Herramientas de programación en Python

1. Fundamentos de programación

1.2. Introducción a Funciones

1. Uso de funciones. Funciones internas y de módulos
2. Funciones y procedimientos
3. Diseño de funciones

Pedro Gomis

pedro.gomis@upc.edu

Introducción

- El paradigma de la **programación modular** surgió para hacer un programa más claro, leíble, menos complejo y, principalmente, para **reutilizar subprogramas**.
- De esta forma, al programador se le facilita modificar y corregir los **códigos por separado** y también crear una biblioteca de subprogramas utilizables por otros programas
- Los **subprogramas** (llamados también **subrutinas**) se refieren al conjunto de instrucciones que están separadas del programa principal y realizan cálculos o tareas.
- Se refieren, generalmente, a las **funciones** en los diversos lenguajes de programación, incluyendo **Python**
- Pero también hay **subprogramas** que realizan tareas **sin devolver resultados** (como sí lo hacen las típicas funciones matemáticas), que son llamados en algunos lenguajes de programación ***procedures*** (modula-3, Pascal, ADA)

1. Uso de funciones. Funciones internas y de módulos

- En diversos ejemplos de programas hemos utilizado funciones internas (***built-in functions***), `abs(x)`, `print()`, `input()`, `int()`, etc.

- En **matemáticas** una **función** devuelve un resultado dependiendo de cada valor de su(s) variable(s) independiente(s). Por ejemplo:

  $f(x) = 3x^2 + 1$ calcula, para $x = 2$, el valor $f(2) = 13$

  $f2(x, y) = x^2 + y^2$ calcula $f(2,2) = 8$

  $f(x) = \sqrt{x}$ calcula $f(9) = 3$

- Las **funciones** que **devuelven resultados** (**funciones productivas** o ***fruitful functions***) en los lenguajes de programación se comportan de manera similar.

1. Uso de funciones. Funciones internas y de módulos

- En los lenguajes informáticos una **función** es una instrucción o un **bloque de instrucciones** que realizan un cálculo o una tarea, y que tiene un identificador como nombre (igual que las variables).

Usar o invocar una función

- Uso o **llamado de una función** (también se dice **invocar** la función):

```
>>> type(7)       # el valor 7 es el: argumento de la función
<class 'int'>
>>> int(23.6)
23
>>> print('Escribe el resultado de 3*2/12:', 3*2/12)
Escribe el resultado de 3*2/12: 0.5
```

1. Uso de funciones. Funciones internas y de módulos

- Ejemplos de **funciones internas** (***built-in functions***)*:

Función interna	Devuelve
`type`	tipo de dato
`id`	Identidad o ubicación de memoria
`bin`	string del binario equivalente al entero dado
`int, float, str`	entero, real, string del valor dado
`input`	string del texto leído del teclado
`print`	Valores a imprimir en pantalla
`abs`	valor absoluto de un número
`round`	redondea un real a los decimales especificados

- **Importar** funciones de librerías (no funciones internas):

```
from math import sqrt
x = abs(-9) + 3
y = sqrt(x)
print('Raíz cuadrada de',x, '=', round(y, 3))
```

```
>>> Raíz cuadrada de 12 = 3.464
```

* https://docs.python.org/3/library/functions.html

1. Uso de funciones. Funciones internas y de módulos

- Los **argumentos** son los valores que le pasamos a las funciones.
- Los **argumentos** de las funciones pueden ser ***expresiones***, incluso ***expresiones con funciones***, como se observa a continuación :

```
y = sqrt(4 + x**2)
z = 4 + sqrt(abs(-64))
```

```
>>> z = 4 + sqrt(abs(-64))
>>> print(z)
12.0
```

- Estas funciones devuelven un valor (*return value*) luego de ser llamadas.
- Pueden haber múltiples argumentos de entrada. Por ejemplo, las funciones **round** o **randint** requieren de dos argumentos y devuelven un valor:

```
from random import randint
n = randint(10, 100)/3.2567
print(n, round(n,2))
```

1. Uso de funciones. Funciones internas y de módulos

- Lista de algunas funciones de l módulo ***math*** (matemáticas)*

`Módulo math`	devuelve
`pi`	valor π = 3.141592653589793
`e`	valor e = 2.718281828459045
`ceil(x)`	entero mayor que x, hacia ∞
`floor(x)`	entero menor que x, hacia -∞
`trunc(x)`	redondea hacia 0
`factorial(x)`	x!
`exp(x)`	e**x
`log(x)`	logaritmo natural (base e), ln(x)
`log10(x)`	logaritmo base 10
`sqrt(x)`	raíz cuadrada de x
`sin(x), cos(x), tan(x)`	seno, coseno, tangente de x
`degrees(x)`	ángulo x de radianes a grados
`radians(x)`	ángulo x de grados a radianes

* La lista completa de módulos se puede consultar en: https://docs.python.org/3/library/numeric.html

