Tema 8: Conceptos avanzados sobre funciones

Héctor Xavier Limón Riaño

May 31, 2024

Contents

1 Ámbito de identificadores (scope)

- Un identificador es cualquier cosa que el programador pueda nombrar en el programa: variables, nombres de funciones, nombres de clases, etc.
- El ámbito o scope se refiere al área del programa donde se puede tener acceso a un identificador
- Por ejemplo, si defines una variable dentro de una función, no puedes tener acceso a dicha variable fuera de dicha función

```
def mi_fun():
    x = 'hola' # definida en el ámbito de la función
    return x

mi_fun()
print(x) # error, no está definida la variable en este ámbito
```

- Los ámbitos se anidan, esto es, están dentro de otros, si un identificador no se encuentra en el ámbito actual, se busca en el siguiente nivel de ámbito.

```
variable = 3 # ámbito global

def fun():
    return variable + 4 # se busca en global

print(fun())
```

7

- Si dos identificadores se llaman igual pero se encuentran en diferente ámbito, entonces se toma el identificador más cercano al ámbito activo

```
variable = 'hola ' # ámbito global

def saludar(nombre):
    variable = 'adios '
    return variable + nombre

print(saludar('pepe'))
print(variable)

adios pepe
hola
```

- Cuidado: en `Python` si defines dos identificadores en el mismo nivel de ámbito que se llaman igual, se hará una `sobre-escritura`, quedándose sólo la última definición
- En otros lenguajes como `Java` podría generarse un error en casos similares

```
def hola():
    print('hola')

def hola():
    print('adios')

hola()

adios
```

1.1 Tipos de ámbito

1.1.1 Global

- Es el ámbito más general, en `Python` puede identificarse fácilmente al ver que no hay identación
- El ámbito se crea cuando inicia el programa
- Este ámbito muere hasta que termina el programa

- CUIDADO: es una mala práctica bien conocida el usar variables globales, estás pueden ser la fuente de errores por efectos colaterales (explicado más adelante)
- En este ámbito sólo debería haber definiciones de funciones, clases y constantes (como las variables pero no cambian de valor)

1.1.2 Función/método

- Son los identificadores que se crean dentro del cuerpo de una función o método
- Esto incluye a los parámetros de la función
- Se crea un ámbito de este tipo por cada invocación a función/método
- El ámbito muere cuando la función/método termina
- Visualmente son los identificadores que están definidos dentro del cuerpo de la función/método

```
def externa():
    def interna(): # función dentro de función
        var = 3
        return 1
    return interna() # se puede hacer la referencia

print(externa()) # sin errores
print(interna()) # error, no está definida en ámbito global
```

1.1.3 Clase/objeto

- Se crea un ámbito de este tipo cada vez que se crea un objeto
- El ámbito muere cuando el objeto pierde todas sus referencias (ver subtema de recolección de basura)
- Visualmente son los identificadores definidos dentro del cuerpo de una clase
- `Python` realmente no tiene este ámbito, como si lo tienen lenguajes como `Java`, por eso requiere del uso explícito de `self` para hacer referencia al ámbito del objeto

```
class Prueba():
    def __init__(self, propiedad):
        self.propiedad = propiedad # ámbitos diferentes

    def metodo(self):
        print(propiedad) # error no existe
        print(self.propiedad) # OK
```

1.1.4 Bloque

- Es un tipo de ámbito que se crea cuando hay sentencias de bloque (como las sentencias de control)
- El ámbito muere cuando termina el bloque
- `Python` no tiene este tipo de ámbito, aunque hay lenguajes como `Java` que si lo tienen

```
if True:
    variable = 6 # se definió en global

print(variable) # no hay problema
```

```
6
```

1.2 Recolección de basura

- Cuando el ámbito de un identificador desaparece, también desaparecen sus identificadores asociados
- Considera que en el caso de las variables, éstas realmente son referencias a posiciones en memoria
- Puede haber varias referencias a la misma memoria

```
l1 = [1, 2, 3]
l2 = l1 # es la misma memoria
l2[0] = 99
print(l1[0])
```

