con sus entradas, sino que también parten de los siguientes principios:

- El principio de **reusabilidad**, que nos dice que si por ejemplo tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función. Esto nos evitaría tener código repetido, y que modificarlo fuera más fácil, ya que bastaría con cambiar la función una vez.
- Y el principio de **modularidad**, que defiende que en vez de escribir largos trozos de código, es mejor crear módulos o funciones que agrupen ciertos fragmentos de código en funcionalidades específicas, haciendo que el código resultante sea más fácil de leer.

Pasando argumentos de entrada

Empecemos por la función más sencilla de todas. Una función sin parámetros de entrada ni parámetros de salida.

```
def di_hola():
    print("Hola")
```

Hemos declarado o definido la función. El siguiente paso es llamarla con `di_hola()`. Si lo realizamos veremos que se imprime Hola.

```
di_hola() # Hola
```

Vamos a complicar un poco las cosas pasando un argumento de entrada. Ahora si pasamos como entrada un nombre, se imprimirá Hola y el nombre.

```
def di_hola(nombre):
    print("Hola", nombre)
di_hola("Juan")
```

```
# Hola Juan
```

Python permite pasar argumentos también de otras formas. A continuación las explicamos todas.

Argumentos por posición

Los argumentos por posición o **posicionales** son la forma más básica e intuitiva de pasar parámetros. Si tenemos una función `resta()` que acepta dos parámetros, se puede llamar como se muestra a continuación.

```
def resta(a, b):
    return a-b
resta(5, 3) # 2
```

Al tratarse de parámetros posicionales, se interpretará que el primer número es la `a` y el segundo la `b`. El número de parámetros es fijo, por lo que si intentamos llamar a la función con solo uno, dará error.

```
#resta(1) # Error! TypeError
```

Tampoco es posible usar mas argumentos de los tiene la función definidos, ya que no sabría que hacer con ellos. Por lo tanto si lo intentamos, Python nos dirá que toma 2 posicionales y estamos pasando 3, lo que no es posible.

```
#TypeError: resta() takes 2 positional arguments but 3 were given
#resta(5,4,3) # Error
```

Argumentos por nombre

Otra forma de llamar a una función, es usando el nombre del argumento con `=` y su valor. El siguiente código hace lo mismo que el código anterior, con la diferencia de que los argumentos no son posicionales.

```
resta(a=3, b=5) # -2
```

Al indicar en la llamada a la función el nombre de la variable y el valor, el orden ya no importa, y se podría llamar de la siguiente forma.

```
resta(b=5, a=3) # -2
```

Como es de esperar, si indicamos un argumento que no ha sido definido como parámetro de entrada, tendremos un error.

```
#resta() got an unexpected keyword argument 'c'
#resta(b=5, c=3) # Error!
```

Argumentos por defecto

Tal vez queramos tener una función con algún parámetro opcional, que pueda ser usado o no dependiendo de diferentes circunstancias. Para ello, lo que podemos hacer es asignar un valor **por defecto** a la función. En el siguiente caso `c` valdría cero salvo que se indique lo contrario.

```
def suma(a, b, c=0):
    return a+b+c
```

```
suma(5,5,3) # 13
```

Dado que el parámetro `c` tiene un valor por defecto, la función puede ser llamada sin ese valor.

```
suma(4,3) # 7
```

Podemos incluso asignar un valor por defecto a todos los parámetros, por lo que se podría llamar a la función sin ningún argumento de entrada.

```
def suma(a=3, b=5, c=0):
    return a+b+c
suma() # 8
```

Las siguientes llamadas a la función también son válidas

```
suma(1)    # 6
suma(4,5)  # 9
suma(5,3,2) # 10
```

O haciendo uso de lo que hemos visto antes y usando los nombres de los argumentos.

```
suma(a=5, b=3) #8
```

Argumentos de longitud variable

En el ejemplo con argumentos por defecto, hemos visto que la función puede ser llamada con diferente número de argumentos de entrada, pero esto no es realmente una función con argumentos de longitud variable, ya que existe un número máximo.

Imaginemos que queremos una función `suma()` como la de antes, pero en este caso necesitamos que sume todos los números de entrada que se le pasen, sin importar si son 3 o 100. Una primera forma de hacerlo sería con una lista.

```
def suma(numeros):
    total = 0
    for n in numeros:
        total += n
    return total
suma([1,3,5,4]) # 13
```

La forma es válida y cumple nuestro requisito, pero realmente no estamos trabajando con argumentos de longitud variable. En realidad tenemos un solo argumento que es una lista de números.

Por suerte, Python tiene una herramienta muy potente. Si declaramos un argumento con `*`, esto hará que el argumento que se pase sea empaquetado en una tupla de manera automática. No confundir `*` con los punteros en otros lenguajes de programación, no tiene nada que ver.

```
def suma(*numeros):
    print(type(numeros))
    # <class 'tuple'>
    total = 0
    for n in numeros:
        total += n
    return total
suma(1, 3, 5, 4) # 13
```

El resultado es igual que el anterior, y podemos ver como efectivamente `numeros` es de la clase `tuple`. También podemos hacer otras llamadas con diferente número de argumentos

```
suma(6) # 6
suma(6, 4, 10) # 20
suma(6, 4, 10, 20, 4, 6, 7) # 57
```

Usando doble `**` es posible también tener como parámetro de entrada una lista de elementos almacenados en forma de clave y valor. En este caso podemos iterar los valores haciendo uso de `items()`.

```
def suma(**kwargs):
    suma = 0;
    for key, value in kwargs.items():
        print(key, value)
        suma += value
    return suma

suma(a=5, b=20, c=23) # 48
```

De igual manera, podemos pasar un diccionario como parámetro de entrada.

```
def suma(**kwargs):
    suma = 0
    for key, value in kwargs.items():
        print(key, value)
        suma += value
    return suma
di = {'a': 10, 'b':20}
```

```
suma(**di) # 30
```

Sentencia return

El uso de la sentencia `return` permite realizar dos cosas:

- Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada.
- Devolver uno o varios parámetros, fruto de la ejecución de la función.

En lo relativo a lo primero, una vez se llama a `return` se para la ejecución de la función y se vuelve o retorna al punto donde fue llamada. Es por ello por lo que el código que va después del `return` no es ejecutado en el siguiente ejemplo.

```
def mi_funcion():
    print("Entra en mi_funcion")
    return
    print("No llega")
mi_funcion() # Entra en mi_funcion
```

Por ello, sólo llamamos a `return` una vez hemos acabado de hacer lo que teníamos que hacer en la función.

Por otro lado, se pueden **devolver parámetros**. Normalmente las funciones son llamadas para realizar unos cálculos en base a una entrada, por lo que es interesante poder devolver ese resultado a quien llamó a la función.

```
def di_hola():
    return "Hola"
di_hola()
# 'Hola'
```

También es posible devolver mas de una variable, separadas por `,`. En el siguiente ejemplo tenemos una función que calcula la suma y media de tres números, y devuelve su resultado.

