Curso Básico de Python

4. Funciones

Contenido

Una función permite ejecutar un conjunto de sentencias con solo invocar el nombre de esa función. La definición de una función asigna un nombre a un bloque de sentencias, que se ejecutarán toda vez que se escriba el nombre de la función, como si fuera una sentencia más. Cuando se escribe el nombre de una función en un intérprete de comandos o dentro de un programa se dice que *se llama o se invoca la función*. La definición de la función no ejecuta las sentencias que forman el cuerpo de la función; éste solo será ejecutado cuando la función sea llamada. Además del nombre, la función se define con un conjunto de *parámetros*, nombres de variables donde pueden almacenarse valores al llamar la función. Al llamar la función, se la invoca con *argumentos*, valores que serán recogidos en los parámetros de la función. Una vez ejecutadas las sentencias dentro de la función, ésta puede devolver uno más *valores de retorno* al entorno donde esa función fue invocada.

Lecturas recomendadas:

1. Tutorial de Python, sección 4.7 Definiendo funciones.
2. Tutorial de Python, sección 4.8 Más sobre definición de funciones, 4.8,1 a 4.8.5.
3. Atender especialmente las secciones 4.8.7 Cadenas de texto de documentación, necesaria para la generación automática de documentación, y 4.9 Estilo de codificación, necesaria para escribir código fácilmente legible.

Advertencia: algunas partes de estas secciones pueden no resultar claras en primera lectura; las notas subsiguientes pueden contribuir a aclarar algunos puntos; se sugiere volver luego a leer y experimentar estas secciones más dificultosas.

1 Sintaxis

Definición de una función:

```
def nombre de función ( [lista de parámetros] ):
   bloque de sentencias
   [return [lista de valores] ]
```

La lista de parámetros es opcional, así como la sentencia `return`, que puede devolver o no una lista de valores.
Para invocar una función:

```
nombre de la función ( [lista de argumentos] )
```

La lista de argumentos al invocar la función deberá corresponder con la lista de parámetros definida en la función.

2 Algunas funciones simples

Una función elemental, con un solo parámetro y sin devolver ningún valor:

```
def hola(nombre):     # define función de nombre hola con parámetro nombre
   print("Hola", nombre)   # el parámetro nombre contiene el nombre recibido
   return            # esta función no devuelve ningún valor
```

Si bien al no devolver nada la sentencia `return` puede omitirse, se considera buena práctica incluirla. Esta función puede usarse así:

```
hola("Antonio")   # invoca la función con el nombre como argumento
```

También as posible pedir el nombre, guardarlo en una variable, e invocar la función con esa variable:

```
mi_nombre = input("Su nombre: ") # guarda un nombre en la variable mi_nombre
Su nombre: Antonio               # el usuario escribe un nombre
hola(mi_nombre)   # invoca la función con mi_nombre como argumento
Hola Antonio      # la función imprime el saludo con el nombre recibido
```

Los parámetros pueden tener un valor por defecto, lo cual permite omitirlos.

```
def hola2(nombre, pais="Uruguay"):
   print("Nombre:", nombre)
   print("País  :", pais)
   return
```

Esta función puede invocarse de varias maneras:

```
hola2("Antonio")
hola2("Antonio", pais="Brasil")
hola2("Antonio", "Argentina")
#hola2()       # ERROR, falta el argumento obligatorio
```

Una función puede devolver uno o varios valores:

```
from math import sqrt
def cuad(num):
   '''Calcula potencia de 2 y raíz cuadrada del número recibido.'''
   n_cuad = num ** 2        # num alcuadrado
   n_raiz = sqrt(num)       # raíz cuadrada de num
   return (n_cuad, n_raiz)  # devuelve potencia de 2 y raíz cuadrada de num
```

Esta función puede invocarse así:

```
n_pot2, n_raiz2 = cuad(4)    # guarda en variables los valores devueltos
print(n_pot2, n_raiz2)       # imprime 16 2.0
```

Una función aún por escribir, no hace nada:

```
def fn_skel():
   '''Esqueleto de una función por escribir.'''
   pass
```

3 La lista de parámetros

Hay varias formas de pasar valores a una función.

