💻Clase 09 - Funciones

📘 Funciones

La función de Python es un bloque de código que realiza una tarea específica y bien definida.

Las dos ventajas principales de las funciones son:

- (a) Nos ayudan a dividir nuestro programa en múltiples tareas. Para cada tarea podemos definir una función. Esto hace que el código sea modular.
- (b) Las funciones proporcionan un mecanismo de reutilización. La misma función se puede llamar cualquier número de veces.

🛠️ Tipos de funciones en Python

Hay dos tipos de funciones de Python:

- (a) Funciones integradas – Ej. `len()`, `sorted()`, `min()`, `max()`, entre otros.
- (b) Funciones definidas por el usuario.

A continuación, se muestra un ejemplo de función definida por el usuario.

🔧 Definición de una función

```
# Definición de una función
def func():
    print('Mis opiniones podrían cambiar')
    print('Mis opiniones no cambian')
```

▶️ Primera llamada

```
func()  # Primera llamada
```

```
Mis opiniones podrían cambiar
Mis opiniones no cambian
```

▶️ Segunda llamada

```
func()  # Segunda llamada
```

```
Mis opiniones podrían cambiar
Mis opiniones no cambian
```

🧠 Convención de Python para nombres de funciones

- Utilice siempre caracteres en minúsculas.
- Conecte varias palabras usando `_`. Ejemplo: `cal_si()`, `split_data()`, etc.
- Una función se puede redefinir. Al llamar a la función, se llamará a su **última definición**.
- Las definiciones de funciones se pueden anidar. Cuando lo hacemos, la función interna puede acceder a las variables de la función externa. La función externa debe llamarse para que se ejecute la función interna.

🧩 Ejemplo de funciones anidadas

```
def fun1():
    print('Entré a la función 1')
    def fun2():  # función anidada
        print('Entré a la segunda función')
    print('Sigo dentro en la función 1')
    fun2()
```

```
fun1()
```

```
Entré a la función 1
Sigo dentro en la función 1
Entré a la segunda función
```

`fun2` se ejecuta siempre y cuando se *invoque*:

🧪 Ejemplo de función anidada sin invocar la función interna

```
def fun1():
    print('Entré a la función 1')
    def fun2():  # función anidada
        print('Entré a la segunda función')
    print('Sigo dentro en la función 1')
    # fun2()
```

```
fun1()
```

```
Entré a la función 1
Sigo dentro en la función 1
```

🚫 No se puede invocar una función anidada desde fuera

```
fun2()  # No puedo invocar funciones 'interiores' desde aquí
```

```
NameError: name 'fun2' is not defined
```

🔍 Verificando el tipo de `fun1`

```
print(type(fun1))
```

```
<class 'function'>
```

🔄 Comunicación con Funciones

> 💡 La comunicación con las funciones se realiza utilizando parámetros/argumentos que se le pasan y los valores que esta devuelve.

La forma de pasar valores a una función y devolver el valor de ella se muestra a continuación:

```
def calcular_suma(x, y, z):
    return x + y + z

# pasar 10, 20, 30 a calcular_suma(), luego recopilar el valor devuelto por él
s1 = calcular_suma(10, 20, 30)
```

```
print(s1)
```

Salida:

```
60
```

🧪 Pasando variables como argumentos a una función

```
def calcular_suma(x, y, z):
    return x + y + z

# pasar a, b, c a calcular_suma(), recopilar el valor devuelto por él:
a, b, c = 1, 2, 3
s2 = calcular_suma(a, b, c)
print(s2)
```

Salida

```
6
```

⚠️ No olvidar escribir `return` :

```
def calcular_suma(x, y, z):
    x + y + z

# pasar a, b, c a calcular_suma(), recopilar el valor devuelto por él:
a, b, c = 1, 2, 3
s2 = calcular_suma(a, b, c)
print(s2)
```

Salida:

```
None
```

⚠️ Asignar una variable no es suficiente si no se retorna

```
def calcular_suma(x, y, z):
    w = x + y + z

# pasar a, b, c a calcular_suma(), recopilar el valor devuelto por él:
a, b, c = 1, 2, 3
s2 = calcular_suma(a, b, c)
print(s2)
```

```
None
```

📦 Variables locales no son accesibles fuera de la función

```
def calcular_suma(x, y, z):
    w = x + y + z

# pasar a, b, c a calcular_suma(), recopilar el valor devuelto por él:
a, b, c = 1, 2, 3
s2 = calcular_suma(a, b, c)
print(s2)
print(w)
```

```
None
NameError: name 'w' is not defined
```

🔁 Instrucción `return` en funciones

La declaración de retorno (`return`) devuelve el control **y** el valor de una función.

