Herramientas de programación en Python

1. Fundamentos de programación

1.5 Programación modular. Variables globales Tipos de Argumentos de Funciones, Recursividad, Espacio de nombres, Ámbitos.

Pedro Gomis

pedro.gomis@upc.edu

1. Funciones productivas y no productivas

- En Python hay funciones:
  - **Productivas** o ***fructíferas:*** **return** devuelven un resultado
  - **Nulas** (estériles o no productivas, en inglés: **void**): devuelven **None**. Pueden compararse con los ***procedures*** de otros lenguajes de programación.

```
def sumaNP(x, y):
    """ suma en función no productiva """
    print(x+y)
def suma(x, y):
    """ suma en función productiva con return"""
    return x + y
```

```
>>> sumaNP(3,7)
10
>>> suma(3, 7)
10
>>> print(sumaNP(3, 7))
10
None
```

```
>>> y = 2*suma(3, 7)
>>> y
20
>>> y = 2*sumaNP(3, 7)
10
Traceback y = 2*sumaNP(3, 7)
TypeError: unsupported operand type(s) for *: 'int'
and 'NoneType'
```

1. Funciones productivas y no productivas

- Otro ejemplo **funciones *productivas*** y **funciones *nulas o void***

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

Shell Python

```
>>> f1(5)
12.5
>>> f2(5)
12.5
>>> 4 + f1(5)
16.5
>>> 4 + f2(5)
12.5
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    4 + f2(5)
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

```
In [2]: f1(5)
Out[2]: 12.5

In [3]: f2(5)
12.5
```

Ipython* (Spyder)

* https://ipython.org/

1. Funciones productivas y no productivas

- La sentencia **pass**:

- Si se usa la declaración **pass** dentro de una función la operación es nula. Esto significa que cuando se ejecuta, no pasa nada. Es útil como marcador de posición en situaciones en las que se requiere una declaración por sintaxis, pero no es necesario ejecutar código.

```
def hagoNada():
    pass
```

```
>>> hagoNada()
```

- **pass** en una estructura. Ejemplo en **if**:

```
x = 25
if x < 20:
    pass
else:
    print(x)
```

```
25
```

2. Variables locales y globales

Variables locales

- En la función `ganancia_dB` las variables `gain` y `db` son **variables locales**, es decir, las usa solo esta función y al acabar su ejecución se **borran**.
- Los parámetros `x` e `y` también actúan como variables locales.

```
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')
```

2. Variables locales y globales

Variables globales

- Las funciones pueden incluir también **variables globales**. Se declaran dentro de la función precedidas por la palabra reservada **`global`**.
- Si una ***variable del programa principal*** tiene ***el mismo identificador*** que la definida como ***global dentro de una función***, entonces será modificada por cualquier asignación de la variable **`global`**.

```
def f(x):
    global a
    y = a*x
    a = 3
    return y

a = 7
print('Función:',f(4))
print('Valor de la variable global a:',a)
```

```
Función: 28
Valor de la variable global a: 3
>>>
```

Las variables globales se debe usar con **moderación**.

3. Valores de argumentos por omisión (default)

- Al diseñar funciones se puede prever que un argumento tenga un ***valor por omisión*** si el usuario lo omite al llamar la función. Por ejemplo, la función **ganancia_dB** puede considerar que el usuario introduzca o no el valor de **y**:

```
from math import log10
def  ganancia_dB(x, y=100):
    """
     Calcula ganancia en dB: 20log(y/x)
    """
    return 20*log10(y/x)
Vi = 10
print('Ganancia =', ganancia_dB(Vi),'dB')
print('Ganancia =', ganancia_dB(Vi+5, 50),'dB')
```

```
>>>
Ganancia = 20.0 dB
Ganancia = 10.457574905606752 dB
```

3. Valores de argumentos por omisión (default)

- Ejemplo de valor por omisión

```
def hola(nombre = "a todos"):
    """ Saluda a una persona """
    print("Hola " + nombre + "!")

