

APUNTES PYTHON

TEMA 1 - INTRODUCCIÓN A PYTHON

En este curso vamos a ver los conceptos básicos de programación en Python.

CARACTERÍSTICAS

Python es un lenguaje de programación bastante actual. Fue creado por Guido Van Rossum a comienzos de los 90. Una curiosidad sobre el origen del nombre de este lenguaje es que, al contrario de lo que algunos piensan, no viene del nombre de la serpiente pitón (python en inglés), sino que su creador se lo puso debido a su gran afición al grupo de los Monty Python.

Es un lenguaje que se considera fácil de aprender ya que su sintaxis se compone de una gramática sencilla y legible, muy clara y cercana al lenguaje humano, así es un lenguaje de alto nivel. La curva de aprendizaje de Python por tanto es bastante más suave que la de otros lenguajes de programación.

Algunas de sus principales características son:

- **Tipado fuerte y dinámico:** Esto significa que el lenguaje diferencia de una forma muy fuerte de qué tipo es una variable, de modo que, si se intenta usar una variable de un tipo cuando debería ser de otro, nos dará un error, por ejemplo no se podrán sumar números y texto. También es dinámico, es decir que el lenguaje establece el tipo de la variable en tiempo de ejecución. En Python, el tipo de la variable vendrá dado por el continente y no por el contenedor, así, si al declarar una variable le asigno, por ejemplo, una cadena de caracteres, en ese momento el continente (la cadena) habrá hecho que el continente (la variable) pase a ser de tipo String.
- **Es orientado a objetos:** Con lo cual dispondrá de muchas características de este tipo de programación como la sobrecarga de constructores, la herencia múltiple, encapsulación, interfaces o polimorfismo. Hay que reseñar que este lenguaje es **multiparadigma**, es decir, que aunque está principalmente diseñado para la programación orientada a objetos, también se podría usar para programación imperativa (sentencias de bucle) o programación funcional (módulos y funciones)
- **Es open source:** es decir, es de código abierto y libre distribución.
- **Es un lenguaje interpretado:** es decir, no hará falta compilar el proyecto antes.
- **Es multiplataforma:** puede usarse en muchos dispositivos y sistemas operativos, ya que se han creado intérpretes para Unix, Linux, Windows y MacOs.
- **Es versátil:** con él se pueden crear tanto aplicaciones de escritorio como aplicaciones web o de servidor.
- **Tiene una librería estándar muy amplia:** gracias a su popularidad existen una gran cantidad de librerías y funciones ya hechas, y tiene un gran soporte gracias a lo

comunidad. También tiene soporte e información para una gran cantidad de bases de datos.

SINTAXIS BÁSICA

El término **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 los diferentes apartados de esta sección se verán los Tipos de Datos y Operadores que Python maneja, así como la definición y uso de las variables.

Algunos aspectos básicos a tener en cuenta a la hora de programar con este lenguaje de programación son:

