Programación estructurada

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Funciones con nombre

1.1. Definición de funciones con nombre

- En programación imperativa también podemos definir funciones.
- Al igual que ocurre en programación funcional, una función en programación imperativa es una construcción sintáctica que acepta argumentos y produce un resultado.
- Pero a diferencia de lo que ocurre en programación funcional, una función en programación imperativa es una **secuencia de sentencias**.
- Las funciones en programación imperativa conforman los bloques básicos que nos permiten **descomponer un programa en partes** que se combinan entre sí.
- Todavía podemos construir funciones mediante expresiones lambda, pero Python nos proporciona otro mecanismo para definir funciones en estilo imperativo: las **funciones con nombre**.
- La sintaxis para definir una función con nombre es:

```
def <nombre>([<parámetros>]):
    <cuerpo>
```

- donde:

```
<cuerpo> ::= <sentencia>+
```

- Por ejemplo:

```
def saluda(persona):
    print('Hola', persona)
    print('Encantado de saludarte')

def despide():
    print('Hasta luego, Lucas')
```

- Notas importantes:
  - Tiene que haber, al menos, *una* sentencia.
  - Las sentencias van **indentadas** (o *sangradas*) dentro de la definición de la función, con el mismo nivel de indentación.
  - El final de la función se deduce al encontrarse una sentencia con un **nivel de indentación superior** (en el caso de arriba, otro `def`).

Conclusión:

En Python, la **estructura** del programa viene definida por la **indentación** del código.

1.2. Paso de argumentos

- Existen distintos mecanismos de paso de argumentos, dependiendo del lenguaje de programación utilizado.
- Los más conocidos son los llamados **paso de argumentos *por valor*** y **paso de argumentos *por referencia***.

- En Python existe un único mecanismo de paso de argumentos llamado **paso de argumentos *por asignación*** o también, a veces, **paso de argumentos *por nombre***.
- En la práctica resulta bastante sencillo.
- Consiste en suponer que **el argumento *se asigna* al parámetro** correspondiente, con toda la semántica relacionada con los *alias* de variables, inmutabilidad, mutabilidad, etcétera.
- Por ejemplo:

```
1 def saluda(persona):
2     print('Hola', persona)
3     print('Encantado de saludarte')
4
5 saluda('Manolo')  # Saluda a Manolo
6 x = 'Juan'
7 saluda(x)         # Saluda a Juan
```

- En la línea 5 se asigna a `persona` el valor `Manolo` (como si se hiciera `persona = Manolo`).
- En la línea 7 se asigna a `persona` el valor de `x`, como si se hiciera `persona = x`, lo que sabemos que crea un *alias* (que no afectaría ya que el valor pasado es una cadena, y por tanto inmutable).
- En caso de pasar un argumento mutable:

```
1 def cambia(l):
2     print(l)
3     l.append(99)
4
5 lista = [1, 2, 3]
6 cambia(lista)     # Imprime [1, 2, 3]
7 print(lista)      # Imprime [1, 2, 3, 99]
```

- La función es capaz de **cambiar el estado de la lista que se ha pasado como argumento** ya que, al llamar a la función, el argumento `lista` se pasa a la función **asignándola** al parámetro `l`, haciendo que ambas variables sean *alias* una de la otra (se refieren al mismo objeto) y, por tanto, la función está modificando la misma variable que se ha pasado como argumento (`lista`).

