

SISTEMA DE GESTIÓN EMPRESARIAL

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Programando en Python

Indentación

IF

FOR y WHILE

Errores

Funciones, módulos y paquetes o librerías

Cadenas de textos

Ficheros

Orientación a objetos

Indentación y flujo secuencial

Indentación

La indentación en Python facilita tanto la comprensión como la lectura del código, pero ¿qué es la indentación?

Es un desfase de una o varias líneas de código hacia la derecha (mejor dicho varios espacios hacia la derecha), es decir, una especie de **sangrado**. Se trata de algo **muy importante** en Python y una de sus principales características.

Se usa para **agrupar sentencias** y así poder **diferenciar un bloque** de otro, ya que, en Python, **no hay terminadores** de sentencia (como el punto o el punto y coma que se usan en lenguajes como Java, C o C++) **ni llaves** para abrir y cerrar bloques. Esto evita que los programadores comenten los errores típicos que se comenten al olvidar estos delimitadores.

A partir de este punto, veremos algunos ejemplos de indentación, pero como adelanto, podremos analizar el siguiente:

Ejemplo de indentación

if <condición>:

instrucción si es verdadera la condición

Nueva instrucción no indentada # fuera del bloque condicional

Un IDE como **PyCharm** realiza el sangrado o **indentación automáticamente**, mientras que si nos decantamos por usar un editor de textos, como puede ser el bloc de notas, deberemos realizarlo nosotros manualmente haciendo uso de espacios o tabuladores.

Flujo de programación secuencial

En este punto, veremos la sentencia condicional IF, que se basa en que si se cumple una condición, realiza una instrucción concreta.

En Python, se declara así:

Declaración IF

if <condición 1>:

instrucción 1: se cumple la condición 1

elif < condición 2>:

if <condición 2.1>:

instrucción 2: se cumplen las condiciones 2 y 2.1

else:

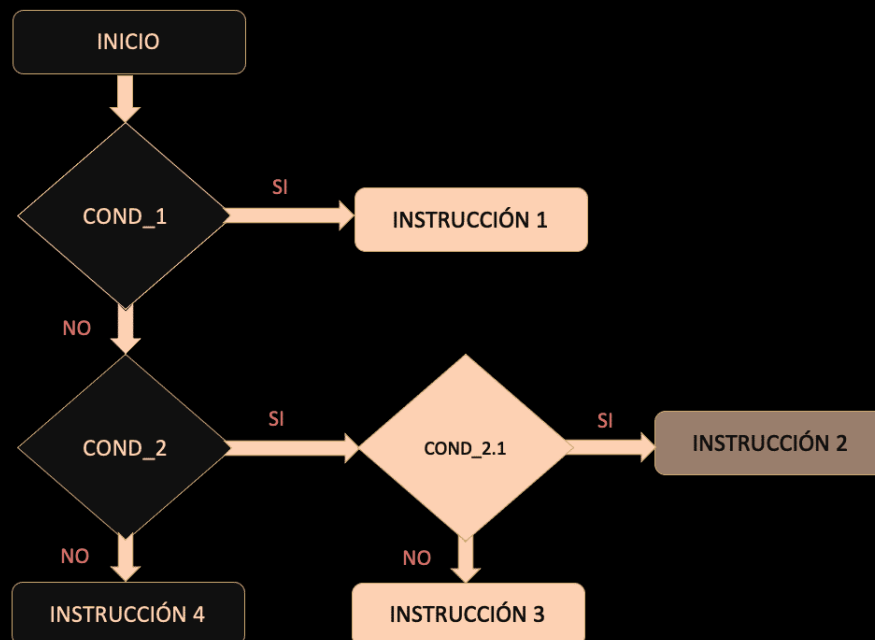
instrucción 3: se cumple la condición 2, pero no la 2.1

else:

instrucción 4: no se cumplen las condiciones 1 ni 2

Correspondiéndose con el siguiente diagrama de flujo:

Diagrama de flujo del ejemplo de la sentencia condicional



Pudiéndose usar un formato acortado para el if y para el if-else:

Formato reducido IF

Con IF:

```
if a > b: print(f"{a} es mayor que {b}")
```

Con IF ... ELSE:

```
print(a) if a > b else print(b)
```

Flujo de programación iterativo y range

Python dispone de dos comandos **iterativos**: WHILE y FOR, y una función para generar **progresiones aritméticas**: RANGE.

