1 Funciones en Python

En el lenguaje Python se pueden construir funciones para ser usadas reiteradas veces. Las funciones en Python tienen que estar definidas y por lo tanto cargadas antes de ser usadas, recordar que Python no es un lenguaje compilado. Por eso es muy común que todo código comience con la definición de varias funciones o con una carga de librerías que contienen esas funciones.

Cómo en otros lenguajes la idea de función es escribir una sóla vez, cualquier estructura que se repita muchas veces en un cálculo. Pero esa idea a ido evolucionando a convertirse en un código que desarrolla una tarea en particular y que puede guardarse en forma separada y luego reutilizarse en otros programas. La idea entonces es que en el caso de resolver un problema en particular, guardar el código de las funciones utilizadas y que esta solución de software quede en disponibilidad para que uno u otros puedan acceder a ese código y reutilizarlo.

El éxito de Python se debe entonces a la existencia de una cantidad de librerias especializadas en una cuestión particular que contienen funciones muy útiles sobre ese tema.

Para definir una función se usa la orden **def** seguido de su nombre y entre paréntesis las variables o los datos que se ingresan a la función. A estos últimos también se los llama argumentos. Al final del paréntesis tiene que ir el símbolo **:** para indicar al intérprete Python que las líneas siguientes con una sangría similar (un bloque) pertenecen a la función.

La sangría, al igual que en el caso de las estructuras de control, determina que órdenes son de esa función en particular, es decir determina el bloque de órdenes propias de la función. Como dentro de una función puede haber estructuras de control internas, estas tendrán más espacios de sangría que las que se indican como sentencias que son parte del bloque. Pero eso no es un problema, se siguen considerando dentro del bloque que forma la función.

Las funciónes suelen terminar con la orden **return**, que puede o no llevar los valores resultantes del cálculo, los cuales serán enviados al lugar donde la función fue llamada. La orden **return** es optativa, puede incluso no estar.

Veamos un ejemplo de una función que eleva al cubo valor que se ingresa:

[1]:
```
def cubo(x):
    print(x**3)
    return

cubo(5)
```

```
125
```

[2]:
```
# pero podría no haber puesto el return

def cubo(x):
    print(x**3)

cubo(5)
```

```
125
```

Notar que la función se dio por finalizada porque terminó la sangría, y no se usó la orden return. Este ejemplo demuestra la importancia de la sangría en Python para determinar inicio y fin de bloques de código.

Pero la pregunta es: ¿Devolvió la función un resultado?

[3]:
```
def func_cubo(x):
    """
    Retorno el cubo de un número.
    A este comentario se la llama docstring y puede accederse desde el
    notebook como una ayuda para recordar detalles del uso de la función
    como por ejemplo que variables ingresan y en que orden.
    Note las 3 comillas que se usan para indicarlo y cerrarlo
    """
    return(x**3)    # <-- El return puede hacer cuentas!!!

a = func_cubo(3)

print(a)
print("")

# pido el "help" para recordar detalles de la función y me
# devuelve el texto del docstring

help(func_cubo)
```

```
27

Help on function func_cubo in module __main__:

func_cubo(x)
    Retorno el cubo de un número.
    A este comentario se la llama docstring y puede accederse desde el
    notebook como una ayuda para recordar detalles del uso de la función
    como por ejemplo que variables ingresan y en que orden.
    Note las 3 comillas que se usan para indicarlo y cerrarlo
```

[4]:
```
# Otra manera hacer shift tab dentro del parentesis,
# aunque en algunas computadoras no funciona
# ¿Lo hace en la tuya?

func_cubo(3)
```

[4]: 27

Como en otros lenguajes puedo poner en los argumentos de la función variables o números

[5]:
```
a=3
print(a)

print(func_cubo(a))
```

```
3
27
```

Estos argumentos que hemos usado hasta ahora son requeridos y son obligatorios, ya que así están en la definición de la función. Debe llamarse a la función con estos argumentos cargados con valores numéricos o variables que los contengan. La falta de uno de estos argumentos en una llamada genera un error. Debido a que su orden importa se suelen llamar argumentos de posición o **positional arguments**.

Pero existen otros argumentos llamados de teclado o **keyword arguments** donde se indica un valor default (default = predeterminado) por si no se lo indica en el momento que se llama la función.