> Declarar return sin una expresión devuelve None.

```
def calcular_suma(x, y, z):
    w = x + y + z
    return

s3 = calcular_suma(10, 20, 30)
print(s3)
```

```
None
```

```
def calcular_suma(x, y, z):
    w = x + y + z
    return w

s3 = calcular_suma(10, 20, 30)
print(s3)
```

```
60
```

📦 Devolviendo múltiples valores desde una función

Para devolver múltiples valores de una función, podemos ponerlos en una **lista**, **tupla** o **diccionario** y luego devolverlo.

Con listas:

```
def diez_numeros():
    numeros = []
    for i in range(1, 11):
        numeros.append(i)
    return numeros

print(diez_numeros())
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ejercicio 1 : Reducir el código anterior usando list comprehensions

---

🧾 Ejemplo con múltiples argumentos

Aquí hay otro ejemplo. Esta vez pasamos varios argumentos cuando llamamos a la función.

```
def millas_recorridas(sem_1, sem_2, sem_3, sem_4):
    return [sem_1, sem_2, sem_3, sem_4]

total_mensual = millas_recorridas(25, 30, 28, 40)
print(total_mensual)
```

```
[25, 30, 28, 40]
```

📦 Con tuplas:

```
def persona(nombre, edad):
    return nombre, edad

print(persona("henry", 5))
```

```
('henry', 5)
```

```
def persona(nombre, edad):
    return (nombre, edad)

print(persona("henry", 5))
```

```
('henry', 5)
```

🗺️ Con diccionarios:

```
def ciudad_pais(ciudad, pais):
    ubicacion = {}
    ubicacion[ciudad] = pais
    return ubicacion

ubicacion_favorita = ciudad_pais("Boston", "Estados Unidos")
print(ubicacion_favorita)
```

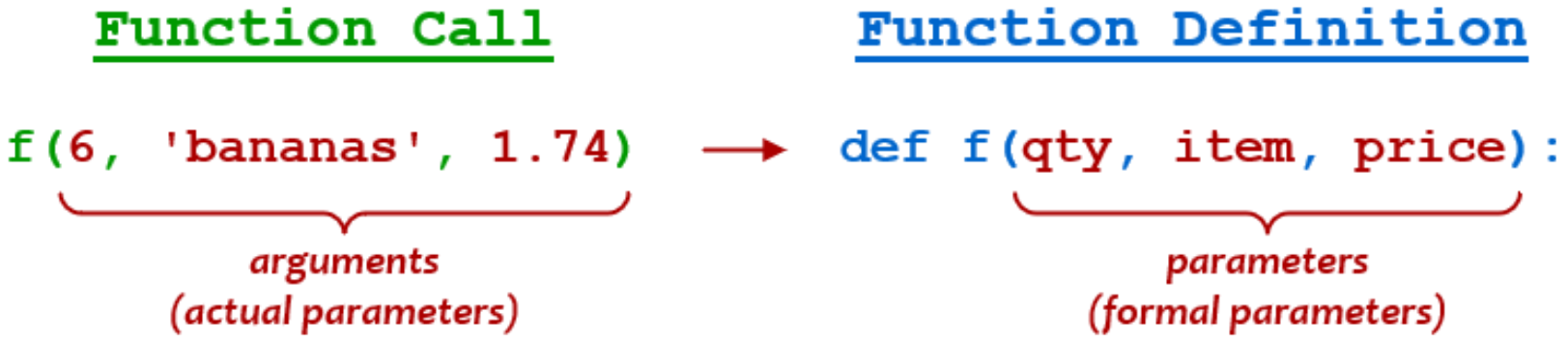
```
{'Boston': 'Estados Unidos'}
```

🧾 Tipos de Argumentos

Los argumentos en una función en Python pueden ser de 4 tipos:

- (a) Argumentos posicionales
- (b) Argumentos por palabras clave (keyword)
- (c) Argumentos posicionales de longitud variable
- (d) Argumentos de palabras clave y longitud variable

> Los argumentos posicionales y de palabra clave a menudo se denominan argumentos **"requeridos"**, mientras que los argumentos de longitud variable se denominan argumentos "opcionales".



📌 Argumentos posicionales

Los argumentos posicionales deben pasarse en el orden posicional correcto.

Por ejemplo, si una función espera que se le pase un `int`, `float` y `string`, entonces al llamar a esta función, los argumentos deben pasarse en el mismo orden:

✅ Llamada correcta

```
def func(i, j, k):
    print(i + j)
    print(k.upper())

func(10, 3.14, 'Rigmarole')  # Llamada correcta
```

```
13.14
RIGMAROLE
```

❌ Llamada incorrecta

```
def fun(i, j, k):
        print(i + j)
    print(k.upper())

fun('Rigmarole', 3.14, 10)  # Orden incorrecto
```

```
TypeError: can only concatenate str (not "float") to str
```

⚠️ Número de argumentos posicionales

💡 Al pasar argumentos posicionales, el número de argumentos pasados debe coincidir con el número de parámetros recibidos.

