Programación procedimental (II)

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/07/15 a las 14:33:00

1. Calidad

1.1. Documentación

Docstrings

- Una **cadena de documentación** (*docstring*) es un literal de tipo cadena que aparece como primera sentencia en un módulo o función.
- Las *docstrings* son comentarios que tienen la finalidad de **documentar** el módulo o la función correspondientes.
- Por convenio, las *docstrings* siempre se delimitan mediante triples dobles comillas (`""" """`).
- La función `help` muestran la *docstring* del objeto para el que se solicita la ayuda.
- Internamente, la *docstring* se almacena en el atributo `__doc__` del objeto.

Ejemplo

```python
"""Módulo de ejemplo (ejemplo.py)."""

def saluda(nombre):
    """Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.

    Returns:
        str: El saludo.
    """
    return "¡Hola, " + nombre + "!"
```

- Existen dos formas distintas de *docstrings*:
  - **De una sola línea (*one-line*):** para casos muy obvios que necesiten poca explicación.
  - **De varias líneas (*multi-line*):** para casos donde se necesita una explicación más detallada.

```
>>> import ejemplo
>>> help(ejemplo)
Help on module ejemplo:

NAME
    ejemplo - Módulo de ejemplo (ejemplo.py).

FUNCTIONS
    saluda(nombre)
        Devuelve un saludo.

        Args:
            nombre (str): El nombre de la persona a la que saluda.

        Returns:
            str: El saludo.

FILE
    /home/ricardo/python/ejemplo.py

>>> help(ejemplo.saluda)
Help on function saluda in module ejemplo:

saluda(nombre)
    Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.

    Returns:
        str: El saludo.
```

- Lo que hace básicamente la función `help(objeto)` es acceder al contenido del atributo `__doc__` del objeto y mostrarlo de forma legible.
- Siempre podemos acceder directamente al atributo `__doc__` para recuperar la *docstring* original usando *objeto*`.__doc__`:

```
>>> import ejemplo
>>> print(ejemplo.__doc__)
Módulo de ejemplo (ejemplo.py).
>>> print(ejemplo.saluda.__doc__)
Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.

    Returns:
        str: El saludo.
```

- Esta información también es usada por otras herramientas de documentación externa, como `pydoc`.

¿Cuándo y cómo usar cada forma de *docstring*?

- ***Docstrings* de una sola línea:**
  - Más apropiada para funciones sencillas.
  - Las comillas de apertura y cierre deben aparecer en la misma línea.
  - No hay líneas en blanco antes o despues de la *docstring*.
  - Debe ser una frase acabada en punto que describa el efecto de la función («Hace *esto*», «Devuelve *aquello*»...).
  - No debe ser una signatura, así que lo siguiente está mal:

```
def funcion(a, b):
    """funcion(a, b) -> tuple"""
```

Esto está mejor:

```
def funcion(a, b):
    """Hace esto y aquello, y devuelve una tupla."""
```

- ***Docstrings* de varias líneas:**
  - Toda la *docstring* debe ir indentada al mismo nivel que las comillas de apertura.
  - La primera línea debe ser un resumen informativo y caber en 80 columnas.

    Puede ir en la misma línea que las comillas de apertura, o en la línea siguiente.
  - A continuación, debe ir una línea en blanco, seguida de una descripción más detallada.
  - La *docstring* de un módulo debe enumerar los elementos que exporta, con una línea resumen para cada uno.
  - La *docstring* de una función debe resumir su comportamiento y documentar sus argumentos, valores de retorno, efectos laterales, excepciones que lanza y precondiciones (si tiene).

pydoc

- El módulo `pydoc` es un generador automático de documentación para programas Python.
- La documentación generada se puede presentar en forma de páginas de texto en la consola, enviada a un navegador web o guardada en archivos HTML.
- Dicha documentación se genera a partir de los *docstrings* de los elementos que aparecen en el código fuente del programa.
- La función `help` llama al sistema de ayuda en línea del intérprete interactivo, el cual usa `pydoc` para generar su documentación en forma de texto para la la consola.