Parámetros posicionales. Se distinguen por su posición en la lista: `fn(p1, p2, p3)`.

```
def fn_pos(nombre, edad, pais):
   print (nombre, "tiene", edad, "años, y viene de", pais)
   return

fn_pos("Antonio", 46, "Polonia")
```

Salida: `Antonio tiene 46 años, y viene de Polonia`
Se debe llamar la función indicando todos los argumentos, uno para cada parámetro definido.

Parámetros por palabra clave (keyword). También llamados parámetros nombrados, parámetros por omisión o parámetros por defecto. Se distinguen por un nombre, y tienen asignado un valor, en la forma `fn(par1=p1, par2=p2):` .

```
 def fn_def(nombre="Anónimo", edad=0, pais=None):
    print (nombre, "tiene", edad, "años, y viene de", pais)
    return

fn_def(nombre="Antonio", edad=46, pais="Polonia")
fn_def(edad=46, pais="Polonia", nombre="Antonio")
```

Cuando la función se define con parámetros nombrados es posible cambiar el orden en la llamada, porque el nombre identifica al argumento.

```
fn_def("Antonio", 46, "Polonia")
```

Si se omite el nombre del parámetro, se debe respetar el orden, para poder reconocer los argumentos por su posición.

```
fn_def()
```

Salida: `Anónimo tiene 0 años, y viene de None`
Si se omiten todos los parámetros, usa los valores por defecto.

```
#fn_def("Antonio", pais="Polonia", 46)  # ERROR
# los argumentos posicionales deben ir antes que los nombrados
```

Los parámetros dados por posición deben ir siempre antes de los parámetros nombrados.

Parámetros posicionales adicionales. El asterisco `*` permite pasar una lista de parámetros posicionales arbitraria.

```
def fn_posvars(*varios):
    print("Otros parámetros posicionales:", varios)

fn_posvars('Antonio', 46, 'Polonia', 'pianista')
```

Salida: `Otros parámetros posicionales: ('Antonio', 46, 'Polonia', 'pianista')`
Estos parámetros se reciben como una tupla de valores, distinguidos por su posición.

Parámetros nombrados adicionales. El doble asterisco `**` permite pasar un diccionario de parámetros nombrados, {nombre_parámetro : valor_por_defecto}.

```
def fn_nomvars(**nomvarios):
    print("Otros parámetros nombrados", nomvarios)

fn_nomvars(nombre="Antonio", edad=46, profesion="pianista")
```

Salida: `Otros parámetros nombrados {'nombre': 'Antonio', 'edad': 46, 'profesion': 'pianista'}`

Las formas anteriores pueden usarse todas a la vez en una misma función, en tanto se respete el orden: posicionales, nombrados, otros posicionales, otros nombrados.

```
def fn_pars(edad, nombre="Anónimo", *posvars, **nomvars):
    print("Posicional, edad:", edad)
    print("Nombrado, nombre:", nombre)
    print("Posicionales otros:", posvars)
    print("Nombrados, otros:", nomvars)

fn_pars(46, "Antonio", "pianista", "Polonia", musica="clásica",
preferido="Schumann")
```

Salida:

```
Posicional, edad: 46
Nombrado, nombre: Antonio
Posicionales otros: ('pianista', 'Polonia')
Nombrados, otros: {'musica': 'clásica', 'preferido': 'Schumann'}
```