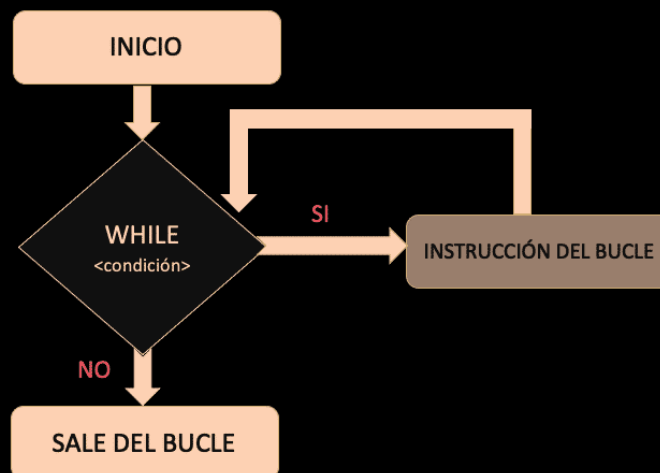
El bucle WHILE

Este bucle repite la/s instrucción/es que se encuentren en su interior mientras se cumpla la condición. En Python, se declara de la siguiente forma:

Declaración while

```
While <condición>:  
    instrucción/es
```

Diagrama de flujo del bucle while



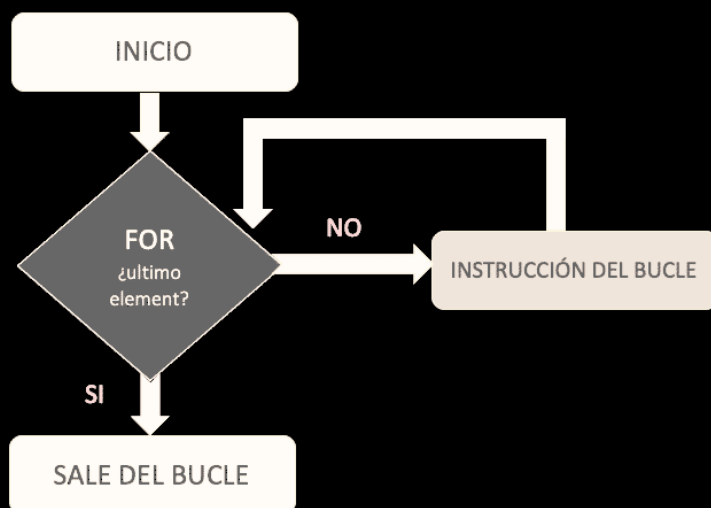
El bucle FOR

A diferencia de otros lenguajes, en Python, este bucle **se usa solo para iterar sobre secuencias** (lista, tupla, diccionario, cadena...) en el **orden en el que aparecen** los elementos en la secuencia. En Python, se declara así:

Declaración for

For <elemento> in <secuencia>:
Instrucción/es

Diagrama de flujo del bucle for



Para los bucles **WHILE** y **FOR**, existen las siguientes sentencias:

- Break: **para la iteración**, incluso si la condición es verdadera.
- Continue: para la iteración actual del bucle y **pasar a la siguiente**.

La función range

La función range() genera listas con progresiones aritméticas, muy útil para iterar con bucles.

Por ejemplo:

Range(3) generaría → [0, 1, 2].

Esta función **empieza en 0 por defecto** y va **incrementando** los valores **de 1 en 1**.

Pero podemos hacer que empiece en otro número y se incremente de otra forma.

Por ejemplo:

Range(5, 21, 5) generaría → [5, 10, 15, 20]

empieza en 5, termina en 21 y se incrementa de 5 en 5.

Control de errores

Cuando se produce un error en Python, este detiene la ejecución del programa y muestra el error. Pero estos son errores del estándar de Python, que pueden llegar a ser difíciles de entender por los usuarios finales. Por ello, Python nos permite **manejar el control de errores** con las siguientes instrucciones:

- **Try:**

Se usa al inicio del bloque para indicar que en ese bloque se estarán capturando errores.

- **Except:**

Bloque en el que se **maneja el error**. Se pueden definir tantos bloques except como necesitemos, indicando el tipo de error que se controlará. (Nota: como no existen llaves en Python, con Except se indica que empieza el manejo del error en caso de que suceda, similar al catch en Java).

Con el comando `as`, se nos permitirá **capturar el objeto** de excepción.

- **Finally:**

Bloque que **se ejecutará siempre** al final, tanto si hay errores como si no.

Además, Python nos permite hacer uso de la instrucción `else` para definir un bloque de código **cuando no se produzcan errores**.

También, con el comando `raise`, Python nos da la opción de **lanzar excepciones** indicando, incluso, el tipo de excepción si fuese necesario.