```
def fun(i, j, k):
    print(i + j)
    print(k.upper())

fun(3.14, 10)  # falta un argumento
```

```
TypeError: fun() missing 1 required positional argument: 'k'
```

🧩 Argumentos por palabras clave (Keyword Arguments)

💡 Los argumentos de palabras clave se pueden pasar fuera de orden.

El intérprete de Python utiliza palabras clave (nombres de variables) para hacer coincidir los valores pasados con los argumentos utilizados en la definición de la función.

✅ Orden diferente, pero con nombres correctos

```
def imprimelo(i, a, str):
    print(i, a, str)

imprimelo(a = 3.14, i = 10, str = 'Sicilian')  # keyword, ok
```

```
10 3.14 Sicilian
```

```
def imprimelo(i, a, str):
    print(i, a, str)

imprimelo(str = 'Sicilian', a = 3.14, i = 10)  # keyword, ok
```

```
10 3.14 Sicilian
```

❌ Nombres incorrectos de palabras clave

```
def imprimelo(ix  r):
    print(i, a, str)

imprimelo(s = 'Sicilian', i = 10, a = 3.14)  # error, keyword name
```

```
TypeError: imprimelo() got an unexpected keyword argument 's'
```

> Se informa un error en la última llamada porque los nombres de las variables en la llamada y la definición no coinciden.

🧩 Combinando argumentos posicionales y de palabras clave

💡 En una llamada, podemos usar argumentos posicionales y de palabras clave.

> Si lo hacemos, **los argumentos posicionales deben preceder a los argumentos de palabras clave**.

✅ Ejemplos válidos:

```
def imprimelo(i, a, str):
    print(i, a, str)

imprimelo(10, a = 3.14, str = 'Ngp')  # ok
```

```
10 3.14 Ngp
```

```
def imprimelo(i, a, str):
    print(i, a, str)

imprimelo(10, str = 'Ngp', a = 3.14)  # ok
```

```
10 3.14 Ngp
```

❌ Ejemplos inválidos (posicional después de keyword):

```
def imprimelo(i, a, str):
    print(i, a, str)

imprimelo(str = 'Ngp', 10, a = 3.14)  # error, positional after keyword
```

```
SyntaxError: positional argument follows keyword argument
```

```
def imprimelo(i, a, str):
    print(i, a, str)

imprimelo(str = 'Ngp', a = 3.14, 10)  # error, positional after keyword
```

```
SyntaxError: positional argument follows keyword argument
```

✨ Argumentos posicionales de longitud variable con `args`

💡 A veces, el número de argumentos posicionales que se pasarán a una función no es seguro.

En tales casos, los argumentos posicionales de longitud variable se pueden recibir usando `*args`.

✅ Un solo argumento:

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10)  # 1 arg, ok
```

```
10
```

✅ Dos argumentos:

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10, 3.14)  # 2 arg, ok
```

```
103.14
```

✅ Sin argumentos:

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo()
```

Salida:

```

```

🧪 Más ejemplos con `args` (argumentos posicionales de longitud variable)

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10, 3.14, 'Sicilian')  # 3 arg, ok
```

```
103.14Sicilian
```

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10, 3.14, 'Sicilian', 'Punekar')  # 4 arg, ok
```

```
103.14SicilianPunekar
```

ℹ️ ¿Qué es `args`?

> 💡 El `args` utilizado en la definición de `imprimelo()` es una **tupla** y `*` indica que contendrá **todos los argumentos pasados** a la función `imprimelo()`.

La tupla se puede iterar mediante el uso de un bucle `for`.

🧩 Uso de `*kwargs` (argumentos de palabras clave de longitud variable)

A veces, el número de argumentos de palabras clave (**keywords**) que se pasarán a una función no es seguro. En tales casos, los argumentos de longitud variable se pueden recibir usando `**kwargs`.

