



**Python 3**

## 5. Funciones

Carolina Mañoso, Ángel P. de Madrid y Miguel Romero

# Índice

## ◆ Funciones:

- Definición
- Parámetros



# Función: Definición (1/2)

- ◆ Una función es un fragmento de código con un nombre asociado que realiza una tarea concreta (y puede devuelve un valor).
  - Permite reutilizar código.
- ◆ Las funciones aparecen:
  - Las propias de Python (`input()`, `print()`, `int()`, `float()`, ...) que se suelen conocer con el nombre de *built-in*.
  - De los módulos preinstalados de Python.
  - Directamente programadas en el código.

# Función: Definición (1/2)

## ◆ Para definir una función:

- Se utiliza la palabra clave `def` seguida del nombre de la función más paréntesis de apertura y cierre. Como toda estructura de control, la definición finaliza con dos puntos ":".
- Las sentencias que forman el cuerpo de la función (el algoritmo que desarrolla) empiezan en la línea siguiente con sangría.
- La primera sentencia del cuerpo de la función puede ser una cadena de texto de documentación de la función (`docstring`)


```
def mi_funcion():  
    """ Esta función escribe hola mundo """  
    print("Hola mundo desde la función")
```

## Función: Definición (2/2)

- ◆ Una función no ejecuta nada hasta que no es **llamada**:

```
mi_funcion() #llamada
```

- ◆ Una función puede retornar valores, (en vez de imprimirlos) usando **return**. Entonces la función puede ser asignada a una variable, que contendrá esos valores:

```
def mi_funcion_con_return():  
    """ Esta función retorna hola mundo """  
    return "Hola mundo" 
```

```
#Llamada a la función  
saludo = mi_funcion_con_return()  
print("Escribo desde el programa:",saludo)
```

# Funciones y parámetros (1/8)

- ◆ Un parámetro es un valor que la función espera recibir cuando es llamada a fin de realizar acciones con el mismo. En la definición estos parámetros van entre los paréntesis separados por comas:

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

- ◆ Los parámetros son variables:

- Locales a la función.
- Toman valor en el momento de la invocación.
- Pueden ser variables simples o listas.

- ◆ Los argumentos:

- Son los valores concretos con los que se invoca la función.
- Existen fuera de la función.

# Funciones y parámetros (2/8)

## ◆ Ejemplo:

```
def mi_funcion_con_param(nombre):  
    print(nombre)
```

```
mi_funcion_con_param('Maria') # Llamada
```

- Los parámetros son variables de ámbito local, si uso `nombre` fuera de la definición nos dará error:

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

# Funciones y parámetros (3/8)

- ◆ Una variable que existe fuera de una función se puede acceder dentro de la función.
- ◆ Una variable que existe dentro de una función no se puede acceder fuera de la función.
- ◆ Si se define la misma variable dentro de la función, se accede a esa nueva variable, no a la exterior.
  - Si se define con la palabra clave `global` dentro de la función entonces se accede a la variable exterior.



# Funciones y parámetros (4/8)

- ◆ Cuando los parámetros son *variables simples* son variables de ámbito local, corresponde a un paso de parámetros *por valor*:

```
def incremento_uno(num):  
    num += 1  
    print("Escribo num desde dentro:", num)  
    return num
```

```
num1 = 1  
num2 = incremento_uno(num1)  
print("Primer_num1 después de la función: ", num1)    → 1  
print("Segundo_num2 después de la función: ", num2)   → 2  
print("Escribo num desde fuera ", num)               → error
```

- ◆ Sin embargo, cuando los parámetros son listas es un paso de parámetros *por referencia* y pueden tener un comportamiento diferente:

- Si cambiamos valores, la lista original no es afectada!
- Si cambiamos la lista en sí, entonces SÍ!

# Funciones y parámetros (5/8)

```
def incremento_uno_lista(lista):  
    lista = lista + [1]  
    return lista  
  
lista1 = [4,3,2]  
lista2 = incremento_uno_lista(lista1)  
print("primera lista", lista1) → [4, 3, 2]  
print("segunda lista", lista2) → [4, 3, 2, 1]
```

```
def incremento_uno_lista_con_append(lista):  
    lista.append(1)  
    return lista  
  
lista1 = [4,3,2]  
lista2 = incremento_uno_lista_con_append(lista1)  
print("primera lista", lista1) → [4, 3, 2, 1]  
print("segunda lista", lista2) → [4, 3, 2, 1]  
  
lista1 = [4,3,2]  
lista2 = incremento_uno_lista_con_append(lista1[:])  
print("primera lista", lista1) → [4, 3, 2]  
print("segunda lista", lista2) → [4, 3, 2, 1]
```

# Práctica: Funciones y parámetros

## ◆ Paso de un parámetro

- ◆ Programa 1: Calcula el cuadrado de un número y lo devuelve.

## ◆ Paso de dos parámetros

- ◆ Programa 2: Calcula la suma de dos números y la devuelve.

## ◆ Sucesión de Fibonacci

- ◆ Programa 3: Calcula la sucesión de Fibonacci.

$$f_0 = 0$$

$$f_1 = 1$$

...

$$f_n = f_{n-1} + f_{n-2}$$

# Práctica: Funciones y parámetros. Solución (1/4)

## ◆ Paso de un parámetro

- ◆ Calcula el cuadrado de un número.

