T06_Funciones_y_Documentacion

January 29, 2021

1 Python de cero a experto

Autor: Luis Miguel de la Cruz Salas

1.1 Pythonico es más bonito

1.1.1 Funciones

Las funciones son la primera forma de estructurar un programa. Esto nos lleva al paradigma de programación estructurada, junto con las construcciones de control de flujo. Las funciones nos permiten agrupar y reutilizar líneas de código.

La sintáxis es:

```
def nombre de la función(parm1,parm2,...):
    bloque de código
    return resultado
```

[1]:
```
# La siguiente función calcula la secuencia de Fibonacci
def fib(n):  # La función se llama fib y recibe el parámetro n
    a, b = 0, 1
    while a < n:
        print(a, end=',')
        a, b = b, a+b
```

[2]:
```
fib(10) # ejecutamos la función fib con el argumento 10
```

```
0,1,1,2,3,5,8,
```

[5]:
```
type(fib)
```

[5]:
```
function
```

Le podemos poner otro nombre a la función

[6]:
```
Fibonacci = fib
```

[7]:
```
Fibonacci(200)
```

```
0,1,1,2,3,5,8,13,21,34,55,89,144,
```

[8]:
```
type(Fibonacci)
```

[8]:
```
function
```

[9]:
```
print(id(fib), id(Fibonacci))
```

```
140569922779744 140569922779744
```

1.1.2 Ámbitos

Las funciones (y otros operadores también), crean su propio ámbito, de tal manera que las etiquetas declaradas dentro de funciones son locales.

[10]:
```
a = 20 # Objeto global etiquetado con a
def f():
    a = 21 # Objeto local etiquetado con a
    return a
```

[11]:
```
print(a)
```

```
20
```

[12]:
```
print(f())
```

```
21
```

Para usar el objeto global dentro de la función debemos usar `global`

[13]:
```
a = 20
def f():
    global a
    return a
```

[14]:
```
print(a)
```

```
20
```

[15]:
```
print(f())
```

```
20
```

1.1.3 Retorno de una función

La palabra reservada `return` regresa un objeto, que en principio contiene el resultado de las operaciones realizadas por la función.

[16]:
```
# Función que calcula la posición y velocidad en el tiro vertical de un objeto.
def verticalThrow(t,v0):
```

```
    g = 9.81 # [m / s**2]
    y = v0 * t - 0.5 * g * t**2
    v = v0 - g * t
    return (y, v)  # regresa la posición [m] y la velocidad [m/s]
```

[17]:
```
t = 2.0   # [s]
v0 = 20 # [m/s]
verticalThrow(t, v0)
```

[17]:
```
(20.38, 0.379999999999999)
```

[18]:
```
resultado = verticalThrow(t, v0)
```

[19]:
```
print(resultado)
```

```
(20.38, 0.379999999999999)
```

1.1.4 Parámetros por omisión

[20]:
```
# Función que calcula la posición y velocidad en el tiro vertical de un objeto.
def verticalThrow(t, v0 = 20):
    g = 9.81 # [m / s**2]
    y = v0 * t - 0.5 * g * t**2
    v = v0 - g * t
    return y, v  # regresa la posición [m] y la velocidad [m/s]
```

[21]:
```
pos, vel = verticalThrow(t)
```

[22]:
```
print(pos, vel)
```

```
20.38 0.379999999999999
```

1.1.5 Argumentos posicionales y `keyword`

Un argumento es el valor que se le pasa a una función cuando se llama. Hay dos tipos de argumentos:

- *Positional argument* : un argumento que no es precedido por un identificador, o pasado en una tupla precedido por `*`:

  ```
  verticalThrow(3, 50)
  verticalThrow(*(3, 50))
  ```

- *Keyword argument* : un argumento precedido por un identificador en la llamada de una función (o que se pasa por valor en un diccionario precedido por `**`):

  ```
  verticalThrow(t=3, v0=50)
  verticalThrow(**{'t': 3, 'v0': 50})
  ```