[6]:
```
def func_rara(x, y, z, a=0, b=1):
    """
    Esta función tiene 5 argumentos, pero dos de ellos
    tienen valores por default (entonces no son obligatorios)
    """
    return a + b * (x**2 + y**2 + z**2)**0.5


D = func_rara(3, 4, 5)
print(D)
```

```
7.0710678118654755
```

[7]:
```
E = func_rara(3, 4, 5, 10, 100)
print(E)
```

```
717.1067811865476
```

Veamos un ejemplo que es muy bueno para mostrar el uso de este tipo de argumentos.

Supongamos que queremos hacer un programa que pase medidas hechas en el sistema imperial, es decir pies y pulgadas a centímetros. la medida puede estar en pies, o en pulgadas o en los dos sistemas.

Recorar que 1 pulgada = 2.54 cm y 1 Pie = 12 pulgadas

[8]:
```
def cm(pies=0, pulgadas=0):
    pulgadas_a_cm = pulgadas * 2.54
    pies_a_cm = pies * 12 * 2.54
    return pulgadas_a_cm + pies_a_cm
```

En esta función pies y pulgadas tienen ya prefijado valores de inicio, es decir un default que es en ambos casos un “0”.

Veamos como puede correr esta función y aprovechar los valores default de las argumentos de teclado.

[9]:
```
# Sólo tengo un valor en pies y nada en pulgadas

a = cm(pies=20)
print("20 pies son: ",a)

# Sólo tengo un valor en pulgadas y nada en pies

b = cm(pulgadas=22.5)
print("22 pulgadas son: ",b)

#Tengo parte de la medida en pies y otra parte en pulgadas

c = cm(pies=5, pulgadas=3.5)
print("5 pies y 3.5 pulgadas son: ",c)

# E incluso puedo invertir los argumentos y lo puedo hacer porque
# tiene nombres las variables

d = cm(pulgadas=3.5,pies=5)
print("3.5 pulgadas y 5 pies on: ",d)
```

```
20 pies son:  609.6
22 pulgadas son:  57.15
5 pies y 3.5 pulgadas son:  161.29000000000002
3.5 pulgadas y 5 pies on:  161.29000000000002
```

Puedo mezclar argumentos de ambos tipos, pero los de tipo “keyword” tienen que ir al final.

Pero estos últimos no tiene un orden si los escribo como **variable=dato**

También puedo asignar la función a una variable con cierto nombre y utilizar ese nombre para el cálculo. Veamos:

[10]:
```
X = func_cubo

print(X(5))
```

```
125
```

Aunque lo que sucedió a que el objeto que contiene la función tiene ahora más de un nombre. Es decir, no se copió.

2 *args y **kargs

Veamos el caso de una facilidad que tiene Python de hacer variable la cantidad de argumentos tanto los posicionales como los argumentos “Keywords” que envio a una función.

2.1 Caso de una cantidad arbitraria de argumentos posicionales (*args)

Si por ejemplo realizo un programa que utiliza una función que calcula el promedio por ejemplo para 3 números, lo haría de la siguente manera: Es decir le envio 3 argumentos

[11]:
```
def prom(x,y,z):
    prom = (x+y+z)/3
    return prom

print(prom(2,3,4))
```

```
3.0
```

De esta manera funcionaría, pero no me serviría para promediar más de 3 números o menos de 2. Existe una manera de universalizar para que el programa me sirva para una cantidad no determina de números. Es decir, que yo pueda mandar una cantidad de arguemtos posicionales arbitraria.

Para realizar esta tarea tengo que indicar que la cantidad de argumentos no se conoce, sólo se sabrá al momento de la llamada a la función. Para eso uso ***args** como argumento al momento de la defición de la función.