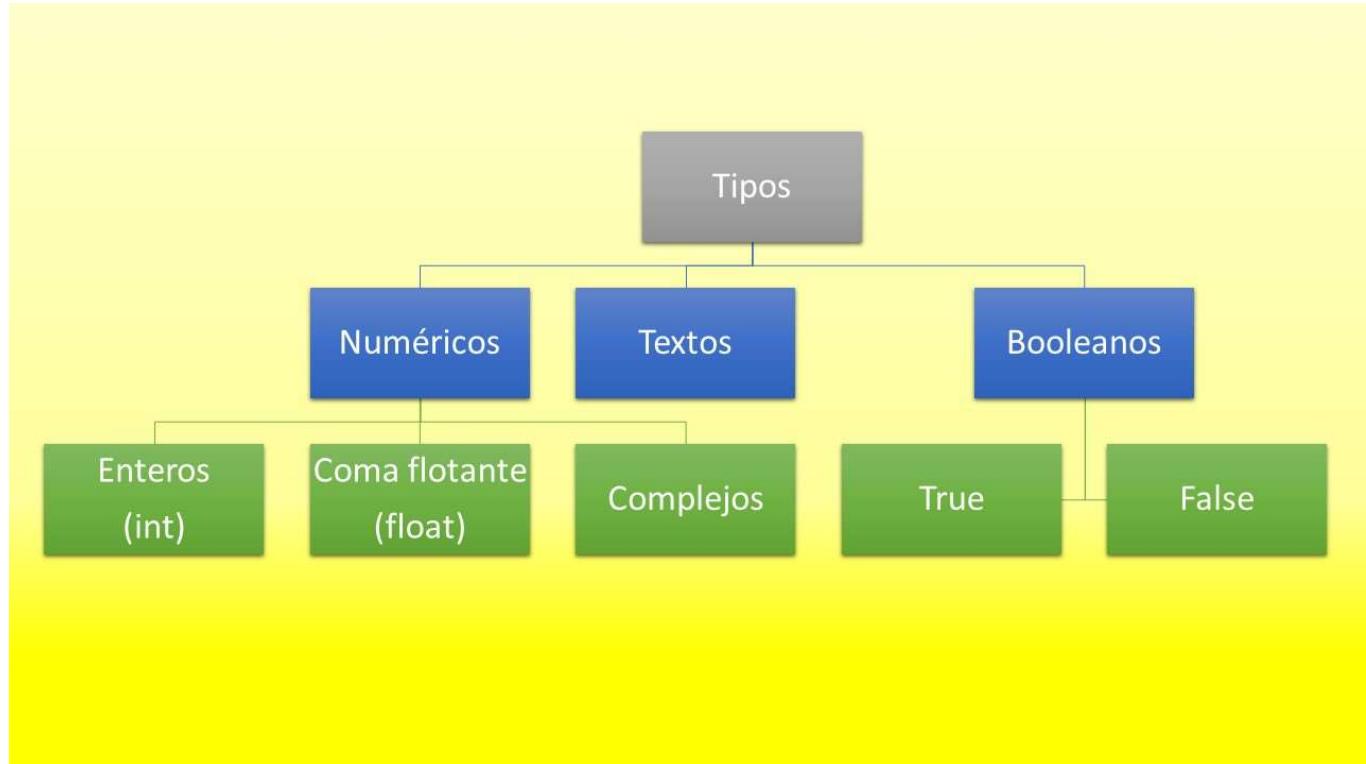
- Los **comentarios**:
 - De una sola línea se realizará mediante el carácter # delante de dicha línea
 - De varias líneas se realizará colocando tres comillas simples (o dobles) al principio y al final del comentario """ ó ''''''
- La **indentación** : en Python resulta de vital importancia ya que se reconocerán los bloques de código gracias a la indentación. Esta indentación suele ser de **cuatro espacios** como norma general.

TIPOS DE DATOS

TIPOS DE DATOS

Los diferentes tipos de datos que podemos encontrar en Python son:

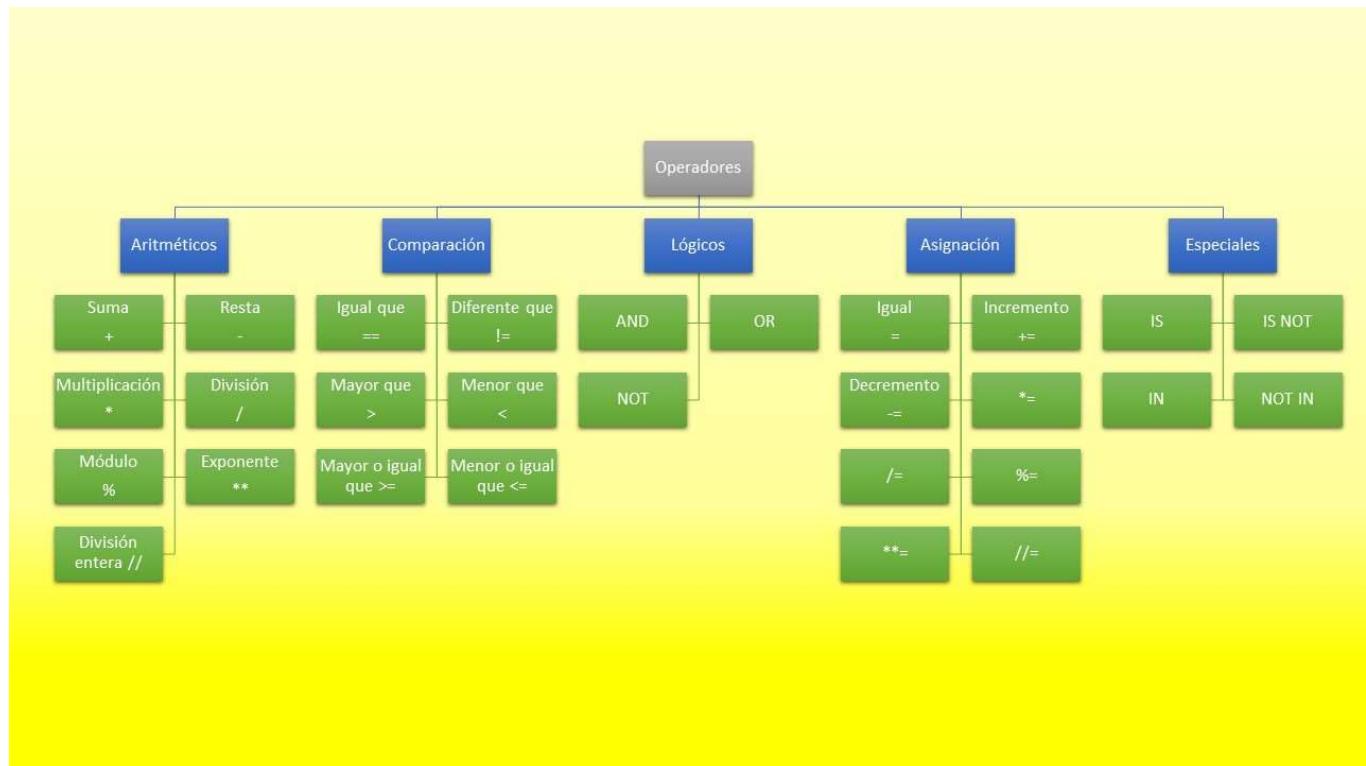
- Numéricos:
 - Enteros (int) : 123
 - Entero largo (con precisión ilimitada) : 123456789
 - Coma flotante (float): 14.6
 - Complejos : 9+7j
- Textos
- Booleanos:
 - True
 - False



OPERADORES

OPERADORES

El siguiente esquema muestra el resumen de los operadores de los que dispone Python



VARIABLES

VARIABLES

Es un espacio en la memoria del ordenador donde se almacena un valor.

Este valor puede cambiar durante la ejecución del programa.

La forma de asignar un valor a una variable en Python es usando el símbolo = .

Ejemplo :

```
1 | numero = 5
```

Si queremos saber el tipo de una variable podemos usar la función predefinida *type*. Por ejemplo, para la variable que hemos definido anteriormente:

```
1 | type(numero)
```

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

Como se puede observar la variable ha tomado el tipo del valor que se le ha asignado.

Algunas características sobre las variables son:

- Nombres no permitidos para las variables:
 - 2mivariable -> no pueden empezar por un número
 - mi-variable -> no puede contener guiones
 - mi variable -> no puede contener espacios
- Se pueden realizar asignaciones múltiples:

```
1 | x, y, z = 1, 2, 3
2 |
3 | print(x)
4 | print(y)
5 | print(z)
```

1
2
3

PALABRAS RESERVADAS

Python tiene un conjunto de **palabras reservadas** que no podemos utilizar para nombrar variables ni funciones, ya que las reserva internamente para su funcionamiento.

Para acceder al listado de palabras reservadas de Python no tendremos más que hacer lo siguiente:

```
1 | import keyword  
2 | print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',  
◀ ━━━━━━ ▶
```

TIPOS Y ESTRUCTURAS

LISTAS

LISTAS

Son estructuras similares a los arrays en otros lenguajes de programación. Nos permiten guardar muchos datos de una forma estructurada. Se definen poniendo sus elementos entre corchetes. Las características principales de las listas en Python son:

- Permiten guardar datos de tipos diferentes

```
1 | milista = ["hola", 3, 55, False]
```

- Se pueden añadir elementos de forma dinámica

```
1 | milista.append("adiós")
2 | print(milista)
```

```
["hola", 3, 55, False,"adiós"]
```