#Ejemplo de paso de un parámetro

```
def cal_cuadrado(n):  
    """Calcula el cuadrado de un número"""  
    return n**2
```



```
#Llamada a la función  
cuadrado = cal_cuadrado(4)  
print(cuadrado)
```

**Nota:** >>> help(cal\_cuadrado)

# Práctica: Funciones y parámetros. Solución (2/4)

## ◆ Paso de dos parámetros

- ◆ Calcula la suma de dos números.

#Ejemplo de paso de dos parámetros

```
def cal_suma(a,b):  
    """Calcula la suma de dos números"""  
    return a+b
```

```
#Llamada a la función  
suma = cal_suma(2,3)  
print(suma)
```

Nota: Ejecutar el programa pasando como parámetros dos cadenas.

# Práctica: Funciones y parámetros. Solución (3/4)

```
def fib1(n):  
    """Escribe la sucesión de Fibonacci (solucion 1)"""  
    a, b = 0, 1  
    while a < n:  
        print(a)  
        a, b = b, a + b  
#primera llamada  
fib1(30)  
-----  
def fib2(n):  
    """Escribe la sucesión de Fibonacci con return (solucion 2)"""  
    result = []  
    a, b = 0, 1  
    while a < n:  
        result = result + [a] #result += [a] o result.append(a)  
        a, b = b, a + b  
    return result  
#segunda llamada  
f30 = fib2(30)  
print(f30)
```

# Práctica: Fibonacci, implementación recursiva (4/4)

◆ Observemos nuevamente:

$$f_0 = 0$$

$$f_1 = 1$$

...


$$f_n = f_{n-1} + f_{n-2}$$

◆ Si escribimos así el código... **recursividad**: *la función se llama a sí misma.*

```
def fib3(n):  
    ''' Definición recursiva de la sucesión de Fibonacci.  
        fib3(n) devuelve el n-ésimo elemento de la sucesión '''  
    if n == 0:  
        f = 0  
    elif n == 1:  
        f = 1  
    else:  
        f = fib3(n-1) + fib3(n-2)  
    return f  
  
for i in range(9):  
    print(fib3(i))
```

# Funciones y parámetros (6/8)

- ◆ Es posible asignar valores por defecto a los parámetros (se le asigna el valor en la definición).
  - La función podrá ser llamada con menos argumentos de los que espera.

```
def mi_funcion(nombre, mensaje='hola') :   
    print(mensaje, nombre)
```

```
mi_funcion('Maria') # Llamada
```

- Pero puedo darle otro valor, si quiero, en la llamada:

```
mi_funcion('Antonio', 'adiós') # Llamada
```



# Funciones y parámetros (7/8)

- ◆ Se pueden definir funciones con un número arbitrario de parámetros. Estos argumentos llegan a la función como una **tupla**. En la definición, se coloca el último parámetro precedido por un asterisco (\*):

```
def mi_funcion(param_fijo, *param_arbitrarios):
```

- Ejemplo:

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

```
varios(1,2) # Llamada1  
varios(1,2,3) # Llamada2  
varios(1,2,3,4) # Llamada3
```

# Funciones y parámetros (8/8)

- ◆ Llamamos *desempaquetado de parámetros* a la situación inversa, la función espera una lista fija de parámetros, pero estos están disponibles en una lista o tupla. En este caso, en la llamada, se debe colocar un asterisco (\*) antes del nombre de la lista o tupla que es pasada.

```
def calcular(importe, descuento):  
    return importe - (importe*descuento/100)
```

```
datos = [1500, 10]  
print(calcular(*datos)) #llamada 
```

# Práctica

◆ **Cálculo del área de un triángulo.** Realice un programa que calcule el área de un triángulo. Para ello:

- Defina una función, `area_triangulo`, con dos parámetros de entrada, la base y la altura, y que retorne el área de un triángulo.
- Desde el programa principal, pida al usuario la base y la altura.
- Muestre en pantalla el área calculada.

Nota: Recuerde que el área del triángulo se calcula con la fórmula  $(Base \times Altura) / 2$

# Práctica: Solución

```
#Programa que calcula el área de un cuadrado,  
mediante una función
```

```
#Definición de la función
```

```
def area_triangulo(b, a):
```

```
    '''Calcular el área de un triangulo'''
```

```
    return b * a / 2
```

```
#Llamada a la función
```

```
print("Vamos a calcular el área de un triángulo")
```

```
base = int(input("¿Longitud de la base (m)?: "))
```

```
altura = int(input("¿Valor de la altura (m)?: "))
```

```
print("El      área      del      triángulo      es",  
      area_triangulo(base, altura), "m2")
```

**Nota 1:** Observe los tipos de datos de la base, de la altura y del área.

**Nota 2:** Ejecute `help(area_triangulo)`.

# Aviso



Python 3 by C. Mañoso, A. P. de Madrid, M. Romero is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional License](https://creativecommons.org/licenses/by-nc-sa/4.0/).

Esta colección de transparencias se distribuye con fines meramente docentes.

Todas las marcas comerciales y nombres propios de sistemas operativos, programas, hardware, etc. que aparecen en el texto son marcas registradas propiedad de sus respectivas compañías u organizaciones.