1. Uso de funciones. Funciones internas y de módulos

- Lista de algunas funciones del módulo ***random*** (de valores aleatorios) *

`Módulo random`	devuelve aleatorio
`randint(a,b)`	entero en el rango [a, b]
`randrange(a,b,paso)`	de range(a, b, paso)
`shuffle(s)`	baraja la secuencia s
`choice(s)`	escogido de la secuencia s
`random()`	real en el rango [0.0 1.0)
`seed()`	inicializa generador aleatorios

* La lista completa de módulos se puede consultar en: https://docs.python.org/3/library/index.html

1. Uso de funciones. Funciones internas y de módulos

- **Importar** funciones de un módulo:

```
from math import sqrt, log10
x = sqrt(10)
dB = 20*log10(x/2)
```

- O todo el módulo:

```
import math
x = math.sqrt(10)
dB = 20*math.log10(x/2)
```

```
import math as m
x = m.sqrt(10)
dB = m.log10(x/2)
```

2. Funciones y procedimientos

- En algunos lenguajes de programación (Modula-3, Pascal, ADA), se ***distinguen*** aquellos subprogramas que realizan cálculos y **devuelven resultados** (como las funciones matemáticas), que se llaman **funciones.**
- Mientas otros subprogramas ***realizan tareas o acciones sin devolver resultados***. Éstos se llaman **procedimientos** (***procedures***).
- En **Python**, hay **funciones** (la típica de matemáticas), como por ejemplo, l**og10**(100) que devuelve 2.0.
- Pero hay **funciones**, como **shuffle**(*s*) del módulo random, que baraja o pone en posiciones aleatorias los elementos de la secuencia *s*. ***No devuelve resultado alguno,*** sino que realiza una acción sobre un objeto y éste queda modificado. Este tipo de función equivale a un ***procedure*** de otros lenguajes.

2. Funciones y procedimientos

- En Python las funciones pueden ser:
  - **Productivas** o ***fructíferas*** (devuelven un resultado)
  - **Nulas** (estériles o no productivas, en inglés: **void**). Realmente, devuelven el resultado **None**. Pueden compararse con los ***procedures*** de otros lenguajes de programació.

log10(x) es una **función productiva** (***fruitful function***): devuelve resultado

shuffle(s) es una **función nula o estéril** (***void function***) pues realiza una acción sobre s pero NO devuelve un resultado.

3. Diseño de funciones

- En Python la **definición de una función** es de la forma:

```
def   nombre_funcion(parametros):
      cuerpo_de_la_funcion
```

- Los **parámetros** se relacionan con los valores que le son pasados como **argumentos**.
- El **cuerpo** del subprograma se indenta (sangra) también con cuatro espacios en blanco.
- Al volver a la columna 1 del editor se termina el bloque de código que define la función y volvemos al programa principal.
- Conviene definir todas las funciones que use nuestro programa al comienzo de éste.
- Sin embargo, Python solo requiere que la función esté definida antes de que sea usada.

3. Diseño de funciones

- **Parámetros y Argumentos**

```
def suma(a, b):
    c = a + b
    return c
```

Parámetros: a, b

```
>>> x = 8
>>> y = 4.5
>>> suma(x, y)
12.5
```

Argumentos: x, y

Argumentos: x, y

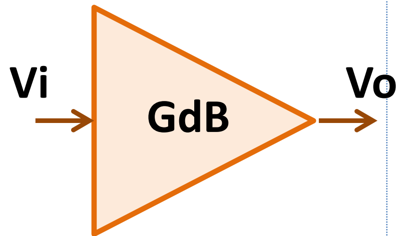
Parámetro:
Es el nombre usado en la definición de la función, a modo de variable, que se relaciona con el valor que se le pasa como ***argumento*** desde el programa principal u otra función. Dentro de la función será usado como variable local

Argumento
Valor pasado a una función cuando ésta es llamada. El valor se pasa al ***parámetro*** correspondiente de la función

3. Diseño de funciones

Ejemplo:

Ganancia GdB




```
from math import log10
def  ganancia_dB(x, y):
    """
     Calcula ganancia en dB: 20log(y/x)
    """
    gain = y/x
    dB = 20*log10(gain)
    return dB
Vi = 10
Vo = 1000
GdB = ganancia_dB(Vi, Vo)
print('Ganancia =', GdB,'dB')
```

- **Nombre** de la función: ganancia_dB, sus **parámetros** son x e y, la palabra reservada **return** indica que la función es **productiva *o fruitful function*** (devuelve un resultado).
- Al ***devolver*** un resultado de tipo dato simple, como float en esta función, lo pudiéramos usar en una expresión aritmética:

```
Ganancia = 40.0 dB
```

3. Diseño de funciones

- El estilo ***docstring*** que incluimos en la función (***string*** entre comillas triples)

```
"""
 Calcula ganancia en dB: 20log(y/x)
"""
```

sirve también para documentar en la consola de Python lo que hace la función.

