

# TEMA 4: Funciones en Python

Dpto. de Matemática Aplicada a la Ingeniería Aeroespacial

Informática, 1<sup>er</sup> semestre  
2022 – 2023

**Profesor:** Juan Antonio Hernández Ramos ([juanantonio.hernandez@upm.es](mailto:juanantonio.hernandez@upm.es))

**Coordinador:** Javier de Vicente Buendía ([fj.devicente@upm.es](mailto:fj.devicente@upm.es))

**Colaborador:** Víctor Javier Llorente Lázaro ([victorjavier.llorente@upm.es](mailto:victorjavier.llorente@upm.es))

- **PROGRAMACIÓN MODULAR**

El código se desarrolla en diferentes capas:

- programación de **nivel más alto** en **programa principal**
- **niveles de detalle** en **subprogramas** (que desarrollan tareas específicas) posiblemente incluyendo varios niveles

Ventajas de la programación modular:

- **ahorra repetición de código** que hace misma tarea (que debe ser parametrizada para tratar con diferentes datos)
- resulta **más fácil depurar** partes de código por separado
- **códigos resultantes** son **más claros** (y fáciles de mantener)

- **EJEMPLOS DE SUBPROGRAMAS: FUNCIONES INTRÍNSECAS**

Funciones intrínsecas representan subprogramas (de uso muy común) que forman parte del propio lenguaje:

- `max(x,y,...)` *# por defecto en Python*
- `math.sin(x)` *# necesario importar módulo “math”*
- `math.prod(x,y,...)` *# idem.*
- `numpy.matmul(m1,m2)` *# necesario importar módulo “numpy”*

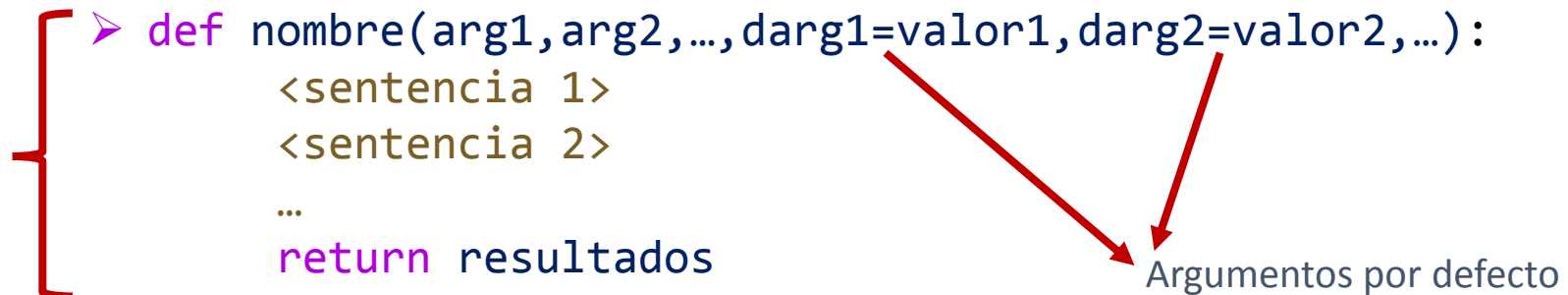
Es posible ampliar la biblioteca de funciones intrínsecas con otras **desarrolladas por usuario**. Algunos ejemplos: cálculo de determinante, identificación de números primos, conversión a expresión binaria...

## • FUNCIONES

Una **función** es una sección de un programa que calcula un valor de manera independiente al resto del programa. Una función tiene tres componentes importantes:

- los **parámetros**, que son los valores que recibe la función como entrada
- el **código de la función**, que son las operaciones que hace la función
- el **resultado** (o **valor de retorno**), que es el valor final que entrega la función

Siempre hay que definirla antes de utilizarla. def al comienzo del programa principal.



```
➤ def nombre(arg1,arg2,...,darg1=valor1,darg2=valor2,...):  
    <sentencia 1>  
    <sentencia 2>  
    ...  
    return resultados
```

Diagram illustrating the function definition syntax. Red arrows point from the default values in the parameter list (`darg1=valor1` and `darg2=valor2`) to the text "Argumentos por defecto".

