Python - Funciones definidas por el usuario

Sumario

Funciones definidas por el usuario

Una función, es la forma de agrupar expresiones y sentencias (algoritmos) que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas.

Definición de funciones

Para definir una función se utiliza la instrucción **def** más un nombre de función descriptivo, seguido luego de paréntesis de apertura y cierre.

- Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone irá identado con 4 espacios.

```
def funcion1():
    # Aquí va el cuerpo de la función
```

- Una función no es ejecutada hasta que no es invocada. Para invocarla simplemente se le llama por su nombre:

```
funcion1()
```

- Cuando una función realice un **retorno de datos**, por ejemplo, éstos pueden ser asignados a una variable:

```
def funcion1():
    return "Hola mundo"

frase = funcion1()

print(frase)
```

- Se puede poner una ayuda en la función. Se hace añadiendo una "cadena literal":

```
def funcion1():
    """Función saludo. Devuelve 'Hola mundo'"""
    return "Hola mundo"

ayuda = funcion1.__doc__
print(ayuda)
```

Paso de parámetros a funciones

Un parámetro es un valor que la función espera recibir cuando sea llamada, a fin de ejecutar acciones en base al mismo.
Se pueden pasar a la función ninguno, uno o más parámetros separados por comas.

```
def funcion1(param1, param2, param3):
    # Cuerpo de la función
```

◊ Los parámetros se comportan como **variables de ámbito local** dentro de la misma.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

nombre = 'Manuel'
```

```
def funcion1(nombre):
    print('{} : Hola desde dentro de la función'.format(nombre))

print('{} : Hola desde fuera de la función'.format(nombre))
print()
funcion1('Patricia')
```

Teniendo como salida:

```
# ./prueba.py
Manuel : Hola desde fuera de la función

Patricia : Hola desde dentro de la función
```

◊ Será posible asignar **valores por defecto** a los parámetros de las funciones, pudiendo así llamar a la función con menos argumentos de los que se espera:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

def saludo (nombre, mensaje='Hola'):
    print('{}, {}'.format(mensaje, nombre))

saludo('Andrea')
print()
saludo('Daniel', 'Adios')
```

Teniendo como salida:

```
# ./prueba.py
Hola, Andrea

Adios, Daniel
```

◊ Otro modo de llamar a una función es pasándole los argumentos esperados como pares de **clave=valor**:

```
...
saludo(mensaje="Buenos días", nombre="Daniel")
...
```

◊ Es posible que las funciones esperen recibir un número **arbitrario** de argumentos. Estos argumentos llegarán a la función en forma de tupla.
- Para definir argumentos arbitrarios en una función, se antecede al parámetro con un asterisco (*):

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-


def imprimir(pfijo, *parbitrarios):
    print('Parámetro fijo: {}'.format(pfijo))
    # Los parámetros arbitrarios crean una tupla
    print('Parámetros arbitrarios: ')
    for param in parbitrarios:
        print('\t - {}'.format(param))

# Llamamos a la función:
imprimir('Fijo', 'arbitrario_1', 'arbitrario_2', 'arbitrario_3')
```

- Si una función espera recibir parámetros fijos y "de número arbitrario", **los de número arbitrario siempre deben suceder a los fijos.**
- Es posible, también, obtener parámetros arbitrarios como pares de *clave=valor*. En estos casos, al nombre del parámetro deben precederlo dos asteriscos:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

def imprimir(pfijo, *parbitrarios, **parbitclave):
    print('Parámetro fijo: {}'.format(pfijo))
    print()
```

```
    # Los parámetros arbitrarios crean una tupla
    print('Parámetros arbitrarios: ')
    for param in parbitrarios:
        print('\t - {}'.format(param))
    print()

    # Los parámetros arbitrarios tipo clave, se recorren como los "diccionarios"
    for clave in parbitclave:
        print('El valor de {} es {}'.format(clave, parbitclave[clave]))

# Llamamos a la función:
imprimir('Fijo', 'arbitrario_1', 'arbitrario_2', 'arbitrario_3', clave1='Uno', clave2='Dos')
```

◊ **Desempaquetado de parámetros**
A veces puede ocurrir una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o una tupla. En este caso, el asterisco (*) deberá preceder al nombre de la **lista** o **tupla** que es pasada como parámetro durante la llamada a la función:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = [357, 10]
print("El importe queda en {} Euros".format(calcular(*datos)))
```

Si los datos estuviesen contenidos en un **diccionario** se deberían pasar a la función precedidos de dos asteriscos (**):

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = {"importe": 357, "descuento": 10}
print("El importe queda en {} Euros".format(calcular(**datos)))
```

