



# Curso de Programación Python

Por: Ing. Darwin Calle  
[dACL010811@gmail.com](mailto:dACL010811@gmail.com)

# CONTENIDO

- 1** **Unidad I**  
Introducción a Python
- 2** **Unidad II**  
Tipos de Datos Simples
- 3** **Unidad III**  
Estructuras de Control
- 4** **Unidad IV**  
Funciones en Python

# CONTENIDO

- 5 **Unidad V**  
Archivos - Ficheros
- 6 **Unidad VI**  
Tipos de Datos Estructurados
- 7 **Unidad VII**  
Control de Excepciones
- 8 **Unidad VIII**  
Librería Datetime

# CONTENIDO



# Introducción a Python

**Python es un lenguaje de programación de alto nivel multiparadigma que permite:**

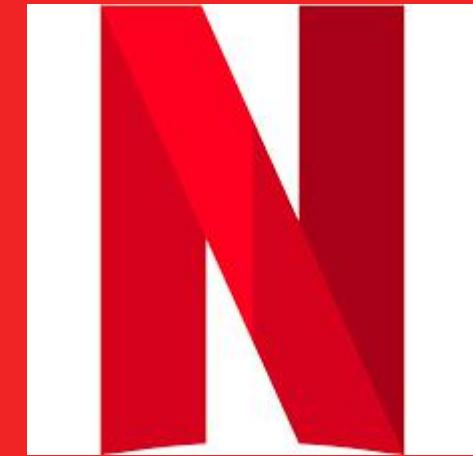
- Programación imperativa
- Programación funcional
- Programación orientada a objetos
- Fue creado por Guido van Rossum en 1990 aunque actualmente es desarrollado y mantenido por la Python Software Foundation

## Ventajas:

- Es de código abierto (certificado por la OSI).
- Es interpretable y compilable - Lenguaje de Alto Nivel
- Es fácil de aprender gracias a que su sintaxis es bastante legible para los humanos.
- Es un lenguaje maduro (29 años).
- Es fácilmente extensible e integrable en otros lenguajes (C, java).
- Esta mantenido por una gran comunidad de desarrolladores y hay multitud de recursos para su aprendizaje.



NetFlix



Yahoo!



Quienes utilizan Python

Instagram



Google



# Introducción a Python

1

## Tipos de ejecución

- Interpretado en la consola de Python
- Interpretado en fichero
- Compilado a bytecode
- Compilado a ejecutable del sistema

## Entornos de Desarrollo

- Consola integrada de python
- Sublime Text
- Atom
- Eclipse
- PyCharm
- Visual Studio Code
- Jupyter Notebook
- ipython : modo consola
- Control de Versiones : gitlab o GitHub



# Entorno de Desarrollo



## Instalación de Python

Linux, Windows y MAC

## Configuración Entorno

Visual Studio Code - GitHub - GitLab

## PIP8

Mejores Prácticas - Clean Code

# Tipos de Datos Simples

## Tipos de datos primitivos simples

- **Números (numbers):** Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números.  
Ejemplo. 0, -1, 3.1415.
- **Cadenas (strings):** Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas simples o dobles.  
Ejemplo. 'Hola', "Adiós".
- **Booleanos (boolean):** Contiene únicamente dos elementos True y False que representan los valores lógicos verdadero y falso respectivamente.

**Estos datos son inmutables, es decir, su valor es constante y no puede cambiar.**

# Tipos de Datos Simples

## Tipos de datos primitivos compuestos (contenedores)

- **Listas (lists):** Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. Se representan con corchetes y los elementos se separan por comas.  
Ejemplo. [1, “dos”, [3, 4], True].
- **Tuplas (tuples).** Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. A diferencia de las listas son inmutables, es decir, que no cambian durante la ejecución. Se representan mediante paréntesis y los elementos se separan por comas.  
Ejemplo. (1, ‘dos’, 3)
- **Diccionarios (dictionaries):** Colecciones de objetos con una clave asociada. Se representan con llaves, los pares separados por comas y cada par contiene una clave y un objeto asociado separados por dos puntos.  
Ejemplo. {‘pi’:3.1416, ‘e’:2.718}.

# Tipos de Datos Simples

## Verificar la clase de un tipo de dato

- La clase a la que pertenece un dato se obtiene con el comando **type()**
- **Números** (clases int y float) : int (1), int (-2) , float (2.3)
- **Operadores aritméticos** : Prioridad : Funciones-predefinidas; Potencias; Productos y cocientes; Sumas y restas.
- **Operadores lógicos con números:**  
== (igual que),  
> (mayor que),  
< (menor que),  
>= (mayor o igual que),  
<= (menor o igual que),  
!= (distinto de)

**Tipado dinámico:** un lenguaje de tipado dinámico es aquel cuyas variables, no requieren ser definidas asignando su tipo de datos, sino que éste, se auto-asigna en tiempo de ejecución, según el valor declarado.

# Tipos de Datos Simples

## Cadenas (clase str)

Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas sencillas ' o dobles "

### Subcadenas :

- `c[i:j:k]` : Devuelve la subcadena de c desde el carácter con el índice i hasta el carácter anterior al índice j, tomando caracteres cada k.

-----

### Operaciones con Cadenas :

- `c1 + c2` : Devuelve la cadena resultado de concatenar las cadenas c1 y c2.
- `c * n` : Devuelve la cadena resultado de concatenar n copias de la cadena c.
- `c1 in c2` : Devuelve True si c1 es una cadena concenida en c2 y False en caso contrario.
- `c1 not in c2` : Devuelve True si c1 es una cadena no concenida en c2 y False en caso contrario.

Cadena	P	y	t	h	o	n
Indice positivo	0	1	2	3	4	5
Indice negativo	-6	-5	-4	-3	-2	-1

### Operaciones de comparación de cadenas :

- `c1 == c2` : Devuelve True si la cadena c1 es igual que la cadena c2 y False en caso contrario.
- `c1 > c2` : Devuelve True si la cadena c1 sucede a la cadena c2 y False en caso contrario.
- `c1 < c2` : Devuelve True si la cadena c1 antecede a la cadena c2 y False en caso contrario.
- `c1 >= c2` : Devuelve True si la cadena c1 sucede o es igual a la cadena c2 y False en caso contrario.
- `c1 <= c2` : Devuelve True si la cadena c1 antecede o es igual a la cadena c2 y False en caso contrario.
- `c1 != c2` : Devuelve True si la cadena c1 es distinta de la cadena c2 y False en caso contrario.