4 Alcance de las variables

Las variables recibidas como parámetros no son modificadas dentro de la función, en cambio, las variables que representan estructuras, como listas o diccionarios, sí se modifican dentro de la función.

```
def fn_mod(par_1, ls_1):      # parámetros: una variable y una lista
    par_1 = "nuevo valor"     # cambia el valor de la variable
    ls_1 += [33]              # agrega un elemento a la lista
    return par_1, ls_1        # devuelve variable y lista

mi_nombre = "Antonio"
mi_lista = [1, 2, 3]
res_1, res_2 = fn_mod(mi_nombre, mi_lista)
print(res_1, res_2)           # salida: nuevo valor [1, 2, 3, 33]
print(mi_nombre, mi_lista)    # salida: Antonio [1, 2, 3, 33]
```

Una variable modificada dentro de una función es otra variable, no afecta la variable externa a la función. En cambio, la lista sí es modificada por la función. En la función se copia el valor de la variable externa sin alterarla, es un pasaje *por valor*; en cambio el pasaje de la lista es *por referencia*, es la misma lista externa la que modifica la función en su interior.

Las variables definidas dentro de una función, así como los parámetros, son *variables locales*; las variables definidas en el módulo Python (archivo .py) son *variables globales*. Una variable global y una variable local pueden tener el mismo nombre, pero son dos variables distintas. No pasa lo mismo con estructuras de datos, como una lista o un diccionario; estas estructuras pueden ser modificads dentro de la función.

5 Ejercicios

1. Estudiar y correr el módulo `params_mod.py` según se indica en los comentarios dentro del código.

2. El módulo `params_nomain.py` define tres funciones y muestra distintas formas de pasar parámetros.
   a) Estudiar el código. Observar las distintas formas de pasar parámetros, así como el formato de los comentarios, que permite presentar la ayuda en forma legible.
   b) Para invocar las funciones, se debe primero importar el módulo. Probar estos comandos:
   ```
   import params_nomain               # importa el módulo
   dir(params_nomain)                 # muestra las funciones definidas
   help("params_nomain.f1")           # toma el formato de los comentarios
   # f2(9, "cuarenta")                # ERROR, no se indica el módulo
   params_nomain.f3(9, "cuarenta")    # bien, indica el módulo
   ```
   c) Reiniciar el shell (en IDLE, Ctrl-F6). Probar ahora importando todas las funciones del módulo:
   ```
   from params_nomain import *
   ```
   Verificar que ahora acepta la invocación directa de todas las funciones, sin necesidad de indicar el módulo; las funciones fueron importadas al entorno de trabajo.
   d) Reiniciar nuevamente el shell. Importar ahora solo una función:
   ```
   from params_nomain import f2
   ```

Verificar que la función `f2` está disponible, pero las otras dos no, aún indicando el módulo. ¿Por qué?

3. El módulo `params_main.py` es idéntico a `params_nomain.py` pero agrega al final un bloque de sentencias que se ejecuta si el módulo ha sido invocado como programa:
```
if __name__ == '__main__':
    bloque de sentencias
```
en cuyo caso ejecuta el bloque de sentencias subsiguiente. Ejecutar el módulo escribiendo en una terminal
```
$ python3 params_main.py
```

4. Estudiar y correr el módulo `tabla_mult_func.py`, donde se define una función para mostrar los 10 primeros múltiplos de un número.
Examinar ahora el módulo `tabla_mult_try.py`, donde se asegura que el ingreso sea un número. La sentencia
```
try:
    bloque de sentencias
except NombreExcepcion:
    bloque de sentencias
```
permite capturar muchos tipos de errores, reconocidos por el nombre de la excepción.

5. Estudiar y correr el ejemplo `alcance.py`. Observar la modificación de valores de variables dentro de la función, y su resultado en el módulo. ¿Qué hace la sentencia `global`? ¿Es necesario usar `global` para una lista? ¿Será necesario usar `global` para un diccionario? Probarlo experimentalmente.

6. Documentación del código. El directorio `ejemplos-html` contiene la documentación de los ejemplos, generada automáticamente con la aplicación pydoctor, en forma de archivos HTML. Para visualizarla, abrir el archivo `index.html` con un navegador. Esta documentación surge de los comentarios (*docstrings*) incluidos en el código, escritos en el formato adecuado.
