TEMA 4: Funciones en Python

Dpto. de Matemática Aplicada a la Ingeniería Aeroespacial

Informática, 1er semestre
2022 – 2023

Profesor: Juan Antonio Hernández Ramos (juanantonio.hernandez@upm.es)

Coordinador: Javier de Vicente Buendía (fj.devicente@upm.es)

Colaborador: Víctor Javier Llorente Lázaro (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
```

Argumentos por defecto

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

Se define función real `sinc` mediante:

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

```
# 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 **v** ∈ R^n y p ∈ N se define la norma L^p de **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 R^2 con dos argumentos:
  - vector **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

```python
# Norma L^p de vector en 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) =  9.051485767702112
```

Importante

- **Vectores y matrices** se representan mejor con arreglos de tipo **array**
- Es más eficiente trabajar con la librería **numpy** que con arreglos de tipo listas en Python para estos objetos matemáticos

siguiente tema

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

Dados un vector **v** ∈ R^n y p ∈ N se define la norma L^p de **v** mediante:

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

```python
# Norma L^p de vector en 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])
norma1, norma2, normainf = normas(w)
print('L1 = ',norma1)
print('L2 = ',norma2)
print('Linf = ',normainf)
```

Función que llama a otras funciones

Valores de salida múltiples

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

• 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

Variables globales

```python
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)
```

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

```python
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`,

```python
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**

```python
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
```

```python
def fact(n):
    res = 1
    while n > 1:
        res = res * n
        n -= 1
    return res

n = 5
print(n,'! = ',fact(n))
```

```
5 ! =  120
```

```python
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)`

```python
# 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
```

```python
# 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)`

```python
# 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
```

```python
# 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)`

```python
# 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
```

```python
# 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

• APLICACIÓN MATEMÁTICA DEL PARADIGMA FUNCIONAL: DERIVADA

La derivada de una función en un punto se define como:

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Una aproximación sería utilizar la formula anterior pero con una h muy pequeña: $0 < h \ll 1$

Valor optimo dependiente de $f(x)$

```python
def func(x):
    return <expresion>

def derivada(f, x0):
    h = 1e-7
    return ( f(x0 + h) - f(x0) ) / h

derivada(f = func,x0 = ...)
```

```
f(x) = exp(x), x0 = 1
--------------------------------------------------
h = 1e-01 df(1) = 2.85884195 error = 1.406e-01
h = 1e-02 df(1) = 2.73191866 error = 1.364e-02
h = 1e-03 df(1) = 2.71964142 error = 1.359e-03
h = 1e-04 df(1) = 2.71841775 error = 1.359e-04
h = 1e-05 df(1) = 2.71829542 error = 1.359e-05
h = 1e-06 df(1) = 2.71828319 error = 1.359e-06
h = 1e-07 df(1) = 2.71828197 error = 1.399e-07
h = 1e-08 df(1) = 2.71828182 error = 6.603e-09
h = 1e-09 df(1) = 2.71828204 error = 2.154e-07
h = 1e-10 df(1) = 2.71828338 error = 1.548e-06
h = 1e-11 df(1) = 2.71831446 error = 3.263e-05
h = 1e-12 df(1) = 2.71871414 error = 4.323e-04
h = 1e-13 df(1) = 2.71782596 error = 4.559e-03
h = 1e-14 df(1) = 2.70894418 error = 9.338e-02
h = 1e-15 df(1) = 3.10862447 error = 3.903e-01
h = 1e-16 df(1) = 0.0        error = 2.718e+00
h = 1e-17 df(1) = 0.0        error = 2.718e+00
h = 1e-18 df(1) = 0.0        error = 2.718e+00
h = 1e-19 df(1) = 0.0        error = 2.718e+00
```