1.3. La sentencia `return`

- Para devolver el resultado de la función al código que la llamó, hay que usar una sentencia `return`.
- Cuando el intérprete encuentra una sentencia `return` dentro de una función:
  - se finaliza la ejecución de la función,
  - se devuelve el control al punto del programa en el que se llamó a la función y
  - la función devuelve como resultado el valor de retorno definido en la sentencia `return`.
- Por ejemplo:

```
1  def suma(x, y):
2      return x + y
3
4  a = input('Introduce el primer número: ')
5  b = input('Introduce el segundo número: ')
6  resultado = suma(a, b)
7  print('El resultado es:', resultado)
```

- La función se define en las líneas 1–2. El intérprete lee la definición de la función pero no ejecuta sus sentencias en ese momento (lo hará cuando se *llame* a la función).
- En la línea 6 se llama a la función `suma` pasándole como argumentos los valores de `a` y `b`, asignándose a `x` e `y`, respectivamente.
- Dentro de la función, se calcula la suma `x + y` y la sentencia `return` finaliza la ejecución de la función, devolviendo el control al punto en el que se la llamó (la línea 6) y haciendo que su valor de retorno sea el valor calculado en la suma anterior (el valor de la expresión que acompaña al `return`).
- El valor de retorno de la función sustituye a la llamada a la función en la expresión en la que aparece dicha llamada, al igual que ocurre con las expresiones lambda.
- Por tanto, una vez finalizada la ejecución de la función, la línea 6 se reescribe sustituyendo la llamada a la función por su valor.
- Si, por ejemplo, suponemos que el usuario ha introducido los valores `5` y `7` en las variables `a` y `b`, respectivamente, tras finalizar la ejecución de la función tendríamos que la línea 6 quedaría:

  ```
  resultado = 12
  ```

  y la ejecución del programa continuaría por ahí.
- También es posible usar la sentencia `return` sin devolver ningún valor.
- En ese caso, su utilidad es la de finalizar la ejecución de la función en algún punto intermedio de su código.
- Pero en Python todas las funciones devuelven algún valor.
- Lo que ocurre en este caso es que la función devuelve el valor `None`:

```
def hola():
  print('Hola')
  return
  print('Adiós')  # aquí no llega

hola()
```

imprime:

```
Hola
```

```
def hola():
  print('Hola')
  return
  print('Adiós')

x = hola()  # devuelve None
```

```
print(x)
```

imprime:

```
Hola
None
```

1.4. Ámbito de variables

- La función `suma` se podría haber escrito así:

  ```
  def suma(x, y):
      res = x + y
      return res
  ```

  y el efecto final habría sido el mismo.

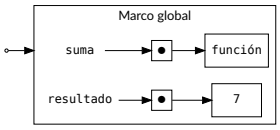
- La variable `res` que aparece en el cuerpo de la función es una **variable local** y sólo existe dentro de la función. Por tanto, esto sería incorrecto:

```
1 def suma(x, y):
2     res = x + y
3     return res
4
5 resultado = suma(4, 3)
6 print(res)  # da error
```

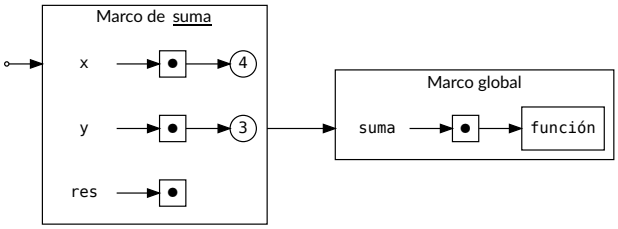
  Da error porque la variable `res` no está definida en el ámbito actual.

1.4.1. Variables locales

- Al igual que pasa con las expresiones lambda, las definiciones de funciones generan un nuevo ámbito.
- Tanto los parámetros como las variables que se definan en el cuerpo de la función son **locales** a ella, y por tanto sólo existen dentro de ella.
  - El ámbito de un parámetro es el cuerpo de la función.
  - El ámbito del resto de las variables locales es desde su definición hasta el final del cuerpo de la función.
- Eso significa que se crea un nuevo marco en el entorno, que contendrá, en principio, los parámetros y las variables locales a la función.



