# Introducción a las funciones

## Introducción a la programación estructurada y modular

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador, utilizando únicamente subrutinas (funciones o procedimientos) y tres estructuras: secuencia, alternativas y repetitivas.

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación (divide y vencerás).

La programación estructural y modular se lleva a cabo en python3 con la definición de funciones.

## Definición de funciones

Veamos un ejemplo de definición de función:

```
>>> def factorial(n):
...   """Calcula el factorial de un número"""
...   resultado = 1
...   for i in range(1,n+1):
...     resultado*=i
...   return resultado
```

Podemos obtener información de la función:

```
>>> help(factorial)
Help on function factorial in module __main__:
factorial(n)
    Calcula el factorial de un número
```

Y para utilizar la función:

```
>>> factorial(5)
120
```

## Ámbito de variables. Sentencia global

Una variable local se declara en su ámbito de uso (en el programa principal y dentro de una función) y una global fuera de su ámbito para que se pueda utilizar en cualquier función que la declare como global.

```
>>> def operar(a,b):
...    global suma
...    suma = a + b
...    resta = a - b
...    print(suma,resta)
...
>>> operar(4,5)
9 -1
>>> resta
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'resta' is not defined
>>> suma
9
```

Podemos definir variables globales, que serán visibles en todo el módulo. Se recomienda declararlas en mayúsculas:

```
>>> PI = 3.1415
>>> def area(radio):
...    return PI*radio**2
...
>>> area(2)
12.566
```

## Parámetros formales y reales

- Parámetros formales: Son las variables que recibe la función, se crean al definir la función. Su contenido lo recibe al realizar la llamada a la función de los parámetro reales. Los parámetros formales son variables locales dentro de la función.
- Parámetros reales: Son la expresiones que se utilizan en la llamada de la función, sus valores se copiarán en los parámetros formales.

## Paso de parámetro por valor o por referencia

En Python el paso de parámetros es siempre por referencia. El lenguaje no trabaja con el concepto de variables sino objetos y referencias. Al realizar la asignación a = 1 no se dice que "a contiene el valor 1" sino que "a referencia a 1". Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.

Evidentemente si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función:

```
>>> def f(a):
...     a=5
>>> a=1
>>> f(a)
>>> a
1
```

Sin embargo si pasamos un objeto de un tipo mutable, si podremos cambiar su valor:

```
>>> def f(lista):
...   lista.append(5)
...
>>> l = [1,2]
>>> f(l)
>>> l
[1, 2, 5]
```

Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, no es recomendable hacerlo en Python. En otros lenguajes es necesario porque no tenemos opción de devolver múltiples valores, pero como veremos en Python podemos devolver tuplas o lista con la instrucción `return`.

## Llamadas a una función

Cuando se llama a una función se tienen que indicar los parámetros reales que se van a pasar. La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función. Si la función no tiene una instrucción `return` el tipo de la llamada sera `None`.

```
>>> def cuadrado(n):
...   return n*n

>>> a=cuadrado(2)
>>> cuadrado(3)+1
10
>>> cuadrado(cuadrado(4))
256
>>> type(cuadrado(2))
<class 'int'>
```

Cuando estamos definiendo una función estamos creando un objeto de tipo `function`.

```
>>> type(cuadrado)
<class 'function'>
```

Y por lo tanto puedo guardar el objeto función en otra variable:

```
>>> c=cuadrado
>>> c(4)
16
```