- Cuando un ámbito termina, es posible que desaparezcan referencias a datos en memoria
- Si un dato en memoria no tiene referencias vivas (en ámbitos activos) entonces esa memoria es propensa a ser reciclada
- Al proceso de reciclado se le conoce como `recolección de basura`
- El recolector de basura es un proceso especial del intérprete que se encarga de reciclar memoria que ya no tiene referencias vivas, y que por lo tanto no puede ser recuperada
- Muchos lenguajes tienen recolectores de basura: `Java`, `Python`, `Java Script`, `C#`, entre otros.
- Mientras que en otros lenguajes como `C` y `C++` la liberación de memoria es un proceso manual que le corresponde al programador, lo cual tiene ventajas y desventajas

2 Paso de parámetros a funciones

- Se refiere a los parámetros (también llamadas argumentos) que se pasan al mandar a llamar a una función
- Dependiendo del tipo de dato, es posible que los parámetros se manden de forma diferente
- Entender esas diferencias es fundamental en programación para poder prevenir y diagnosticar errores
- Lo primero es entender que las variables son referencias a direcciones en memoria, por si mismas no guardan valores, sólo señalan dónde se encuentra un dato

2.1 Paso por valor

- Es la forma en que se pasan tipos simples (primitivos)
- Los parámetros que se pasan de esta forma se pueden entender como "copias" del valor original

- De esta forma, si modificas el valor (la referencia realmente) no afecta a otras partes del código
- Aunque realmente en lenguajes como Python que no tienen primitivos (los tipos simples y todos los tipos son objetos) se siguen pasando referencias

```
def fun(var):
    var = 1 # la referencia se va a otro lado
    return var

var = 33 # este identificador está en otro ámbito
fun(var)
print(var) # no se afectó var
```

```
33
```

2.2 Paso por referencia

- Es la forma en que se pasan los tipos estructurados (estructuras de datos)
- Se puede entender más directamente como el paso de referencias a memoria y no como copias de valores
- Hay que tener cuidado con este paso de parámetros dado que son lo fuente de posibles errores (como se explica en la sección siguiente)

```
def fun(lista): # paso por referencia
    lista[0] = 66 # se modifica memoria
    return 1

l = [1, 2, 3, 4]
fun(l)
print(l)
```

```
[66, 2, 3, 4]
```

3 Efectos colaterales

- Dado que al pasar por referencia se puede manipular memoria, es posible que al hacer cambios en una función se afecte a otras partes del código
- A lo anterior se le conoce como `efecto colateral`, esto es, tras ejecutar la función, se produjeron resultados que posiblemente no se esperaban a priori
- Los efectos colaterales son una fuente común de bugs en los programas

```
def sumatoria(lista):
    res = 0
    while lista:
        res += lista.pop() # se va destruyendo la lista, efecto colateral
    return res

l = [1, 2, 3, 4]
print(sumatoria(l)) # aparentemente todo bien
print(sumatoria(l)) # no era lo que esperaba
```

```
10
0
```

- Una forma de solucionar el problema es haciendo copias
- Si se quiere estar seguro de que no hay efectos colaterales, se pueden usar también estructuras no mutables como las tuplas

```
def sumatoria(lista):
    lista = lista.copy() # copia
    res = 0
    while lista:
        res += lista.pop() # se va destruyendo la copia
    return res

l = [1, 2, 3, 4]
print(sumatoria(l)) # aparentemente todo bien
print(sumatoria(l)) # todo bien
```

```
10
10
```

- Los efectos colaterales también tienen que ver con crear archivos, modificar configuraciones del sistema o hacer cualquier cosa que no sea obvia y que cambia el estado del sistema de formas impredecibles y silenciosas

4 Tipos de parámetros en las funciones

4.1 Posicionales obligatorios

- Son el tipo de parámetros que se han estado utilizando hasta el momento
- Dependen de un orden, por lo que se les llama posicionales
- En Python es obligatorio pasar todos los parámetros posicionales al invocar la función

```
def fun(pos1, pos2, pos3):
    return pos1 + pos2 + pos3

fun(1, 2, 3) # el mapeo es pos1=1, pos2=2, pos3=3
```