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

Documentación

Ahora que ya tenemos nuestras propias funciones creadas, tal vez alguien se interese en ellas y podamos compartírselas. Las funciones pueden ser muy complejas, y leer código ajeno no es tarea fácil. Es por ello por lo que es importante **documentar** las funciones. Es decir, añadir comentarios para indicar como deben ser usadas.

```
def mi_funcion_suma(a, b):
    """
    Descripción de la función. Como debe ser usada,
    que parámetros acepta y que devuelve
    """
    return a+b
```

Para ello debemos usar la triple comilla `"""` al principio de la función. Se trata de una especie de comentario que podemos usar para indicar como la función debe ser usada. No se trata de código, es un simple comentario un tanto especial, conocido como `docstring`.

Ahora cualquier persona que tenga nuestra función, podrá llamar a la función `help()` y obtener la ayuda de como debe ser usada.

```
help(mi_funcion_suma)
```

Otra forma de acceder a la documentación es la siguiente.

```
print(mi_funcion_suma.__doc__)
```

Para saber más: Las descripciones de las funciones suelen ser un poco mas detalladas de lo que hemos mostrado. En la PEP257 se define en detalle como debería ser.

Anotaciones en funciones

Existe una funcionalidad relativamente reciente en Python llamada **function annotation** o anotaciones en funciones. Dicha funcionalidad nos permite añadir metadatos a las funciones, indicando los tipos esperados tanto de entrada como de salida.

```
def multiplica_por_3(numero: int) -> int:
    return numero*3


multiplica_por_3(6) # 18
```

Las anotaciones son muy útiles de cara a la documentación del código, pero no imponen ninguna norma sobre los tipos. Esto significa que se puede llamar a la función con un parámetro que no sea `int`, y no obtendremos ningún error.

```
multiplica_por_3("Cadena")
# 'CadenaCadenaCadena'
```

Paso por valor y referencia

En muchos lenguajes de programación existen los conceptos de paso por **valor** y por **referencia** que aplican a la hora de como trata una función a los parámetros que se le pasan como entrada. Su comportamiento es el siguiente:

- Si usamos un parámetro pasado por **valor**, se creará una copia local de la variable, lo que implica que cualquier modificación sobre la misma no tendrá efecto sobre la original.
- Con una variable pasada como **referencia**, se actuará directamente sobre la variable pasada, por lo que las modificaciones afectarán a la variable original.

En Python las cosas son un poco distintas, y el comportamiento estará definido por el tipo de variable con la que estamos tratando. Veamos un ejemplo de paso por **valor**.

```
x = 10
def funcion(entrada):
    entrada = 0
funcion(x)

print(x) # 10
```

Iniciamos la `x` a 10 y se la pasamos a `funcion()`. Dentro de la función hacemos que la variable valga 0. Dado que Python trata a los `int` como pasados por **valor**, dentro de la función se crea una copia local de `x`, por lo que la variable original no es modificada.

No pasa lo mismo si por ejemplo `x` es una lista como en el siguiente ejemplo. En este caso Python lo trata como si estuviese pasada por **referencia**, lo que hace que se modifique la variable original. La variable original `x` ha sido modificada.

```
x = [10, 20, 30]
def funcion(entrada):
    entrada.append(40)

funcion(x)
print(x) # [10, 20, 30, 40]
```

El ejemplo anterior nos podría llevar a pensar que si en vez de añadir un elemento a `x`, hacemos `x=[]`, estaríamos destruyendo la lista original. Sin embargo esto no es cierto.

```
x = [10, 20, 30]
def funcion(entrada):
    entrada = []

funcion(x)
print(x)
# [10, 20, 30]
```

Una forma muy útil de saber lo que pasa por debajo de Python, es haciendo uso de la función `id()`. Esta función nos devuelve un identificador único para cada objeto. Volviendo al primer ejemplo podemos ver como los objetos a los que "apuntan" `x` y `entrada` son distintos.

```
x = 10
print(id(x)) # 4349704528
def funcion(entrada):
    entrada = 0
    print(id(entrada)) # 4349704208

funcion(x)
```

Sin embargo si hacemos lo mismo cuando la variable de entrada es una lista, podemos ver que en este caso el objeto con el que se trabaja dentro de la función es el mismo que tenemos fuera.

```
x = [10, 20, 30]
print(id(x)) # 4422423560
def funcion(entrada):
    entrada.append(40)
    print(id(entrada)) # 4422423560

funcion(x)
```

Los tipos simples se pasan por valor y los tipos compuestos por referencia.

Funciones lambda

Las funciones `lambda` o anónimas son un tipo de funciones en Python que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño. Resulta complicado explicar las diferencias, y para que te hagas una idea de ello te dejamos con la siguiente cita sacada de la documentación oficial.

"Python lambdas are only a shorthand notation if you're too lazy to define a function."

Lo que significa algo así como, "las funciones lambda son simplemente una versión acortada, que puedes usar si te da pereza escribir una función"

Lo que sería una función que suma dos números como la siguiente.

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

Se podría expresar en forma de una función `lambda` de la siguiente manera.

```
lambda a, b : a + b
```

La primera diferencia es que una función `lambda` no tiene un nombre, y por lo tanto salvo que sea asignada a una variable, es totalmente inútil. Para ello debemos.

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

Una vez tenemos la función, es posible llamarla como si de una función normal se tratase.

```
suma(2, 4)
```

Si es una función que solo queremos usar una vez, tal vez no tenga sentido almacenarla en una variable. Es posible declarar la función y llamarla en la misma línea.

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

Ejemplos

Una función `lambda` puede ser la entrada a una función normal.

```
def mi_funcion(lambda_func):
    return lambda_func(2,4)

mi_funcion(lambda a, b: a + b)
```

Y una función normal también puede ser la entrada de una función `lambda`. Nótese que son ejemplo didácticos y sin demasiada utilidad práctica per se.

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

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

A pesar de que las funciones `lambda` tienen muchas limitaciones frente a las funciones normales, comparten gran cantidad de funcionalidades. Es posible tener argumentos con valor asignado por defecto.

```
(lambda a, b, c=3: a + b + c)(1, 2) # 6
```

También se pueden pasar los parámetros indicando su nombre.

```
(lambda a, b, c: a + b + c)(a=1, b=2, c=3) # 6
```

Al igual que en las funciones se puede tener un número variable de argumentos haciendo uso de *, lo conocido como **tuple unpacking**.

```
(lambda *args: sum(args))(1, 2, 3) # 6
```

Y si tenemos los parámetros de entrada almacenados en forma de `key` y `value` como si fuera un diccionario, también es posible llamar a la función.

```
(lambda **kwargs: sum(kwargs.values()))(a=1, b=2, c=3) # 6
```

Por último, es posible devolver más de un valor.

```
x = lambda a, b: (b, a)

print(x(3, 9))

# Salida (9,3)
```

La **recursividad** o recursión es un concepto que proviene de las matemáticas, y que aplicado al mundo de la programación nos permite resolver problemas o tareas donde las mismas pueden ser divididas en subtareas cuya funcionalidad es la misma. Dado que los subproblemas a resolver son de la misma naturaleza, se puede usar la misma función para resolverlos. Dicho de otra manera, una función recursiva es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma hasta llegar a un punto de salida.
Cualquier función recursiva tiene dos secciones de código claramente divididas:

- Por un lado tenemos la sección en la que la función se llama a sí misma.
- Por otro lado, tiene que existir siempre una condición en la que la función retorna sin volver a llamarse. Es muy importante porque de lo contrario, la función se llamaría de manera indefinida.