[23]:
```
verticalThrow(3,50)
```

[23]: (105.85499999999999, 20.57)

[24]:
```
verticalThrow(*(3,50))
```

[24]: (105.85499999999999, 20.57)

[25]:
```
verticalThrow(t=3,v0=50)
```

[25]: (105.85499999999999, 20.57)

[26]:
```
verticalThrow(**{'t':3,'v0':50})
```

[26]: (105.85499999999999, 20.57)

[27]:
```
verticalThrow(v0=50,t=3)
```

[27]: (105.85499999999999, 20.57)

1.1.6 Número variable de parámetros

[28]:
```
# *args: número variable de Positional arguments empacados en una tupla
# *kwargs: número variable de Keyword arguments empacados en un diccionario
def parametrosVariables(*args, **kwargs):
    print('args es una tupla : ', args)
    print('kwargs es un diccionario: ', kwargs)
    print(set(kwargs))

parametrosVariables('one', 'two','three', 'four', a = 4,  x=1, y=2, z=3,␣
 ↪w=[1,2,2])
```

```
args es una tupla :  ('one', 'two', 'three', 'four')
kwargs es un diccionario:  {'a': 4, 'x': 1, 'y': 2, 'z': 3, 'w': [1, 2, 2]}
{'y', 'z', 'w', 'a', 'x'}
```

[29]:
```
def funcion_kargs(**argumentos):
    for key, val in argumentos.items():
        print(f" key = {key} : value = {val}")
```

[30]:
```
funcion_kargs(nombre = 'Luis', apellido='de la Cruz', edad=15, peso=80.5 )
```

```
 key = nombre : value = Luis
 key = apellido : value = de la Cruz
 key = edad : value = 15
 key = peso : value = 80.5
```

[31]:
```
mi_dicc = {'nombre':'Luis', 'apellido':'de la Cruz', 'edad':15, 'peso':80.5}
```

[32]:
```
funcion_kargs(**mi_dicc)
```

```
key = nombre : value = Luis
key = apellido : value = de la Cruz
key = edad : value = 15
key = peso : value = 80.5
```

1.1.7 Funciones como parámetros de otras funciones

[33]:
```
def g():
    print("Iniciando la función 'g()'")

def func(f):
    print("Iniciando la función 'func()'")
    print("Ejecución de la función 'f()', nombre real '" + f.__name__ + "()'")
    f()

func(g)
```

```
Iniciando la función 'func()'
Ejecución de la función 'f()', nombre real 'g()'
Iniciando la función 'g()'
```

[34]:
```
import math

def integra(func,a,b,N):
    print(f"Integral de {func.__name__} en el intervalo ({a},{b}) usando {N}␣
 ↪puntos")
    h = (b - a) / N
    resultado = 0
    x = [a + h*i for i in range(N+1)]
    for xi in x:
        resultado += func(xi) * h
    return resultado

print(integra(math.sin, 1,2,5))
print(integra(math.cos, 2,3,10))
```

```
Integral de sin en el intervalo (1,2) usando 5 puntos
1.128335692301793
Integral de cos en el intervalo (2,3) usando 10 puntos
-0.8378441308575052
```

1.1.8 Funciones que regresan otra función

[35]:
```
def funcionPadre(n):

    def funcionHijo1():
        return "Resultado de funcionHijo1()"

    def funcionHijo2():
        return "Resultado de funcionHijo2()"

    if n == 10:
        return funcionHijo1
    else:
        return funcionHijo2

f1 = funcionPadre(10)
f2 = funcionPadre(11)

print(f1())
print(f2())

#print(f1)
#print(f2)
```

```
Resultado de funcionHijo1()
Resultado de funcionHijo2()
```