- Se pueden indexar:

```
1 | print(milista[1]) #Accedemos al segundo elemento de la lista. Los índ
```

3

```
1 | print(milista[-2]) #Accedemos al segundo elemento de la lista empezan
```

False

```
1 | print(milista[1:3]) #Accedemos a los elementos de índice 1 y 2. El últ
```



```
[ 3 , 55]
```

```
1 | print(milista[2:]) #Accedemos a los elementos de la lista desde el índ
```



```
[ 55, False, "adiós"]
```

JOIN

JOIN

Esta función permite convertir una lista en una cadena de texto, usando un separador.

Se llamaría de la forma *separador.join(milista)*. Por ejemplo:

```
1 | milista = ["azul", "verde", "amarillo", "rojo"]
2 | print(", ".join(milista))
```

azul, verde, amarillo, rojo

SPLIT

SPLIT

Esta función convierte una cadena de texto en una lista. También puede especificarse un separador. Si este no se especifica se tomará como separador el espacio. Por ejemplo

```
1 | micadena = "Yo soy un estudiante de informática"
2 | milista = micadena.split()
3 | print(milista)
```

```
['Yo', 'soy', 'un', 'estudiante', 'de', 'informática']
```

ó

```
1 | micadena = "perro y gato y ratón"
2 | milista = micadena.split("y")
3 | print(milista)
```

```
['perro ', ' gato ', ' ratón']
```

LEN

LEN

Esta función nos devuelve la longitud de la lista.

```
1 | lista = ["azul", "rojo", "verde", "amarillo"]  
2 | print(len(lista))
```

4

Nota: hay que tener en cuenta que los índices empiezan en 0, así la longitud será el valor del índice del último elemento + 1

COUNT

COUNT

Devolverá el número de elementos de la lista que coincidan con el argumento introducido.

```
1 | lista = ["azul", "rojo", "verde", "amarillo", "verde"]  
2 | print(lista.count("verde"))
```

2

SORT

Esta función ordenará los elementos de la lista de menor a mayor (si se trata de caracteres empezará por las mayúsculas y seguirá con las minúsculas)

Ejemplo:

```
1 | lista = [2,4,5,1,2]
2 | lista.sort()
3 | print(lista)
```

[1, 2, 2, 4, 5]

En caso de lista de caracteres

```
1 | lista = ["H","o","l","a"]
2 | lista.sort()
3 | print(lista)
```

['H', 'a', 'l', 'o']

Ojo! Cuando se ordena una lista, se modifica la lista original, por lo que si tenemos que guardar la lista original habrá que hacer una copia antes de ordenarla, o bien usar la función sorted().

Ejemplo:

```
1 | lista = ["H","o","l","a"]
2 | print("La lista ordenada es: " ,sorted(lista))
3 | print("La lista original es: " , lista)
```

```
('La lista ordenada es: ', ['H', 'a', 'l', 'o'])
('La lista original es: ', ['H', 'o', 'l', 'a'])
```

TUPLAS

TUPLAS

Las tuplas son un elemento enumerable (iterable) que se parecen a las listas pero tienen algunas diferencias:

- Son **inmutables**, es decir, no cambian en el tiempo y por tanto no se pueden aplicar sobre ellas los métodos de las listas que las modifiquen (p.ej. append, remove...)
- Son más rápidas y ocupan menos espacio en memoria
- Permiten formatear cadenas
- Se pueden usar como claves en un diccionario

Ejemplos de usos de las tuplas:

- Se puede definir una tupla sin paréntesis, a esto se le llama **empaquetado de tupla**

```
1 | mitupla = "hola", 3, 45, True  
2 | print(mitupla)
```

```
("hola", 3, 45, True)
```

- El **desempaquetado de tupla** será el proceso inverso, y permitirá asignar los valores de la tupla a variables de una forma sencilla

```
1 | saludo, num1, num2, b = mitupla  
2 | print(num2)
```

45

- Para definir una tupla con un sólo elemento habrá que poner una coma al final

```
1 | mitupla = ("hola",)  
2 | print(mitupla)
```

("hola",)

- Índice en las tuplas

```
1 | mitupla = ("hola", 3, 45, True)  
2 | print(mitupla[1])
```