4.2 Opcionales

- Como su nombre lo indica son opcionales, esto es, se pueden pasar o no al invocar la función
- Son posicionales en el sentido de que se corresponden con la posición de acuerdo a como se invoca la función
- Al declarar la función es obligatorio establecer el valor por defecto del parámetro, de hecho esta es la forma de distinguirlos de parámetros posicionales normales
- Al declarar la función, deben de ir después de los parámetros posicionales obligatorios, de lo contrario se confundiría el intérprete

```
def fun(normal, opcional=0, op2=1):
    return normal + opcional + op2
```

```
print(fun(3)) # ne se pasó segundo param, valor 0 por defecto
print(fun(3, 4)) # aquí si se pasó, se considera valor pasado
print(fun(3, 1, 2))
```

```
3
7
```

- Este tipo de parámetros elimina la necesidad de tener sobrecarga de funciones (visto en un subtema más adelante)

4.3 Keyword (nombrados)

- Son parámetros no posicionales, esto es, los puedes pasar en cualquier orden
- Se declaran igual que los parámetros opcionales, esto es, en Python, cualquier parámetro definido como opcional es también keyword (esto no es así en todos los lenguajes)
- Al igual que los opcionales, se deben definir después de los parámetros posicionales obligatorios
- Al invocar la función, se puede pasar directamente la relación de valores del parámetro utilizando su nombre

```
def fun(normal, nombrado1=0, nombrado2=0):
    return normal + nombrado1 + nombrado2

print(fun(1)) # siguen siendo opcionales
print(fun(1, nombrado2=11)) # puedo pasar alguno
print(fun(1, nombrado2=11, nombrado1=22)) # puedo pasar en diferente orden
```

```
1
12
34
```

4.4 Variables

- Son un tipo especial de parámetro que representa un número variable de parámetros

- Permiten que una función pueda recibir cualquier número de parámetros sin importar cómo fue definida
- Al definir la función, cualquier argumento que inicie con * se considera variable
- Sólo puede haber uno de estos y debe ir al final de la lista de parámetros (o puede ser casi al final como se menciona más adelante) a menos que tengas parámetros nombrados
- Todos los parámetros extra que pases se van a ir a una tupla

```
def sumatoria(*args):
    res = 0
    for val in args:
        res += val
    return res

print(sumatoria())
print(sumatoria(1))
print(sumatoria(1, 2))
print(sumatoria(1, 2, 3))
```

```
0
1
3
6
```

```
  def fun(param1, op1=0, *resto):
      print(resto)
f
  fun(1) # Al menos se tiene que pasar un param
  fun(1, 2)
  fun(1, 2, 3)
  fun(1, 2, 3, 4)
```

```
()
()
(3,)
(3, 4)
```

- Es posible pasar una lista (u otra estructura secuencial) a una función que recibe parámetros variables mediante un proceso que se llama `desempaquetar`
- Básicamente se agrega un `*` al pasar la lista en la invocación de la función

```
def sumatoria(*args):
    res = 0
    for val in args:
        res += val
    return res

lista = [1, 2, 3, 4]
print(sumatoria(*lista)) # se pasa como parámetros
```

```
10
```

4.4.1 Parámetros variables nombrados

- Un caso especial de parámetros variables son los parámetros variables nombrados
- Es una idea similar, pero para pasar múltiples parámetros nombrados
- Se utiliza `**` al declarar el parámetro
- Los parámetros nombrados extra se van a un diccionario (vistos en el siguiente tema del curso)
- Deben ir al final de la lista de parámetros, después de los parámetros variables (de haberlos)

```
def fun(**kargs):
    print(kargs)

fun(nombre='pepe', apellido='pecas', edad=15)
```

```
{'nombre': 'pepe', 'apellido': 'pecas', 'edad': 15}
```

- Al igual que con parámetros variables normales se pueden desempaquetar valores, en este caso de un diccionario

- Para desempaquetar se usa ** al invocar

```
def fun(**kargs):
      print(kargs)

diccionario = {'nombre': 'pepe', 'edad': 15, 'apellido': 'pecas'}
fun(**diccionario)

{'nombre': 'pepe', 'edad': 15, 'apellido': 'pecas'}
```