Utilizan el orden establecido en el código ASCII.

# Tipos de Datos Simples

## Funciones de cadenas

- **len(c)** : Devuelve el número de caracteres de la cadena c.
- **min(c)** : Devuelve el carácter menor de la cadena c.
- **max(c)** : Devuelve el carácter mayor de la cadena c.
- **c.upper()** : Devuelve la cadena con los mismos caracteres que la cadena c pero en mayúsculas.
- **c.lower()** : Devuelve la cadena con los mismos caracteres que la cadena c pero en minúsculas.
- **c.title()** : Devuelve la cadena con los mismos caracteres que la cadena c con el primer carácter en mayúsculas y el resto en minúsculas.
- **c.split(delimitador)** : Devuelve la lista formada por las subcadenas que resultan de partir la cadena c usando como delimitador la cadena delimitador. Si no se especifica el delimitador utiliza por defecto el espacio en blanco.

## Cadenas formateadas ( **format()** )

- **c.format(valores)**: Devuelve la cadena c tras sustituir los valores de la secuencia valores en los marcadores de posición de c. Los marcadores de posición se indican mediante llaves {} en la cadena c, y el reemplazo de los valores se puede realizar por posición, indicando en número de orden del valor dentro de las llaves, o por nombre, indicando el nombre del valor, siempre y cuando los valores se pasen con el formato nombre = valor.
- **{:n}** : Alinea el valor a la izquierda rellenando con espacios por la derecha hasta los n caracteres.
- **{:>n}** : Alinea el valor a la derecha rellenando con espacios por la izquierda hasta los n caracteres.
- **{:^n}** : Alinea el valor en el centro rellenando con espacios por la izquierda y por la derecha hasta los n caracteres.
- **{:nd}** : Formatea el valor como un número entero con n caracteres rellenando con espacios blancos por la izquierda.
- **{:n.mf}** : Formatea el valor como un número real con un tamaño de n caracteres (incluido el separador de decimales) y m cifras decimales, rellenando con espacios blancos por la izquierda.

Utilizan el orden establecido en el código ASCII.

# Tipos de Datos Simples

## Datos lógicos o booleanos (clase bool)

- Contiene únicamente dos elementos True y False que representan los valores lógicos verdadero y falso respectivamente.
- False tiene asociado el valor 0 y True tiene asociado el valor 1.

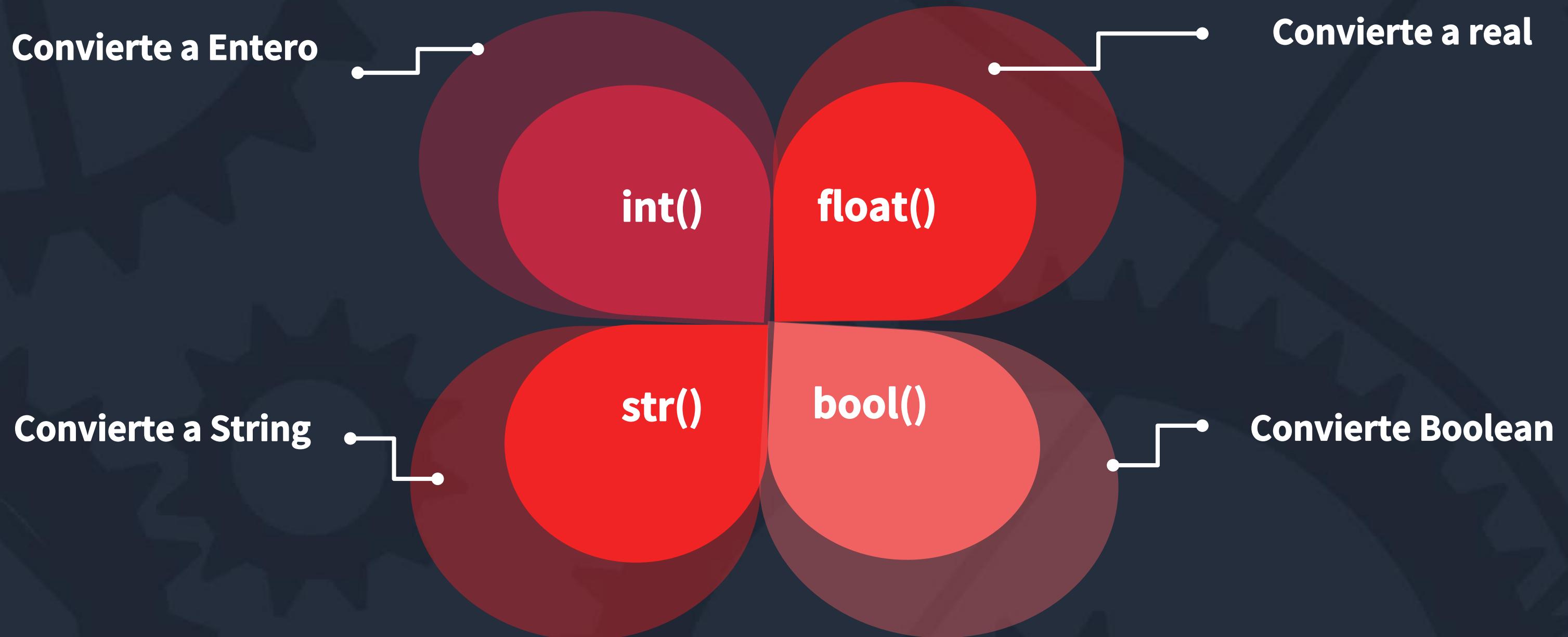
## Operaciones con valores lógicos

- **Operadores lógicos:** == (igual que), > (mayor), < (menor), >= (mayor o igual que), <= (menor o igual que), != (distinto de).
- **not b (negación)** : Devuelve True si el dato booleano b es False , y False en caso contrario.
- **b1 and b2** : Devuelve True si los datos booleanos b1 y b2 son True, y False en caso contrario.
- **b1 or b2** : Devuelve True si alguno de los datos booleanos b1 o b2 son True, y False en caso contrario.