Veamos unos ejemplos con el **factorial** y la **serie de fibonacci**.

Calcular factorial

Uno de los ejemplos mas usados para entender la recursividad, es el cálculo del factorial de un número `n!`. El factorial de un número `n` se define como la multiplicación de todos sus números predecesores hasta llegar a uno. Por lo tanto `5!`, leído como cinco factorial, sería `5*4*3*2*1`.
Utilizando un enfoque tradicional no recursivo, podríamos calcular el factorial de la siguiente manera.

```
def factorial_normal(n):
    r = 1
    i = 2
    while i <= n:
        r *= i
        i += 1
    return r

factorial_normal(5) # 120
```

Dado que el factorial es una tarea que puede ser dividida en subtareas del mismo tipo (multiplicaciones), podemos darle un enfoque recursivo y escribir la función de otra manera.

```
def factorial_recursivo(n):
    if n == 1:
        return 1
    else:
        return n * factorial_recursivo(n-1)

factorial_recursivo(5) # 120
```

Lo que realmente hacemos con el código anterior es llamar a la función `factorial_recursivo()` múltiples veces. Es decir, `5!` es igual a `5 * 4!` y `4!` es igual a `4 * 3!` y así sucesivamente hasta llegar a 1.

Algo muy importante a tener en cuenta es que si realizamos demasiadas llamadas a la función, podríamos llegar a tener un error del tipo `RecursionError`. Esto se debe a que todas las llamadas van apilándose y creando un contexto de ejecución, algo que podría llegar a causar un `stack overflow`. Es por eso por lo que Python lanza ese error, para protegernos de llegar a ese punto.

Serie de Fibonacci

Otro ejemplo muy usado para ilustrar las posibilidades de la recursividad, es calcular la serie de fibonacci. Dicha serie calcula el elemento `n` sumando los dos anteriores `n-1` + `n-2`. Se asume que los dos primeros elementos son 0 y 1.

```
def fibonacci_recursivo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursivo(n-1) + fibonacci_recursivo(n-2)
```

Podemos ver que siempre que la `n` sea distinta de cero o uno, se llamará recursivamente a la función, buscando los dos elementos anteriores. Cuando la `n` valga cero o uno se dejará de llamar a la función y se devolverá un valor concreto. Podemos calcular el elemento 7, que será 0,1,1,2,3,5,8,13, es decir, 13.

```
fibonacci_recursivo(7)
# 13
```

8 Excepciones

Las excepciones en Python son una herramienta muy potente que la gran mayoría de lenguajes de programación modernos tienen. Se trata de una **forma de controlar el comportamiento de un programa cuando se produce un error**.
Esto es muy importante ya que salvo que tratemos este error, **el programa se parará**, y esto es algo que en determinadas aplicaciones no es una opción válida.
Imaginemos que tenemos el siguiente código con dos variables `a` y `b` y realizamos su división `a/b`.

```
a = 4
b = 2
c = a/b
```

Pero imaginemos ahora que por cualquier motivo las variables tienen otro valor, y que por ejemplo `b` tiene el valor `0`. Si intentamos hacer la división entre cero, **este programa dará un error** y su ejecución terminará de manera abrupta.

```
a = 4; b = 0
print(a/b)
# ZeroDivisionError: division by zero
```

Ese "error" que decimos que ha ocurrido es lanzado por Python (*raise* en Inglés) ya que la división entre cero es una operación que matemáticamente no está definida.
Se trata de la excepción `ZeroDivisionError`. En el siguiente enlace, tenemos un listado de todas las excepciones con las que nos podemos encontrar.
Veamos un ejemplo con otra excepción. ¿Que pasaría si intentásemos sumar un número con un texto? Evidentemente esto no tiene ningún sentido, y Python define una excepción para esto llamada `TypeError`.

```
print(2 + "2")
```

En base a esto es **muy importante controlar las excepciones**, porque por muchas comprobaciones que realicemos es posible que en algún momento ocurra una, y si no se hace nada **el programa se parará**.
¿Te imaginas que en un avión, un tren o un cajero automático tiene un error que lanza `raise` una excepción y se detiene por completo?
Una primera aproximación al control de excepciones podría ser el siguiente ejemplo. Podemos realizar una comprobación manual de que no estamos dividiendo por cero, para así evitar tener un error tipo `ZeroDivisionError`.
Sin embargo es complicado escribir código que contemple y que prevenga todo tipo de excepciones. Para ello, veremos más adelante el uso de `except`.

```
a = 5
b = 0
# A través de esta comprobación prevenimos que se divida entre cero.
if b!=0:
    print(a/b)
else:
    print("No se puede dividir!")
```

Uso de raise

También podemos ser nosotros los que levantemos o lancemos una excepción. Volviendo a los ejemplos usados en el apartado anterior, **podemos ser nosotros los que levantemos** `ZeroDivisionError` o `NameError` usando `raise`. La sintaxis es muy fácil.

```
raise ZeroDivisionError("Información de la excepción")
```

O podemos lanzar otra de tipo `NameError`.

```
raise NameError("Información de la excepción")
```

Se puede ver como el `string` que hemos pasado se imprime a continuación de la excepción. Se puede llamar también sin ningún parámetro como se hace a continuación.

```
raise ZeroDivisionError
```

Visto esto, ya sabemos como una excepción puede ser lanzada. Existen dos maneras principalmente:

- Hacemos una operación que no puede ser realizada (como dividir por cero). En este caso Python se encarga de lanzar automáticamente la excepción.
- O también podemos lanzar nosotros una excepción manualmente, usando `raise`.
- Habría un tercer caso que sería lanzar una excepción que no pertenece a las definidas por defecto en Python. Pero eso te lo explicamos aquí.

A continuación veremos que podemos hacer para controlar estas excepciones, y que hacer cuando se lanza una para que no se interrumpa la ejecución del programa.

Uso de try y except

La buena noticia es que las excepciones que hemos visto antes, **pueden ser capturadas** y manejadas adecuadamente, **sin que el programa se detenga**. Veamos un ejemplo con la división entre cero.

```
a = 5; b = 0
try:
    c = a/b
except ZeroDivisionError:
    print("No se ha podido realizar la división")
```

En este caso no verificamos que `b!=0`. Directamente intentamos realizar la división y en el caso de que se lance la excepción `ZeroDivisionError`, la capturamos y la tratamos adecuadamente.

La diferencia con el ejemplo anterior es que ahora no se para el programa y se puede seguir ejecutando. Prueba a ejecutar el código y ver que pasa. Verás como el programa ya no se para.

Entonces, lo que hay dentro del `try` es la sección del código que podría lanzar la excepción que se está capturando en el `except`. Por lo tanto cuando ocurra una excepción, se entra en el `except` pero **el programa no se para**.