Ejemplo 1. Implementar una fábrica de polinomios de segundo grado:

$$p(x) = ax^2 + bx + c$$

[36]:
```
def polinomio(a, b, c):

    def polSegundoGrado(x):
        return a * x**2 + b * x + c

    return polSegundoGrado

p1 = polinomio(2, 3, -1) # 2x^2 + 3x - 1
p2 = polinomio(-1, 2, 1) # -x^2 + 2x + 1

for x in range(-2, 2, 1):
    print(f'x = {x:3d} \t p1(x) = {p1(x):3d} \t p2(x) = {p2(x):3d}')
```

```
x =  -2          p1(x) =   1     p2(x) =  -7
x =  -1          p1(x) =  -2     p2(x) =  -2
```

```
x =   0          p1(x) =  -1     p2(x) =   1
x =   1          p1(x) =   4     p2(x) =   2
```

Ejemplo 2. Implementar una fábrica de polinomios de cualquier grado:

$$\sum_{k=0}^{n} a_k x^k = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

[37]:
```
def polinomioFactory(*coeficientes):

    def polinomio(x):
        res = 0
        for i, coef in enumerate(coeficientes):
            res += coef * x ** i
        return res

    return polinomio

p1 = polinomioFactory(5)           # a_0 = 5
p2 = polinomioFactory(2, 4)        # 4 x + 2
p3 = polinomioFactory(-1, 2, 1)    # x^2 + 2x - 1
p4 = polinomioFactory(0, 3, -1, 1) # x^3 - x^2 + 3x + 0

for x in range(-2, 2, 1):
    print(f'x = {x:3d} \t p1(x) = {p1(x):3d} \t p2(x) = {p2(x):3d} \t p3(x) =␣
 ↪{p3(x):3d} \t p4(x) = {p4(x):3d}')
```

```
x =  -2          p1(x) =   5     p2(x) =  -6     p3(x) =  -1     p4(x) = -18
x =  -1          p1(x) =   5     p2(x) =  -2     p3(x) =  -2     p4(x) =  -5
x =   0          p1(x) =   5     p2(x) =   2     p3(x) =  -1     p4(x) =   0
x =   1          p1(x) =   5     p2(x) =   6     p3(x) =   2     p4(x) =   3
```

1.1.9 Documentación con *docstring*

Python ofrece dos tipos básicos de comentarios para documentar el código:

1. Lineal. Este tipo de comentarios se llevan a cabo utilizando el símbolo especial **#**. El intérprete de Python sabrá que todo lo que sigue delante de este símbolo es un comentario y por lo tanto no se toma en cuenta en la ejecución:

```
a = 10 # Este es un comentario
```

2. Docstrings En programación, un *docstring* es una cadena de caracteres embedidas en el código fuente, similares a un comentario, para documentar un segmento de código específico. A diferencia de los comentarios tradicionales, las docstrings no se quitan del código cuando es analizado, sino que son retenidas a través de la ejecución del programa. Esto permite al programador inspeccionar esos comentarios en tiempo de ejecución, por ejemplo como un sistema de ayuda interactivo o como metadatos. En Python se utilizan las triples comillas para definir un *docstring*.

```
def funcion(x):
    '''
    Esta es una descripción de la función ...
    '''

def foo(y):
    """
    También de esta manera se puede definir una docstring
    """
```

[38]:
```
def suma(a,b):
    '''
    Esta función calcula  la suma de los parámetros a y b.
    Regresa el resultado de la suma
    '''
    return a + b
```

[39]:
```
suma
```

[39]:
```
<function __main__.suma(a, b)>
```

[40]:
```
# En numpy se usa la siguiente definición de docstrings
def suma(a,b):
    '''
    Calcula la suma de los dos parámetros a y b.

    Args:
        a: int Numero a sumar
        b: int Numero a sumar
    Return:
        c: int Suma del numero a y b
    '''
    c = a + b
    return c
```

[41]:
```
suma
```

[41]:
```
<function __main__.suma(a, b)>
```

[ ]:
```

```