hola("Gerard")
hola()
```

```
>>>
Hola Gerard!
Hola a todos!
```

3. Valores de argumentos por omisión (default)

Ejemplo. Arcotangente. El desarrollo en serie de Taylor de la función *arcotangente*, donde el resultado se halla en radianes, es:

$$arctang(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots$$

Diseñar una función `arctan(x)` que, dado un valor de x, devuelva la arcotangente en ***grados sexagesimales*** mediante la aproximación de la serie que acabe cuando un nuevo término aporte menos de un **épsilon** a la suma.

El resultado se devolverá redondeado a 2 cifras decimales. Usaremos un número máximo de iteraciones, por ejemplo, para i<10000.

Compara con la función `atan` del módulo `math` de Python

3. Valores de argumentos por omisión (default)

- **Arcotangente**. $arctang(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots$

```
from math import pi, atan
def arctan(x, epsilon=1e-9):
    """
    >>> arctan(1)
    45.0
    >>> round(atan(1)*180/pi, 2)
    45.0
    """
    n=1
    signo = 1
    term = x
    suma = x
    while abs(term)>epsilon and n<10000:
        n += 2
        signo=-signo
        term = signo*x**n/n
        suma += term
    return round(suma*180/pi,2)
```

4. Argumentos de palabra clave (*keyword arguments*)

- Cuando una **función** tenga **varios parámetros** con valores de argumento por omisión, éstos se pueden omitir pero también **pueden ser llamados en cualquier orden**, siempre que se use el **nombre del parámetro y su argumento asignado**

```
def  fkey(x, y=1, z=1, w=0):
    return (x + y)*(z + w)
print(fkey(2))                  # x=2, y=1, z=1, w=0
print(fkey(2, w = 4))           # x=2, y=1, z=1, w=4
print(fkey(1, z = 2, y = 3)) # x=1, y=3, z=2, w=0
```

```
>>>
3
15
8
```

- Los **argumentos** con el nombre del parámetro y su valor, del grupo de parámetros con valores por omisión, se colocan sin importar el orden en que se introducen.

5. Módulos: integrar funciones en una biblioteca

- Las **funciones** que hemos definido se pueden guardar en un **módulo** para ser reutilizadas cuando queramos.
- Un **módulo** será un ***fichero*** de extensión **.py** que contiene la definición de un grupo de funciones y otros valores. Representan una **biblioteca** de **funciones** de un determinado **tema**.
- Por ejemplo, supongamos que comenzamos a formar una biblioteca simple de **funciones de medidas** (área, perímetro) **geométricas**.

```
from math import pi
def AreaCirc(radio):
    return pi*radio**2
def PerimCirc(radio):
    return 2*pi*radio
def VolCilindro(radio, h):
    return h*pi*radio**2
def AreaRect(b, h):
    return b*h
def PerimRect(b, h):
    return 2*(b+h)
def VolOrtoed(b, h, a):
    return b*h*a
```

Guardamos en el fichero Geomet.py

5. Módulos: integrar funciones en una biblioteca

- Igual que los módulos internos y math, el módulo Geomet.py lo podemos importar para usar sus funciones y constantes. Por ejemplo,

```
from Geomet import pi, AreaCirc, AreaRect
print(AreaCirc(0.5))
print(AreaRect(3, 4))
```

```
>>>
0.7853975
12
```

- Se pueden importar todas las funciones y valores del módulo, en lugar de cada una de ellas, con *

```
from Geomet import *
```

- Se puede importar el módulo completo y usar los valores o funciones de éste:

```
import Geomet as G
x = G.pi*4
print(x)
print(G.AreaCirc(0.5))
```

6. Recursividad

- Una **estructura recursiva** puede contener otra del mismo tipo, es decir se integra por partes de sí misma.
- Por ejemplo, un ***objeto fractal**** se caracteriza por su **estructura** básica que **se repite a diferentes escalas**.
- En informática**, recursión** es un método de programar en la cual una **función se llama a sí misma** una o más veces en el cuerpo de la función.
- La condición más importante es que la función acabe y no se quede recurriendo interminablemente.
- El ejemplo más claro de una función recursiva es el cálculo del **factorial** (denotado por el símbolo**!**) de un **número natural**. El factorial ***se define en términos de sí mismo***:

$$n! = \begin{cases} 1 & si\ n = 0 \\ n * (n-1)! & si\ n > 0 \end{cases}$$

- Por ejemplo 4! es 4 veces 3!, que a su vez es 3 veces 2!, que a su vez es 2 veces 1!, que es 1 vez 0!, que es 1. Esto resulta en: 4! es igual a 4 por 3 por 2 por 1 y por 1, que es 24.

* Ejemplo: https://blog.goodaudience.com/fractals-and-recursion-in-python-d11d87fcf9cd

6. Recursividad

- Diseño de una función que calcule el **factorial** usando una composición ***iterativa de recorrido*** (NO recursiva):

```
def factorial_iter(n):
    if n == 0 :
        result = 1
    else:
        result = 1
        for i in range(1, n+1):
            result = result*i
    return result
```

Comprobar que esta versión es más compactada: si n es 0 o 1, no se ejecuta **range** (vacío) y **for** no itera

```
def factorial_iter(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result
```

```
>>> factorial_iter(0)
1
>>> factorial_iter(1)
1
>>> factorial_iter(5)
120
```

6. Recursividad

- La función **factorial** de forma **recursiva** es mucho más **simple**:

```
def factorial_r(n):
    if n == 0:
        return 1
    else:
        return n*factorial_r(n-1)
```

- Si usáramos los principios clásicos de la ***programación estructurada*** con la función con **un solo punto de salida** (otros lenguajes). Sería:

```
def factorial_r(n):
    if n == 0:
        result = 1
    else:
        result = n*factorial_r(n-1)
    return result
```

```
# También incluida en módulo math
from math import factorial
factorial(5)
```

```
>>> factorial_r(5)
120
```

6. Recursividad

Otra secuencia típica que puede ser calculada de forma **recursiva** es la formada por los números de **Fibonacci***: `1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …`

(*la versión moderna suele iniciarse en 0:* 0, 1, 1, 2, 3, 5, …)*

Los números de Fibonacci se pueden definir como:

$F_1 = 1$, $F_2 = 1$, $F_k = F_{k-1} + F_{k-2}$ (para k >2)

Fibonacci	1	1	2	3	5	8	13	21	34	55	89	144	…
k	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	…

Una función recursiva para calcular el número en la posición k de la secuencia:

```
def fibo(k):
    if k == 1 or k == 2:
        result = 1
    else:
        result = fibo(k-1) + fibo(k-2)
    return result
```

```
>>> fibo(8)
21
>>> fibo(10)
55
```

* En matemáticas modernas la secuencia empieza para k = 0. Ver: http://www.python-course.eu/recursive_functions.php

6. Recursividad

- **Problema**! Las funciones recursivas necesitan mucha más memoria para ejecutarse ya que se crean objetos a medida que se ejecutan.
- Muchos problemas sólo pueden resolverse de manera recursiva. Sin embargo, en aquellos en los que no es imprescindible se recomienda utilizar un método iterativo alternativo.

Cálculo de posición k de Fibonacci con iteraciones

```
def fiboI(k):
    if k == 1 or k == 2:
        result = 1
    else:
        a, b = 1, 1
        for i in range(k-1):
            a, b = b, a+b
        result = a
    return result
```

Comparamos tiempo (s) de cálculo con ambas opciones: fibo (recursiva) fiboI (iteraciones)

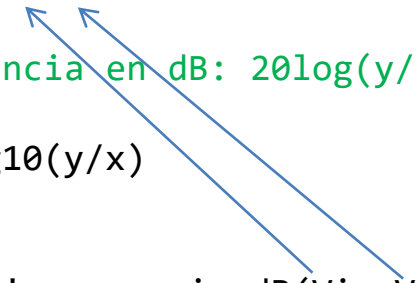
```
import time as t
for i in range(30,36):
    ini = t.time()
    print('k =',i,', F:',fibo(i),', fibo:',
          round(t.time()-ini,3), end=' ')
    ini=t.time()
    f=fiboI(i)
    print(', fiboI: %.15f' % (t.time()-ini))
```

Tiempos dependen de la velocidad del computador