También se puede capturar diferentes excepciones como se ve en el siguiente ejemplo.

```
try:
    #c = 5/0      # Si comentas esto entra en TypeError
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError
except ZeroDivisionError:
    print("No se puede dividir entre cero!")
except TypeError:
    print("Problema de tipos!")
```

Puedes también hacer que un determinado número de excepciones se traten de la misma manera con el mismo bloque de código. Sin embargo suele ser más interesante tratar a diferentes excepciones de diference manera.

```
try:
    #c = 5/0      # Si comentas esto entra en TypeError
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError
except (ZeroDivisionError, TypeError):
    print("Excepcion ZeroDivisionError/TypeError")
```

Otra forma si no sabes que excepción puede saltar, puedes usar la clase genérica `Exception`. En este caso se controla cualquier tipo de excepción. De hecho todas las excepciones heredan de `Exception`. Ver documentación.

```
try:
    #c = 5/0      # Si comentas esto entra en TypeError
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError
except Exception:
    print("Ha habido una excepción")
```

No obstante hay una forma de saber que excepción ha sido la que ha ocurrido.

```
try:
    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError
except Exception as ex:
    print("Ha habido una excepción", type(ex))


# Ha habido una excepción <class 'TypeError'>
```

Uso de else

Al ya explicado `try` y `except` le podemos añadir un bloque más, el `else`. Dicho bloque **se ejecutará si no ha ocurrido ninguna excepción**. Fíjate en la diferencia entre los siguientes códigos.

```
try:
    # Forzamos una excepción al dividir entre 0
    x = 2/0
except:
    print("Entra en except, ha ocurrido una excepción")
else:
    print("Entra en else, no ha ocurrido ninguna excepción")

#Entra en except, ha ocurrido una excepción
```

Sin embargo en el siguiente código la división se puede realizar sin problema, por lo que el bloque `except` no se ejecuta pero el `else` si es ejecutado.

```
try:
    # La división puede realizarse sin problema
    x = 2/2
except:
    print("Entra en except, ha ocurrido una excepción")
else:
    print("Entra en else, no ha ocurrido ninguna excepción")

#Entra en else, no ha ocurrido ninguna excepción
```

Uso de finally

A los ya vistos bloques `try`, `except` y `else` podemos añadir un bloque más, el `finally`. Dicho bloque **se ejecuta siempre**, haya o no haya habido excepción.

Este bloque se suele usar si queremos ejecutar algún tipo de **acción de limpieza**. Si por ejemplo estamos escribiendo datos en un fichero pero ocurre una excepción, tal vez queramos borrar el contenido que hemos escrito con anterioridad, para no dejar datos inconsistenes en el fichero.

En el siguiente código vemos un ejemplo. Haya o no haya excepción el código que haya dentro de `finally` será ejecutado.

```
try:
    # Forzamos excepción
    x = 2/0
except:
    # Se entra ya que ha habido una excepción
    print("Entra en except, ha ocurrido una excepción")
finally:
```

```
    # También entra porque finally es ejecutado siempre
    print("Entra en finally, se ejecuta el bloque finally")


#Entra en except, ha ocurrido una excepción
#Entra en finally, se ejecuta el bloque finally
```

Ejemplos

Un ejemplo muy típico de excepciones es en el **manejo de ficheros**. Se intenta abrir, pero se captura una posible excepción. De hecho si entras en la documentación de open se indica que `OSError` es lanzada si el fichero no puede ser abierto.

```
# Se intenta abrir un fichero y se captura una posible excepción
try:
    with open('fichero.txt') as file:
        read_data = file.read()
except:
    # Se entra aquí si no pudo ser abierto
    print('No se pudo abrir')
```

Como ya hemos comentado, en el `except` también se puede capturar una excepción concreta. Dependiendo de nuestro programa, tal vez queramos tratar de manera distinta diferentes tipos de excepciones, por lo que es una buena práctica especificar que tipo de excepción estamos tratando.

```
# Se intenta abrir un fichero y se captura una posible excepción
try:
    with open('fichero.txt') as file:
        read_data = file.read()
# Capturamos una excepción concreta
except OSError:
    print('OSError. No se pudo abrir')
```

En este otro ejemplo vemos el uso de los bloques `try`, `except`, `else` y `finally` todos juntos.

```
try:
    # Se fuerza excepción
    x = 2/0
except:
    print("Entra en except, ha ocurrido una excepción")
else:
    print("Entra en el else, no ha ocurrido ninguna excepción")
finally:
    print("Entra en finally, se ejecuta el bloque finally")
```

También se puede capturar una excepción de tipo `SyntaxError`, que hace referencia a errores de sintaxis. Sin embargo el código debería estar libre de este tipo de fallos, por lo que tal vez nunca deberías usar esto.

```
try:
    print("Hola"))
```

```
except SyntaxError:
    print("Hay un error de sintaxis")
```

9 POO

Se trata de un paradigma de programación introducido en los años 1970s, pero que no se hizo popular hasta años más tarde.
Este modo o paradigma de programación nos permite organizar el código de una manera que se asemeja bastante a como pensamos en la vida real, utilizando las famosas **clases**. Estas nos permiten agrupar un conjunto de variables y funciones que veremos a continuación.
Cosas de lo más cotidianas como un perro o un coche pueden ser representadas con clases. Estas clases tienen diferentes características, que en el caso del perro podrían ser la edad, el nombre o la raza. Llamaremos a estas características, **atributos**.
Por otro lado, las clases tienen un conjunto de funcionalidades o cosas que pueden hacer. En el caso del perro podría ser andar o ladrar. Llamaremos a estas funcionalidades **métodos**.
Por último, pueden existir diferentes tipos de perro. Podemos tener uno que se llama Toby o el del vecino que se llama Laika. Llamaremos a estos diferentes tipos de perro **objetos**. Es decir, el concepto abstracto de perro es la **clase**, pero Toby o cualquier otro perro particular será el **objeto**.

Definiendo clases
Vista ya la parte teórica, vamos a ver como podemos hacer uso de la programación orientada a objetos en Python. Lo primero es crear una clase, para ello usaremos el ejemplo de perro.

```
# Creando una clase vacía
class Perro:
    pass
```

Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. Nótese el uso del `pass` que no hace realmente nada, pero daría un error si después de los `:` no tenemos contenido.
Ahora que tenemos la **clase**, podemos crear un **objeto** de la misma. Podemos hacerlo como si de una variable normal se tratase. Nombre de la variable igual a la clase con `()`. Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
# Creamos un objeto de la clase perro
mi_perro = Perro()
```

Definiendo atributos

A continuación vamos a añadir algunos atributos a nuestra clase. Antes de nada es importante distinguir que existen dos tipos de atributos:

- Atributos de **instancia**: Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia, en nuestro caso de cada perro.
- Atributos de **clase**: Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos.

Empecemos creando un par de **atributos de instancia** para nuestro perro, el `nombre` y la `raza`. Para ello creamos un método `__init__` que será llamado automáticamente cuando creemos un objeto. Se trata del **constructor**.

```
class Perro:
    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza
```

Ahora que hemos definido el método *init* con dos parámetros de entrada, podemos crear el objeto pasando el valor de los atributos. Usando `type()` podemos ver como efectivamente el objeto es de la clase `Perro`.

```
mi_perro = Perro("Toby", "Bulldog")
print(type(mi_perro))
# Creando perro Toby, Bulldog
# <class '__main__.Perro'>
```

Seguramente te hayas fijado en el `self` que se pasa como parámetro de entrada del método. Es una variable que representa la instancia de la clase, y deberá estar siempre ahí.

El uso de `__init__` y el doble `__` no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje. En este caso sería lo que se conoce como **constructor**. Hay gente que llama a estos métodos mágicos.

Por último, podemos acceder a los atributos usando el objeto y `.`. Por lo tanto.