- **Un primer ejemplo: función sinc(x)**

Se define función real sinc mediante:

$$\text{sinc}(x) = \begin{cases} \frac{\sin(x)}{x}, & \text{si } x \neq 0 \\ 1, & \text{si } x = 0 \end{cases}$$

```
# Función sinc
```

```
import math
```

```
def sinc(x):
```

```
    if (x == 0.0):
```

```
        return 1.0
```

```
    else:
```

```
        return math.sin(x)/x
```

```
a = float(input('Deme un número: '))
```

```
print('sinc(',a,') = ',sinc(a))
```

- Se desea programar una función con argumento de tipo real y resultado de tipo real que implemente esta función
- Desde el código principal se llamará a esta función mediante sinc(variable), como se hace para las funciones intrínsecas

```
Deme un número: 5.0
```

```
sinc( 5.0 ) = -0.1917848549326277
```

```
Deme un número: 0
```

```
sinc( 0.0 ) = 1
```

```
Deme un número: 0.0001
```

```
sinc( 0.0001 ) = 0.9999999983333334
```

- **Un segundo ejemplo: norma  $L^p$**

Dados un vector  $\mathbf{v} \in \mathbb{R}^n$  y  $p \in \mathbb{N}$  se define la norma  $L^p$  de  $\mathbf{v}$  mediante:

$$\|\mathbf{v}\|_p = \left( \sum_{i=1}^n |v_i|^p \right)^{1/p}$$

- Se implementa una función para cálculo norma  $L^p$  de vector en  $\mathbb{R}^2$  con dos argumentos:
  - vector  $\mathbf{v}$  (del que se calcula la norma)
  - número natural  $p$  (especifica norma empleada)
- Se llamará a la función con `norma(vector, indice)` donde el segundo argumento recoge el índice  $p$  de la norma

```
# Norma  $L^p$  de vector en  $\mathbb{R}^2$ 
```

```
import numpy as np
```

```
def norma(v,p):
    Lp = (abs(v[0])**p + abs(v[1])**p)**(1/p)
    return Lp
```

```
w = np.array([5.6, 8.7])
print('norma(w,1) = ',norma(w, 1))
print('norma(w,4) = ',norma(w, 4))
```

```
norma(w,1) = 14.299999999999999
norma(w,4) = 9.05148576770212
```

- **Un segundo ejemplo: norma  $L^p$**

Dados un vector  $\mathbf{v} \in \mathbb{R}^n$  y  $p \in \mathbb{N}$  se define la norma  $L^p$  de  $\mathbf{v}$  mediante:

$$\|\mathbf{v}\|_p = \left( \sum_{i=1}^n |v_i|^p \right)^{1/p}$$

Función que llama a otras funciones

Valores de salida múltiples

```
L1 = 14.299999999999999
L2 = 10.346496991735897
Linf = 8.7
```

```
# Norma  $L^p$  de vector en  $\mathbb{R}^2$ 
```

```
import numpy as np
```

```
def norma(v,p):
```

```
    if (p == 'inf'):
```

```
        Lp = max(abs(v[0]), abs(v[1]))
```

```
    else:
```

```
        Lp = (abs(v[0])**p + abs(v[1])**p)**(1/p)
```

```
    return Lp
```

```
def normas(v):
```

```
    return norma(v,1), norma(v,2), norma(v,'inf')
```

```
w = np.array([5.6, 8.7])
```

```
normal1, norma2, normainf = normas(w)
```

```
print('L1 = ',normal1)
```

```
print('L2 = ',norma2)
```

```
print('Linf = ',normainf)
```

- **VARIABLES GLOBALES Y LOCALES**

Cada función define un nuevo **espacio de nombres (name space)**, también llamado **ámbito (scope)**. Si queremos asignar valor a una variable en una función, pero no queremos que Python la considere local, debemos declararla como `global` o `nonlocal`,

Nuevas variables locales ←

```
def f(x):  
    y = 1  
    x = x + y  
    print('x = ', x)  
    return x  
  
x = 3  
y = 2  
z = f(x)  
print('z = ', z)  
print('x = ', x)  
print('y = ', y)
```

Variables globales ←

```
x = 4  
z = 4  
x = 3  
y = 2
```


```
def f(x):  
    global y  
    x = x + y  
    print('x = ', x)  
    return x  
  
x = 3  
y = 2  
z = f(x)  
print('z = ', z)  
print('x = ', x)  
print('y = ', y)
```

```
x = 5  
z = 5  
x = 3  
y = 2
```

- **VARIABLES GLOBALES Y LOCALES**

Cada función define un nuevo **espacio de nombres (name space)**, también llamado **ámbito (scope)**. Si queremos asignar valor a una variable en una función, pero no queremos que Python la considere local, debemos declararla como `global` o `nonlocal`,

```
def f(x):  
    def g(x):  
        nonlocal y  
        y = 3  
        return x + y  
  
    y = 1  
    x = g(x)  
    print('x = ', x)  
    print('y = ', y)  
    return x  
  
x = 3  
y = 2  
z = f(x)  
print('z = ', z)  
print('x = ', x)  
print('y = ', y)
```



x =	6
y =	3
z =	6
x =	3
y =	2

El calificativo `global` solo puede usarse para variables globales a nivel de módulo. Para poder modificar la variable a nivel de función, necesitamos un nuevo calificador: `nonlocal`.