- En la línea de órdenes del sistema operativo, se puede usar `pydoc` pasándole el nombre de una función, módulo o atributo:

  1. Si no se indican más opciones, se visualizará en pantalla la documentación del objeto indicado:

```
$ pydoc sys
$ pydoc len
$ pydoc sys.argv
```

  2. Con la opción `-w` se genera un archivo HTML:

```
$ pydoc -w ejemplo
wrote ejemplo.html
```

  3. Con la opción `-b` se arranca un servidor HTTP y se abre el navegador para visualizar la documentación:

```
$ pydoc -b
Server ready at http://localhost:45373/
Server commands: [b]rowser, [q]uit
server>
```

1.2. Pruebas

doctest

- `doctest` es una herramienta que permite realizar pruebas de forma automática sobre una función.
- Para ello, se usa la *docstring* de la función.
- En ella, se escribe una *simulación* de una pretendida ejecución de la función desde el intérprete interactivo de Python.
- La herramienta comprueba si la salida obtenida coincide con la esperada según dicta la *docstring* de la función.
- De esta forma, la *docstring* cumple dos funciones:
  - Documentación de la función.
  - Especificación de casos de prueba de la función.

```python
# ejemplo.py
def factorial(n):
    """Devuelve el factorial de n, un número entero >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n debe ser >= 0
    """

    import math
    if not n >= 0:
        raise ValueError("n debe ser >= 0")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result
```

```
$ python -m doctest ejemplo.py
$ python -m doctest ejemplo.py -v
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n debe ser >= 0
ok
1 items had no tests:
    ejemplo
1 items passed all tests:
   3 tests in ejemplo.factorial
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

pytest

- `pytest` es una herramienta que permite realizar pruebas automáticas sobre una función o programa Python, pero de una manera más general que con `doctest`.
- La forma más sencilla de usarla es crear una función llamada `test_`⟨*nombre*⟩ por cada función ⟨*nombre*⟩ que queramos probar.
- Esa función `test_`⟨*nombre*⟩ será la encargada de probar automáticamente el funcionamiento correcto de la función ⟨*nombre*⟩.
- Dentro de la función `test_`⟨*nombre*⟩, usaremos la orden **`assert`** para comprobar si se cumple una determinada condición.
- En caso de que no se cumpla, se entenderá que la función ⟨*nombre*⟩ no ha superado dicha prueba.
- En Python 3, la herramienta se llama `pytest-3` y se instala mediante:

```
$ sudo apt install python3-pytest
```

```python
# test_ejemplo.py
def inc(x):
    return x + 1


def test_respuesta():
    assert inc(3) == 5
```

```
$ pytest-3
======================== test session starts ========================
platform linux -- Python 3.8.5, pytest-4.6.9, py-1.8.1, pluggy-0.13.0
rootdir: /home/ricardo/python
collected 1 item

test_ejemplo.py F                                                  [100%]

============================== FAILURES ==============================
_________________________ test_respuesta _________________________

    def test_respuesta():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_ejemplo.py:7: AssertionError
====================== 1 failed in 2.48 seconds ======================
```

- `pytest` sigue la siguiente estrategia a la hora de localizar pruebas:
  - Si no se especifica ningún argumento, empieza a buscar recursivamente empezando en el directorio actual.
  - En esos directorios, busca todos los archivos `test_*.py` o `*_test.py`.
  - En esos archivos, localiza todas las funciones cuyo nombre empiece por `test`.

2. Funciones locales a funciones

2.1. Definición

Definición

- En Python también podemos definir funciones dentro de funciones:

```
def f(...):
    def g(...):
        ...
```

- Cuando definimos una función `g` dentro de otra función `f`, decimos que:
  - `g` es un **función local** o **interna** de `f`.
  - `f` es la **función externa** de `g`.
- También se dice que:
  - `g` es una **función anidada** dentro de `f`.
  - `f` **contiene** a `g`.
- Como `g` se define **dentro** de `f`, sólo es visible dentro de `f`, ya que el **ámbito** de `g` es el cuerpo de `f`.
- El uso de funciones locales evita la superpoblación de funciones en un espacio de nombres cuando esa función sólo tiene sentido usarla en un ámbito más local.

- 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`.
- Por tanto, no se puede usar fuera de `fact`, ya que sólo existe en el ámbito de la función `fact` (es decir, en el cuerpo de la función `fact`).
- Como `fact_iter` sólo existe para ser usada como función auxiliar de `fact`, tiene sentido definirla como una función local de `fact`.
- De esta forma, no contaminaremos el espacio de nombres global con el nombre `fact_iter`, que es el nombre de una función que sólo debe ser usada y conocida por `fact`, y que queda oculta dentro de `fact`.

- Tampoco se puede usar `fact_iter` dentro de `fact` *antes* de definirla:

```
1  def fact(n):
2      print(fact_iter(n, 1))  # UnboundLocalError: se usa antes de definirse
3      def fact_iter(n, acc):  # aquí es donde empieza su definición
4          if n == 0:
5              return acc
6          else:
7              return fact_iter(n - 1, acc * n)
```

- Esto ocurre porque la sentencia **`def`** de la línea 3 crea una ligadura entre `fact_iter` y una variable que apunta a la función que se está definiendo, pero esa ligadura y esa variable sólo empiezan a existir cuando se ejecuta la sentencia **`def`** en la línea 3, y no antes.
- Por tanto, en la línea 2 aún no existe la función `fact_iter` y, por tanto, no se puede usar ahí, dando un error `UnboundLocalError`.

  Esto puede verse como una extensión a la regla que vimos anteriormente sobre cuándo considerar a una variable como local, cambiando «*asignación*» por «*definición*» y «*variable*» por «*función*».

- Como ocurre con cualquier otra función, las funciones locales también determinan un ámbito.
- Ese ámbito, como siempre ocurre, estará anidado dentro del ámbito en el que se define la función.
- En este caso, el ámbito de `fact_iter` está anidado dentro del ámbito de `fact`.
- Asimismo, como ocurre con cualquier otra función, cuando la ejecución del programa entre en el ámbito de `fact_iter` se creará un nuevo marco en el entorno.
- Y, como siempre, ese nuevo marco apuntará al marco del ámbito que lo contiene, es decir, el marco de la función que contiene a la función local.

  En este caso, el marco de `fact_iter` apuntará al marco de `fact`, el cual a su vez apuntará al marco global.

2.2. nonlocal

nonlocal

- Una función local puede **acceder** al valor de las variables locales a la función que la contiene, ya que se encuentran dentro de su ámbito (aunque en otro marco).
- En cambio, cuando una función local quiere **cambiar** mediante una asignación el valor de una variable local a la función que la contiene, deberá declararla previamente como **no local** con la sentencia `nonlocal`.
- De lo contrario, al intentar cambiar el valor de la variable, el intérprete crearía una nueva variable local a la función actual, que haría sombra a la variable que queremos modificar y que pertenece a otra función.
- Es algo similar a lo que ocurre con la sentencia `global` y las variables globales, pero en ámbitos intermedios.
- La sentencia «`nonlocal n`» es una **declaración** que informa al intérprete de que la variable `n` debe buscarla en el entorno saltándose el marco de la función actual y el marco global.

```
1  def fact(n):
2      def fact_iter(acc):
3          nonlocal n
4          if n == 0:
5              return acc
6          else:
7              acc *= n
8              n -= 1
9              return fact_iter(acc)
10     return fact_iter(1)
11
12 print(fact(5))
```

- La función `fact_iter` puede consultar el valor de la variable `n`, ya que es una variable local a la función `fact` y, por tanto, está en el entorno de `fact_iter` (para eso no hace falta declararla como **no local**).
- Como, además, `n` está declarada **no local** en `fact_iter` (en la línea 3), la función `fact_iter` también puede modificar esa variable y no hace falta que la reciba como argumento.
- Esa instrucción le indica al intérprete que, a la hora de buscar `n` en el entorno de `fact_iter`, debe saltarse el marco de `fact_iter` y el marco global y, por tanto, debe empezar a buscar en el marco de `fact`.

3. Funciones genéricas

3.1. Definición y uso

Definición y uso

- Las funciones genéricas se definen de la siguiente forma:

```
def func[T](arg: T): ...
```

- En esta definición, `T` es una **variable de tipo**, es decir, un identificador que representa a un *tipo* cualquiera que en este momento no está determinado.
- Las variables de tipo sirven, por ejemplo, para indicar que todos los elementos de una colección son del mismo tipo.
- Por otra parte, `[T]` representa un **parámetro de tipo**, que sirve para expresar el hecho de que la función que estamos definiendo es genérica y funciona con valores de muchos tipos distintos.
- Esta forma de expresar funciones describe un cierto tipo de polimorfismo llamado **polimorfismo paramétrico**, donde la misma función puede actuar sobre valores de tipos muy diversos. Por eso, a esas funciones las podemos llamar **funciones polimórficas**.

- Por ejemplo, la función que devuelve el máximo elemento de una lista, se puede anotar de la siguiente forma:

```
def maximo[T](l: list[T]) -> T:
  ...
```

y podríamos llamarla con cualquier tipo de lista, siempre que todos los elementos de la lista sean del mismo tipo:

```
>>> maximo([1, 2, 3, 4])
4
>>> maximo(['a', 'b', 'c', 'd'])
'd'
```

- Según la signatura de la función, no sería correcto pasarle una lista con elementos de diferentes tipos, pero el intérprete no detectaría el error ya que no hace comprobación de tipos. De todas formas, la función probablemente no funcionaría bien en ese caso, ya que es una violación de su especificación:

```
>>> maximo([1, 'a', True, 2.5])  # Incorrecto si se comprueban los tipos
???                              # No se sabe cómo se comportará la función en este caso
```

4. Bibliografía

Bibliografía

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.