```
print(mi_perro.nombre) # Toby
print(mi_perro.raza)   # Bulldog
```

Hasta ahora hemos definido atributos de instancia, ya que son atributos que pertenecen a cada perro concreto. Ahora vamos a definir un **atributo de clase**, que será común para todos los perros. Por ejemplo, la especie de los perros es algo común para todos los objetos Perro.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza
```

Dado que es un atributo de clase, no es necesario crear un objeto para acceder al atributos. Podemos hacer lo siguiente.

```
print(Perro.especie)
# mamífero
```

Se puede acceder también al atributo de clase desde el objeto.

```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.especie
# 'mamífero'
```

De esta manera, todos los objetos que se creen de la clase perro compartirán ese atributo de clase, ya que pertenecen a la misma.

Definiendo métodos

En realidad cuando usamos `__init__` anteriormente ya estábamos definiendo un método, solo que uno especial. A continuación vamos a ver como definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro.

Vamos a codificar dos métodos, ladrar y caminar. El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente `self` hace referencia a la instancia de la clase. Se puede definir un método con `def` y el nombre, y entre `()` los parámetros de entrada que recibe, donde siempre tendrá que estar `self` el primero.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'
```

```
    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

Por lo tanto si creamos un objeto `mi_perro`, podremos hacer uso de sus métodos llamándolos con `.` y el nombre del método. Como si de una función se tratase, pueden recibir y devolver argumentos.

```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)

# Creando perro Toby, Bulldog
# Guau
# Caminando 10 pasos
```

Herencia en Python

Para entender la herencia, es fundamental entender la programación orientada a objetos, por lo que te recomendamos empezar por ahí antes.

La **herencia** es un proceso mediante el cual se puede crear una clase **hija** que hereda de una clase **padre**, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobreescribir los métodos o atributos, o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos como se puede usar la herencia en Python, con la clase `Perro` que hereda de `Animal`. Así de fácil.

```
# Definimos una clase padre
class Animal:
    pass

# Creamos una clase hija que hereda de la padre
class Perro(Animal):
    pass
```

De hecho podemos ver como efectivamente la clase `Perro` es la hija de `Animal` usando `__bases__`

```
print(Perro.__bases__)
# (<class '__main__.Animal'>,)
```

De manera similar podemos ver que clases descienden de una en concreto con `__subclasses__`.

```
print(Animal.__subclasses__())
# [<class '__main__.Perro'>]
```

¿Y para que queremos la herencia? Dado que una clase hija hereda los atributos y métodos de la padre, nos puede ser muy útil cuando tengamos clases que se parecen entre sí pero tienen ciertas particularidades. En este caso en vez de definir un montón de clases para cada animal, podemos tomar los elementos comunes y crear una clase `Animal` de la que hereden el resto, respetando por tanto la filosofía **DRY**. Realizar estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás, es una tarea de lo más compleja en el mundo de la programación.

Para saber más: El principio DRY (Don't Repeat Yourself) es muy aplicado en el mundo de la programación y consiste en no repetir código de manera innecesaria. Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias. Las clases y la herencia a no repetir código.

Extendiendo y modificando métodos

Continuemos con nuestro ejemplo de perros y animales. Vamos a definir una clase padre `Animal` que tendrá todos los atributos y métodos genéricos que los animales pueden tener. Esta tarea de buscar el denominador común es muy importante en programación. Veamos los atributos:

•Tenemos la **especie** ya que todos los animales pertenecen a una.
•Y la **edad**, ya que todo ser vivo nace, crece, se reproduce y muere.

Y los métodos o funcionalidades:

•Tendremos el método **hablar**, que cada animal implementará de una forma. Los perros ladran, las abejas zumban y los caballos relinchan.
•Un método **moverse**. Unos animales lo harán caminando, otros volando.
•Y por último un método **descríbeme** que será común.

Definimos la clase padre, con una serie de atributos comunes para todos los animales como hemos indicado.

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad

    # Método genérico pero con implementación particular
    def hablar(self):
        # Método vacío
        pass

    # Método genérico pero con implementación particular
    def moverse(self):
        # Método vacío
        pass

    # Método genérico con la misma implementación
    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

Tenemos ya por lo tanto una clase genérica `Animal`, que generaliza las características y funcionalidades que todo animal puede tener. Ahora creamos una clase `Perro` que hereda del `Animal`. Como primer ejemplo vamos a crear una clase vacía, para ver como los métodos y atributos son heredados por defecto.

```
# Perro hereda de Animal
class Perro(Animal):
    pass

mi_perro = Perro('mamífero', 10)
mi_perro.describeme()
# Soy un Animal del tipo Perro
```

Con tan solo un par de líneas de código, hemos creado una clase nueva que tiene todo el contenido que la clase padre tiene, pero aquí viene lo que es de verdad interesante. Vamos a crear varios animales concretos y sobreescrbir algunos de los métodos que habían sido definidos en la clase `Animal`, como el `hablar` o el `moverse`, ya que cada animal se comporta de una manera distinta. Podemos incluso crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la `Abeja` con `picar()`.

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando con 4 patas")

class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando con 4 patas")

class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")

    # Nuevo método
    def picar(self):
```

```
        print("Picar!")
```

Por lo tanto ya podemos crear nuestros objetos de esos animales y hacer uso de sus métodos que podrían clasificarse en tres:

- Heredados directamente de la clase padre: `describeme()`
- Heredados de la clase padre pero modificados: `hablar()` y `moverse()`
- Creados en la clase hija por lo tanto no existentes en la clase padre: `picar()`

```
mi_perro = Perro('mamífero', 10)
mi_vaca = Vaca('mamífero', 23)
mi_abeja = Abeja('insecto', 1)

mi_perro.hablar()
mi_vaca.hablar()
# Guau!
# Muuu!

mi_vaca.describeme()
mi_abeja.describeme()
# Soy un Animal del tipo Vaca
# Soy un Animal del tipo Abeja

mi_abeja.picar()
# Picar!
```

Uso de super()

En pocas palabras, la función `super()` nos permite acceder a los métodos de la clase padre desde una de sus hijas. Volvamos al ejemplo de `Animal` y `Perro`.

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    def hablar(self):
        pass

    def moverse(self):
```

```
        pass

    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

Tal vez queramos que nuestro `Perro` tenga un parámetro extra en el constructor, como podría ser el `dueño`. Para realizar esto tenemos dos alternativas:

- Podemos crear un nuevo `__init__` y guardar todas las variables una a una.
- O podemos usar `super()` para llamar al `__init__` de la clase padre que ya aceptaba la `especie` y `edad`, y sólo asignar la variable nueva manualmente.

```
class Perro(Animal):
    def __init__(self, especie, edad, dueño):
        # Alternativa 1
        # self.especie = especie
        # self.edad = edad
        # self.dueño = dueño

        # Alternativa 2
        super().__init__(especie, edad)
        self.dueño = dueño
mi_perro = Perro('mamífero', 7, 'Luis')
mi_perro.especie
mi_perro.edad
mi_perro.dueño
```

Herencia múltiple

En Python es posible realizar **herencia múltiple**. En otros posts hemos visto como se podía crear una clase padre que heredaba de una clase hija, pudiendo hacer uso de sus métodos y atributos. La herencia múltiple es similar, pero una clase **hereda de varias clases** padre en vez de una sola. Veamos un ejemplo. Por un lado tenemos dos clases `Clase1` y `Clase2`, y por otro tenemos la `Clase3` que hereda de las dos anteriores. Por lo tanto, heredará todos los métodos y atributos de ambas.