Un ejemplo completo sería el siguiente:

Ejemplo de control de errores

`try:`

```
#instrucciones (10/0) ("a"+2) (2+2)  
except ArithmeticError as error: #captura errores aritméticos  
    print("Se ha producido el siguiente error aritmético:", str(error))  
except Exception as error: #captura el resto de errores  
    print("Se ha producido el siguiente error:", str(error))  
else: #entra si no se producen errores  
    print("No se han producido errores")  
finally: #entra siempre  
    print("bloque finally, se ejecuta siempre")
```


De esta forma:

- Si la instrucción fuese $10/0$, devolvería el mensaje de error aritmético y seguidamente el del finally.
- Si la instrucción fuese $"a" + 2$, devolvería el mensaje de error del exception y luego el del finally.
- Si la instrucción fuese $2+2$, devolvería 4, el mensaje indicando que no se han producido errores y el del finally.

Ejemplos de programación Python

Sentencia condicional IF, los bucles WHILE y FOR con RANGE

Hablaremos de algunas de las sentencias que hemos visto hasta ahora en este tema. Empezaremos hablando de la **sentencia `if`**.

Por ejemplo, en este caso tenemos dos variables `x` e `y`.

Si `x` es mayor que `y`, pintaremos "x mayor que y".

Si `x` es menor que `y`, pintaremos "x menor que y".

Y si no, pintaremos "x igual a y".

En este caso, por ejemplo, si `x` valiese 7, pintaría "x menor que y", porque es menor que 8.

Si fuese mayor, pintaría "x mayor que y".

Y si fuesen iguales, entraríamos por el `else` y pintaría "x igual a y".

```
IF SENTENCIA IF
x=8
y=8
if x > y:
    print("X > Y")
elif x < y:
    print("X < Y")
else:
    print("X = Y")
```

Esta misma sentencia IF se puede usar de una forma más corta, como se muestra a continuación:

Pintaremos "X > Y", si `x` es mayor que `y`.

Si no, pintaremos "X < Y", si `x` es menor que `y`.

Y si no, pintaremos "X = Y".

Podremos ir encadenando tantos `else` como necesitemos.

Por ejemplo, en este caso, si `x` es menor que `y` y ejecutamos, nos pinta que "X < Y".

Si `x` es mayor que `y` y ejecutamos, nos pinta que "X > Y".

Y si `x` fuese igual que `y` y ejecutamos, nos pinta que "X = Y".

Con esto ya hemos visto la sentencia `if`.

```
# SENTENCIA IF ACORTADA
x=5
y=5
print("X > Y") if x > y else print("X < Y") if x < y else print("X = Y")
```

Ahora, por ejemplo, queremos el **bucle `while`**.

En este caso, lo que haremos será recorrer un bucle incrementando en 1 la variable `x` mientras `x` sea menor que 20.

Si el resto dividido entre 2 es igual a 0, eso significa que el número es par. Entonces, no pintaremos nada.

Si `x` es igual a 15, terminaremos el bucle y, en otros casos, iremos pintando el valor de `x`. Por ejemplo, en el primer caso, `x` valdrá 0, que el resto es 0. Entonces, no pintará nada.

Con el `continue`, volverá arriba. `x` le incrementa 1. Entonces, `x` valdrá 1. El resto dividido entre 2 no es igual a 0. Tampoco es el valor 15. Entonces, lo pintará e incrementará 1. Ahora volvemos al 2, que es par, por lo que incrementará el valor de `x`, continuará y volverá aquí arriba. `x` ahora es 3, no es par, no es 15, lo pintará. Y así hasta el valor 15, que hará que salga del bucle igual. En este caso, pintaremos los números entre 1 y 15 que no sean pares.

```
# BUCLE WHILE
x = 0
while x < 20:
    if x % 2 == 0: # si el resto de dividirlo entre 2 es 0 -> es par
        x += 1
        continue
    if x == 15:
        break
    print(x)
    x += 1
```

Ahora hablaremos sobre el **bucle `for`**.

El bucle `for` va a recorrer una lista, y en este caso es una lista de deportes, de cadenas de texto de deportes. Por cada elemento de la lista de deportes, que llamaremos `deporte`, si el **primer carácter es una 'f', continúa**. Es decir, que no hace nada. Vuelve al siguiente elemento de la lista. Si el deporte es "equitación", termina de recorrer el bucle, y si no, pinta. En este caso, entrará el primer elemento, la 'f', como el primer carácter es la 'f', pues el "fútbol", el primer carácter es la 'f', pues continúa y no lo pinta. El siguiente "baloncesto" no es la 'f' ni es "equitación", entonces lo pinta. "Fórmula 1", si es la 'f', continúa. "Tenis" no es la 'f' ni es "equitación", entonces lo pinta. Y en el siguiente elemento, que si es "equitación", no es 'f' pero es "equitación", entonces **hace el `break` que sale del bucle. Pintará entonces "baloncesto" y "tenis"**, como podemos apreciar en este ejemplo.

```
# BUCLE FOR
deportes = ["futbol", "baloncesto", "formula 1", "tenis",
            "equitación", "ciclismo"]
for deporte in deportes:
    if deporte[0] == "f":
        continue
    elif deporte == "equitación":
        break
    print(deporte)
```

A continuación, hablaremos del uso del `range`.

