Funciones

Mario González
Facultad de Ingeniería y Ciencias Aplicadas
Universidad de las Américas, Quito, Ecuador

Agosto, 2021

Definiendo funciones I

- En Python, la definición de funciones se realiza mediante la instrucción `def` más un nombre de función descriptivo, seguido de paréntesis de apertura y cierre.
- Aplican las mismas reglas que para el nombre de las variables.
- Como toda estructura de control en Python, la definición de la función finaliza con dos puntos `(:)`
- Los parámetros de entrada o argumentos deben ser colocados dentro de estos paréntesis.

```
def mi_funcion(parametros):
    "docstring_de_la_funcion"
    instrucciones
    return [expresion]
```

Definiendo funciones II

- La primera declaración de una función puede ser una declaración opcional, la cadena de documentación de la función o docstring.
- El bloque de código dentro de una función se inicia con dos puntos (:) y está identado.
- La declaración de retorno `return [expresion]` sale de una función, opcionalmente devolviendo una expresión.
- Una sentencia `return` sin argumentos es el mismo que `return None`.

```
def mi_funcion(parametros):
    "docstring_de_la_funcion"
    instrucciones
    return [expresion]
```

Definiendo funciones III

- Ejemplo:

```
def saludo(p1,p2):
    """
    Imprime un saludo para dos personas.
    Espera como entrada dos strings.
    """
    print("Hola, "+p1+" y "+p2)
    return
```

Llamando funciones I

- Definir de una función le da un nombre, especifica los parámetros que se van a incluir en la función y las estructuras de los bloques de código.
- Una función está puede ejecutarse **llamandola desde otra función** o directamente desde la línea de comandos de Python.

```
>>> saludo('Pedro','Jose')
Hola, Pedro y Jose
```

- Obteniendo información de una función:

```
help(saludo)

Help on function saludo in module funciones:

saludo(p1, p2)
    Imprime un saludo para dos personas.
    Espera como entrada dos strings.
(END)
```

Argumentos de una función I

- Se puede llamar a una función mediante el uso de los siguientes tipos de argumentos formales:
- argumentos obligatorios,
- argumentos de palabras clave,
- argumentos predeterminados,
- argumentos de longitud variable.

Argumentos de una función II

- **Argumentos requeridos** son los argumentos que se pasan a una función en el orden posicional correcto. En este caso, el número de argumentos en la llamada a la función debe coincidir exactamente con la definición de la función.

```
def fruta(nom,cant):
    """
    Imprime un nombre y una cantidad.
    Espera como entrada un string (nom) y un entero (cant).
    """
    print(nom+' tiene '+str(cant)+' naranjas.')
    return
```

- Llamando a la función:

```
>>>fruta('Jose',5)
Jose tiene 5 naranjas.

>>>fruta(5,'Jose')

>>>fruta(5,'Jose')
```

Argumentos de una función III

- **Argumentos de palabras clave** están relacionados con las llamadas a funciones. Cuando se utilizan argumentos de palabra clave en una llamada a la función, se identifican los argumentos por el nombre del parámetro.
- El intérprete de Python es capaz de utilizar las palabras clave previstas para que coincida con los valores de los parámetros.

```
>>>fruta(nom='Jose',cant=5)
Jose tiene 5 naranjas.

>>>fruta(cant=5,nom='Jose')
Jose tiene 5 naranjas.
```

Argumentos de una función IV

- **Un argumento por defecto** es un argumento que asume un valor por defecto si un valor no se proporciona en la llamada de una función.

```
def saludo(p1='Fulano'):
    """
    Imprime un saludo impersonal,
    o para una persona.
    Espera como entrada un string.
    """
    print("Hola, "+p1)
    return

>>>saludo()
Hola, Fulano

>>>saludo('Jose')
Hola, Jose
```

Argumentos de una función V

- Una versión más de esta función:

```
def saludo(p1='Fulano',p2=None):
    """
    Imprime un saludo impersonal,
    o para una o dos personas.
    Espera como entrada dos strings.
    """
    if p2==None:
        print("Hola, "+p1)
    else:
        print("Hola, "+p1+" y "+p2)
    return
```

Argumentos de una función VI

- Puede que tengamos que procesar una función para más argumentos que los especificados al definir la función.
- **Los argumentos de longitud variable** no se nombran en la definición de la función, a diferencia de los requeridos y argumentos por defecto.

```
def mi_funcion([args_formales,] *args_var_tupla):
    "docstring_de_la_funcion"
    instrucciones
    return [expresion]
```

- Un asterisco (*) se coloca antes del nombre de la variable que contiene los valores de todos los argumentos variables sin palabra clave. Esta tupla permanece vacía si no se especifican argumentos adicionales durante la llamada a la función.

Argumentos de una función VII

- A continuación se presenta un ejemplo sencillo:

```
def impInfo(arg1, *tuplavar):
    print('Historial de temperatura en '+arg1)
    for var in tuplavar:
        print(var, end=' ')
    return

>>>impInfo('Quito',25,23,24,25)
Historial de temperatura en Quito
25 23 24 25

impInfo('Quito')
```

Funciones anónimas I

- Estas funciones son llamadas anónimas porque no se declaran de la manera estándar mediante el uso de la palabra clave `def`. Se puede utilizar la palabra clave `lambda` para crear pequeñas funciones anónimas.
- Las funciones `lambda` pueden tomar cualquier número de argumentos, pero regresar sólo un valor en forma de una expresión. No pueden contener comandos o múltiples expresiones.
- Las funciones `lambda` no pueden acceder a las variables que no sean los de su lista de parámetros y los que están en el espacio de nombres global.
- Sintaxis:

```
lambda [arg1 [,arg2,.....argn]]:expresion
```

Funciones anónimas II

- Ejemplo:

```
suma = lambda arg1, arg2: arg1 + arg2

>>>print("Total : ", suma(10,20))
Total: 30
```

Variables Globales vs. Locales I

- Las variables que se definen dentro del cuerpo de la función tienen un alcance local y las que se definen fuera tienen un alcance global.
- Esto significa que se puede acceder a las variables locales sólo dentro de la función en la que se declaran, mientras que se puede acceder a las variables globales en todo el cuerpo programa por todas las funciones.
- Cuando se llama a una función, las variables declaradas dentro de ella se ponen al alcance.

Variables Globales vs. Locales II

Ejemplo:

```
total=5

def suma(arg1, arg2):
        total = arg1 + arg2
        print("Total dentro de la funcion:", total)

suma(8,2)

print("Total (global):",total)
```

Variables Globales vs. Locales III

Ejemplo:

```
total=5

def suma(arg1, arg2):
        global total
        total = arg1 + arg2
        print("Total dentro de la funcion:", total)

suma(8,2)

print("Total (global):",total)
```

Módulos I

- Un módulo de Python es simplemente un archivo fuente de Python (.py), que puede exponer clases, funciones y variables globales.

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Módulos II

- Para importar un modulo en python usamos:

```
>>> import fibo
>>> fibo.fib(5)
1 1 2 3

>>> fibo.fib2(5)
[1, 1, 2, 3]

>>> import fibo as f
>>> f.fib(5)
1 1 2 3

>>> from fibo import fib2
>>> fib2(5)
[1, 1, 2, 3]

>>> from fibo import *
>>> from fibo import fib, fib2
```

Módulos III

```
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

def main(argv):
        if len(argv)>0:
                print(fib2(int(argv[0])))
        else:
                print("Usage:")
                print("  python fibo.py <integer>")
                print("Example")
                print("  python fibo.py 5")

if __name__ == "__main__":
        import sys
        main(sys.argv[1:])
```