5 Sobrecarga de funciones

- Se refiere a poder definir varias veces la misma función pero con variaciones en los parámetros
- Dependiendo del lenguaje estas variaciones pueden tener que ver con el tipo de cada parámetro y/o con el número de parámetros
- Por ejemplo, en C++ se puede querer definir una función suma que funcione con enteros o con flotantes, en este caso se definiría la función dos veces, una vez por cada tipo de parámetro
- A esto se le conoces como sobrecarga de funciones

```
int sumar(int num1, int num2) {
  return num1 + num2;
}

float sumar(float num1, float num2) {
  return num1 + num2;
}
```

- En Python este concepto no existe
- Al tenerse tipos dinámicos no hay necesidad en muchas ocasiones
- Si necesitas que haya variaciones en el número de parámetros puedes usar parámetros opcionales o nombrados
- Si se intenta, se hará una sobreescritura, esto es, se remplaza la función original por la última que se definió

```
def sumar(val):
    return val + 1

def sumar(val1, val2=0):
    return val1 + val2

print(sumar(1)) # error, se esperaban dos parámetros
```

6 Documentar funciones

- La documentación es información que se provee para quienes quieren utilizar tu código
- Por ejemplo, la función `help` muestra la documentación de ayuda que el programador estableció

```
help(''.capitalize)

Help on built-in function capitalize:

capitalize() method of builtins.str instance
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest lower
    case.
```

- Se pueden documentar: módulos, funciones, métodos y clases
- La documentación la establece el programador con cadenas especiales, usualmente se usa una cadena multi-linea

```
def fun(var1, var2):
    """
    Cadena de documentación...
    """
    return 1

help(fun)

Help on function fun in module __main__:
```

```
fun(var1, var2)
    Cadena de documentación...
```

- Una buena documentación de funciones incluye:
  - Una descripción breve de lo que hace la función
  - Una descripción breve de cada parámetro
  - Una descripción de lo que la función regresa

```
def sumatoria(lista):
    """
    Calcula la sumatoria de los elementos de lista.

    lista: lista de números de entrada
    returns: un número con la sumatoria
    """
    res = 0
    for elemento in lista:
        res += elemento
    return res
```

6.1 Anotaciones de tipos

- Entre otras cosas, son una forma de mejorar la documentación de una función
- Sirven para establecer el tipo de datos que se espera reciba y regrese la función
- No establecen restricciones de tipo (como en lenguajes como `C`), son simplemente un apoyo

```
def sumatoria(lista: list) -> int:
    """
    Calcula la sumatoria de los elementos de lista.

    lista: list, lista de enteros
    returns: int, sumatoria total
    """
    res = 0
```

```
for elemento in lista:
    res += elemento
return res
```

7 Ejercicios sugeridos

7.1 Ejercicio 1

- Hacer un script de linea de comandos que regresa la fecha y hora actual de acuerdo a dos posibles fuentes:
  - Google
  - Sistema
- El usuario puede establecer "google" o "sistema" como parámetro del script para determinar la fuente
- Utiliza funciones con parámetros opcionales
- Documenta apropiadamente tus funciones

7.2 Copiar archivos txt

- Hacer un script de linea de comandos que puede recibir hasta dos parámetros:
  - Ruta directorio de entrada
  - Ruta directorio de salida
- Considerar que si se recibe sólo un parámetro, éste representa la ruta de salida, mientras que la ruta de entrada sería la ruta actual (donde sea que esté la linea de comandos en el momento de ejecutar el programa)
- El script copia todos los archivos con extensión txt del directorio de entrada hacia el directorio de salida
- Utiliza funciones con parámetros opcionales
- Documenta apropiadamente tus funciones

7.2.1 Ayuda

- Sacar rutas de un directorio

```
import os

ruta = '/tmp'
rutas = ['%s/%s' % (ruta, arch) for arch in os.listdir(ruta)]
```

- Copiar archivos

```
import shutil

shutil.copy(ruta_fuente, ruta_destino)
```