Pruebas unitarias

March 30, 2022

#Pruebas Unitarias

En la mayoría de los campos profesionales, los productos se prueban rigurosamente antes de comercializarse para garantizar la calidad y el funcionamiento previsto.

Se prueban medicamentos, cosméticos, vehículos, teléfonos y computadoras portátiles para garantizar que brinden un cierto nivel de calidad prometido al consumidor. Debido al impacto y alcance del software en nuestra vida diaria, es importante probar nuestro software minuciosamente antes de lanzarlo a los usuarios para evitar problemas de uso.

Hay diferentes métodos y técnicas de probar nuestro software, en esta sección aprenderemos una de las pruebas más fundamentales que se deben realizar a las aplicaciones, las pruebas unitarias.

0.1 Pruebas unitarias frente a otras formas de prueba

Existen diferentes formas de probar el software, principalmente agrupadas en pruebas funcionales y no funcionales. * **Pruebas no funcionales**: tiene como objetivo probar y verificar aspectos no funcionales del software, como la confiabilidad, la seguridad, la usabilidad y la escalabilidad.

- **Pruebas funcionales**: esto implica probar nuestro software con los requisitos funcionales para garantizar que proporcione la funcionalidad requerida. Por ejemplo, podemos probar si nuestra plataforma de compras envía un correo electrónico al usuario después de realizar un pedido simulando este escenario y revisando el correo electrónico.

Las pruebas unitarias se incluyen en las pruebas funcionales, este tipo de prueba se refiere a un método de prueba en el que un programa se divide en varios componentes (funciones) y cada unidad se prueba funcionalmente y por separado de otras funciones o unidades.

Aquí, la unidad se refiere a la parte más pequeña del sistema que pueda realizar una sola función y se pueda probar. El objetivo de las pruebas unitarias es verificar que cada componente del sistema funcione según lo previsto, asegurando así que todo el sistema se ejecute apropiadamente y cumpla con los requisitos funcionales.

0.2 El módulo `pytest`

pytest es el marco de pruebas más popular para Python. Usando pytest podemos probar cualquier cosa, desde scripts básicos de python hasta bases de datos o interfaces de usuario.

Instalación: Podemos instalar pytest desde PyPI usando el comando (Google colab ya la trae instalada)

```
pip install pytest
```

Pytest nos permite realizar distintos experimentos para garantizar que el comportamiento de las funciones que diseñemos es el esperado, para esto primero debemos analizar la sentencia `assert`

0.2.1 `assert`

La sentencia `assert` nos permite verificar una condición lógica, cuando su resultado es verdadero, el programa continúa su ejecución, si por el contrario obtenemos un resultado falso, la sentencia generará una excepción de tipo `AssertionErrorException`, por lo tanto podríamos decir que esta sentencia nos permite "garantizar" que un programa siga un mismo comportamiento a la hora de ejecutar cualquier operación.

Esta sentencia, nos permite construir con el módulo pytest experimentos donde esperemos una respuesta específica de una función y en caso contrario, marcar como fallida la prueba.

0.3 Sintaxis script de pruebas

```
from modulo_funciones import *
import pytest

def test_prueba1():
    assert condicion1

def test_prueba2():
    with pytest.raises(Excepcion):
       func()
```

- Es importante tener en cuenta que **obligatoriamente** cada función de prueba debe iniciar con la palabra `test` de lo contrario el sistema no la ejecutará.
- Por buena práctica debemos separar nuestros algoritmos del script donde diseñemos las pruebas, por lo tanto, es necesario importar las funciones con `import` y el nombre del archivo que las contenga.
- El nombre del script de las pruebas también **debe iniciar con la palabra test** de lo contrario pytest no lo reconocerá.
- Si esperamos que una prueba genere una excepción, podemos indicarle este comportamiento a pytest con la instrucción `with pytest.raise(Excepcion)` donde debemos reemplazar `Excepcion` por el tipo específico de excepción que lanzará la función.

0.4 Escritura y ejecución de bloques de código como scripts

Google Colab nos permite convertir nuestros bloques de código en scripts independientes de Python. Para hacer esto, basta con añadir el siguiente comando en la primera línea del bloque:

`%%writefile nombre_archivo.py`

para llamar el script que guardamos, podemos usar el siguiente comando:

`%run nombre_archivo`

[ ]:
```
%%writefile nombre_archivo.py
print('hola mundo')
```

```
Writing nombre_archivo.py
```

[ ]:
```
%run nombre_archivo
```

```
hola mundo
```

0.5 Ejecución de pruebas con pytest

Antes de poder demostrar cómo se ejecuta una batería de pruebas unitarias dentro de pytest, debemos crear las funciones que probaremos. Diseñemos un programa que retorne el área de cualquier rectángulo y almacenémoslo en `rectangulo.py`. Usaremos la fórmula:

$$rea = base \times altura$$

[ ]:
```
%%writefile rectangulo.py

def area_rectangulo(base,altura):
    """
    Función que retorna el área de un rectángulo dados
    su base y su altura
    """
    return base*altura
```

```
Overwriting rectangulo.py
```