En este caso, por ejemplo, vamos a recorrer un bucle para una variable `x`:

en el rango **de 5 hasta 45**,

de 10 en 10, y vamos a pintar `x`.

Es decir, empezaría por el valor 5, luego le incrementaría 10, y así hasta que el valor llegue a 45.

```
# RANGE
for x in range(5, 45, 10):
    print(x)
```

Ejemplo: Hallar el número entero mayor y menor de una lista

Necesitamos implementar un programa que devuelva el **número entero mayor y menor de una lista heterogénea**, es decir, podrá tener números enteros, reales, caracteres, cadenas, etc.

Por ejemplo, para una lista como la que sigue: `lista = [1, 2, "árbol", 5, 1.5, 6, "a"]`

el programa devolverá: Menor: 1 / Mayor: 6

Para ello, en primer lugar, crearemos las variables 'menor' y 'mayor' y las inicializaremos a vacío (**None**).

A continuación, recorreremos la lista, y por cada elemento, comprobaremos si es un entero. En caso afirmativo, chequearemos si dicho entero es menor que nuestra variable menor; entonces, la variable menor tomará ese valor.

A continuación, haremos lo mismo para la variable mayor, es decir, si el elemento es mayor que la variable mayor, entonces, la variable mayor tomará dicho valor. De esta forma, una vez que se termine de recorrer la lista, tendremos en 'menor' el valor menor y en 'mayor' el valor mayor. Pero ¿qué ocurrirá en la primera iteración del bucle cuando compare el primer elemento entero con la variable 'menor' si esta tiene un valor None?

El programa dará el siguiente error:

Error que se da al comparar un tipo 'int' con un 'NoneType'

`TypeError: '<' not supported between instances of 'int' and 'NoneType'`

El motivo es que **no se puede comparar un tipo 'int' con un 'NoneType'**. Por ello, antes de realizar las comparaciones, **si las variables 'menor' y 'mayor' están vacías**, les **daremos el valor** del primer elemento entero que nos encontremos, puesto que en ese momento será el menor y el mayor a la vez en la lista.

De esta forma, el código quedará como sigue:

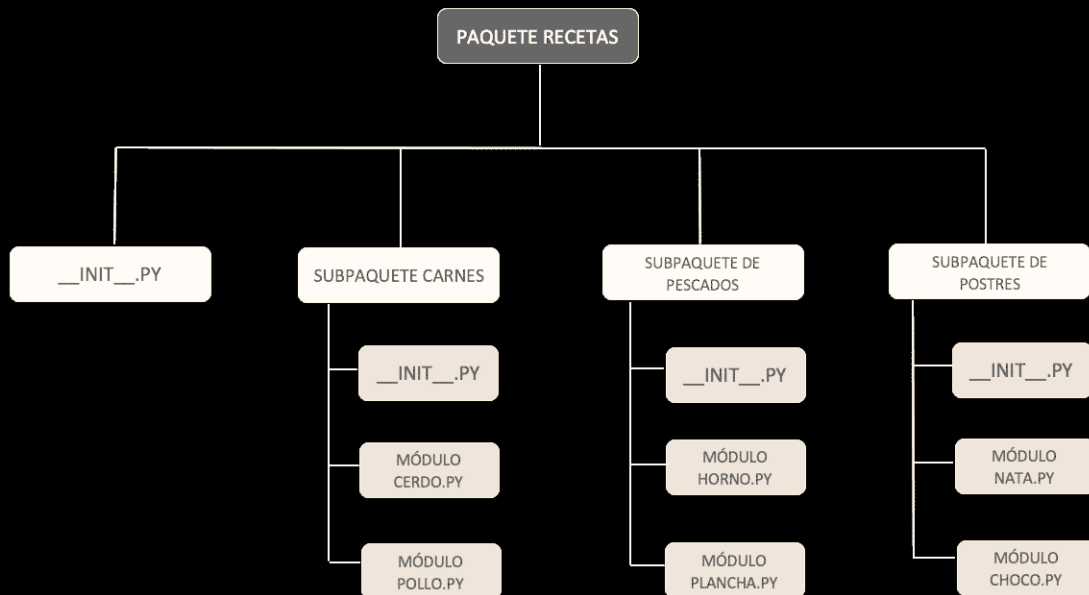
```
menor = None #reinicializamos la menor
mayor = None #reinicializamos la mayor
lista = [1, 2, "árbol", 5, 1.5, 6, "a"]
for x in lista:
    if type(x) is int:
        if menor == None and mayor == None: #es la primera iteración con un entero
            menor = mayor = x
        if x < menor:
            menor = x
        elif x > mayor:
```

```
        mayor = x  
print ("Menor:", menor)  
print ("Mayor:", mayor)
```

Estructura del código: Funciones

Estructura del código

Estructura del código en Python



Estructurando el código

Para estructurar el código y hacerlo más potente, legible, reutilizable y portátil Python ofrece las funciones, los **módulos** y los **paquetes** o librerías.