## Tabla de Verdad

x	y	not x	x and y	x or y
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

# Coversiones datos primitivos



# Variables en Python

## Consideraciones :

- Una variable es un espacio para almacenar datos modificables, en la memoria de un ordenador.
- Una variable es un identificador ligado a algún valor. Reglas para nombrarlas:
  - Comienzan siempre por una letra, seguida de otras letras o números.
  - No se pueden utilizarse palabras reservadas del lenguaje.
  - A diferencia de otros lenguajes no tienen asociado un tipo y no es necesario declararlas antes de usarlas (tipado dinámico).
  - Para asignar un valor a una variable se utiliza el operador = y para borrar una variable se utiliza la instrucción **del**.

Buenas prácticas : PEP 8

## Ejemplos :

```
1 lenguaje = 'Python'
2 x = 3.14
3 y = 3 + 2
4 # Asignación múltiple
5 a1, a2 = 1, 2
6 # Intercambio de valores
7 a, b = b, a
8 # Incremento (equivale a x = x + 2)
9 x += 2
10 # Decremento (equivale a x = x - 1)
11 x -= 1
12 # Valor no definido
13 x = None
14 del x
```

### Entrada por terminal:

Se utilizará la función `input()`

### Salida por terminal:

Se utilizará la función `print()`

### Asignación múltiple - Encoding

```
a,b,c = 1,"Hola",True
# -*- coding: utf-8 -*-
```

# Comentarios

## Tipos de comentarios:

Los comentarios pueden ser de dos tipos: de una sola línea o multi-línea y se expresan de la siguiente manera:

```
# Esto es un comentario de una sola línea  
mi_variable = 15
```

```
"""Y este es un comentario de varias  
líneas"""\nmi_variable = 15
```

```
mi_variable = 15 # Este comentario es de  
una línea también
```

### PEP 8: comentarios

Comentarios en la misma línea del código deben separarse con dos espacios en blanco. Luego del símbolo # debe ir un solo espacio en blanco.

Correcto:

a = 15 # Edad de María

Incorrecto:

a = 15 # Edad de María

# PEP 8

### PEP 8: variables

Utilizar nombres descriptivos y en minúsculas. Para nombres compuestos, separar las palabras por guiones bajos. Antes y después del signo =, debe haber uno (y solo un) espacio en blanco

Correcto: mi\_variable = 12

Incorrecto: MiVariable = 12 | mivariable = 12 | mi\_variable=12 | mi\_variable = 12

### PEP 8: constantes

Utilizar nombres descriptivos y en mayúsculas separando palabras por guiones bajos.

Ejemplo: MI\_CONSTANTE = 12

### PEP 8: operadores

Siempre colocar un espacio en blanco, antes y después de un operador

# Estructuras de Control



**Una estructura de control**, es un bloque de código que permite agrupar instrucciones de manera controlada. En este capítulo, hablaremos sobre dos estructuras de control:

- Estructuras de control condicionales
- Estructuras de control iterativas

**¿Qué es la identación?** En un lenguaje informático, la identación es lo que la sangría al lenguaje humano escrito (a nivel formal). Así como para el lenguaje formal, cuando uno redacta una carta, debe respetar ciertas sangrías, los lenguajes informáticos, requieren una identación.

## PEP 8: identación

Una identación de **4 (cuatro) espacios en blanco**, indicará que las instrucciones identadas, forman parte de una misma estructura de control.

## Fundamentos estructuras de control

## if - elif - else

- Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje:  
**if (si)**, **elif (sino, si)** y **else (sino)**.
  - La evaluación de condiciones, solo puede arrojar 1 de 2 resultados: verdadero o falso (True o False).
  - Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** (o de comparación).
  - Y para evaluar más de una condición simultáneamente, se utilizan **operadores lógicos**.
- (\*) 1 indica resultado verdadero de la condición, mientras que 0, indica falso.

# while - for

- Sentencias de control iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.
- En Python se dispone de dos estructuras cíclicas:
  - El bucle while
  - El bucle for

## while

- Este bucle, se encarga de ejecutar una misma acción “mientras que” una determinada condición se cumpla o sea verdadera.

**while True:**  
**statement(s)**

## for

- El bucle for, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla:

**for e in range(start,stop):**  
**print(e)**

**for element in list:**  
**condiciones**

# Funciones en Python

Una función, es la forma de agrupar expresiones y sentencias (algoritmos) que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas.

## Características :

- Se realiza mediante la instrucción **def** más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre.
- La definición de la función finaliza con dos puntos (**:**) y el algoritmo que la compone, irá **identado con 4 espacios**

## Ejemplo:

```
def mi_funcion():
    # aquí el algoritmo
```

```
def mi_funcion(argumentos):
    # aquí el algoritmo
```

```
# Asignación de valores desde funciones
frase = mi_funcion()
print frase
```

## Características de las funciones



# Funciones en Python

## Parámetros:

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de variables de ámbito local.

Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder.

**Nota :** En la definición de una función los valores que se reciben **se denominan parámetros**, pero durante la llamada los valores que se envían **se denominan argumentos**.

**Los argumentos se pueden indicar de la siguiente forma:**

- 1. Argumentos posicionales:** Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.
- 2. Argumentos por nombre:** Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma `parametro = argumento`.
- 3. Argumento por Defecto:** Se puede asignar a cada parámetro un argumento por defecto.

### Código Resultado

```
def resta(a, b):
    return a - b

resta(30, 10) # argumento 30 => posición 0 => parámetro a
                # argumento 10 => posición 1 => parámetro b
```

### Código Resultado

```
resta(b=30, a=10)
```

### Código Resultado

```
def resta(a=None, b=None):
    if a == None or b == None:
        print("Error, debes enviar dos números a la función")
        return # indicamos el final de la función aunque no devuelva nada
    return a-b

resta()
```

## PEP 8: Funciones

A la definición de una función la deben anteceder dos líneas en blanco.

Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo `=`.

# Funciones en Python

1. **Argumentos indeterminados:** Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.

## Características :

- Es posible pasar un número variable de argumentos a un parámetro. Esto se puede hacer de **dos formas**.
- **\*parametro:** Se antepone un asterisco al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos separados por comas. Los argumentos **se guardan en una tupla** que se asocia al parámetro.
- **\*\*parametro:** Se anteponen dos asteriscos al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos por pares nombre = valor, separados por comas. Los argumentos **se guardan en un diccionario** que se asocia al parámetro.

