

# Funciones

Antonio Espín Herranz

# Funciones

- Las funciones se definen con la palabra **def**, no es necesario indicar el tipo de los parámetros ni el tipo devuelto por la función.
- Si en un fichero tenemos varias funciones pueden estar por encima o por debajo de la **función llamante**.
- Las tabulaciones indican el contenido de la función.

```
def mi_funcion(param1, param2):  
    print(param1 )  
    print(param2)
```

# Funciones

- Una cadena de texto como primera línea del cuerpo de la función, hace referencia a la documentación.
- Estas cadenas se conocen con el nombre de ***docstring*** (*cadena de documentación*) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):  
    """Esta funcion imprime los dos valores pasados como  
    parametros"""  
    print(param1)  
    print(param2)
```

- A parte de las funciones los objetos también pueden tener **docstring**.

# Funciones

- La llamada a la función:  
`mi_funcion("hola", 2)`
- Se pueden cambiar el orden de los parámetros en la llamada si se indica el nombre:  
`mi_funcion(param2 = 2, param1 = "hola")`
- Si no se indican los dos parámetros:  

```
>>> mi_funcion("hola")  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: mi_funcion() takes exactly 2 arguments (1 given)
```

# Funciones

- Se pueden indicar parámetros por defecto:

```
def imprimir(texto, veces = 1):  
    print( veces * texto)
```

```
>>> imprimir("hola")
```

Hola

```
>>> imprimir("hola", 2)
```

holahola

# Funciones

- Para definir funciones con **un número variable de argumentos** colocamos un último parámetro para la función cuyo nombre debe precederse de un signo \*:

```
def varios(param1, param2, *otros):  
    for val in otros:  
        print (val)
```

- Posibles llamadas:

```
varios(1, 2)  
varios(1, 2, 3)  
varios(1, 2, 3, 4)
```

Los parámetros se almacenan en una **Tupla**.

Cuando llamamos a la función  
Con 2 parámetros la tupla estará vacía  
Y así sucesivamente.

# Funciones

- Otra forma de pasar parámetros a un función y que almacene en un **diccionario** en vez de una tupla sería con **\*\***
- Función:

```
def varios(param1, param2, **otros):  
    for i in otros.items():  
        print(i)
```
- Llamada:

```
varios(1, 2, tercero = 3)
```

# Expansión Tuplas

- Partiendo de una tuplas y una función:  
t = (1,2,3)  
def funcion(param1, param2, param3):  
 pass
- Podemos llamar a la función expandiendo la tupla: Se coloca un \* por delante. **funcion(\*t)**
- Con listas también funciona.



# Expansión Diccionarios

- De la misma forma funciona con un diccionario, pero con un `**` y las claves del diccionario deben de coincidir con los nombres de los parámetros:

```
t = (1,2,3)
```

```
def funcion(param1, param2, param3):
```

```
    Pass
```

- `dic = {"param1":1, "param2":2, "param3":3}`
- `funcion(**dic)`

# Parámetros por valor / referencia

- En el caso del paso de parámetros se hace una distinción entre **objetos mutable e inmutables**.
- Todos los objetos pasan por referencia y en python todo son objetos pero teniendo en cuenta la propiedad anterior.
- Las tuplas son objetos inmutables, si pasamos una tupla a una función se creará una instancia nueva que es la que sufre la modificación.

# Ejemplo

```
def f(x, y):  
    x = x + 3  
    y.append(23)  
    print(x, y)
```

**SALIDA:**  
25 [22, 23]  
22 [22, 23]

```
# Código principal  
x = 22  
y = [22]  
f(x, y)  
print(x, y)
```

- **x** es un objeto **inmutable** **NO** cambia → **por valor**.
- **y** es un objeto **mutable** **cambia** → **por referencia**.

# Objetos mutables e inmutables

- Objetos **inmutables** (que **no pueden cambiar** su valor) como:
  - Números (de cualquier tipo)
  - Cadenas
  - Tuplas
- Objetos **mutables** (que **pueden cambiar**) como:
  - Listas
  - Diccionarios
  - Set (sin indexación)

# return

- La palabra reservada **return** se utiliza para devolver valores de una función.
- En el caso de python podemos devolver varios valores: **por debajo lleva una tupla.**

- Ejemplo:

```
def f(x, y):  
    return x * 2, y * 2
```

```
# Llamada a la función:  
a, b = f(1, 2)
```

Se puede generar procedimientos:  
funciones que no devuelven nada.  
No hay return.

# Alcances

- **Resumen:**

- Los objetos declarados fuera de una función son globales
- Los objetos declarados dentro de una función son locales
- Los objetos globales **siempre se pueden leer** dentro de una función
- Para **modificar un objeto global dentro de una función**
  - Si es **inmutable**, hay que usar global dentro de la función
  - Si es **mutable** Para modificarlo mediante una asignación, hay que usar global
  - Para modificarlo mediante sus métodos, no es necesario usar global.

# Ejemplo

```
1. myGlobal = 5

2. def func1():
3.     global myGlobal
4.     myGlobal = 42

5. def func2():
6.     print (myGlobal)

7. func1()
8. func2()
```

¿Qué imprime en estas situaciones?

- Se comenta la línea 3 de código
- No se comenta la línea 3

```
# datos globales:
```

```
num = 234
```

```
L=[1,2,3]
```

## Otro ejemplo

```
def funcion():
```

```
    print("num:",num)
```

```
    print("L:",L)
```

```
def funcion2():
```

```
    global num,L
```

```
    num+=2
```

```
    L+=[45]
```

```
def funcion3():
```

```
    L.append(8)
```

```
funcion()
```

```
funcion2()
```

```
funcion()
```

```
funcion3()
```

```
funcion()
```



# Ámbitos y espacios de nombres

```
x = 0
def outer():
    x = 1

    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 0
```

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 2
# global: 0
```

# globals()

- La función **globals()** devuelve un diccionario con todas las variables y funciones que están definidas dentro del módulo actual.
- Dentro de este diccionario se encuentran propiedades especiales del módulo como son: `__name__`, `__doc__`, etc.