Esto nos permitirá:

Crear código -> empaquetarlo -> distribuirlo -> reutilizarlo.

Para comprender mejor los tres conceptos, de los que hablaremos a continuación, haremos una analogía con las recetas de cocina. De esta forma:

Las **funciones** vendrían a ser las recetas en sí, por ejemplo, la **receta** de una tarta de chocolate.

Los **módulos** los **libros de recetas**, por ejemplo, un libro de receta de postres.

Los paquetes (o librerías) estanterías llenas de libros de recetas, por ejemplos, libros de recetas de postres de diferentes autores con diferentes recetas.

Funciones: 'Las recetas'

Son como cajas negras que **realizan una serie de instrucciones cuando son llamadas**, de forma que al usarlas, solo nos preocupará saber qué hacen, pero no cómo lo hacen internamente con código Python. Suelen tener un nombre característico de la acción que realiza y **pueden recoger parámetros y devolver valores**.

Un ejemplo es la función `print(value1, value2, ..., sep=' ', end='\n'...)`, cuya finalidad es imprimir por pantalla. Además, tiene los parámetros (opcionales) 'sep', 'end', etc., de los que ya hemos hablado.

Para **crear** una función, basta con usar:

Creación de una función

```
def<nombre_función> (parámetros si fuese necesario):  
    <implementación>  
    return <valor> opcional: si devolviese algún/os valor/es
```

Y para **llamarla**, basta con usar su nombre:

Llamar a una función

<nombre_función>(parámetros si tuviese)

Un posible ejemplo de declaración y uso de función sería el siguiente, dando como resultado: 7 y 12:

Ejemplo declaración y uso de función

```
def sum_mul (num1, num2):  
    return num1 + num2, num1 * num2  
sum, mul = sum_mul(3,4)  
print(sum)  
print(mul)
```

***Nota:** cuidado con las variables globales y locales. Las variables que se definan **dentro de la función** solo serán visibles ahí dentro. Si queremos **modificar dentro de la función una variable global**, hay que usar la **etiqueta 'global'**.

Estructura del código: Módulos y librerías

Módulos: 'Los libros de recetas'

Son ficheros que **contienen las funciones en código Python** y se denominan con **<nombre_modulo>.py**.

Antes de usarlo, debemos tener instalados el módulo, y para **comprobar los módulos** que tenemos instalados, bastará con ejecutar el comando `help('modules')` dentro de la consola de Python. Este dispone de muchos módulos estándar consultables desde aquí.

Para instalar un módulo, se puede hacer con el comando:

pip + install + <nombre del módulo>

Una vez creado o instalado el módulo, ya podemos **usarlo** haciendo uso de la sentencia:

import <módulo>

Por ejemplo, podremos crear el módulo 'holamundo.py' (extensión .py) con varias funciones en su interior entre las que se encuentra:

```
def holamundo():  
    print("hola mundo")
```

A continuación, en nuestro programa, podremos importarlo y usar dicha función con la instrucción `def holamundo()`:

Importar holamundo

```
import holamundo  
holamundo.holamundo()
```

Otra forma de importarlo sería usando el atributo from, así no tendremos que indicar el módulo al usar la función:

Importar con from

```
from holamundo import holamundo  
holamundo
```

Los paquetes o librerías: 'Las estanterías'

Los paquetes no son más que una carpeta que contiene varios módulos (archivos '*.py') y/o paquetes. Se caracterizan por tener un **archivo de inicialización** '**__init__.py**' (el contenido **suele estar vacío**). Python dispone de una librería estándar que le otorga al lenguaje multitud de funcionalidades. Consúltala aquí: <https://docs.python.org/3/library/index.html>.

Pero, además, existen una gran cantidad de librerías de terceros fácilmente instalables: <https://pypi.org/>.

Con el **comando pip**, podremos:

- **instalar paquetes** (pip install <paquete>),
- **instalar una versión** concreta (pip install -Iv <paquete>==<versión>),
- **actualizarlo** (pip install -U <paquete>==<versión>),
- **desinstalarlo** (pip uninstall <paquete>)
- **listar** todos (pip list).