### Código      Resultado

```
def indeterminados_posicion(*args):  
    for arg in args:  
        print(arg)  
  
indeterminados_posicion(5, "Hola", [1,2,3,4,5])
```

### Código      Resultado

```
def indeterminados_nombre(**kwargs):  
    print(kwargs)  
  
indeterminados_nombre(n=5, c="Hola", l=[1,2,3,4,5])
```

# Funciones en Python

## 1. Ámbito de los parámetros y variables de una función.

- Los parámetros y las variables declaradas dentro de una función son de **ámbito local**, mientras que las definidas fuera de ella son de **ámbito global**.
- Tanto los parámetros como las variables del ámbito local de una función sólo están accesibles durante la ejecución de la función.
- Si en el ámbito local de una función existe una variable que también existe en el ámbito global, durante la ejecución de la función la variable global queda eclipsada por la variable local y no es accesible hasta que finaliza la ejecución de la función.

## 2. Documentación de funciones

```
def saludo(nombre):
    """Esta función realiza un saludo por consola

    Args:
        nombre (str): Nombre de la persona
    """
    lenguaje = "Django"

    print("Hola ", nombre, "estas programando en:",lenguaje)
```

## 3. Sobre la finalidad de las funciones

*Una buena práctica, indica que la finalidad de una función, debe ser realizar una única acción, reutilizable y por lo tanto, tan genérica como sea posible.*

# Ficheros con Python

Al utilizar ficheros para guardar los datos estos perdurarán tras la ejecución del programa, pudiendo ser consultados o utilizados más tarde.

Las operaciones más habituales con ficheros son:

- Crear un fichero.
- Escribir datos en un fichero.
- Leer datos de un fichero.
- Borrar un fichero.

## 1.- Creación o escritura :

**open(ruta, 'w')** : Crea el fichero con la ruta ruta, lo abre en modo escritura (el argumento 'w' significa write) y devuelve un objeto que lo referencia.

```
from io import open

texto = "Una línea con texto\nOtra línea con texto"

# Ruta donde crearemos el fichero, w indica escritura
fichero = open('fichero.txt', 'w')

# Escribimos el texto
fichero.write(texto)

# Cerramos el fichero
fichero.close()
```

# Ficheros con Python

## 1.- Lectura : Lectura total del archivo : **read()**

Código Resultado

```
from io import open

# Ruta donde leeremos el fichero, r indica lectura (por defecto ya es r)
fichero = open('fichero.txt','r')

# Lectura completa
texto = fichero.read()

# Cerramos el fichero
fichero.close()

print(texto)
```

- Podemos usar el método **readlines()** del fichero para generar una lista con las líneas:

Código Resultado

```
from io import open
fichero = open('fichero.txt','r')

# Leemos creando una lista de líneas
texto = fichero.readlines()

fichero.close()
print(texto)
```

- También se puede leer un fichero utilizando la instrucción estándar **with**, la misma que nos ayudará a gestionar los recursos del objeto de manera automática.

Código Resultado

```
with open("fichero.txt", "r") as fichero:
    for linea in fichero:
        print(linea)
```

## 2.- Escritura (**append**) : Escritura de un archivo, al final del mismo.

Código Resultado

```
from io import open

# Ruta donde leeremos el fichero, a indica extensión (puntero al final)
fichero = open('fichero.txt','a')

fichero.write('\nOtra línea más abajo del todo')
fichero.close()
```

# Ficheros con Python

- El método `seek()`:
- Es posible posicionar el puntero en el fichero manualmente e indicando un número de caracteres para luego leer una cantidad de caracteres con el método `read`:

Código	Resultado
<pre>fichero = open('fichero.txt','r') fichero.seek(0)    # Puntero al principio fichero.read(10)   # Leemos 10 caracteres</pre>	

# Tipos de Datos Estructurados

1. **Listas** : Las listas se tratan de un tipo compuesto de dato que puede almacenar distintos valores (llamados ítems o elementos) ordenados entre [ ] y separados con comas.

## Se caracterizan por:

- Tienen orden.
- Pueden contener elementos de distintos tipos.
- Son mutables, es decir, pueden alterarse durante la ejecución de un programa.
- **función list(c)** : Crea una lista con los elementos de la secuencia o colección c.
- **Acceso al los elementos l[i]** : Devuelve el elemento de la lista l con el índice i.

2.- **SubListas** : l [i:j:k] : Devuelve la sublista desde el elemento de l con el índice i hasta el elemento anterior al índice j, tomando elementos cada k.

## 3.- Operaciones que no modifican las listas:

- **len(l)** : Devuelve el número de elementos de la lista l.
- **min(l)** : Devuelve el mínimo elemento de la lista l siempre que los datos sean comparables.
- **max(l)** : Devuelve el máximo elemento de la lista l siempre que los datos sean comparables.
- **sum(l)** : Devuelve la suma de los elementos de la lista l, siempre que los datos se puedan sumar.
- **dato in l** : Devuelve True si el dato dato pertenece a la lista l y False en caso contrario.
- **l.index(dato)** : Devuelve la posición que ocupa en la lista l el primer elemento con valor dato.
- **l.count(dato)** : Devuelve el número de veces que el valor dato está contenido en la lista l.
- **all(l)** : Devuelve True si todos los elementos de la lista l son True y False en caso contrario.
- **any(l)** : Devuelve True si algún elemento de la lista l es True y False en caso contrario.

# Listas en Python

## 4.- Operaciones que modifican las listas:

- **I1 + I2 :** Crea una nueva lista concatenan los elementos de la listas I1 y I2.
- **I.append(dato) :** Añade dato al final de la lista I.
- **I.extend(sequencia) :** Añade los datos de sequencia al final de la lista I.
- **I.insert(índice, dato) :** Inserta dato en la posición índice de la lista I y desplaza los elementos una posición a partir de la posición índice.
- **I.remove(dato) :** Elimina el primer elemento con valor dato en la lista I y desplaza los que están por detrás de él una posición hacia delante.
- **I.pop([índice]) :** Devuelve el dato en la posición índice y lo elimina de la lista I, desplazando los elementos por detrás de él una posición hacia delante.
- **I.sort() :** Ordena los elementos de la lista I de acuerdo al orden predefinido, siempre que los elementos sean comparables.
- **I.reverse() :** invierte el orden de los elementos de la lista I.