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

imprimelo(a = 10)  # keyword, ok
```

```
a 10
```

🧪 Más ejemplos con `*kwargs` (argumentos de palabras clave de longitud variable)

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

imprimelo(a = 10, b = 3.14)  # keyword, ok
```

```
a 10b 3.14
```

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

imprimelo(a = 10, b = 3.14, s = 'Sicilian')  # keyword, ok
```

```
a 10b 3.14s Sicilian
```

📦 `**kwargs` a partir de un diccionario

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

diccionario = {'Estudiante': 'Mary ', 'Edad': 23}

imprimelo(**diccionario)  # ok
```

```
Estudiante Mary Edad 23
```

ℹ️ ¿Qué es `*kwargs`?

El `**kwargs` utilizado en la definición de `imprimelo()` es un **diccionario** que contiene nombres de variables como **claves** y sus valores como **valores**, mientras que `**` indica que contendrá **todos los argumentos pasados** a `imprimelo()`.

🧠 Reglas sobre `args` y `**kwargs`

Podemos usar cualquier otro nombre en lugar de `args` y `kwargs`.

No podemos usar **más de un** `*args` ni más de un `**kwargs` al definir una función.

🧩 Orden de los argumentos en funciones

Si una función va a recibir argumentos **requeridos** y **opcionales**, entonces deben ocurrir en el siguiente orden:

- Argumentos posicionales
- Argumentos posicionales de longitud variable (`args`)
- Argumentos de palabras clave (`keywords`)
- Argumentos de palabras clave de longitud variable (`*kwargs`)

✅ Ejemplo con todos los tipos de argumentos

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# Ni args ni kwargs
imprimelo(10, 20, x = 30, y = 40)
```

```
10 2030 40
```

🧪 Combinando `args`, `*kwargs`, argumentos posicionales y de palabra clave

---

✅ `100`, `200` van a `args`, nada a `kwargs`

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# 100, 200 van a args, nada va a kwargs
imprimelo(10, 20, 100, 200, x = 30, y = 40)
```

```
10 2010020030 40
```

✅ `a`, `b`, `c` van a `kwargs`

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
```

```
        print(name, value, end = '')

# 100, 200 van para args.
# 'a': 5, 'b': 6, 'c': 7 van para kwargs
imprimelo(10, 20, 100, 200, x = 30, y = 40, a = 5, b = 6, c = 7)
```

```
10 2010020030 40a 5b 6c 7
```

❌ Error: no se pasan `x` e `y`

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# 30, 40 van para args y no queda nada para los argumentos requeridos x, y
# ERROR
imprimelo(10, 20, 30, 40)
```

```
TypeError: imprimelo() missing 2 required keyword-only arguments: 'x' and 'y'
```

⚙️ Argumentos con valores predeterminados

Al definir una función, se puede dar un valor predeterminado a los argumentos.

Se usará ese valor predeterminado **si no pasamos el valor** para ese argumento durante la llamada.

✅ Solo se pasa un argumento (los demás toman valores por defecto)

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(10)  # a toma el valor de 10, b = 100, c = 3.14
print(w)
```

```
113.14
```

✅ Se pasan dos argumentos

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(20, 50)  # a = 20, b = 50, c = 3.14
print(w)
```

```
73.14
```

✅ Se pasan los tres argumentos

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(30, 60, 6.28)  # a = 30, b = 60, c = 6.28
print(w)
```

```
96.28
```

✅ Se usan argumentos por palabra clave fuera de orden

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(1, c = 3, b = 5)  # a = 1, b = 5, c = 3
print(w)
```

```
9
```

> ⚠️ Tenga en cuenta que, al definir una función, los argumentos predeterminados deben ir después de los argumentos no predeterminados.

📦 Desempaquetando argumentos

Supongamos que una función contiene argumentos posicionales y los argumentos a pasar están en una lista, tupla o conjunto.

En tal caso, necesitamos **desempaquetar** la lista/tupla/conjunto usando el operador `*` antes de pasarlo a la función.

✅ Desempaquetando una lista

```
def imprimelo(a, b, c, d, e):
    print(a, b, c, d, e)

lista = [10, 20, 30, 40, 50]
imprimelo(*lista)
```

```
10 20 30 40 50
```

✅ Desempaquetando una tupla

```
def imprimelo(a, b, c, d, e):
    print(a, b, c, d, e)

tupla = ('A', 'B', 'C', 'D', 'E')
imprimelo(*tupla)
```

```
A B C D E
```

✅ Desempaquetando un conjunto

```
def imprimelo(a, b, c, d, e):
    print(a, b, c, d, e)
```

```
conjunto = {1, 2, 3, 4, 5}
imprimelo(*conjunto)
```

```
1 2 3 4 5
```

> ⚠️ Nota: En el caso del conjunto (set), los elementos pueden aparecer en un orden diferente porque los set en Python no tienen orden garantizado.