Para usar un paquete, bastará con hacer un import <paquete> en nuestro proyecto y podremos hacer uso de todos sus módulos y funciones. Si quisiéramos importar un módulo concreto, podríamos hacer:

```
import <paquete>.<módulo>
```

Ficheros y cadenas de texto

En Python, disponemos de funciones que nos permitirán crear, abrir, modificar, cerrar, eliminar y tratar ficheros.

Apertura de ficheros

El comando Open tiene dos parámetros:

- el nombre del fichero
- y el modo, que puede ser:
 - r: en modo lectura.
 - a: en modo **agregar o crear** si no existe.
 - w: en modo escritura.
 - x: modo **creación**.

```
open("<nombre_fichero>","<modo>")
```

Adicionalmente, se puede especificar si se desea tratar en modo:

- **texto** ("t" por defecto),
- o **binario** ("b").

De esta forma, para abrir un fichero en **modo escritura binaria**, podríamos usar:

```
f = open("archivo.txt", "wb")
```

Si no se especifica nada en <modo> **por defecto**, lo abre en modo **texto y lectura**.

Cierre de ficheros

Es una buena costumbre cerrar los ficheros cuando ya no se están tratando. Para ello, se usa la función close().

Continuando con el ejemplo anterior:

```
f.close()
```

Lectura de ficheros

Una vez abierto, podremos leer su contenido haciendo uso de la función `read()` (para leer todo su contenido) o `readline()` (para leer su contenido por líneas). Siguiendo el ejemplo:

- `print(f.read())`: imprimirá el **contenido** del archivo.
- `print(f.readline())`: imprimirá la **primera línea** del archivo.

Escritura de ficheros

Una vez abierto, podremos escribir en el mismo usando la instrucción `write()`. Continuando con el ejemplo anterior:

```
f.write("texto nuevo")
```

- Si lo abrimos con el `'w'`, se **sobrescribirá** el contenido del fichero;
- si lo abrimos con `'a'`, se **agregará**.

Otras funciones con ficheros

Existen muchas más funciones de las que destacaríamos:

- `tell()` que devuelve en qué **posición del archivo** estamos.
- `seek()` para **irnos a una posición** concreta del archivo.
- `flush()` para que **se realice** todo lo que tiene en **buffer**.

Trabajando con cadenas de texto

Analizaremos, con ejemplos en PyCharm, el uso de las cadenas de texto en Python.

Veremos cómo pueden ser declaradas tanto con comillas simples (`'`) como con dobles (`"`), el uso de `"\` y `"\n"`, las subcadenas, y algunas funciones.

Analizaremos el uso de las cadenas de texto en Python, con ejemplos.

Para empezar, comentar que Python permite tanto el uso de la **comilla simple** como la **comilla doble** para generar las cadenas de texto. De esta forma, la variable `x` y la variable `y`, una con comilla simple y otra con comilla doble, imprimirán el mismo valor.

```
# comillas simples y dobles
x = 'Hola mundo'
y = "Hola mundo"
```

```
print(x)
print(y)
```

Esto también se extrapola a los textos de varias líneas. Por ejemplo, podemos usarlos con **cadenas simples o con cadenas dobles**. En este ejemplo, las líneas se imprimen con un retorno de carro o un intro al final.

```
# varias Lineas de texto
x = '''esto es una cadena
de texto de
varias Lineas'''
y = """esto es una cadena
de texto de
varias Lineas"""
print(x)
print(y)
```

Si no quisiéramos que se imprimiese ese retorno de carro, podríamos hacer uso de la **barra invertida**. En este caso, la barra invertida nos indica que **todo es una misma línea**. Si imprimimos el mismo texto con las barras estas al final, podemos ver que se imprime todo en una misma línea.

```
# la \
x = '''esto es una cadena \
de texto de \
varias Lineas'''
print(x)
```

resultado:

esto es una cadena de texto de varias Lineas

Si quisiéramos, de alguna forma, que todo lo escribiéramos en una línea, pero se imprimiese un intro, el carácter para el **retorno de carro es el "\n"**. En este caso, aunque todo esté en una misma línea, al imprimirlo, podremos ver que a partir del "\n" se imprime en otra línea.

```
# la \n
x = "esto es una Linea \n esto es otra"
print(x)
```

La concatenación de cadenas de caracteres se puede hacer de varias formas, pero la más fácil es usando el **carácter "+"**.

En este caso, tenemos x con "hola " e y con "mundo", y al imprimir la concatenación se imprime "hola mundo".

```
# concatenación
x = "hola "
y = "mundo"
```

Pasamos a hablar del uso de la función `len()`, que es la **longitud**.

En este caso, tenemos una cadena que es "tamaño de la cadena", que si queremos imprimir la longitud de esa cadena, podremos hacer uso de la función `len()`. En este caso, nos indica que la cadena tiene 19 caracteres.

```
# función len()
x = "tamaño de la cadena"
print(len(x))
```

Ahora pasaremos a comentar el uso de las cadenas de texto como array.