Note el uso del símbolo * para indicar que se trata de una cantidad aleatoria de argumentos que se guardarán en la tupla con nombre **args**

Por ejemplo:

[12]:
```
# función con una cantidad variable de argumentos
def prom(*args):
    sum = 0
    for i in args:
        # calculo la suma total
        sum = sum + i
    # calculo el promedio
    promedio = sum / len(args)  # en vez len() podría haber puesto un contador,
                                # pero usar len() es más rápido.
    print('Promedio =', promedio)


prom(156, 23, 23)
print()
prom(2,3,4,5,6)
```

```
Promedio = 67.33333333333333

Promedio = 4.0
```

2.2 Caso de una cantidad arbitraria de argumentos de teclado (**kwargs)

Si quisiera hacer la misma operación pero con una cantidad de argumentos de teclado, tendría que usar como argumento de la función, por ejemplo, el nombre ****kwargs**. El doble símbolo ** indica que se trata de argumentos de teclado. Los argumentos pasarán entonces a la función la cual los verá internamente

como un diccionario. Por lo cual, los datos serán accesibles a través de un sistema llave-dato. Es decir para una llave determinada corresponde cierto dato.

Ejemplo:

[13]:
```
def porcentaje(**kwargs):
    sum = 0
    for sub in kwargs:
        # get argument name
        sub_nombre = sub
        # get argument value
        sub_nota = kwargs[sub]
        print(sub_nombre, "=", sub_nota)

# envio varios arguemtnos de teclado
porcentaje(matemática=56, física=61, química=73)
print()
```

```
matemática = 56
física = 61
química = 73
```

Por lo cual debemos considerar que hay 4 tipos de argumentos en la llamada de una función en Python:

- Argumentos default (los que indico al construir la función)
- Argumentos posicionales (se indentifican con la variable por la posición)
- Argumentos de Teclado (Son argumentos en los cuales modifico los valores indicados en el default)
- Argumentos que tienen una cantidad variable de elementos (*args y **kwargs)

Veamos ejemplos:

[14]:
```
def add(a,b=5,c=10):
    return (a+b+c)
```

Y podría hacer:

Por lo cual ahora los argumentos son posicionales, es decir identifico cuál es cuál por su posición.

O podría hacer lo siguiente:

[15]:
```
print(add(b=24,c=5,a=12))
```

```
41
```

Con lo cual ahora son argumentos de teclado, donde cada variable es indicada por nombre y valor

2.2.1 Reglas para los argumentos

- Los argumentos default deben seguir después de los argumentos que no son default
- Los argumentos de teclado deben seguir después de los aguemtnos posicionales.

- Los argumentos de teclado deben indicar un nombre de una variable declarada en la definición de la función. Su orden no es importante.
- Ningúno de los argumentos puede recibir más de un valor
- Los argumentos default son opcionales al momento de la llamada a la función.

Las funciones pueden devolver más de un resultado. En cierta manera violando la idea que uno ha aprendido en análisis matemático. Cuando son varios los resultados este es retornado en forma de tupla.

Por ejemplo:

[16]:
```
def cuadrado_cubo(x):
    x2 = x*x
    x3 = x2*x
    return(x2,x3)

print("Para x=3",cuadrado_cubo(3))
print("Para x=4",cuadrado_cubo(4))
print("Para x=5",cuadrado_cubo(5))
```

```
Para x=3 (9, 27)
Para x=4 (16, 64)
Para x=5 (25, 125)
```

Note que los resultados están englobados por ( ) indicando que es una tupla.

La orden **dir** sirve para listar las funciones cargadas en la memoria. En esa lista habrá funciones del sistema y las que agregué en las celdas anteriores.

[17]:
```
print(dir())
```

```
['D', 'E', 'In', 'Out', 'X', '_', '_4', '__', '___', '__builtin__',
'__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'_dh', '_i', '_i1', '_i10', '_i11', '_i12', '_i13', '_i14', '_i15', '_i16',
'_i17', '_i2', '_i3', '_i4', '_i5', '_i6', '_i7', '_i8', '_i9', '_ih', '_ii',
'_iii', '_oh', 'a', 'add', 'b', 'c', 'cm', 'cuadrado_cubo', 'cubo', 'd', 'exit',
'func_cubo', 'func_rara', 'get_ipython', 'porcentaje', 'prom', 'quit']
```

Puede verse que las funciones cubo, func_cubo, func_rara y X que creamos en este notebook hasta ahora, están como funciones en el sistema Python que estamos corriendo.

También podría devolver más de un valor. La construcción sería de esta forma, ya que la orden return debería indicar todos los valores a devolver

```
def algo():
    ...
    return(a,b)

x,y=algo()
```

3 Función Lambda (Lambda expression)

La función lambda es para crear funciones de una línea, y son equivalentes en todos sus aspectos a la función sentencia de Fortran.