## 5.- Copia de listas :

Existen dos formas de copiar listas:

- **Copia por referencia I1 = I2:** Asocia la variable I1 la misma lista que tiene asociada la variable I2, es decir, ambas variables apuntan a la misma dirección de memoria. Cualquier cambio que hagamos a través de I1 o I2 afectará a la misma lista.
- **Copia por valor I1 = list(I2):** Crea una copia de la lista asociada a I2 en una dirección de memoria diferente y se la asocia a I1. Las variables apuntan a direcciones de memoria diferentes que contienen los mismos datos. Cualquier cambio que hagamos a través de I1 no afectará a la lista de I2 y viceversa.

# Tuplas en Python

**Tuplas :** Son unas colecciones muy parecidas a las listas con la peculiaridad de que son **inmutables**.

**Se caracterizan por:**

- Tienen orden.
- Pueden contener elementos de distintos tipos.
- Son inmutables, es decir, no pueden alterarse durante la ejecución de un programa.
- Otra forma de crear tuplas es mediante la función tuple().
- tuple(c) : Crea una tupla con los elementos de la secuencia o colección c.
- Se pueden indicar los elementos separados por comas, mediante una cadena, o mediante una colección de elementos iterable.

Código	Resultado
<pre>tupla = (100, "Hola", [1,2,3], -50) tupla</pre>	

- El acceso a los elementos de una tupla se realiza del mismo modo que en las listas. También se pueden obtener subtuplas de la misma manera que las sublistas.
- Las operaciones de listas que no modifican la lista también son aplicables a las tuplas.

Código	Resultado
<pre>print(tupla) print(tupla[0]) print(tupla[-1]) print(tupla[2:]) print(tupla[2][-1])</pre>	

# Diccionarios en Python

**Diccionarios :** Es una colección de pares formados por una **clave** y un **valor** asociado a la clave.

**Se caracterizan por:**

- Se construyen poniendo los pares entre llaves { } separados por comas, y separando la clave del valor con dos puntos ":"
- No tienen orden.
- Pueden contener elementos de distintos tipos.
- Son mutables, es decir, pueden alterarse durante la ejecución de un programa.
- Las claves son únicas, es decir, no pueden repetirse en un mismo diccionario, y pueden ser de cualquier tipo de datos inmutable

## Acceso a los elementos de un diccionario

- **d[clave]** : Devuelve el valor del diccionario d asociado a la clave. Si en el diccionario no existe esa clave devuelve un error.
- **d.get(clave, valor)**: Devuelve el valor del diccionario d asociado a la clave. Si en el diccionario no existe esa clave devuelve valor, y si no se especifica un valor por defecto devuelve None.

### Código

```
colores = {'amarillo':'yellow', 'azul':'blue'}  
colores
```

### Resultado

# Diccionarios en Python

**Diccionarios :** Es una colección de pares formados por una **clave** y un **valor** asociado a la clave.

## 1.- Operaciones que no modifican un diccionario:

- **len(d)** : Devuelve el número de elementos del diccionario d.
- **min(d)** : Devuelve la mínima clave del diccionario d siempre que las claves sean comparables.
- **max(d)** : Devuelve la máxima clave del diccionario d siempre que las claves sean comparables.
- **sum(d)** : Devuelve la suma de las claves del diccionario d, siempre que las claves se puedan sumar.
- **clave in d** : Devuelve True si la clave pertenece al diccionario d y False en caso contrario.
- **d.keys()** : Devuelve un iterador sobre las claves de un diccionario.
- **d.values()** : Devuelve un iterador sobre los valores de un diccionario.
- **d.items()** : Devuelve un iterador sobre los pares clave-valor de un diccionario.

## 2.- Operaciones que modifican un diccionario:

- **d[clave] = valor** : Añade al diccionario d el par formado por la clave y el valor.
- **d.update(d2)**: Añade los pares del diccionario d2 al diccionario d.
- **d.pop(clave, alternativo)** : Devuelve el valor asociado a la clave del diccionario d y lo elimina del diccionario. Si la clave no está devuelve el valor alternativo.
- **d.popitem()** : Devuelve la tupla formada por la clave y el valor del último par añadido al diccionario d y lo elimina del diccionario.
- **del d[clave]** : Elimina del diccionario d el par con la clave.
- **d.clear()** : Elimina todos los pares del diccionario d de manera que se queda vacío.

```
personajes = []

gandalf = {'Nombre': 'Gandalf', 'Clase': 'Mago', 'Raza': 'Humano'}
legolas = {'Nombre': 'Legolas', 'Clase': 'Arquero', 'Raza': 'Elfo'}
gimli = {'Nombre': 'Gimli', 'Clase': 'Guerrero', 'Raza': 'Enano'}
```

# Control de Excepciones

7

## Errores y Excepciones

**Errores :** Los errores detienen la ejecución del programa y tienen varias causas.

### Errores de sintaxis:

- Identificados con el código SyntaxError, son los que podemos apreciar repasando el código, por ejemplo al dejarnos de cerrar un paréntesis:

### Errores de nombre:

- Se producen cuando el sistema interpreta que debe ejecutar alguna función, método... pero no lo encuentra definido. Devuelven el código NameError:

### Errores semánticos:

- Estos errores son muy difíciles de identificar porque van ligados al sentido del funcionamiento y dependen de la situación. Algunas veces pueden ocurrir y otras no.
- La mejor forma de prevenirlos es programando mucho y aprendiendo de tus propios fallos, la experiencia es la clave.



# Control de Excepciones

**1.- Excepciones:** Las excepciones son bloques de código que nos permiten continuar con la ejecución de un programa pese a que ocurra un error.

## 2.- Bloques try - except:

- Para prevenir el fallo debemos poner el código propenso a errores en un bloque try y luego encadenar un bloque except para tratar la situación excepcional mostrando que ha ocurrido un fallo.
- Nos permite controlar situaciones excepcionales que generalmente darían error y en su lugar mostrar un mensaje o ejecutar una pieza de código alternativo.