Al fin y al cabo, un string en Python no deja de ser como un array, que ya conocemos de otros lenguajes. Por ello, podemos imprimir:

```
x = "0123456789"
```

- el **primer carácter** de la cadena con esta instrucción:

```
print(x[0])
resultado: 0
```

- los caracteres comprendidos entre la posición 8 y 9 de esta cadena:

```
print(x[7:9])
resultado: 78
```

- los caracteres comprendidos entre el inicio y la posición 7:

```
print(x[:7])
```

resultado: 0123456

- los comprendidos entre la posición 7 y el fin de la cadena:

```
print(x[7:])
```

resultado: 789

- los comprendidos entre el final menos 7, o sea, 7 caracteres para que llegue al final y el final:

```
print(x[-7:])
```

resultado: 3456789

- o en este último caso, los caracteres que están comprendidos desde la posición 3 hasta el final menos 1 de esta cadena.

```
print(x[3:-1])
```

resultado: 345678

Podremos decirle que nos imprima la cadena "hola mundo" todo en minúscula:

```
# imprime todo en minúsculas:  
x = "Hola Mundo"  
print(x.lower())
```

O todo en mayúscula:

```
# imprime todo en minúsculas:  
print(x.upper())
```

También podremos reemplazar caracteres o palabras de un string por otros.

En este caso, imprimiremos el resultado de reemplazar "hola" por "adiós". Y ahora tenemos "adiós mundo", pero en este caso solamente lo hemos hecho a la hora de imprimir:

```
# reemplazo de caracteres:  
x = "Hola mundo"  
print(x.replace('Hola', 'Adiós'))
```

Si intentamos hacer uso de la función `find()` para buscar la palabra "adiós" en la variable `x` nos va a decir que no está, nos va a **devolver menos 1**, porque realmente no lo hemos reemplazado, hemos imprimido el reemplazo:

```
print(x.replace('Hola', 'Adiós'))
```

Si imprimimos la `x` solamente, vemos que se imprime "hola mundo". Este "adiós mundo" es la impresión del reemplazo y esta `x` es la variable que realmente no ha sido cambiada.

```
print(x)
```

resultado:

-1

Hola mundo

Explicaremos el uso de la función `split()`, la cual nos **divide** nuestra **cadena** en **elementos de una lista**. Podemos apreciar que se imprime por un lado "hola" y por otro "mundo" como dos objetos en una lista.

```
# función split():  
x = "Hola mundo"  
y = x.split()  
print(y)
```

resultado: ['Hola', 'mundo']

Ejemplo de conversión de decimal a binario usando ficheros

Necesitamos crear un programa en Python que lea un fichero denominado 'decimal.txt' que contiene un número decimal y que lo convierta a binario y escriba el resultado en otro fichero denominado 'binario.txt'.

En primer lugar, crearemos una función para convertir el decimal que se le pasa por parámetro a binario (sin hacer uso de la función bin()). A continuación, en el programa principal, abriremos el fichero decimal, leeremos su contenido y lo cerraremos. Seguidamente, llamaremos a la función que hemos creado para convertir el decimal, abriremos el fichero de salida, escribiremos el resultado en él y lo cerraremos.

La función la denominaremos ConvertirBin, tendrá un parámetro de entrada denominado 'decimal' y devolverá una cadena de unos y ceros.

Tendrá la siguiente forma:

```
def ConvertirBin(decimal):  
    binario = ""  
    # mientras la división entera del decimal entre 2 no sea 0  
    while decimal // 2 != 0:  
        # concatena el resto de dividirlo entre 2 al resultado por la izquierda  
        binario = str(decimal % 2) + binario  
        # volcar en el decimal el resultado de la división entera entre 2  
        decimal = decimal // 2  
    return str(decimal) + binario
```

A continuación, abriremos el fichero, lo leeremos y lo cerraremos:

```
fich = open("decimal.txt", "r")  
dec = fich.read()  
fich.close()
```

Después, llamaremos a la función que hemos creado para convertir el decimal, ojo, pero el que hemos leído está en formato str, deberemos hacerle un **casting a int** al pasarlo a la función:

Llamada a la función para convertir a binario

```
bin = ConvertirBin(int(dec))
```

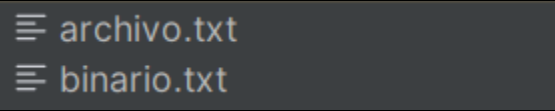
Finalmente, **abriremos el fichero de salida, escribiremos** el resultado en él y lo cerraremos.