🧠 Desempaquetando diccionarios con argumentos por palabras clave

> Si una función espera argumentos por palabras clave y los valores están en un diccionario, podemos desempaquetarlos usando **.

```
def imprimelo(nombre='Sandra', edad=75, profesion='Data Analyst'):
    print(nombre, edad, profesion)

diccionario = {'nombre': 'Ana', 'edad': 50, 'profesion': 'Data Scientist'}

imprimelo(*diccionario)      # Pasa solo las claves (incorrecto)
imprimelo(**diccionario)     # Pasa las claves con sus valores
```

```
nombre edad profesion
Ana 50 Data Scientist
```

> La primera llamada pasa las claves, la segunda pasa los valores correctamente.

🧠 Alcance y vida útil de parámetros

> Los parámetros y variables **definidos dentro de una función** tienen un **alcance local**, por lo tanto:

- No son visibles desde fuera de la función.
- Solo existen mientras se ejecuta la función.
- Se destruyen una vez que salimos de la función.

> En cambio, las variables externas a la función sí son visibles dentro de ella, pero:

- Podemos **leer** sus valores,
- No podemos **modificarlas** directamente,
- Para modificarlas, debemos declararlas como **globales** usando la palabra clave `global`.

> ⚠️ Cuando se llama a una función, el intérprete primero busca en el ámbito local y luego en el ámbito global.
>
> Si no encuentra el nombre, se lanza un error del tipo `NameError`.

🔷 Uso de una variable externa a la función

```
def f():
    print(s)

s = 'Python'
f()
```

```
Python
```

> ✔️ La variable s se define fuera, pero se puede leer dentro.

🔷 Variables interna y externa con el mismo nombre

```
def f():
    s = 'Mundo'
    print(s)

s = 'Python'
f()
print(s)
```

```
Mundo
Python
```

> ✔️ La variable `s` dentro de la función es distinta de la `s` global. No se sobrescribe.

🧠 Alcance y vida útil de variables en Python

En Python, **todas las variables definidas dentro de una función son locales por defecto**, a menos que se indique lo contrario. Esto significa que:

- Solo existen **mientras la función se está ejecutando**.
- **No pueden ser accedidas desde fuera** de esa función.
- Al salir de la función, **se destruyen automáticamente**.

> 💡 Usar variables locales es una **buena práctica de programación**, ya que ayuda a evitar errores y conflictos. En general, **es preferible pasar información a una función usando parámetros** o **devolver un valor** con `return`, en lugar de usar o modificar variables externas.

🧠¿Qué pasa con las variables externas (globales)?

Las variables definidas **fuera de una función** se conocen como **variables globales**. Estas variables:

- Son visibles **dentro de las funciones** (pero solo para lectura).
- **No pueden ser modificadas directamente** dentro de una función, a menos que se declaren como `global`.

Si una función necesita **modificar** una variable global, se debe declarar explícitamente con la palabra clave `global`.

```
s = "Hola"

def cambiar():
    global s
    s = "Adiós"

cambiar()
print(s)  # Salida: Adiós
```

🧠 Errores comunes: usar variables globales sin declararlas

Si intentas **usar y modificar** una variable global dentro de una función **sin declararla como global**, Python lanzará un error porque considera que estás trabajando con una variable local no inicializada:

```
s = "Hola"

def fallo():
    print(s)   # Error: variable local 's' usada antes de asignar valor
```

```
    s = "Adiós"

fallo()
```

Salida de error:

```
UnboundLocalError: local variable 's' referenced before assignment
```

> Si definimos la variable externa como global dentro de la función, ya no aparecerá ningún error. Y podremos cambiar el valor de la variable global desde dentro de la función.

✅ Con `global`

```
s = 'Alfa'

def f():
    global s
    print(f'El valor 1 de s es: {s}')
    s = 'Mundo'
    print(f'El valor 2 de s es: {s}')

s = 'Python'
f()
print()
print(f'El valor 3 de s es: {s}')
```

Salida:

```
El valor 1 de s es: Python
El valor 2 de s es: Mundo

El valor 3 de s es: Mundo
```

❌ Sin `global`

```
s = 'Alfa'

def f():
    # global s
    s = 'Mundillo'
    print(f'El valor 1 de s es: {s}')
    s = 'Mundo'
    print(f'El valor 2 de s es: {s}')

s = 'Python'
f()
print()
print(f'El valor 3 de s es: {s}')
```

Salida:

```
El valor 1 de s es: Mundillo
El valor 2 de s es: Mundo

El valor 3 de s es: Python
```

Practica 08