- Usando help(nombre_de_la_funcion)

```
>>> help(ganancia_dB)
Help on function ganancia_dB in module __main__:
ganancia_dB(x, y)
    Calcula ganancia en dB: 20log(y/x)
```

3. Diseño de funciones

3.1 Paso de parámetros entre programa y funciones

- En el paso de los parámetros ***el orden es importante*** (ver excepciones en la sección de ***keyword arguments***)

```
from math import log10
def  ganancia_dB(x, y):
    """
     Calcula ganancia en dB: 20log(y/x)
    """
    return 20*log10(y/x)
y = 10
x = 1000
print('Ganancia =', ganancia_dB(y, x),'dB')
```

```
>>>
Ganancia = 40.0 dB
```

3. Diseño de funciones

3.2 Funciones productivas y funciones nulas (procedimientos)

- El siguiente ejemplo muestra los dos tipos de subprogramas típicos en lenguajes de programación: **funciones** (***productivas***) y **procedimientos** (***funciones nulas o void***):

```
def  f1(x):
    return (x/2 + 1)
def  f2(x):
    print(x/2 + 1)
```

```
>>> f1(5)
3.5
>>> f2(5)
3.5
>>> print(f1(10))
6.0
>>> print(f2(10))
6.0
None
>>> y = 4 + f1(6)     # f1(6) devuelve 4, el valor 8 se le asigna a y
>>> y = 4 + f2(6)     # f2(6) muestra el valor 4; pero errónea expresión
4.0
Traceback (most recent call last):   File "<pyshell#11>", line 1, in
<module>     y = 4 + f2(6)
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

3. Diseño de funciones

3.3 Consideraciones prácticas: uso de programa principal o no

```
def suma(a, b):
    return a+b

res = suma(3, 5)
print(res)
```

```
8
```

```
def suma(a, b):
    return a+b

if __name__ == "__main__":
    res = suma(3, 5)
    print(res)
```

```
8
```

Resultados **idénticos**, con estos matices:
Si se guarda la función en un fichero (por ejemplo *suma.py)* la opción de la derecha al ser ejecutada en el programa principal importa el fichero *suma.py* como un módulo, el nombre del módulo (__name__) será __main__ y se ejecuta lo que está en el if.
Pero si el módulo suma.py se importa desde otro programa, solo se ejecutan sus funciones (suma() en este caso) y no lo dentro del if.
La opción izquierda se ejecutaría todo tanto al ejecutarlo directamente o al llamarlo de un módulo

3. Diseño de funciones

3.4 Uso de DOCTEST para probar funciones

- Existe en Python el módulo **doctest** que incluye la función **testmod**
- **testmod** busca fragmentos de texto en el ***docstring*** como las sesiones interactivas de Python, y luego ejecuta esas sesiones para verificar que funcionan exactamente como se muestra.
- Por ejemplo, la función **`ganancia_dB`** se puede ejecutar varias veces en la consola de Python, o se puede prever qué valores debe dar:

```
>>> ganancia_dB(1,1000)
60.0
>>> ganancia_dB(10,1000)
40.0
>>> ganancia_dB(1,1)
0.0
>>> ganancia_dB(10,1)
-20.0
```

3. Diseño de funciones

3.4 Uso de DOCTEST para probar funciones

- La función incluye el string de prueba y la llamada a la función

```
from math import log10
def  ganancia_dB(x, y):
    """Calcula ganancia en dB: 20log(y/x)
    >>> ganancia_dB(1,1000)
    60.0
    >>> ganancia_dB(10,1000)
    40.0
    >>> ganancia_dB(1,1)
    0.0
    >>> ganancia_dB(10,1)
    -20.0
    """
    return 20*log10(y/x)

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

3. Diseño de funciones

3.4 Uso de DOCTEST para probar funciones

- Ejecutando el programa se escribe el resultado de evaluar la función con la sesión interactiva incluida en el ***docstring***:

```
>>>
 RESTART: D:\Informatica\P6\Amplificador_doctest.py
Trying:
    ganancia_dB(1,1000)
Expecting:
    60.0
ok
Trying:
    ganancia_dB(10,1000)
Expecting:
    40.0
ok
Trying:
    ganancia_dB(1,1)
Expecting:
    0.0
ok
Trying:
    ganancia_dB(10,1)
Expecting:
    -20.0
ok
1 items had no tests:
    __main__
1 items passed all tests:
   4 tests in __main__.ganancia_dB
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

3. Diseño de funciones

3.4 Uso de DOCTEST para probar funciones

- Ejemplo de función con error en el diseño (- en lugar de +)

```python
def suma(a, b):
    """Realiza la suma de dos números.

    Ejemplos:
    >>> suma(1, 2)
    3
    >>> suma(10, 10)
    20
    """
    return a-b


if __name__ == "__main__":
    res = suma(3, 4)
    print(res)
    import doctest
    doctest.testmod()
```

```
-1
**********************************************************
File "D:/Python3/suma.py", line 5, in __main__.suma
Failed example:
   suma(1, 2)
Expected:
   3
Got:
   -1
**********************************************************
File "D:/Python3/suma.py", line 7, in __main__.suma
Failed example:
   suma(10, 10)
Expected:
   20
Got:
   0
**********************************************************
1 items had failures:
  2 of   2 in __main__.suma
***Test Failed*** 2 failures.
```