```
k =  30 , F: 832040  , fibo: 0.292 , fiboI: 0.000000000000000
k =  31 , F: 1346269 , fibo: 0.471 , fiboI: 0.000000000000000
k =  32 , F: 2178309 , fibo: 0.764 , fiboI: 0.000000000000000
k =  33 , F: 3524578 , fibo: 1.232 , fiboI: 0.000000000000000
k =  34 , F: 5702887 , fibo: 2.008 , fiboI: 0.000000000000000
k =  35 , F: 9227465 , fibo: 3.234 , fiboI: 0.000000000000000
>>>
```

7. Paso de parámetros por valor y por referencia. Por objeto

```
from math import log10
def  ganancia_dB(x, y):
    """
     Calcula ganancia en dB: 20log(y/x)
    """
    return 20*log10(y/x)
Vi = 10
V0 = 1000
print('Ganancia =', ganancia_dB(Vi, V0),'dB')
```



Paso de parámetros por valor

Un ***paso de parámetros por valor*** significa que el valor del argumento entra por el parámetro a la función, pero si el parámetro es modificado dentro de la función, su valor no se referencia a la variable del argumento. Es decir, esa variable no cambia en el programa principal

- Los ***parámetros x*** e ***y*** de la función reciben los valores de los ***argumentos*** **Vi**, **Vo** (como si fuera una asignación) y entran a la función.
- Los parámetros de las funciones en Python se definen, como en el caso de las variables, con un identificador válido.
- Por ahora diseñamos funciones que usan solo parámetros de tipos de datos simples (***int***, ***float***, ***bool***) o ***strings***.
- Estos tipos de datos no permiten que alguna acción modifique sus valores (es decir son inmutables).
- Esto quiere decir que ***el paso de parámetros es por valor*** siempre en los datos inmutables.

7. Paso de parámetros por valor y por referencia. Por objeto

Paso de parámetros por referencia

- Pero, si el argumento contiene variables **mutables**, como las listas, y la función modifica el valor de las variables, entonces el nuevo valor **sí** se **referencia** al **objeto** de la variable que se usó de argumento, tomando este nuevo valor. Equivale a **paso de parámetros por referencia**.
- Esta referencia se hace inmediatamente, sin esperar que acabe la función.

```
def FuncSumLstSqr(lst):
    for i in range(len(lst)):
        lst[i] = lst[i]**2
    return sum(lst)

lista = [1,2,3,4,5]
FuncSumLstSqr(lista)
```

En general, en Python se **pasa por objetos**