LLamadas de retorno

En Python, es posible (al igual que en la gran mayoría de los lenguajes de programación), llamar a una función dentro de otra, de forma fija y de la misma manera que se la llamaría desde fuera de dicha función. Veamos un ejemplo:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

def mensaje_sol(solucion):
    return 'El resultado es: {}'.format(solucion)

def sumar(num1, num2):
    sol = 0
    sol = int(num1)+int(num2)
    print(mensaje_sol(str(sol)))

sumar(3,5)
```

Funciones anónimas

La palabra clave **lambda** nos permite crear una "función rápida". Se hace del siguiente modo:

```
f = lambda x, y: x + y

suma = f(1, 2)

print(suma)
```

Estas funciones no pueden distinguirse por su nombre y, por eso, se les denomina **funciones anónimas**.
Con respecto a parámetros, funciona todo lo visto con las funciones definidas con **def**.

Generadores

Los generadores son funciones que nos permitirán obtener sus resultados poco a poco. Es decir, cada vez que llamemos a la función nos darán un nuevo resultado. Por ejemplo, una función para generar todos los números pares que cada vez que la llamemos nos devuelva el siguiente número par.

> ¿Podemos construir una función que nos devuelva todos los números pares? Esto no es posible si no usamos generadores. Como sabemos los números pares son infinitos.
> Un generador es una función especial que produce secuencias completas de resultados en lugar de ofrecer un único valor. En apariencia es como una función típica pero en lugar de devolver los valores con **return** lo hace con la declaración **yield**. Hay que precisar que el término generador define tanto a la propia función como al resultado que produce.
> Una característica importante de los generadores es que tanto las variables locales como el punto de inicio de la ejecución se guardan automáticamente entre las llamadas sucesivas que se hagan al generador, es decir, a diferencia de una función común, una nueva llamada a un generador no inicia la ejecución al principio de la función, sino que la reanuda inmediatamente después del punto donde se encuentre la última declaración **yield** (que es donde terminó la función en la última llamada).

Veamos un ejemplo con una función que crea una lista basada en la Conjetura de Collatz:

```
#Esta lista tiene como primer elemento n.
#Si n es impar el siguiente elemento es igual a 3*n + 1
#si no, es igual a n // 2 (división redondeando para abajo)
#cuando se llega a 1, la lista está completa.
#La longitud de la lista no se sabe a priori, depende de n.

def wondrous(n):
    milista = []
    while n != 1:
        milista.append(n)
        n = n // 2 if n % 2 == 0 else 3*n + 1
    else:
        milista.append(1)
        return milista

for i in wondrous(13):
    print(i)
```

> Si queremos sólo un conjunto dado de elementos de una de las listas generadas, vemos que perdemos recursos del equipo, pues se genera la lista completa:

```
print("Vemos todos los elementos de la lista: ")
for i in wondrous(13):
    print(i, end = " ")

print("\n\nVemos sólo los primeros cuatro elementos de la lista: ")
n = 0
for i in wondrous(13):
    n = n + 1
    if n < 5:
        print(i, end = " ")
    else:
        break

#Vemos todos los elementos de la lista:
#13 40 20 10 5 16 8 4 2 1

#Vemos sólo los primeros cuatro elementos de la lista:
#13 40 20 10
```

Podemos adaptar esta función a un generador:

> Desde el punto de vista de la sintaxis, una función generadora se distingue de las funciones vistas por el hecho de que el control al llamador es devuelto con la expresión *yield* en vez de con una instrucción *return*.
> Esta función devuelve un iterador del tipo *generator*. La llamada a la función no provoca la ejecución del código, sino que crea un generador y lo devuelve.

```
>>> g = foo('inicio')
>>> type(g)
<class 'generator'>
```

Para movernos con la lista

```
>>> g.__next__()
```

Veamos nuestro ejemplo donde generamos los primeros cuatro elementos de la lista:

```
def wondrous(n):
    while n != 1:
        yield n
        n = n // 2 if n % 2 == 0 else 3*n + 1
    else:
        yield 1

#Cuando se llama a la función, devuelve un generador, mediante el cual
##podemos producir los elementos de la secuencia uno a uno:
g = wondrous(13)
#Generamos e imprimimos los cuatro primeros números de la lista
for i in range(15):
    num = g.__next__()
    print (num)
    #Si llegamos a "1" la lista NO sigue. Evitamos errores así:
    if num == 1:
        break
```

Fuentes:

- El gran libro de Python

Librerías de funciones

Podemos crear un archivo con funciones y llamarlo desde otros *scripts* Python.

- El archivo debe tener la extensión .py, pero, para llamarlo, no hay que ponerle esa extensión.

Así, si creamos un archivo con funciones denominado *funciones.py* podemos acceder a ellas desde otro archivo Python del siguiente modo:

```
import funciones
...
funciones.funcion1()
```

-- Volver