Entorno en la línea 6



Entorno dentro de la función `suma`

1.4.2. Variables globales

- Desde dentro de una función es posible usar variables globales.
- Se puede **acceder** al valor de una variable global directamente:

```
x = 4

def prueba():
    print(x)

prueba()  # imprime 4
```

- Pero para poder **modificar** una variable global ello es necesario que la función la declare previamente como *global*.
- De no hacerlo así, el intérprete supondría que el programador quiere crear una variable local que tiene el mismo nombre que la global:

```
x = 4

def prueba():
    x = 5  # esta variable es local

prueba()
print(x)  # imprime 4
```

- Como en Python no existen las declaraciones de variables, el intérprete tiene que *averiguar* por sí mismo qué ámbito tiene una variable.

- Lo hace con una regla muy sencilla:

  Si hay una **asignación** a una variable **dentro** de una función, esa variable se considera **local**.

- El siguiente código genera un error «*UnboundLocalError: local variable 'x' referenced before assignment*». ¿Por qué?

```
x = 4

def prueba():
    x = x + 4
    print(x)

prueba()
```

- Como la función asigna un valor a `x`, Python considera que `x` es local.
- Pero en la expresión `x + 4`, la variable `x` aún no tiene ningún valor asignado, por lo que genera un error «*variable local `x` referenciada antes de ser asignada*».

1.4.2.1. `global`

- Para declarar una variable como global, se usa la sentencia `global`:

```
x = 4

def prueba():
    global x  # se declara que la variable x es global
    x = 5     # cambia el valor de la variable global x

prueba()
print(x)  # imprime 5
```

- Las reglas básicas de uso de la sentencia `global` en Python son:
  1. Cuando se crea una variable dentro de una función, por omisión es local.
  2. Cuando se define una variable fuera de una función, por omisión es global (no hace falta usar la sentencia `global`).
  3. Se usa la sentencia `global` para cambiar el valor de una variable global dentro de una función.
  4. El uso de la sentencia `global` fuera de una función no tiene ningún efecto.
  5. La sentencia `global` debe aparecer *antes* de que se use la variable global correspondiente.

1.4.2.2. Efectos laterales

- Cambiar el estado de una variable global es uno de los ejemplos más claros y conocidos de los llamados **efectos laterales**.
- Recordemos que una función tiene (o *provoca*) efectos laterales cuando provoca cambios de estado observables en el exterior de la función, más allá de devolver su valor de retorno. Típicamente:

  - Cuando cambia el valor de una variable global
  - Cuando cambia un argumento mutable
  - Cuando realiza una operación de entrada/salida

- Una función que provoca efectos laterales es una **función impura**, a diferencia de las **funciones puras**, que no tienen efectos laterales.
- Una función también puede ser **impura** si su valor de retorno depende de algo más que de sus argumentos (p. ej., de una variable global).
- Un ejemplo de **función impura** (con un efecto lateral provocado por una operación de entrada/salida) podría ser:

```
def suma(x, y):
    res = x + y
    print('La suma vale', res)
    return res

resultado = suma(4, 3) + suma(8, 5)
print(resultado)
```

  Cualquiera que no sepa cómo está construida internamente la función `suma`, se podría pensar que lo único que hace es calcular la suma de dos números, pero resulta que también imprime un mensaje en la salida, por lo que el resultado final que se obtiene no es el que se esperaba:

```
La suma vale 7
La suma vale 13
20
```

- Las llamadas a la función `suma` no se pueden sustituir por su valor de retorno correspondiente. Es decir, que no es lo mismo hacer:

```
resultado = suma(4, 3) + suma(8, 5)
```

  que hacer:

```
resultado = 7 + 13
```

- Por tanto, la función `suma` no cumple la **transparencia referencial**.
- El que una función necesite **acceder al valor de una variable global** supone otra forma de **perder transparencia referencial**, ya que la convierte en **impura** porque su valor de retorno podría depender de algo más que de sus argumentos (en este caso, de la variable global).
- En consecuencia, la función podría producir **resultados distintos en momentos diferentes** ante los mismos argumentos:

```
def suma(x, y):
    res = x + y + z  # impureza: depende del valor de una variable global
    return res

z = 5
print(suma(4, 3))  # imprime 12
z = 2
print(suma(4, 3))  # imprime 9
```