## 3.- Bloque else:

- Es posible encadenar un bloque else después del except para comprobar el caso en que todo funcione correctamente (no se ejecuta la excepción).
- El bloque else es un buen momento para romper la iteración con break si todo funciona correctamente.

```
while(True):
    try:
        n = float(input("Introduce un número: "))
        m = 4
        print("{} / {} = {}".format(n,m,n/m))
    except:
        print("Ha ocurrido un error, introduce bien el número")
    else:
        print("Todo ha funcionado correctamente")
        break # Importante romper la iteración si todo ha salido bien
```

# Control de Excepciones

## 5.- Bloque finally:

Por último es posible utilizar un bloque finally que se ejecute al final del código, ocurra o no ocurra un error.

## 6.- Excepciones Múltiples:

- En una misma pieza de código pueden ocurrir muchos errores distintos y quizás nos interese actuar de forma diferente en cada caso.
- Una buena práctica es asignar una excepción a una variable y de esta forma es posible analizar el tipo de error que sucede gracias a su identificador.

```
try:  
    n = input("Introduce un número: ") # no transformamos a número  
    5/n  
except Exception as e: # guardamos la excepción como una variable e  
    print("Ha ocurrido un error =>", type(e).__name__)
```

```
try:  
    n = float(input("Introduce un número divisor: "))  
    5/n  
except TypeError:  
    print("No se puede dividir el número entre una cadena")  
except ValueError:  
    print("Debes introducir una cadena que sea un número")  
except ZeroDivisionError:  
    print("No se puede dividir por cero, prueba otro número")  
except Exception as e:  
    print("Ha ocurrido un error no previsto", type(e).__name__ )
```

# Control de Excepciones

## 7.- Invocación de excepciones

En algunas ocasiones quizá nos interesa llamar un error manualmente, ya que un print común no es muy elegante:

- **Instrucción raise:**
  - Gracias a **raise** podemos lanzar un error manual pasándole el identificador. Luego simplemente podemos añadir un **except** para tratar esta excepción que hemos lanzado

```
def mi_funcion(algo=None):
    try:
        if algo is None:
            raise ValueError("Error! No se permite un valor nulo")
    except ValueError:
        print("Error! No se permite un valor nulo (desde la excepción)")

mi_funcion()
```

# Librería DataTime



## DataTime :

- Se suele utilizar la librería datetime que incorpora los tipos de datos **date**, **time** y **datetime** para representar fechas y funciones para manejarlas.
- Algunas de las **operaciones más habituales** que permite son:
  - Acceder a los distintos componentes de una fecha (año, mes, día, hora, minutos, segundos y micro segundos).
  - Convertir cadenas con formato de fecha en los tipos date, time o datetime.
  - Convertir fechas de los tipos date, time o datetime en cadenas formateadas de acuerdo a diferentes formatos de fechas.
  - Hacer aritmética de fechas (sumar o restar fechas).
  - Comparar fechas.

## Los tipos de datos date, time y datetime:

- **date(año, mes, dia)** : Devuelve un objeto de tipo date que representa la fecha con el año, mes y dia indicados.
- **time(hora, minutos, segundos, microsegundos)** : Devuelve un objeto de tipo time que representa un tiempo la hora, minutos, segundos y microsegundos indicados.
- **datetime(año, mes, dia, hora, minutos, segundos, microsegundos)** : Devuelve un objeto de tipo datetime que representa una fecha y hora con el año, mes, dia, hora, minutos, segundos y microsegundos indicados.

# Librería DateTime

## Acceso a los componentes de una fecha:

- date.today()** : Devuelve un objeto del tipo date la fecha del sistema en el momento en el que se ejecuta.
- datetime.now()**: Devuelve un objeto del tipo datetime con la fecha y la hora del sistema en el momento exacto en el que se ejecuta.
- d.year** : Devuelve el año de la fecha d, puede ser del tipo date o datetime.
- d.month** : Devuelve el mes de la fecha d, que puede ser del tipo date o datetime.
- d.day** : Devuelve el día de la fecha d, que puede ser del tipo date o datetime.
- d.weekday()** : Devuelve el día de la semana de la fecha d, que puede ser del tipo date o datetime.
- t.hour** : Devuelve las horas del tiempo t, que puede ser del tipo time o datetime.
- t.minutes** : Devuelve los minutos del tiempo t, que puede ser del tipo time o datetime.
- t.second** : Devuelve los segundos del tiempo t, que puede ser del tipo time o datetime.
- t.microsecond** : Devuelve los microsegundos del tiempo t, que puede ser del tipo time o datetime.

## Conversión de fechas en cadenas con diferentes formatos:

- d.strftime(formato)** : Devuelve la cadena que resulta de transformar la fecha d con el formato indicado en la cadena formato. La cadena formato puede contener los siguientes marcadores de posición:  
%Y (año completo), %y (últimos dos dígitos del año), %m (mes en número),  
%B (mes en palabra), %d (día), %A (día de la semana),  
%a (día de la semana abreviado), %H (hora en formato 24 horas), %I (hora en formato 12 horas),  
%M (minutos), %S (segundos), %p (AM o PM),  
%C (fecha y hora completas), %x (fecha completa), %X (hora completa).

### Conversión de cadenas en fechas:

- **strptime(s, formato)** : Devuelve el objeto de tipo date, time o datetime que resulta de convertir la cadena s de acuerdo al formato indicado en la cadena formato. La cadena formato puede contener los siguientes marcadores de posición:

%Y (año completo), %y (últimos dos dígitos del año), %m (mes en número),  
%B (mes en palabra), %d (día), %A (día de la semana), %a (día de la semana abreviado),  
%H (hora en formato 24 horas), %I (hora en formato 12 horas), %M (minutos),  
%S (segundos), %p (AM o PM), %C (fecha y hora completas),  
%x (fecha completa), %X (hora completa).

### Aritmética de fechas:

Para representar el tiempo transcurrido entre dos fechas se utiliza el tipo timedelta.

- **timedelta(dias, segundos, microsegundos)** : Devuelve un objeto del tipo timedelta que representa un intervalo de tiempo con los dias, segundos y micorsegundos indicados.
- **d1 - d2** : Devuelve un objeto del tipo timedelta que representa el tiempo transcurrido entre las fechas d1 y d2 del tipo datetime.
- **d + delta** : Devuelve la fecha del tipo datetime que resulta de sumar a la fecha d el intervalo de tiempo delta, donde delta es del tipo timedelta.