```
class Clase1:
```

```
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass
```

Es posible también que una clase herede de otra clase y a su vez otra clase herede de la anterior.

```
class Clase1:
    pass
class Clase2(Clase1):
    pass
class Clase3(Clase2):
    pass
```

Llegados a este punto nos podemos plantear lo siguiente. Vale, como sabemos de otros posts las clases hijas heredan los métodos de las clases padre, pero también pueden reimplementarlos de manera distinta. Entonces, si llamo a un método que todas las clases tienen en común ¿a cuál se llama?. Pues bien, existe una forma de saberlo.
La forma de saber a que método se llama es consultar el **MRO** o *Method Order Resolution*. Esta función nos devuelve una tupla con el orden de búsqueda de los métodos. Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass

print(Clase3.__mro__)
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class
'__main__.Clase2'>, <class 'object'>)
```

Una curiosidad es que al final del todo vemos la clase `object`. Aunque pueda parecer raro, es correcto ya que en realidad todas las clases en Python heredan de una clase genérica `object`, aunque no lo especifiquemos explícitamente.
Y como último ejemplo,…el cielo es el límite. Podemos tener una clase heredando de otras tres. Fíjate en que el **MRO** depende del orden en el que las clases son pasadas: 1, 3, 2.

```
class Clase1:
```

```
    pass
class Clase2:
    pass
class Clase3:
    pass
class Clase4(Clase1, Clase3, Clase2):
    pass
print(Clase4.__mro__)
# (<class '__main__.Clase4'>, <class '__main__.Clase1'>, <class
'__main__.Clase3'>, <class '__main__.Clase2'>, <class 'object'>)
```

Junto con la herencia, la cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento son otros de los conceptos claves para entender la programación orientada a objetos.

10 Módulos

Un módulo o *module* en Python es un fichero `.py` que **alberga un conjunto de funciones, variables o clases** y que puede ser usado por otros módulos. Nos permiten reutilizar código y organizarlo mejor en *namespaces*. Por ejemplo, podemos definir un módulo `mimodulo.py` con dos funciones `suma()` y `resta()`.

```
# mimodulo.py
def suma(a, b):
    return a + b

def resta(a, b):
    return a - b
```

Una vez definido, dicho módulo puede ser usado o **importado** en otro fichero, como mostramos a continuación. Usando `import` podemos importar **todo el contenido**.

```
# otromodulo.py
import mimodulo

print(mimodulo.suma(4, 3))   # 7
print(mimodulo.resta(10, 9)) # 1
```

También podemos importar únicamente los componentes que nos interesen como mostramos a continuación.

```
from mimodulo import suma, resta

print(suma(4, 3))   # 7
print(resta(10, 9)) # 1
```

Por último, podemos importar todo el módulo haciendo uso de `*`, sin necesidad de usar `mimodulo.*`.

```
from mimodulo import *

print(suma(4, 3))   # 7
print(resta(10, 9)) # 1
```

Rutas y Uso de sys.path

Normalmente los módulos que importamos están en la misma carpeta, pero es posible acceder también a módulos ubicados en una subcarpeta. Imaginemos la siguiente estructura:

```
.
├── ejemplo.py
├── carpeta
│   └── modulo.py
```

Donde `modulo.py` contiene lo siguiente:

```
# modulo.py
def hola():
        print("Hola")
```

Desde nuestro `ejemplo.py`, podemos importar el módulo `modulo.py` de la siguiente manera:

```
from carpeta.modulo import *
print(hola())
# Hola
```

Es importante notar que Python busca los módulos en las rutas indicadas por el `sys.path`. Es decir, cuando se importa un módulo, lo intenta buscar en dichas carpetas. Puedes ver tu `sys.path` de la siguiente manera:

```
import sys
print(sys.path)
```

Como es obvio, verás que la carpeta de tu proyecta está incluida, pero ¿y si queremos importar un módulo en una ubicación distinta? Pues bien, podemos añadir al `sys.path` la ruta en la que queremos que Python busque.

```
import sys
sys.path.append(r'/ruta/de/tu/modulo')
```

Una vez realizado esto, los módulos contenidos en dicha carpeta podrán ser importados sin problema como hemos visto anteriormente.

Cambiando los Nombres con as

Por otro lado, es posible cambiar el nombre del módulo usando `as`. Imaginemos que tenemos un módulo `moduloconnombrelargo.py`.

```
# moduloconnombrelargo.py
hola = "hola"
```

En vez de usar el siguiente `import`, tal vez queramos asignar un nombre más corto al módulo.

```
import moduloconnombrelargo
print(moduloconnombrelargo.hola)
```

Podemos hacerlo de la siguiente manera con `as`:

```
import moduloconnombrelargo as m
print(m.hola)
```

Listando dir

La función `dir()` nos permite ver los nombres (variables, funciones, clases, etc) existentes en nuestro *namespace*. Si probamos en un módulo vacío, podemos ver como tenemos varios nombres rodeados de __. Se trata de nombres que Python crea por debajo.

```
print(dir())
# ['__annotations__', '__builtins__', '__cached__', '__doc__',
# '__file__', '__loader__', '__name__', '__package__', '__spec__']
```

Por ejemplo, `__file__` es creado automáticamente y alberga el nombre del fichero `.py`.

```
print(__file__)
#/tu/ruta/tufichero.py
```

Imaginemos ahora que tenemos alguna variable y función definida en nuestro script. Como era de esperar, `dir()` ahora nos muestra también los nuevos nombres que hemos creado, y que por supuesto pueden ser usados.

```
mi_variable = "Python"
def mi_funcion():
    pass

print(dir())

#['__annotations__', '__builtins__', '__cached__', '__doc__',
# '__file__', '__loader__', '__name__', '__package__', '__spec__',
# 'mi_funcion', 'mi_variable']
```

Por último, vamos a importar el contenido de un módulo externo. Podemos ver que en el *namespace* tenemos también los nombres `resta` y `suma`, que han sido tomados de `mimodulo`.

```
from mimodulo import *
print(dir())

# ['__annotations__', '__builtins__', '__cached__',
# '__doc__', '__file__', '__loader__', '__name__',
# '__package__', '__spec__', 'resta', 'suma']
```

El uso de `dir()` también acepta parámetros de entrada, por lo que podemos por ejemplo pasar nuestro `modulo` y nos dará más información sobre lo que contiene.

```
import mimodulo

print(dir(mimodulo))
# ['__builtins__', '__cached__', '__doc__',
# '__file__','__loader__', '__name__',
# '__package__', '__spec__', 'resta', 'suma']

print(mimodulo.__name__)
# mimodulo
```

```
print(mimodulo.__file__)
# /tu/ruta/mimodulo.py
```

Excepción ImportError

Importar un módulo puede lanzar una excepción, cuando se intenta importar un módulo que no ha sido encontrado. Se trata de `ModuleNotFoundError`.

```
import moduloquenoexiste
# ModuleNotFoundError: No module named 'moduloquenoexiste'
```

