Calidad

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/07/10 a las 15:12:00

Índice

1. Documentación interna

1.1. Concepto

El uso apropiado de una clara disposición del texto, la inclusión de comentarios apropiados y una buena elección de los identificadores, se conoce como **documentación interna** o **autodocumentación**.

La autodocumentación no es ningún lujo superfluo; por el contrario, se considera preciso adquirir desde el principio el hábito de desarrollar programas claros y bien autodocumentados.

1.2. Comentarios

Recordaremos lo más importante que ya sabemos sobre los comentarios:

- Los comentarios van desde el carácter `#` hasta el final de la línea.
- Para crear comentarios de varias líneas, se puede usar una cadena encerrada entre triples comillas (`"""`).
- Deben expresar información que no sea evidente por la simple lectura del código fuente.
- No deben decir lo mismo que el código, porque entonces será un comentario *redundante* que, si no tenemos cuidado, puede acabar diciendo algo incompatible con el código si no nos preocupamos de actualizar el comentario cuando cambie el código (que lo hará).
- Deben ser los justos: ni más ni menos que los necesarios.

1.3. *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

```
"""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`.

1.3.0.1. ¿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).

1.4. 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>
```

1.5. Estándares de codificación

Un estándar de codificación (o estándar de programación) es un conjunto de normas, reglas y recomendaciones que definen cómo debe escribirse el código fuente en un lenguaje de programación dentro de un proyecto, equipo o empresa.

Los objetivos de un estándar de codificación son:

- Mejorar la legibilidad y comprensión del código por parte de cualquier miembro del equipo.
- Facilitar el mantenimiento y la corrección de errores.
- Asegurar consistencia en el estilo y estructura del código.
- Reducir la probabilidad de errores por prácticas incorrectas o confusas.
- Permitir una colaboración eficiente en proyectos donde varias personas contribuyen.

Qué suele incluir un estándar de codificación:

- Nombres de variables, funciones y clases (por ejemplo, usar `snake_case`, `camelCase` o `PascalCase`).
- Formato y estilo: sangrías, espacios, líneas en blanco.
- Estructura de archivos y carpetas en el proyecto.
- Convenciones de comentarios y documentación.
- Buenas prácticas específicas del lenguaje (por ejemplo, en Python, seguir la norma PEP 8).
- Reglas sobre uso de excepciones, manejo de errores y *logging*.
- Restricciones o recomendaciones de diseño de software (por ejemplo, evitar funciones demasiado largas o complejas).

1.5.1. PEP 8

En Python, el estándar de codificación más utilizado es PEP 8, que define:

- Sangría de 4 espacios.
- Uso de `snake_case` para nombres de funciones y variables.
- Uso de `PascalCase` para nombres de clases.
- Líneas de máximo 79 caracteres.
- Espacios alrededor de operadores.

1.5.2. `pylint`

`pylint` es una herramienta que comprueba determinado tipo de errores en el código fuente de un programa Python.

Trata de asegurar que el programa se ajusta a un estándar de codificación.

Localiza determinados patrones que están mal vistos o que pueden mejorarse fácilmente.

Sugiere cambios en el código, recomienda refactorizaciones y ofrece detalles sobre la complejidad del código.

Se puede instalar o manualmente haciendo:

```
$ pip install pylint
```

Desde la consola, `pylint` se ejecuta directamente sobre un archivo `.py`:

```
$ pylint prueba.py
************* Module prueba
prueba.py:1:0: C0114: Missing module docstring (missing-module-docstring)

-----------------------------------------------------------------------
Your code has been rated at 5.00/10 (previous run: 10.00/10, -5.00)
```

Se puede deshabilitar la comprobación de determinados *defectos* usando la opción `--disable`:

```
$ pylint --disable=missing-docstring prueba.py

-----------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 5.00/10, +5.00)
```

Con la opción `--lists-msgs` se pueden consulta la lista de todos los comprobadores predefinidos.

2. Pruebas

2.1. `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.
```

2.2. 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
```

```
# 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`.

2.3. Desarrollo conducido por pruebas

El **desarrollo conducido por pruebas** o **TDD** (del inglés, *test-driven development*) es una práctica de ingeniería de software que agrupa otras dos prácticas:

- Escribir las pruebas primero (*test first development*).
- Refactorización (*refactoring*).

Para escribir las pruebas generalmente se utilizan **pruebas unitarias** (*unit test*).

El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione.

La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

2.3.1. Ciclo de desarrollo

En primer lugar se debe definir una lista de requisitos y después se ejecuta el siguiente ciclo de siete pasos:

1. **Elegir un requisito:** Se elige el que nos dará mayor conocimiento del problema y que además sea fácilmente implementable.
2. **Escribir una prueba:** Se comienza escribiendo una prueba para el requisito, para lo cual el programador debe entender claramente las especificaciones de la funcionalidad a implementar.
3. **Verificar que la prueba falla:** Si la prueba no falla es porque el requisito ya estaba implementado o porque la prueba es errónea.
4. **Escribir la implementación:** Se escribe el código más sencillo que haga que la prueba funcione.
5. **Ejecutar las pruebas automatizadas:** Se verifica si todo el conjunto de pruebas se pasa correctamente.
6. **Refactorizar:** Se modifica el código para hacerlo más mantenible con cuidado de que sigan pasando todas las pruebas.
7. **Actualizar la lista de requisitos:** Se tacha el requisito implementado y se agregan otros nuevos si hace falta.

Todo este ciclo se resume en que, por cada requisito, hay que hacer:

1. **Rojo:** el test falla
2. **Verde:** se pasa el test
3. **Refactorizar:** se mejora el código

Bibliografía

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