Estamos trabajando más que **precisión de máquina**

- **APLICACIÓN MATEMÁTICA DEL PARADIGMA FUNCIONAL: INTEGRAL**

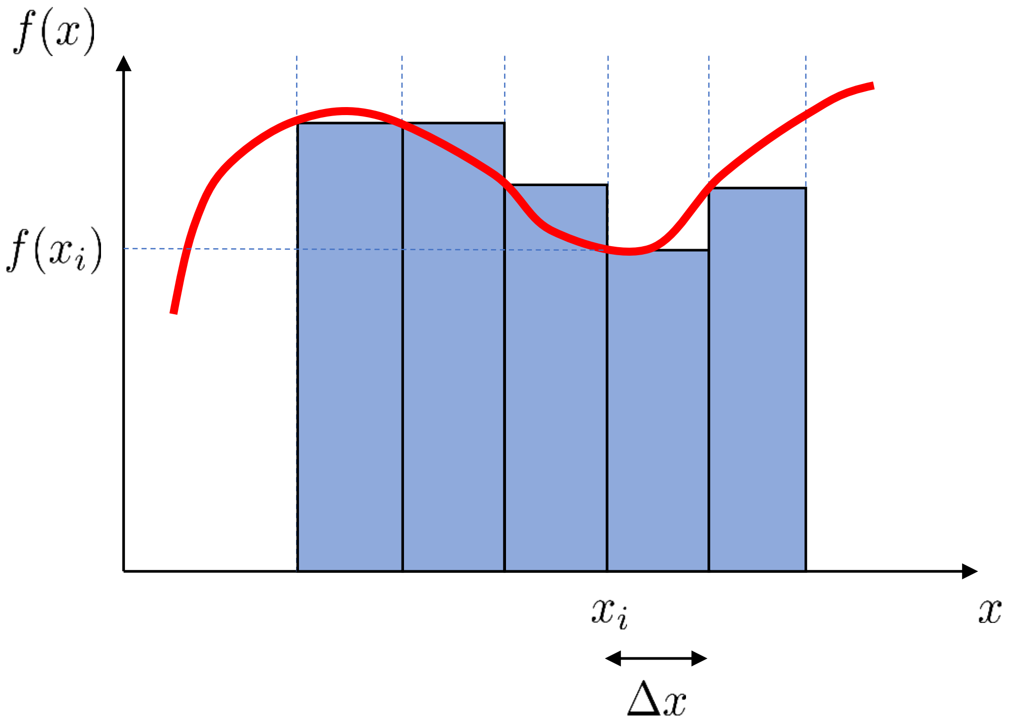
Llevando al límite la suma de Riemann (por la izquierda), se puede demostrar:

$$\int_a^b f(x)\mathrm{d}x = \lim_{N\to\infty} \sum_{i=0}^{N-1} f(x_i)\Delta x$$

donde $x_i = a + i\Delta x$ para $i = 0, \cdots, N-1$ y $\Delta x = (b-a)/N$

```
def integral(f, a, b):
    N = 1000
    Dx = (b - a) / N
    F = [f(a + Dx * i) for i in range(N)]
    return Dx * sum(F)
```

```
f(x) = 4/(1 + x^2), a = 0, b = 1
----------------------------------
N = 10     If = 3.439925988907159
N = 100    If = 3.171575986923129
N = 1000   If = 3.1445924869231243
N = 10000  If = 3.14189265192314
N = 100000 If = 3.141622653573153
```



- **APLICACIÓN MATEMÁTICA DEL PARADIGMA FUNCIONAL: TAYLOR**

El polinomio de Taylor se define como:

$$f(x) \simeq P_N(x;x_0) = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n$$

$$f^{(n)}(x_0) = \lim_{h\to 0} \frac{f^{(n-1)}(x_0+h) - f^{(n-1)}(x_0-h)}{2h}$$

Derivada simétrica para obtener mejores resultados

Si h es muy muy pequeña corremos el riesgo de calcular mal las derivadas de orden superior: $f^{(n)}(x_0) \sim \mathcal{O}(h^{-n})$

```python
def taylor(f, x0, x, N):
    def df(f, n, x):
        if (n == 0):
            return f(x)
        else:
            h = 1e-2
            return (df(f,n - 1,x + h) - df(f,n - 1,x - h)) / (2*h)
    def b(n):
        return df(f,n,x0) * (x - x0) ** n / factorial(n)
    return sum([b(n) for n in range(N + 1)])
```

```
f(x) = exp(x), x = 1, x0 = 0
----------------------------
N = 2  error = 2.18e-01
N = 4  error = 9.90e-03
N = 6  error = 1.81e-04
N = 8  error = 4.64e-05
N = 10 error = 3.87e-04
```

No estamos acercando a precisión de máquina

• SUBRUTINAS

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 (!).

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

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

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**

```python
# 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.

Inicializo.

```python
# 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**

```python
# Módulo operaciones.py

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

```python
# 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)
```

Resumen

- Los argumentos que se pasan a un subprograma en Python es por asignación
- Si queremos que cambien es necesario que sean mutables
- Las subrutinas en Python son ineficientes (y poco frecuentes) frente a las funciones

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: $N \to \infty$

6. Escribe una función que dado un número real x y una lista de coeficientes $a_0, a_1, a_2, \cdots, a_n$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: $x_{i+1} = x_i + h_i,\ \ h_{i+1} = h_i r,\ \ i = 0,1,2, \cdots, N$

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 $[m - 5s, m + 5s]$