3

- Convertir una tupla en lista y viceversa

```
1 | milista = list(mitupla)  
2 | print(milista)
```

["hola", 3, 45, True]

```
1 | mitupla2 = tuple(milista)  
2 | print(mitupla2)
```

("hola", 3, 45, True)

- La instrucción **in** permite saber si un elemento está en una tupla

```
1 | print("hola" in mitupla)
```

True

- El método **count** contará cuantas apariciones del elemento que se pase como parámetro se encuentran en la tupla

```
1 | print(mitupla.count("hola"))
```

1

- El método **len** devolverá la longitud de la tupla

```
1 | print(len(mitupla))
```

4

DICCIONARIOS

DICCIONARIOS

Los **diccionarios** son **colecciones de elementos no ordenados**, cuyos elementos siguen el patrón de *par clave : valor*.

Los diccionarios son **mutables**, al igual que las listas, pero no podrán usar la notación *slice* debido a que su índice no siempre tiene por qué ser un entero.

Algunas operaciones importantes con los diccionarios son las siguientes:

- Para **crear** un diccionario se hará usando {}

```
1 | mi_dict = {}    #Se crea un diccionario vacío
2 | mi_dict = { "angel" : "2 de enero" , "laura" : "10 de marzo" , "alba" :
3 |   print(mi_dict)
```

```
{'angel': '2 de enero', 'laura': '10 de marzo', 'alba': '8 de agosto'}
```

Nota: a partir de la versión 3.7 , Python "guarda" el orden en que se introducen las claves, algo que antes no hacía

- Para **añadir** un nuevo *par clave : valor* en un diccionario:

```
1 | mi_dict["maría"] = "5 de Noviembre"
2 | print(mi_dict)
```

```
{'angel': '2 de enero', 'laura': '10 de marzo', 'alba': '8 de agosto', 'marí
```

- Para acceder a un valor del diccionario usando su clave podemos hacerlo de dos modos:
 - Haciendo referencia directamente a esa clave
 - 1 | `print(mi_dict["laura"])`

10 de marzo

O bien haciendo uso de la función `get`:

- 1 | `print(mi_dict.get("laura"))`

10 de marzo

KEYS/VALUES

Para obtener de forma rápida una lista de las claves de un diccionario podremos usar la función `keys()`. Igualmente para obtener la lista de los valores usaremos `values()`. Un ejemplo sería:

```
1 | mi_dict = {"Luis":30, "Maria":24, "Jose":45, "Leo":39}
2 | print(list(mi_dict.keys()))
3 | print(list(mi_dict.values()))
```

```
['Luis', 'Maria', 'Jose', 'Leo']
[30, 24, 45, 39]
```

Así por ejemplo si quisiéramos acceder al valor del cuarto elemento de mi diccionario sin saber la clave, podría hacerlo del siguiente modo:

```
1 | print(list(mi_dict.values())[3])
```

39

ESTRUCTURAS DE CONTROL

En esta sección veremos las diferentes estructuras de control que podemos encontrar en Python, tanto condicionales como bucles.

CONDICIONALES

En ocasiones necesitamos que un determinado bloque de código se ejecute sólo si se dan unas condiciones particulares. En caso de que estas condiciones no se dieran, nuestro programa debería ejecutar quizá otras instrucciones diferentes.

Para conseguir esto con Python usaremos las estructuras de control condicionales que veremos a continuación.

IF - ELIF - ELSE

La construcción de un bloque IF en Python se realizará de la siguiente forma:

if condicion:

 código a ejecutar si la condición es cierta

elif otracondicion:

 código a ejecutar si otracondicion es cierta

else:

 código a ejecutar si ninguna de las anteriores es cierta

Un ejemplo:

```
1 | x = 7
2 | if x == 7:
3 |     print("El valor es 7")
4 | elif x == 5:
5 |     print("El valor es 5")
6 | else:
7 |     print("El valor no es ni 7 ni 5")
```