- FUNCIONES COMO ARGUMENTOS A FUNCIONES Y FUNCIONES RECURSIVAS

```
def f(x):  
    a = 21  
    return a*x + b  
  
def g(x):  
    b = 3.2  
    return a*x + b  
  
a = 20  
b = -2.5  
print('f(g(1)) = ', f(g(1)))  
print('g(f(1)) = ', g(f(1)))
```

```
f(g(1)) = 484.7  
g(f(1)) = 373.2
```

```
def fact(n):  
    res = 1  
    while n > 1:  
        res = res * n  
        n -= 1  
    return res  
  
n = 5  
print(n, '! = ', fact(n))
```

```
5 ! = 120
```

```
def fact(n):  
    if n == 1:  
        return n  
    return n*fact(n-1)  
  
n = 5  
print(n, '! = ', fact(n))
```

```
5 ! = 120
```

## • USO DE MÓDULOS

- Permiten **agrupar funciones/subrutinas** en un único archivo
- Se compilan todas funciones al mismo tiempo
- Genera **interfaces automáticas** para las funciones (y se almacena información en un archivo adicional)
- Funciones del módulo pueden llamarse unas a otras
- En **programa principal** basta mencionar módulo que se usa

```
def funcion1(argumentos):  
    ...  
    return ...  
  
def funcion2(argumentos):  
    ...  
    return ...  
  
...
```

nombre\_modulo.py

```
import nombre_modulo as abvnombre  
  
...  
valor = abvnombre.funcion1(argumentos)  
...
```

principal.py

- Ejemplo de módulo y uso: función `vectorial(u,v)`

```
# Módulo vectores.py

import numpy as np

pv = np.array([0.0, 0.0, 0.0])

def vectorial(u,v):
    pv[0] = u[1] * v[2] - u[2] * v[1]
    pv[1] = u[2] * v[0] - u[0] * v[2]
    pv[2] = u[0] * v[1] - u[1] * v[0]
    return pv
```

```
# Programa principal

import vectores as mv
import numpy as np

v1 = np.array([5.3, 2.1, 8.6])
v2 = np.array([3.7, 1.2, 7.8])
print('v1 x v2 =',mv.vectorial(v1, v2))
```

```
v1 x v2 = [ 6.06 -9.52 -1.41]
```

- Ejemplo de módulo y uso: función `norma(v,p)`

```
# Módulo vectores.py

import numpy as np

# Definición de func. vectorial

def norma(v, p = 2):
    if (p == 'inf'):
        Lp = max(abs(v))
    else:
        Lp = sum(abs(v) ** p) ** (1.0 / float(p))
    return Lp
```

```
# Programa principal

import vectores as mv
import numpy as np

v1 = np.array([5.3, 2.1, 8.6])
v2 = np.array([3.7, 1.2, 7.8])
print('norma 2 de v1 = ',mv.norma(v1))
print('norma 8 de v2 = ',mv.norma(v2,8))
```

```
norma 2 de v1 = 10.317945531936093
norma 8 de v2 = 7.802497054915356
```

- Ejemplo de módulo y uso: función `norma(v,p)`

```
# Módulo vectores.py

import numpy as np

# Definición de func. vectorial

def norma(v, p = 2):
    if (p == 'inf'):
        Lp = max(abs(v))
    else:
        Lp = sum(abs(v) ** p) ** (1.0 / float(p))
    return Lp
```

```
# Programa principal

import vectores as mv
import numpy as np

v1 = np.array([5.3, 2.1, 8.6])
v2 = np.array([3.7, 1.2, 7.8])
print('norma 2 de v1 = ', mv.norma(v1))
print('norma 8 de v2 = ', mv.norma(p=8, v=v2))
```

```
norma 2 de v1 = 10.317945531936093
norma 8 de v2 = 7.802497054915356
```