Dicha excepción puede ser capturada para evitar la interrupción del programa.

```
try:
    import moduloquenoexiste
except ModuleNotFoundError as e:
    print("Hubo un error:", e)
```

Módulos y Función Main

Un problema muy recurrente es cuando creamos un módulo con una función como en el siguiente ejemplo, y añadimos algunas sentencias a ejecutar.

```
# modulo.py

def suma(a, b):
    return a + b

c = suma(1, 2)
print("La suma es:", c)
```

Si en otro módulo importamos nuestro `modulo.py`, tal como está nuestro código el contenido se ejecutará, y esto puede no ser lo que queramos.

```
# otromodulo.py
import modulo

# Salida: La suma es: 3
```

Dependiendo de la situación, puede ser importante especificar que únicamente queremos que se ejecute el código si el módulo es el `__main__`. Con la siguiente modificación, si hacemos `import modulo` desde otro módulo, este fragmento ya no se ejecutará al ser el módulo main otro.

```
# modulo.py
def suma(a, b):
    return a + b

if (__name__ == '__main__'):
    c = suma(1, 2)
    print("La suma es:", c)
```

Recargando Módulos

Es importante notar que los módulos solamente son cargados una vez. Es decir, no importa el número de veces que llamemos a `import mimodulo`, que sólo se importará una vez.

Imaginemos que tenemos el siguiente módulo que imprime el siguiente contenido cuando es importado.

```
# mimodulo.py

print("Importando mimodulo")

def suma(a, b):
    return a + b

def resta(a, b):
    return a - b
```

A pesar de que llamamos tres veces al `import`, sólo vemos una única vez el contenido del `print`.

```
import mimodulo
import mimodulo
import mimodulo
# Importando mimodulo
```

Si queremos que el módulo sea recargado, tenemos que ser explícitos, haciendo uso de `reload`.

```
import mimodulo
import importlib
importlib.reload(mimodulo)
importlib.reload(mimodulo)
```

11 Gestión de ficheros

Leer archivos en Python

Al igual que en otros lenguajes de programación, en Python es posible **abrir ficheros y leer su contenido**. Los ficheros o archivos pueden ser de lo más variado, desde un simple texto a contenido binario. Para simplificar nos centraremos en **leer un fichero de texto**. Si quieres aprender como escribir en un fichero te lo explicamos en este otro post.

Imagínate entonces que tienes un fichero de texto con unos datos, como podría ser un `.txt` o un `.csv`, y quieres leer su contenido para realizar algún tipo de procesado sobre el mismo. Nuestro fichero podría ser el siguiente.

```
# contenido del fichero ejemplo.txt
Contenido de la primera línea
Contenido de la segunda línea
Contenido de la tercera línea
Contenido de la cuarta línea
```

Podemos abrir el fichero con la función `open()` pasando como argumento el nombre del fichero que queremos abrir.

```
fichero = open('ejemplo.txt')
```

Método `read()`

Con `open()` tendremos ya en `fichero` el contenido del documento listo para usar, y podemos imprimir su contenido con `read()`. El siguiente código imprime **todo el fichero**.

```
print(fichero.read())
#Contenido de la primera línea
#Contenido de la segunda línea
#Contenido de la tercera línea
#Contenido de la cuarta línea
```

Método `readline()`

Es posible también leer un **número de líneas determinado** y no todo el fichero de golpe. Para ello hacemos uso de la función `readline()`. Cada vez que se llama a la función, se lee una línea.

```
fichero = open('ejemplo.txt')
print(fichero.readline())
print(fichero.readline())
# Contenido de la primera línea
# Contenido de la segunda línea
```

Es **muy importante saber** que una vez hayas leído todas las línea del archivo, la función ya no devolverá nada, porque se habrá llegado al final. Si quieres que `readline()` funcione otra vez, podrías por ejemplo volver a leer el fichero con `open()`.

Otra forma de usar `readline()` es pasando como argumento un número. Este número leerá un **determinado número de caracteres**. El siguiente código lee todo el fichero carácter por carácter.

```
fichero = open('ejemplo.txt')
```

```
caracter = fichero.readline(1)
while caracter != "":
    #print(caracter)
    caracter = fichero.readline(1)
```

Método `readlines()`

Existe otro método llamado `readlines()`, que devuelve una lista donde **cada elemento es una línea del fichero**.

```
fichero = open('ejemplo.txt')
lineas = fichero.readlines()
print(lineas)
#['Contenido de la primera línea\n', 'Contenido de la segunda línea\n',
#'Contenido de la tercera línea\n', 'Contenido de la cuarta línea']
```

De manera muy sencilla podemos iterar las líneas e imprimirlas por pantalla.

```
fichero = open('ejemplo.txt')
lineas = fichero.readlines()
for linea in lineas:
    print(linea)
#Contenido de la primera línea
#Contenido de la segunda línea
#Contenido de la tercera línea
#Contenido de la cuarta línea
```

Argumentos de `open()`

Hasta ahora hemos visto la función `open()` con tan sólo un argumento de entrada, el nombre del fichero. Lo cierto es que existe un segundo argumento que es importante especificar. Se trata del **modo de apertura del fichero**. En la documentación oficial se explica en detalle.

- 'r': Por defecto, para leer el fichero.
- 'w': Para escribir en el fichero.
- 'x': Para la creación, fallando si ya existe.
- 'a': Para añadir contenido a un fichero existente.
- 'b': Para abrir en modo binario.

Por lo tanto lo estrictamete correcto si queremos leer el fichero sería hacer lo siguiente. Aunque el modo `r` sea por defecto, es una buena práctica indicarlo para darle a entender a otras personas que lean nuestro código que no queremos modificarlo, tan solo leerlo.

```
fichero = open('ejemplo.txt', 'r')
```

Cerrando el fichero

Otra cosa que debemos hacer cuando trabajamos con ficheros en Python, es **cerrarlos una vez que ya hemos acabado con ellos**. Aunque es verdad que el fichero normalmente acabará siendo cerrado automáticamente, es importante especificarlo para evitar tener comportamientos inesperados.
Por lo tanto si queremos cerrar un fichero sólo tenemos que usar la función `close()` sobre el mismo. Por lo tanto tenemos tres pasos:

- Abrir el fichero que queramos. En modo texto usaremos 'r'.
- Usar el fichero para recopilar o procesar los datos que necesitábamos.
- Cuando hayamos acabado, cerramos el fichero.

```
fichero = open('ejemplo.txt', 'r')
# Usar la variable fichero
# Cerrar el fichero
fichero.close()
```

Existen otras formas de hacerlo, como con el uso de **excepciones** que veremos en otros posts. Un ejemplo sería el siguiente. No pasa nada si aún no entiendes el uso del `try` y `finally`, por ahora quédate con que la sección `finally` **se ejecuta siempre** sin importar si hay un error o no. De esta manera el `close()` siempre será ejecutado.

```
fichero = open('ejemplo.txt')
try:
    # Usar el fichero
    pass
finally:
    # Esta sección es siempre ejecutada
    fichero.close()
```