- Igualmente, el **uso de la sentencia `global`** supone otra forma más de **perder transparencia referencial**, puesto que, gracias a ella, una función puede cambiar el valor de una variable global, lo que la convertiría en **impura** porque podría provocar un **efecto lateral** (la modificación de la variable global).
- En consecuencia, la función podría producir **resultados distintos en momentos diferentes** ante los mismos argumentos:

```
def suma(x, y):
    global z
    res = x + y + z  # impureza: depende del valor de una variable global
    z = z + 1        # efecto lateral: cambia una variable global
    return res

z = 0
print(suma(4, 3))  # imprime 7
print(suma(4, 3))  # la misma llamada a función ahora imprime 8
```

1.5. Declaraciones de tipos

1.5.1. Declaraciones de tipo de argumento

1.5.2. Declaraciones de tipo de devolución

1.6. Funciones locales a funciones

- En Python es posible definir **funciones locales** a una función.
- Las funciones locales también se denominan **funciones internas** o **funciones anidadas**.
- Una función local se define **dentro** de otra función y, por tanto, sólo existe dentro de la función en la que se ha definido.
- Su **ámbito** empieza en su definición y acaba al final del cuerpo de la función que la contiene.
- Evita la superpoblación de funciones en el ámbito más externo cuando sólo tiene sentido su uso en un ámbito más interno.
- Por ejemplo:

```
def fact(n):
    def fact_iter(n, acc):
        if n == 0:
            return acc
        else:
            return fact_iter(n - 1, acc * n)
    return fact_iter(n, 1)

print(fact(5))

# daría un error porque fact_iter no existe en el ámbito global:
print(fact_iter(5, 1))
```

- La función `fact_iter` es local a la función `fact`. No se puede usar desde fuera de `fact`.

- Tampoco se puede usar dentro de `fact` *antes* de haberse definido.
- Lo siguiente daría un error porque intentamos usar `fact_iter` antes de haber definido:

```
def fact(n):
    print(fact_iter(1, 1))  # error: se usa antes de definirse
    def fact_iter(n, acc):  # aquí es donde empieza su definición
        if n == 0:
            return acc
        else:
            return fact_iter(n - 1, acc * n)
    return fact_iter(n, 1)
```

- Las funciones locales definen un nuevo ámbito.
- Ese nuevo ámbito crea un nuevo marco en el entorno.
- Y ese nuevo marco se conecta con el marco del ámbito que lo contiene, es decir, el marco de la función que contiene a la local.

1.6.1. `nonlocal`

- Una función local puede **acceder** al valor de las variables locales a la función que la contiene.
- En cambio, cuando una función local quiere **modificar** el valor de una variable local a la función que la contiene, debe declararla previamente como **no local** con la sentencia `nonlocal`.
- Es algo similar a lo que ocurre con las variables globales.

```
def fact(n):
    def fact_iter(acc):
        nonlocal n
        n = n - 1
        if n == 0:
            return acc
        else:
            return fact_iter(acc * n)
    return fact_iter(n)

print(fact(5))
```

- La función local `fact_iter` puede acceder a la variable `n`, que es local a la función `fact` (para ello no es necesario declararla previamente como **no local**).
- Como la variable `n` está declarada **no local** en `fact_iter`, también puede modificarla.
- De esta forma, ya no es necesario pasar el valor de `n` como argumento a la función `fact_iter` y puede modificarla directamente.

1.7. *Docstrings*

2. Teorema de Böhm-Jacopini

3. Estructuras básicas de control

3.1. Concepto de estructura

3.2. Secuencia

3.3. Selección

3.4. Iteración

4. Metodología de la programación estructurada

4.1. Recursos abstractos

4.2. Diseño descendente

4.3. Refinamiento sucesivo

5. Captura de excepciones