# Programación Orientada a Objetos

40



- En Python todo es un **objeto**. Cuando creas una variable y le asignas un valor entero, ese valor es un objeto; una función es un objeto; las listas, tuplas, diccionarios, conjuntos, ... son objetos; una cadena de caracteres es un objeto.
- Pero, **¿por qué es tan importante la programación orientada a objetos?**
  - Este tipo de programación introduce un nuevo paradigma que nos permite encapsular y aislar datos y operaciones que se pueden realizar sobre dichos datos.

## Clases y objetos en Python:

- Básicamente, una clase es una entidad que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).
- Por su parte, un objeto es una creación o instancia de una clase.





### Constructor de una clase en Python: `__init__()`

- Para crear un objeto de una clase determinada, es decir, instanciar una clase, se usa el nombre de la clase y a continuación se añaden paréntesis (como si se llamara a una función).
- Resumiendo: los objetos son instancias de una clase.

```
class Coche:  
    def __init__(self, color, aceleracion):  
        self.color = color  
        self.aceleracion = aceleracion  
        self.velocidad = 0  
  
    def acelera(self):  
        self.velocidad = self.velocidad + self.aceleracion  
  
c1 = Coche("rojo",60)  
c2 = Coche("azul",10)
```

💡 **NOTA:** Es una convención utilizar la notación CamelCase para los nombres de las clases. Esto es, la primera letra de cada palabra del nombre está en mayúsculas y el resto de letras se mantienen en minúsculas.

# Programación Orientada a Objetos

## Atributos

### Atributos:

- El potencial de la POO esa es la capacidad de definir variables y funciones dentro de las clases, aunque aquí se conocen como atributos y métodos respectivamente.
- **Atributos:**
  - A efectos prácticos los atributos no son muy distintos de las variables, la diferencia fundamental es que sólo existen dentro del objeto.
- **Atributos dinámicos:**
  - Dado que Python es muy flexible los atributos pueden manejarse de distintas formas, por ejemplo se pueden crear dinámicamente (al vuelo) en los objetos.
- **Atributos de clase:**
  - Aunque la flexibilidad de los atributos dinámicos puede llegar a ser muy útil, tener que definir los atributos de esa forma es tedioso. Es más práctico definir unos atributos básicos en la clase.

```
class Galleta:
    pass

galleta = Galleta()
galleta.sabor = "salado"
galleta.color = "marrón"

print(f"El sabor de esta galleta es {galleta.sabor} "
      f"y el color {galleta.color}")
```

```
class Galleta:
    chocolate = False

galleta = Galleta()

if galleta.chocolate:
    print("La galleta tiene chocolate")
else:
    print("La galleta no tiene chocolate")
```

# Programación Orientada a Objetos

## Métodos:

- Si por un lado tenemos las "**variables**" de las clases, por otro tenemos sus "**funciones**", que evidentemente nos permiten definir funcionalidades para llamarlas desde las instancias.
- Definir un método es bastante simple, sólo tenemos que añadirlo en la clase y luego llamarlo desde el objeto con los paréntesis, como si de una función se tratase.
- Lo que tenemos 2 tipos de métodos en las clases: **Métodos de Clase y Métodos de Instancia**.

```
class Galleta:
    chocolate = False

    def saludar():
        print("Hola, soy una galleta muy sabrosa")

Galleta.saludar()
```

```
class Galleta:
    chocolate = False

    def chocolatear(self):
        chocolate = True

galleta = Galleta()
galleta.chocolatear()
print(galleta.chocolate)
```

## Métodos especiales:

- Se llaman especiales porque la mayoría ya existen de forma oculta y sirven para tareas específicas.
- **Constructor :**
  - El constructor es un método que se llama automáticamente al crear un objeto, se define con el nombre `init`.
  - La finalidad del constructor es, como su nombre indica, construir los objetos.

```
class Galleta:
    chocolate = False

    def __init__(self, sabor, color):
        self.sabor = sabor
        self.color = color
        print(f"Se acaba de crear una galleta {self.color} y {self.sabor}. ")

galleta_1 = Galleta("marrón", "amarga")
galleta_2 = Galleta("blanca", "dulce")
```



# Programación Orientada a Objetos

## Destructor:

- Si existe un **constructor** también debe existir un **destructor** que se llame al eliminar el objeto para que encargue de las tareas de limpieza como vaciar la memoria. Ese es el papel del método especial **del**.
- Todos los objetos se borran automáticamente de la memoria al finalizar el programa, aunque también podemos eliminarlos automáticamente pasándolos a la función `del()`:
- En este punto vale comentar algo respecto a los métodos especiales como éste, y es que pese a que tienen accesores en forma de función para facilitar su llamada, es totalmente posible ejecutarlos directamente como si fueran métodos normales:

```
class Galleta:
    def __del__(self):
        print("La galleta se está borrando de la memoria")
galleta = Galleta()
del(galleta)
```

```
class Galleta:
    def __del__(self):
        print("La galleta se está borrando de la memoria")
galleta = Galleta()
galleta.__del__()
```

## String():

- El método `str` es el que devuelve la representación de un objeto en forma de cadena. Un momento en que se llama automáticamente es cuando imprimimos una variable por pantalla.
- Por defecto los objetos imprimen su clase y una dirección de memoria, pero eso puede cambiarse sobreescribiendo el comportamiento:

```
class Galleta:
    def __init__(self, sabor, color):
        self.sabor = sabor
        self.color = color
    def __str__(self):
        return f"Soy una galleta {self.color} y {self.sabor}."
galleta = Galleta("dulce", "blanca")
print(galleta)
print(str(galleta))
print(galleta.__str__())
```



# Programación Orientada a Objetos

## Length():

- Finalmente otro método especial interesante es el que devuelve la longitud. Normalmente está ligado a colecciones, pero nada impide definirlo en una clase. Y sí, digo definirlo y no redefinirlo porque por defecto no existe en los objetos aunque sea el que se ejecuta al pasarlos a la función `len()`.

```
class Cancion:  
  
    def __init__(self, autor, titulo, duracion): # en segundos  
        self.duracion = duracion  
  
    def __len__(self):  
        return self.duracion  
  
cancion = Cancion("Queen", "Don't Stop Me Now", 210)  
  
print(len(cancion))  
print(cancion.__len__())
```