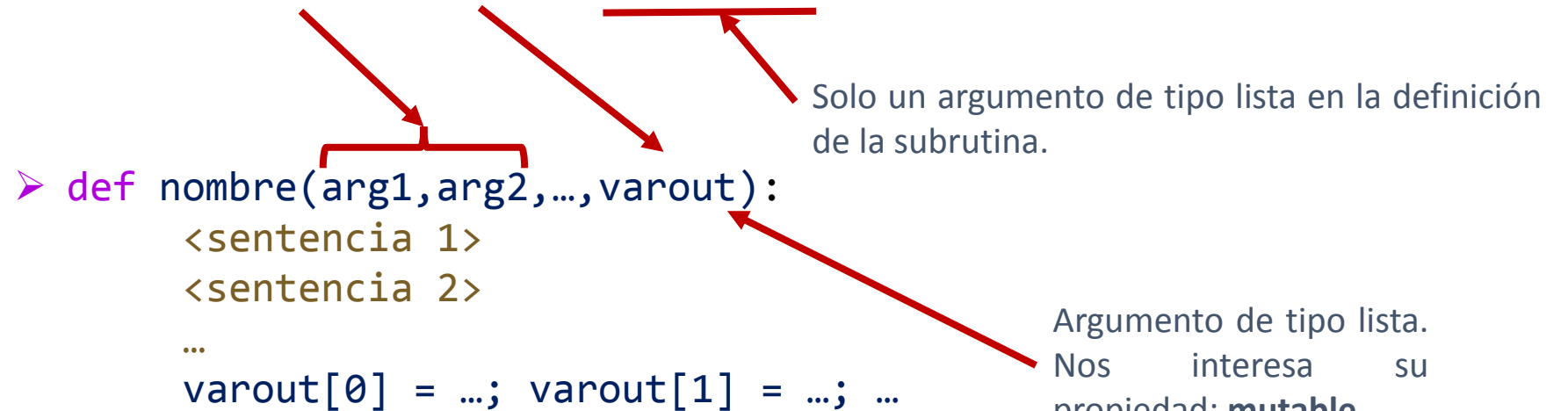
- Orden de argumentos es arbitrario si se usan etiquetas (lo que facilita uso de argumentos opcionales)
- Se emplea nombre de argumento para especificar argumento

## • SUBROUTINAS

Las subrutinas:

- **transfiere control** desde el programa principal para realizar unas operaciones determinadas sobre un conjunto de variables
- se trata de un código parametrizado que **operará sobre ciertas variables** del programa principal
- en la llamada a la subrutina **se especifica a qué variables** tendrá acceso la subrutina

**Al igual que funciones**, constituyen subprogramas que aíslan determinadas tareas; pero **a diferencia de funciones**, las subrutinas tienen un número cualquiera de argumentos de entrada, de salida, y de entrada/salida (!).



```
➤ def nombre(arg1, arg2, ..., varout):
    <sentencia 1>
    <sentencia 2>
    ...
    varout[0] = ...; varout[1] = ...; ...
```

Solo un argumento de tipo lista en la definición de la subrutina.

Argumento de tipo lista.  
Nos interesa su  
propiedad: **mutable**

- **Un ejemplo elemental de subrutina**

Objetivo:

- programa para conversión de tiempos
- se desea convertir un tiempo en segundos a su descomposición en días, horas, minutos y segundos.

Algoritmo:

- se divide tiempo (en segundos) entre 60 para obtener minutos y segundos restantes
- se divide minutos entre 60 para obtener horas y minutos restantes
- se divide horas entre 24 para obtener días y horas restantes

Observación:

- sería cómodo disponer de un subprograma que calcule para dos números enteros el cociente y resto de la división

Esquema de programa principal:

1. se asigna `tiempo` en segundos
2. llama a subprograma `division`: divide `tiempo` entre 60 para obtener `minutos` y `segundos_restantes`
3. llama a subprograma `division`: divide `minutos` entre 60 para obtener `horas` y `minutos_restantes`
4. llama a subprograma `division`: divide `horas` entre 24 para obtener `dias` y `horas_restantes`

Esquema de subprograma `división`:

1. recibe dos números enteros `n1` y `n2`
2. calcula `cociente` de división `n1 / n2`
3. calcula `resto` de división `n1 / n2`

- Un ejemplo elemental de subrutina


Inicializo.