Creación, escritura y cierre del fichero resultante

```
fich = open("binario.txt", "w")  
fich.write(bin)  
fich.close()
```

De esta forma, para el siguiente fichero: decimal.txt (Fichero de lectura)

El programa creará el siguiente resultado: binario.txt (Fichero de escritura)



```
≡ archivo.txt  
≡ binario.txt
```

Orientado a objetos

Python es un lenguaje de programación orientado a objetos. De hecho, **casi todo en Python es un objeto con sus propiedades y métodos**.

Crear una clase

Bastará con usar la **sentencia class**, definir su **método `__init__()`** (esto es recomendable), que será el constructor que se llamará automáticamente al crear un objeto de esa clase y definir los métodos de la clase:

Definir clase y métodos

```
class nombre_clase:
    def __init__(self, propiedad1, propiedad2):
        self.propiedad1 = propiedad1
        self.propiedad2 = propiedad2

    def nombre_metodo(self, parámetro1, parámetro2):
        # acciones del método
```

Siendo `self` un parámetro que hace **referencia a la instancia actual de la clase** que se está llamando, y se deberá usar como **primer parámetro de todos los métodos de la clase** (se puede **nombrar como queramos**)

De esta forma, un ejemplo de creación de una clase podría ser:

Clase DatosUsuario

```
class DatosUsuario:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def escribir_datos(self):
        print(f"El nombre es: {self.nombre} y su edad es: {self.edad}")
```

Crear un objeto de esa clase

Bastará con **crear una variable llamando a la clase**:

Crear objeto

Objeto = <nombre_clase>(<propiedad1>, <propiedad2>...)

Siguiendo el ejemplo anterior:

Creación del objeto usuario de la clase DatosUsuario

```
usuario = DatosUsuario("José", 46)
```

Llamar a un método de una clase

Una vez creada la clase y el objeto de dicha clase, solo será necesario llamar al método poniendo un '.' entre el objeto y el método:

Llamada al método

Objeto.metodo(<parámetro1>,<parámetro2>...)

Siguiendo el ejemplo anterior, la llamada al siguiente método imprimirá el siguiente resultado:

Llamada al método escribir_datos de la clase DatosUsuario

```
usuario.escribir_datos()
```

Resultado: El nombre es: José y su edad es: 46

Ejemplo de uso de librerías de terceros

Necesitamos implementar un programa en Python que pinte por pantalla figuras geométricas dependiendo del número de lados que introduzca el usuario por pantalla.

Hay que tener en cuenta que cada figura geométrica de x lados comparte la propiedad que la suma de sus lados es 360° . Así, sabemos que, por ejemplo, un triángulo deberá girar en cada esquina 120° ($360/3$), un cuadrado deberá girar en cada esquina 90° ($360/4$).

Algo que, a priori, parece tan complejo, es muy fácil haciendo uso de librerías. Si echamos un vistazo a la librería turtle aquí, podemos apreciar que esta tiene las siguientes funciones:

- `Screen()`: para pintar la pantalla.
- `Turtle()`: para crear el objeto Turtle.
- `Forward(x)`: mueve a la tortuga la **distancia x que se le indique**.
- `Left(x)`: **gira** la tortuga hacia la izquierda **x ángulos**.

Nota: Para solucionar el error `ModuleNotFoundError: No module named 'turtle'`, podría funcionar:

https://www.youtube.com/watch?v=0x_MEKr0OJQ

Si no se puede instalar directamente desde la terminal:

`pip install turtle`

o

`apt install python3-tkinter`.

En este caso, habría que usar la terminal para el código, y funciona.

Ya solo nos quedará **implementar el programa** con los datos específicos. Para ello, en primer lugar, **importaremos la librería** turtle:

```
import turtle
```

A continuación, **crearemos los objetos** 'ventana' y 'tortuga' (screen y turtle, respectivamente):

```
ventana = turtle.Screen()  
tortuga = turtle.Turtle()
```

Seguidamente, crearemos la función **'figura'** cuyo **parámetro será el número de lados** y cuya función será la de:

- **"Imprimir una línea de distancia 100,**
- **y girar 360/lados ángulos"** un número 'lados' de veces.

Finalmente, le **preguntaremos al usuario** el número de lados por pantalla y llamaremos a la función 'figura' con esos lados.

De esta forma, el código queda como sigue:

Código para pintar figuras geométricas

```
import turtle

ventana = turtle.Screen()
tortuga = turtle.Turtle()

def figura(lados):
    for i in range(lados):
        tortuga.forward(100)
        tortuga.left(360 / lados)

lados = input("Introduce el número de lados de la figura: ")

figura(int(lados))
ventana.exitonclick()
```

<https://www.w3schools.com/python/default.asp>

<https://www.programiz.com/python-programming/package>