```
In [4]: lista = [1,2,3,4,5]
In [5]: FuncSumLstSqr(lista)
Out[5]: 55
In [6]: lista
Out[6]: [1, 4, 9, 16, 25]
```

`lista` ha sido modificada

Lectura recomendada: https://realpython.com/defining-your-own-python-function/#argument-passing

8. Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

- Las definiciones de clases, objetos, funciones en Python utilizan algunos trucos con los espacios de nombres, que son interesantes para saber cómo funcionan los **alcances** o **ámbitos** (***scope***) y **espacios de nombres** (***namespace***) en los programas.

Espacio de nombres (*namespace*)

- ***Namespace***: Un espacio de nombres (también llamado contexto) es un sistema de nombres o identificadores para hacer que éstos sean únicos y así evitar ambigüedades. Es una relación de nombres a objetos.
- Por ejemplo, la estructura de **directorios** de los sistemas de **archivos**. Se puede usar el ***mismo nombre*** de ***archivo*** en ***diferentes directorios***, a los archivos se puede acceder de manera única a través de los nombres de ruta (*pathname*).
- Un **identificador** definido en un espacio de nombres está asociado con ese espacio de nombres. De esta manera, el mismo **identificador** puede definirse independientemente en múltiples espacios de nombres. (Como los mismos nombres de archivo en diferentes directorios).

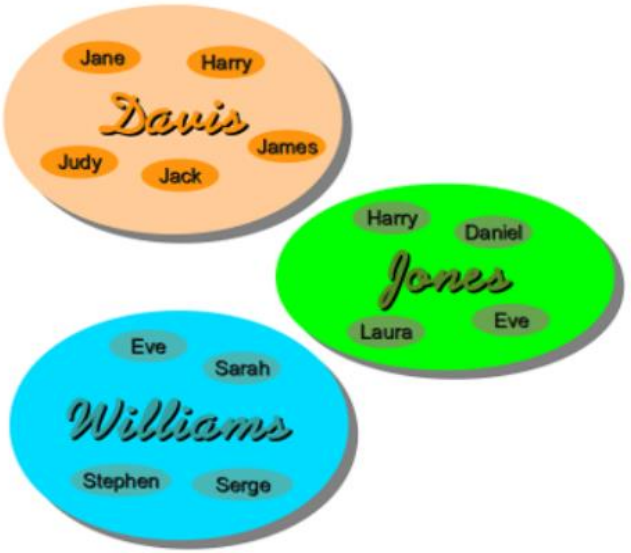
8. Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Espacio de nombres (*namespace*)

Los espacios de nombres en Python se implementan como **diccionarios**: se definen mediante una asignación de nombres (las **claves** del diccionario) a los objetos (los **valores**).

Algunos espacios de nombres en Python:

- Nombres globales de un módulo
- Nombres locales en una invocación de función o método
- Nombres incorporados (built-in): este espacio de nombres contiene funciones built-in (por ejemplo, abs (), int (), max(), …)



Por *ejemplo*, dos módulos diferentes pueden tener ambos definidos una función `maximizar` sin confusión; los usuarios de los módulos deben usar el nombre del módulo como prefijo: `modulo1.maximizar`, `modulo2.maximizar`.

8. Espacios de nombres (***namespace***) y Ámbitos (***Scope***)

Espacio de nombres (***namespace***)

En funciones

Cuando se llama a una función, se crea un espacio de nombres local para esta función. Este espacio de nombres se borra si la función finaliza (por ejemplo, return).

***Scope* (Ámbito)**

Un **ámbito** (*scope*) es una región de un programa donde se puede acceder directamente a un espacio de nombres, es decir, sin usar un prefijo de espacio de nombres.

En otras palabras: el ámbito de un nombre es el área de un programa donde este nombre puede usarse sin ambigüedades, por ejemplo, dentro de una función.

8. Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Ejemplo

Ámbito local

```
def myfunc():
    x = 300
    print(x)

myfunc()
```

```
300
```

```
def myfunc():
    x = 300
    def mi_func_inter():
        print(x)
    mi_func_inter()

myfunc()
```

```
300
```

Ámbito global

```
x = 300

def myfunc():
    x = 200
    print(x)

myfunc()

print(x)
```

```
200
300
```

Ámbito global

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

```
200
```

8. Espacios de nombres (*namespace*) y Ámbitos (*Scope*)

Ejemplo

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
scope_test()
print("In global scope:", spam)
```

Ejemplo que muestra cómo hacer referencia a distintos ámbitos y espacios de nombres, y cómo las declaraciones **global** y **nonlocal** afectan la asignación de variables

Notar cómo la asignación local (default) no cambió la vinculación de spam de scope_test(). La asignación nonlocal cambió la vinculación de spam de scope_test(), y la asignación global cambió la vinculación a nivel de módulo.

Las variables **nonlocal** se usan para trabajar con variables dentro de funciones anidadas, donde la variable no debe pertenecer a la función interna.

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```