En Python se las considera como funciones sin nombre, pero que tienen mucha utilidad en algunos casos.

por ejemplo, si defino

[18]:
```
lambda x: x**2
```

[18]:
```
<function __main__.<lambda>(x)>
```

El sistema me dice que es una función, pero no tiene nombre. Para poder usarla tengo que asginarla a una variable

[19]:
```
f = lambda x: x**2
# si la llamo con un argumento
print(f(2))
```

```
4
```

Funciona!!!

Y se la puede construir con muchas entradas de datos, no solo una. Por ejemplo:

[20]:
```
J = lambda x, y, z: (x**2 + y**2 + z**2)**0.5
J(1,2,3)
```

[20]:
```
3.7416573867739413
```

Entonces la forma general para crear una expresión Lambda sería:

lambda variables: ***expresión matematica en función de las variables que se ingresaron***

Y el caso más extremo sería ni siquiera darle una variable

[21]:
```
f1 = lambda: "¿Para qué sirvo?"

print(f1())
```

```
¿Para qué sirvo?
```

4 Recursividad

La funciones en Python son recursivas, veamos el ejemplo del factorial

$n! = n(n-1)! = n(n-1)(n-2)! = n(n-1)(n-2)....1$

[22]:
```
def fact(n):
    if n <= 0:
        return 1
```

```
    return n*fact(n-1)


print(fact(5))
print(fact(20))
print(fact(10))
```

```
120
2432902008176640000
3628800
```

5 Funciones que están en archivos

5.0.1 Quiero tener en un archivo que se llama ex1, la función siguiente

def f1(x):

```
"""
Este es un ejemplo de la función que devuelve: x**2    - parameter: x
"""
return x**2
```

Lo voy a crear con la celda siguente, ya que al correr la orden: %%writefile ex1.py graba la celda a un archivo

[23]:
```
%%writefile ex1.py
# Lo que está en esta celda se salvará a un archivo con nombre "ex1.py"
def f1(x):
    """
    Este es un ejemplo de la función que devuelve: x**2
    y parámetro es x
    """
    return x**2
```

```
Overwriting ex1.py
```

[24]:
```
import ex1

# Esto importa un archivo que se llama ex1.py del directorio
# o de los directorios que indique poniendo el camino.
# El hecho de que muchos usuarios escriben y comparten funciones
# es lo que impulsó a Python en el área científica.

print(ex1.f1(4))
```

```
16
```

[25]:
```
from ex1 import f1
print(f1(3))

#Cargo de ex1 SÓLO la función f1
```

9

[26]:
```
from ex1 import * # Aquí cargo todas las funciones. Lo cual es una
                  # mala idea se cargan todas las funciones con su
                  # nombre y pueden sobreescribir otros nombres
                  # que estaba usando para las funciones

print(f1(4))
```

16

[27]:
```
import ex1 as tt
print(tt.f1(10))

# Ahora cargo todas las funciones de ex1, pero las tengo
# que llamar con el prefijo tt en el nombre, es decir la
# función f1() es ahora tt.f1
```

100

6 Importando Librerías

No todas las funciones de Python que lo han hecho tan famoso y requerido están en el lenguaje, sino que muchas de las más usadas son externas a Python. Para usar estas funciones hay que "importarlas". Por lo cual hay que indicarle a nuestro programa que las cargue y de esta manera adquiera nuevas capacidades.

Incluso esto es válido para el caso de que se necesite usar funciones matemáticas trascedentes, ya que el Python en su versión original no las trae cargadas. Por ejemplo, Python no tiene funciones intrínsecas de matemática como el Fortran.

Por ejemplo:

calcular: print(sin(3.)) daría error, la funciones trigonoméricas no están construidas por default dentro de Python.

[28]:
```
#print(sin(3.))  # si se eliminina el # y se corre la celda da error.
```

Pero en cambio si importo la funciones matemáticas que están en la librería **math** ya lo puedo usar. Lo haríamos de la siguiente manera:

[29]:
```
import math
print(math.sin(3.))
```

```
0.1411200080598672
```

[30]:
```
help(math.sin)
```

```
Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
```

[31]:
```
print(math.pi)
```

```
3.141592653589793
```