El valor es 7

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, 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(). Su estructura sería la siguiente:

código si se cumple **if** condición **else** código si no se cumple

Un ejemplo:

```
1 | x = 7
2 | print("es 7" if x==7 else "no es 7")
```

es 7

Otro ejemplo:

```
1 | x = 5
2 | x -= 1 if x > 0 else x
3 | print(x)
```

4

MATCH - CASE

A partir de la versión 3.10 de Python podemos hacer uso de una nueva estructura de control **match - case** similar a la estructura *switch - case* presente en muchos otros lenguajes. Con este tipo de estructura se compara un elemento con diferentes patrones, realizando un bloque de contenido determinado si dicho elemento coincide con el patrón. Su estructura será la siguiente:

match elemento:

case patron1:

 bloque de código 1

case patron2:

 bloque de código 2

...

case _:

 bloque de código en cualquier otro caso

Los patrones pueden ser cualquier tipo de dato o estructura. Veamos un ejemplo:

```
1 | x = "b"
2 | match x:
3 |     case "a":
4 |         print("es la letra a")
5 |     case "b":
6 |         print("es la letra b")
7 |     case "c":
8 |         print("es la letra c")
9 |     case _:
10|         print("no es la letra a, ni la b ni la c")
```

es la letra b

BUCLES

En Python tenemos dos tipos de bucles:

- **Determinados**: son aquellos que se ejecutan un número determinado de veces, es decir, a priori se sabe cuantas veces va a haber que ejecutar el código del interior del bucle. En Python este bucle será el **FOR**.
- **Indeterminados**: son aquellos que se ejecutan un número indeterminado de veces, este número dependerá de las circunstancias que se den durante la ejecución del programa. En Python este bucle será el **WHILE**.

BUCLE FOR

En Python la sintaxis del bucle FOR será de la siguiente forma:

for variable **in** elemento a recorrer: --> Declaración del bucle

instrucciones a ejecutar --> Cuerpo del bucle

Ejemplo:

```
1 | for i in ["a", "b", "c"]:  
2 |     print("Hola")
```

```
Hola  
Hola  
Hola
```

BUCLE WHILE

Con el bucle **WHILE** el código se repetirá **mientras** una condición determinada se cumpla. Llamaremos iteración a una ejecución completa del bloque de código. Dicha condición tendrá que ser actualizada durante la ejecución del código, de lo contrario el bucle nunca acabaría.

La estructura de un bucle WHILE será la siguiente:

while condicion:

 código a ejecutar

Un ejemplo:

```
1 | x = 4
2 | while x > 0:
3 |     print("ahora valgo"+str(x))
4 |     x -=1
```

```
ahora valgo4
ahora valgo3
ahora valgo2
ahora valgo1
```

UTILIDADES

RANGE

En ocasiones en los bucles necesitamos iterar una serie de veces o entre una serie de valores. Para ello podemos usar la función **in** seguido de un objeto iterable (una tupla por ejemplo).

En Python existe otra función muy útil para estos casos, llamada **range**, esta 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. Su estructura por tanto sería:

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

Por ejemplo:

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

```
[5, 7, 9, 11, 13, 15, 17, 19]
```

Usándolo en un **for** se podría hacer así:

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

```
0  
1  
2  
3  
4  
5
```

BREAK

La sentencia **break** nos permite alterar el comportamiento de los bucles while y for. Concretamente, permite terminar con la ejecución del bucle. Esto significa que una vez se encuentra la palabra break, el bucle se habrá terminado.

Un ejemplo de uso de esta sentencia sería el siguiente:

```
1 | cadena = 'Python'
2 | for letra in cadena:
3 |     if letra == 'h':
4 |         print("Se encontró la h")
5 |         break
6 |     print(letra)
```

```
P
y
t
Se encontró la h
```

CONTINUE

El uso de **continue** al igual que el ya visto break, nos permite modificar el comportamiento de los bucles while y for. Concretamente, continue se salta todo el código restante en la iteración actual y vuelve al principio en el caso de que aún queden iteraciones por completar. La diferencia entre el break y continue es que el continue no rompe el bucle, si no que pasa a la siguiente iteración saltando el código pendiente.

Un ejemplo sería:

```
1 | cadena = 'Python'
2 | for letra in cadena:
3 |     if letra == 'P':
```

```
4 |     continue  
5 |     print(letra)
```

y
t
h
o
n

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)