Ahora, creemos un script con los siguientes experimentos: * `area_rectangulo(2,2) == 4` * el área no puede ser negativa * solo se deben aceptar valores numéricos

[ ]:
```
%%writefile test_rectangulo.py

from rectangulo import *
import pytest

def test_area():
    """
    comportamiento esperado:
        área de rectángulo 2x2 = 4
    """
    assert area_rectangulo(2,2) == 4

def test_negativo():
    """
    comportamiento esperado:
        el área de cualquier figura no puede ser negativa
    """
    with pytest.raises(ValueError):
            area_rectangulo(2,-2)
```

```
def test_numeros():
    """
    comportamiento esperado:
        la función no debe recibir datos que no sean números
    """
    with pytest.raises(ValueError):
        area_rectangulo('hola',4)
```

```
Overwriting test_rectangulo.py
```

Para ejecutar las pruebas, en un computador, de forma local, basta con ingresar a una consola en el directorio que contenga los script y ejecutar

```
pytest
```

En Google Colab, para indicarle al bloque de código que la instrucción no es de python si no directamente de la consola, añadimos un signo de admiración `!` al inicio.

```
!pytest
```

[ ]:
```
!pytest
```

```
============================= test session starts
==============================
platform linux -- Python 3.7.12, pytest-3.6.4, py-1.11.0, pluggy-0.7.1
rootdir: /content, inifile:
plugins: typeguard-2.7.1
collected 3 items

test_rectangulo.py .FF
[100%]

================================== FAILURES ==================================
_______________________________ test_negativo
--------------------------------

    def test_negativo():
        """
        comportamiento esperado:
            el área de cualquier figura no puede ser negativa
        """
        with pytest.raises(ValueError):
>               area_rectangulo(2,-2)
E               Failed: DID NOT RAISE <class 'ValueError'>
```

```
test_rectangulo.py:18: Failed
_____________________________ test_numeros

_____________________________

    def test_numeros():
        """
        comportamiento esperado:
            la función no debe recibir datos que no sean números
        """
        with pytest.raises(ValueError):
>           area_rectangulo('hola',4)
E           Failed: DID NOT RAISE <class 'ValueError'>

test_rectangulo.py:27: Failed
====================== 2 failed, 1 passed in 0.03 seconds

======================
```

Como vemos en los resultados de la ejecución de la prueba, `test_negativo` falló debido a que al ingresar un valor negativo, no se lanzó una excepción. La prueba `test_numeros` falló porque no se lanzó una excepción al ingresar como argumento un string.

Al inicio de la impresión vemos lo siguiente

```
test_rectangulo.py .FF                                                   [100%]
```

el punto `.` representa la primera prueba, un punto indica que el test pasó exitosamente, las dos F `FF` representan las dos pruebas fallidas y finalmente el `100%` nos indica que pytest pudo ejecutar todas las pruebas del script.

Ahora, ¿cómo podemos corregir nuestra función para que las pruebas pasen correctamente?

- Para garantizar que un parámetro sea de un tipo de dato específico, podemos usar la función `isinstance`

```
isinstance(variable, (tipo1,tipo2,...))
```

  El primer argumento de esta función es la variable que queremos comparar, el segundo argumento es un tipo de dato (`int`, `float`, `str`, etc) o una tupla que contenga dos o más de estos, si la variable es de alguno de los tipos que estamos comparando, la función retornará un `True`, en caso contrario, `False`.

- Para asegurarnos que el área sea positiva, basta con garantizar que ni la base, ni la altura sean negativas.

De este modo, nuestra nueva función será la siguiente.

[ ]:
```
%%writefile rectangulo.py

def area_rectangulo(base,altura):
    """
    Función que retorna el área de un rectángulo dados
```

```
    su base y su altura
    """
    if not isinstance(base,(int,float)):
        raise ValueError('la base debe ser un número real')
    if not isinstance(altura,(int,float)):
        raise ValueError('la altura debe ser un número real')
    if base <= 0 or altura <= 0:
        raise ValueError('la base y la altura deben ser valores positivos')
    return base*altura
```

```
Overwriting rectangulo.py
```

- La instrucción `raise` nos permite invocar una excepción a voluntad en cualquier línea del código, el texto entro los paréntesis es el mensaje que mostrará el error.

Si lanzamos nuevamente la prueba:

[ ]:
```
!pytest
```

```
============================= test session starts

==============================
platform linux -- Python 3.7.12, pytest-3.6.4, py-1.11.0, pluggy-0.7.1
rootdir: /content, inifile:
plugins: typeguard-2.7.1
collected 3 items


test_rectangulo.py …

[100%]

========================== 3 passed in 0.01 seconds

==========================
```

Ahora la prueba nos presenta el mensaje:

```
test_rectangulo.py ...                                               [100%]

========================== 3 passed in 0.01 seconds ==========================
```

¡Tanto los tres puntos `...`, como el mensaje entre los iguales, nos indica que las 3 pruebas se ejecutaron exitosamente!