# Módulo operaciones.py

```
def division(x, y, output):
    cociente = x // y
    resto = x % y
    output[0] = cociente; output[1] = resto
```

Argumentos de salidas para la subrutina  
división escrito en una variable de tipo lista.



Llamada a la subrutina división.



Asigno los nuevos valores.



```
# Programa principal

import operaciones as op

minutos = horas = dias = 0.0
segundos_rest = minutos_rest = horas_rest = 0.0

tiempo = float(input('tiempo total en segundos? '))

args = [minutos, segundos_rest]
op.division(tiempo, 60, args)
minutos = args[0]; segundos_rest = args[1]

args = [horas, minutos_rest]
op.division(minutos, 60, args)
horas = args[0]; minutos_rest = args[1]

args = [dias, horas_rest]
op.division(horas, 24, args)
dias = args[0]; horas_rest = args[1]

print(dias, horas_rest, minutos_rest, segundos_rest)
```



- Un segundo ejemplo (elemental) de subrutina

```
# Módulo operaciones.py

def intercambia(input):
    aux = input[0]
    input[0] = input[1]
    input[1] = aux
```

```
# Programa principal

import operaciones as op

x = 1; y = 2
args = [x, y]
op.intercambia(args)
x = args[0]; y = args[1]
print(x, y)
```

Como vemos, los argumentos que se pasan a un subprograma en Python es por asignación. Si queremos que cambien es necesario que sean mutables (cambien), lo que convierte a las subrutinas en Python en ineficientes (y poco frecuentes) frente a las funciones. En otros lenguajes de programación esto no sucede.

## EJERCICIOS PROPUESTOS

1. Escribir una función que pida la anchura de un triangulo y lo dibuje con el carácter “\*”
2. Escribir una función que ingrese su dirección email e imprima un mensaje indicando si la dirección es válida o no. Una dirección se considerará válida si contiene el símbolo “@”
3. Escribir una función que reciba un número entero y escriba por pantalla su conversión a base binaria
4. Escribir una función que determina si un año es bisiesto o no
5. Escribir una función que muestre el resultado del sumatorio  $\sum_{k=0}^N ar^k$  dado tres número:  $a, r \in \mathbb{R}$  y  $N \in \mathbb{N}$ . El programa deberá decidir si la suma es convergente o no. También deberá dar como resultado la suma infinita, es decir:  $N \rightarrow \infty$
6. Escribe una función que dado un número real  $x$  y una lista de coeficientes  $a_0, a_1, a_2, \dots, a_n$  devuelva el valor del polinomio definido por sus raíces  $a_i$ 's en el punto  $x$
7. Escribe una función que divida el dominio  $[a, b]$  en  $N \in \mathbb{N}$  intervalos cuyos puntos estén equidistantes
8. Escribe una función que divida el dominio  $[a, b]$  con un ratio de expansión,  $r \in \mathbb{R}^+$ , entre intervalos constante. La longitud del primer intervalo,  $h_0$ , debe ser conocida. Si  $r = 1$  deberá devolver la lista de valores del ejercicio anterior. Recordatorio:  $x_{i+1} = x_i + h_i$ ,  $h_{i+1} = h_i r$ ,  $i = 0, 1, 2, \dots, N$

## EJERCICIOS PROPUESTOS

9. Escribe una función en Python  $f(t)$  para calcular la función definida a trozos:

$$f(t) = \begin{cases} 5, & -2\pi < t < -\pi \\ -5, & -\pi < t < 0 \\ 5, & 0 < t < \pi \\ -5, & \pi < t < 2\pi \end{cases}$$

y otra función en Python  $S(t, n)$  para su aproximación mediante una suma,

$$S_n(t) = \frac{10}{\pi} \sum_{k=1}^n \frac{1 - (-1)^k}{k} \sin(kt)$$

donde  $n \in \mathbb{N}$ . El programa deberá devolver el error  $|f(t) - S_n(t)|$  en  $t = -3\pi/2, -\pi/2, \pi/2, 3\pi/2$  y para  $n = 1, 3, 5, 10, 30, 100$

10. Escribe una función en Python `gaussian(x, m=0, s=1)` para calcular la función Gaussiana:

$$f(x) = \frac{1}{\sqrt{2\pi}s} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right]$$

donde  $m, s \in \mathbb{R}$ . El programa deberá devolver por pantalla una tabla de valores  $x$  y  $f(x)$  para  $N$  valores  $x$  espaciados uniformemente en el intervalo  $[m - 5s, m + 5s]$