## Objetos dentro de objetos:

- Hasta ahora no lo hemos comentado, pero al ser las clases un nuevo tipo de dato resulta más que obvio que se pueden poner en colecciones e incluso utilizarlos dentro de otras clases.
- Voy a dejar un pequeño código de ejemplo sobre un catálogo de películas para que lo estudies detenidamente. Pag 46

# Programación Orientada a Objetos

## Objetos de Objetos :

- Analiza el siguiente ejemplo :

```
class Pelicula:  
  
    # Constructor de clase  
    def __init__(self, titulo, duracion, lanzamiento):  
        self.titulo = titulo  
        self.duracion = duracion  
        self.lanzamiento = lanzamiento  
        print('Se ha creado la pelicula:', self.titulo)  
  
    def __str__(self):  
        return '{} ({})'.format(self.titulo, self.lanzamiento)  
  
class Catalogo:  
  
    peliculas = [] # Esta lista contendrá objetos de la clase Pelicula  
  
    def __init__(self, peliculas=[]):  
        self.peliculas = peliculas  
  
    def agregar(self, p): # p será un objeto Pelicula  
        self.peliculas.append(p)  
  
    def mostrar(self):  
        for p in self.peliculas:  
            print(p) # Print toma por defecto str(p)  
  
p = Pelicula("El Padrino", 175, 1972)  
c = Catalogo([p]) # Añado una lista con una pelicula desde el principio  
c.mostrar()  
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974)) # Añadimos otra  
c.mostrar()
```

# Programación Orientada a Objetos

## Encapsulación:

- Finalmente para acabar la introducción vale la pena comentar esta "técnica".
- La encapsulación consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior.
- En Python no existe, pero se puede simular **precediendo atributos y métodos con dos barras bajas \_\_ como indicando que son "especiales"**.

**En el caso de los atributos quedarían así:**

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

e = Ejemplo()
print(e.__atributo_privado)
```

**En el caso de los métodos quedarían así:**

```
class Ejemplo:
    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

e = Ejemplo()
e.__metodo_privado()
```

- Sea como sea para acceder a esos datos se deberían crear métodos públicos que hagan de interfaz.
- En otros lenguajes les llamaríamos **getters y setters** y es lo que da lugar a las propiedades, que no son más que atributos protegidos con interfaces de acceso.

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()

e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```



# 10

- **Interfaces gráficas con Tkinter:**
  - Las interfaces gráficas son medios visuales, mucho más cómodos que una terminal de texto, a través de las cuales nuestros usuarios pueden interactuar y realizar tareas gráficamente.
- Widgets:**
- El módulo Tkinter cuenta con una serie de componentes gráficos llamados Widgets, gracias a los cuales podemos diseñar nuestras interfaces.
  - Los widgets deben seguir una jerarquía a la hora deadirse a la interfaz. Por ejemplo, un Marco (frame) forma parte del objeto raíz Tk. Y a su vez, un botón (button) puede formar parte de un contenedor como la raíz o un marco.
- **Los que veremos en esta introducción a Tkinter son:**
- **Tk:** Contenedor base o raíz de todos los widgets que forman la interfaz. No tiene tamaño propio sino que se adapta a los widgets que contiene.
  - **Frame:** Marco contenedor de otros widgets. Puede tener tamaño propio y posicionarse en distintos lugares de otro contenedor (ya sea la raíz u otro marco).
  - **Label:** Etiqueta donde podemos mostrar algún texto estático.
  - **Entry:** Campo de texto sencillo para escribir texto corto. Nombres, apellidos, números..
  - **Text:** Campo de texto multilínea para escribir texto largo. Descripciones, comentarios...
  - **Button:** Botón con un texto sobre el cual el usuario puede hacer clic.
  - **Radiobutton:** Botón radial que se usa en conjunto donde es posible marcar una opción.
  - **Checkbutton:** Botón cuadrado que se puede marcar con un tic.
  - **Menu:** Estructura de botones centrados en la composición de menús superiores.
  - **Dialogs:** Ventanas emergentes que permiten desde mostrar información al usuario (típico mensaje de alerta o de confirmación) hasta ofrecer una forma gráfica de interactuar con el sistema operativo (seleccionar un fichero de un directorio para abrirlo).
- **Hay otros widgets, pero estos son los más importantes.**

# Manejo de Base de Datos : SQLite

## Lenguaje SQL

11

- **Bases de datos SQLite:**
  - ¿Qué es una base de datos? También conocidas como bancos de datos son simplemente conjuntos de información. Ya conocemos algunos tipos de datos, como los números, las cadenas de caracteres, las fechas, etc.
  - Como son programas complejos centrados en la gestión de información, reciben el nombre de SGBD: **Sistemas Gestores de Bases de Datos**.
- **Modelos:**
  - **Jerárquicas:** Utilizan un modelo los datos que se organiza en forma de árbol invertido, en donde un nodo padre de información puede tener varios hijos...
  - **De red:** Una mejora del modelo jerárquico que permite a un hijo tener varios padres...
  - **Transaccionales:** Cuyo único fin es el envío y recepción de datos a grandes velocidades, estas bases son muy poco comunes
  - **Relacionales:** Éste es el modelo utilizado en la actualidad para representar problemas reales y administrar datos dinámicamente. Es en el que nos vamos a centrar, pero hay otros...
  - **Documentales:** Permiten guardar texto completo, y en líneas generales realizar búsquedas más potentes. Sirven para almacenar grandes volúmenes de información de antecedentes históricos. Junto a las relacionales son de las más utilizadas en el desarrollo web.
  - **Orientadas a objetos:** Este modelo es bastante reciente y propio de los modelos informáticos orientados a objetos, donde se trata de almacenar en la base de datos los objetos completos. Es posible que tome más importancia en el futuro.
  - **Deductivas:** Son bases de datos que permiten hacer deducciones. Se basan principalmente en reglas y hechos que son almacenados en la base de datos, por lo que son algo complejas.

# THANKS!

 Por: Ing. Darwin Calle  
 dacl010811@gmail.com