Y por si no fuera poco, existe otra forma de cerrar el fichero automáticamente. Si hacemos uso se `with()`, el fichero **se cerrará automáticamente una vez se salga de ese bloque** de código.

```
with open('ejemplo.txt') as fichero:
    # Usar el fichero. Se cerrará automáticamente
    pass
```

Ejemplos
Como ya hemos visto `readline()` lee línea por línea el fichero. También hacemos uso de un bucle while para leer líneas mientras que no se haya llegado al final. Es por eso por lo que comparamos `linea != ''`, ya que se devuelve un string vació cuando se ha llegado al final.

```
with open('ejemplo.txt', 'r') as fichero:
    linea = fichero.readline()
    while linea != '':
        print(linea, end='')
        linea = fichero.readline()

#Contenido de la primera línea
#Contenido de la segunda línea
#Contenido de la tercera línea
```

```
#Contenido de la cuarta línea
```

Nos podemos ahorrar alguna línea de código si hacemos lo siguiente, ya que `readlines()` nos devuelve directamente una lista que podemos iterar con las líneas.

```
with open('ejemplo.txt', 'r') as fichero:
    for linea in fichero.readlines():
        print(linea, end='')
#Contenido de la primera línea
#Contenido de la segunda línea
#Contenido de la tercera línea
#Contenido de la cuarta línea
```

Pero puede ser simplificado aún más de la siguiente manera. Nótese que usamos el `end=''` para decirle a Python que no imprima el salto de línea `\n` al final del print.

```
with open('ejemplo.txt', 'r') as fichero:
    for linea in fichero:
        print(linea, end='')
#Contenido de la primera línea
#Contenido de la segunda línea
#Contenido de la tercera línea
#Contenido de la cuarta línea
```

Escribir archivos en Python

A continuación te explicamos **como escribir datos en un fichero usando Python**. Imagínate que tienes unos datos que te gustaría guardar en un fichero para su posterior análisis. Te explicamos como guardarlos en un fichero, por ejemplo, `.txt`. Si también quieres aprender como leer un fichero en Python te lo explicamos en este otro post.

Lo primero que debemos de hacer es crear un objeto para el fichero, con el nombre que queramos. Al igual que vimos en el post de leer ficheros, además del nombre se puede pasar un segundo parámetro que indica el modo en el que se tratará el fichero. Los más relevantes en este caso son los siguientes. Para más información consulta la documentación oficial.

- ‘w’: Borra el fichero si ya existiese y crea uno nuevo con el nombre indicado.
- ‘a’: Añadirá el contenido al final del fichero si ya existiese (*append* end Inglés)
- ‘x’: Si ya existe el fichero se devuelve un error.

Por lo tanto con la siguiente línea estamos creando un fichero con el nombre *datos_guardados.txt*.

```
# Abre un nuevo fichero
```

```
fichero = open("datos_guardados.txt", 'w')
```

Si por lo contrario queremos añadir el contenido al ya existente en un fichero de antes, podemos hacerlo en el modo *append* como hemos indicado.

```
# Abre un nuevo y añade el contenido al final
fichero = open("datos_guardados.txt", 'a')
```

Método `write()`

Ya hemos visto como crear el fichero. Veamos ahora como podemos añadir contenido. Empecemos escribiendo un texto.

```
fichero = open("datos_guardados.txt", 'w')
fichero.write("Contenido a escribir")
fichero.close()
```

Por lo tanto si ahora abrimos el fichero `datos_guardados.txt`, veremos como efectivamente contiene una línea con *Contenido a escribir*. ¿A que es fácil?

Es **muy importante** el uso de `close()` ya que si dejamos el fichero abierto, podríamos llegar a tener un comportamiento inesperado que queremos evitar. Por lo tanto, siempre que se abre un fichero **es necesario cerrarlo** cuando hayamos acabado.

Compliquemos un poco más las cosas. Ahora vamos a guardar una lista de elementos en el fichero, donde cada elemento de la lista se almacenará en una línea distinta.

```
# Abrimos el fichero
fichero = open("datos_guardados.txt", 'w')

# Tenemos unos datos que queremos guardar
lista = ["Manzana", "Pera", "Plátano"]

# Guardamos la lista en el fichero
for linea in lista:
    fichero.write(linea + "\n")

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

Si te fijas, estamos almacenando la línea mas `\n`. Es importante añadir el salto de línea porque por defecto no se añade, y si queremos que cada elemento de la lista se almacena en una línea distinta, será necesario su uso.

Método `writelines()`

También podemos usar el método `writelines()` y pasarle una lista. Dicho método se encargará de guardar todos los elementos de la lista en el fichero.

```
fichero = open("datos_guardados.txt", 'w')
lista = ["Manzana", "Pera", "Plátano"]

fichero.writelines(lista)
fichero.close()

# Se guarda
# ManzanaPeraPlátano
```

Tal vez te hayas dado cuenta de que en realidad lo que se guarda es *ManzanaPeraPlátano*, todo junto. Si queremos que cada elemento se almacene en una línea distinta, deberíamos añadir el salto de línea en cada elemento de la lista como se muestra a continuación.

```
fichero = open("datos_guardados.txt", 'w')
lista = ["Manzana\n", "Pera\n", "Plátano\n"]

fichero.writelines(lista)
fichero.close()

# Se guarda
# Manzana
# Pera
# Plátano
```

Uso del `with`

Podemos ahorrar una línea de código si hacemos uso de lo siguiente. En este caso nos podemos ahorrar la llamada al `close()` ya que se realiza automáticamente. El código anterior se podría reescribir de la siguiente manera.

```
lista = ["Manzana\n", "Pera\n", "Plátano\n"]
with open("datos_guardados.txt", 'w') as fichero:
    fichero.writelines(lista)
```

Ejemplos escribir ficheros en Python

El uso de 'x' hace que **si el fichero ya existe se devuelve un error**. En el siguiente código creamos un fichero e inmediatamente después intentamos crear un fichero con el mismo nombre con la opción 'x'. Por lo tanto se devolverá un error.

```
f = open("mi_fichero.txt", "w")
```

```
# f = open("mi_fichero.txt", "x")
# Error! Ya existe
```

En este otro ejemplo vamos a usar un fichero para establecer una comunicación entre dos funciones. A efectos prácticos puede no resultar muy útil, pero es un buen ejemplo para mostrar la lectura y escritura de ficheros.
Tenemos por lo tanto una función `escribe_fichero()` que recibe un mensaje y lo escribe en un fichero determinado. Y por otro lado tenemos una función `lee_fichero()` que devuelve el mensaje que está escrito en el fichero.
Date cuenta lo interesante del ejemplo, ya que podríamos tener estos dos códigos ejecutándose en diferentes maquinas o procesos, y **podrían comunicarse a través del fichero**. Por un lado se escribe y por el otro se lee.

```
# Escribe un mensaje en un fichero
def escribe_fichero(mensaje):
    with open('fichero_comunicacion.txt', 'w') as fichero:
        fichero.write(mensaje)

# Leer el mensaje del fichero
def lee_fichero():
    mensaje = ""
    with open('fichero_comunicacion.txt', 'r') as fichero:
        mensaje = fichero.read()
    # Borra el contenido del fichero para dejarlo vacío
    f = open('fichero_comunicacion.txt', 'w')
    f.close()
    return mensaje

escribe_fichero("Esto es un mensaje")
print(lee_fichero())
```