

Pruebas con Python

Unittest



Índice

Introducción	3
Un proceso de automatización de verificación	4
Ejemplos de uso	7
Testing con excepciones	9
Verificación de tipos	11
Buenas prácticas en testing unitario	14
Códigos usados en los ejemplos	14

Introducción

Existen numerosas librerías para unit testing.

Posiblemente unittest es el estándar para pruebas de testing. Es el marco de prueba más extendido y además ya viene incorporado en el propio Python. Para usarlo no hay más que importarlo.

Un proceso de automatización de verificación

Con unittest vamos a hacer un proceso de automatización de la verificación de nuestro código.

Por ejemplo, partimos de la siguiente función para calcular un área:

```
from math import pi

def area(r):
    areaC = pi*(r**2)
    return areaC
```

Sabemos que a esta función no podemos darle determinados valores, como por ejemplo valores negativos, strings o booleans.

Vamos a hacer una prueba, de momento sin usar unittest. Vamos a crear una lista con una serie de valores que vamos a pasar como parámetro y vamos a ver el comportamiento que tiene esa función con esos valores.

```
valores = [1, 3, 0, -1, -3, 2+3j, True, 'hola']
```

Viendo el comportamiento de la función al pasarle los parámetros sabremos si la función está trabajando correctamente, aunque nosotros le estemos pasando valores erróneos.

Es una forma de poder conocer con qué valores va a funcionar correctamente mi función y con cuáles no.

Recorremos nuestra función pasándole los parámetros de nuestra lista:

```
for v in valores:
    areaCalculada = area(v)
    print('Para el valor', v, 'el área es', areaCalculada)
```

Y obtenemos el siguiente resultado:

```
... Para el valor 1 el área es 3.141592653589793
Para el valor 3 el área es 28.274333882308138
Para el valor 0 el área es 0.0
Para el valor -1 el área es 3.141592653589793
Para el valor -3 el área es 28.274333882308138
Para el valor (2+3j) el área es (-15.707963267948966+37.69911184307752j)
Para el valor True el área es 3.141592653589793

</>
-----
TypeError                                Traceback (most recent call last)
j:\WORKSPACE\10 PYTHON\029_testing_01.py in line 10
     8 valores = [1, 3, 0, -1, -3, 2+3j, True, 'hola']
     9 for v in valores:
--> 10     areaCalculada = area(v)
    11     print('Para el valor', v, 'el área es', areaCalculada)

j:\WORKSPACE\10 PYTHON\029_testing_01.py in line 3, in area(r)
     2 def area(r):
--> 3     areaC = pi*(r**2)
     4     return areaC

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Vemos que:

- Para los valores **1**, **3** y **0** nos da resultados correctos
- Para **-1** y **-3** nos da valores que no son lo que debería dar, por lo que ya sabemos que, cuando ingresemos radios negativos, tendremos que hacer algo al respecto, como por ejemplo lanzar una excepción.
- Para el número complejo **(2+3j)** el resultado tampoco es lo esperado.
- Para el valor **True** el resultado tampoco es lo esperado.
- Para el valor **'hola'** directamente obtenemos un error de tipo.

Ya sabemos que nuestra función está bien realizada, hace su trabajo, pero dependiendo de los valores que le pasemos, puede dar problemas. Hace cálculos que no se deberían llevar a cabo e incluso da error cuando le pasamos una cadena de texto.

Este es un test muy primitivo. Haciendo uso de unittest podemos automatizar el proceso de prueba de nuestro código.

Para usar unittest tenemos que seguir un protocolo, vamos a tener que crear otro archivo en el que se va a encontrar el propio código de unittest.

Deberemos crear un archivo con el siguiente nombre:

test_nombreArchivo.py

El nombre deberá empezar por la palabra **test_** seguido del nombre del archivo sobre el que vamos a realizar las pruebas. De tal forma que, si nuestro archivo se llama, por ejemplo, **funcionRadio.py**, el archivo de testing se debería llamar **test_funcionRadio.py**.

En nuestro caso el archivo a probar se va a llamar **testing01.py** y el archivo de pruebas **test_testing01.py**

Para ejecutarlo, pondremos en la consola:

```
python -m unittest test_testing01.py
```

Donde **test_testing01.py** será el nombre de mi archivo de pruebas.

Nota: Podemos ejecutar automáticamente las pruebas usando el comando:

python -m unittest discover

Con esta sintaxis no necesitamos mencionar el nombre de archivo de la prueba. Unittest encontrará las pruebas utilizando la convención de nomenclatura que seguimos.

Entonces, debemos nombrar nuestros archivos de prueba con la palabra clave **test** en el arranque.

Importamos **unittest**:

```
import unittest
```

Importamos la función sobre la cual se van a ejecutar los test:

```
from testing01 import area
```

Y en nuestro caso tendremos que importar también **PI**:

```
from math import pi
```

Lo siguiente será crear una clase que herede de la clase **TestCase**:

```
class TestArea(unittest.TestCase):
```

En esta clase es donde vamos a poner nuestros test.

Vamos a crear al primero. Lo que haremos es crear un método cuyo nombre debe empezar por la palabra **test**.

Ahora, dependiendo del tipo de test que queramos hacer, deberemos invocar un método de la clase **TestCase**. Los más usados son:

assertTrue() o **assertFalse()** para verificar una condición.

assertRaises() para asegurar que se lanza una excepción específica. Se utilizan estos métodos en lugar de la sentencia **assert** para que el ejecutor de pruebas pueda acumular todos los resultados de la prueba de cara a realizar un informe.

Los métodos **setUp()** y **tearDown()** permiten definir instrucciones que han de ser ejecutadas antes y después, respectivamente, de cada método de prueba.

assertAlmostEqual() para comprobar si el resultado de una prueba es exactamente igual a un dato que conocemos, por ejemplo una variable en concreto.

El quid de cada test es la llamada a **assertEqual()** para verificar un resultado esperado.

Ejemplos de uso

Veamos ejemplos de uso de alguno de estos métodos.

El primer test que vamos a crear es para saber si nuestra función ha generado un valor que sabemos que es correcto. Vamos a meter un valor conocido y ver si el resultado corresponde a lo que sabemos que debería dar.

Para ello debemos invocar al método **assertAlmostEqual()** pasando como primer parámetro nuestra función con el parámetro de entrada y como segundo parámetro el resultado que sabemos que debe dar.

```
import unittest
from testing01 import area
from math import pi

class TestArea(unittest.TestCase):
    def test_area(self):
        print('-----Test valores de
resultado conocido-----')
        self.assertAlmostEqual(area(1),
pi)
        self.assertAlmostEqual(area(0),
0)
        self.assertAlmostEqual(area(3),
pi*(3**2))
```

Recordemos que tenemos que ejecutar nuestro test poniendo en la consola:

python -m unittest test_testing01.py

```
PS J:\WORKSPACE\10 PYTHON> python -m unittest test_testing01.py
-----Test valores de resultado conocido-----
.
-----
Ran 1 test in 0.000s

OK
```

No nos indica ningún fallo, por lo que podemos decir que las pruebas fueron positivas. Es decir, todos los resultados reales son iguales a los esperados.

Vamos a ejecutar de nuevo, pero forzando a que se produzca un fallo para ver la diferencia. En la línea 10 de nuestro programa hemos cambiado el valor esperado de **0** por un **1**, que sabemos que no sería correcto.

Ejecutamos de nuevo con:

```
python -m unittest test_testing01.py
```

Y el resultado:

```
WORKSPACE > 10 PYTHON > Testing > test_testing01.py > ...
1
2 import unittest
3 from testing01 import area
4 from math import pi
5
6 class TestArea(unittest.TestCase):
7     def test_area(self):
8         print('-----Test valores de resultado conocido-----')
9         self.assertAlmostEqual(area(1), pi)
10        self.assertAlmostEqual(area(0), 1)
11        self.assertAlmostEqual(area(3), pi*(3**2))
12
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
FAIL: test_area (test_testing01.TestArea)
-----
Traceback (most recent call last):
  File "J:\WORKSPACE\10 PYTHON\Testing\test_testing01.py", line 10, in test_area
    self.assertAlmostEqual(area(0), 1)
AssertionError: 0.0 != 1 within 7 places (1.0 difference)

-----
Ran 1 test in 0.000s

FAILED (failures=1)
PS J:\WORKSPACE\10 PYTHON\Testing> 
```

Vemos que nos da un fallo en la línea 10. Era lo esperado.

Testing con excepciones

Uno de los problemas que tenemos con funciones que reciben valores numéricos es que los valores introducidos no se encuentran en el rango adecuado. Por ejemplo, en nuestra función el rango de valores negativos no es factible.

Si ya sabemos que nuestra función no va a sacar valores coherentes si los parámetros de entrada son negativos, lo que tenemos que hacer es crear una excepción que de lance en dichos casos.

En unittest no solo tenemos que comprobar que la función nos devuelva los resultados correctos, también tenemos que asegurarnos de que se disparan las excepciones correctamente según los casos.

Ejemplo:

En nuestro archivo **testing01.py** modificamos la función a comprobar añadiéndole una excepción:

```
def area(r):
    if r<0:
        raise ValueError("No se permiten valores negativos")
    areaC = pi*(r**2)
    return areaC
```

Hemos creado la misma función, pero le hemos añadido una excepción que se disparará en el caso de encontrar parámetros de entrada negativos. Para el resto de valores la función se ejecuta como antes.

En nuestro archivo **test_testing01.py** creamos una nueva función que compruebe la correcta ejecución en caso de valores de entrada negativos.

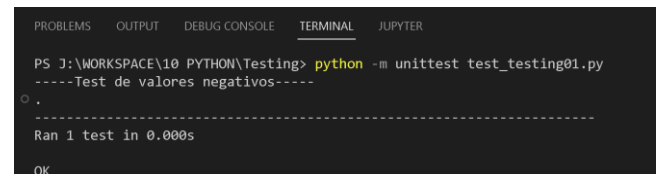
Para ello tendremos que usar el método `assertRaises` al que hay que pasarle tres parámetros; el tipo de excepción, la función a comprobar y el valor de salida esperado:

```
#Test de valores negativos
def test_negativos(self):
    print('-----Test de valores negativos-----')

    #Indicamos el tipo de excepción, la función y el valor esperado.
    self.assertRaises(ValueError, area, -1)
```

Al ejecutar el código con **python -m unittest test_testing01.py**

Nos muestra el siguiente mensaje:



```
PS J:\WORKSPACE\10 PYTHON\Testing> python -m unittest test_testing01.py
-----Test de valores negativos-----
.
Ran 1 test in 0.000s
OK
```

Como vemos, no nos saca ningún mensaje de error, lo que significa que la excepción se ha lanzado correctamente.

Vamos a ver qué hubiese pasado si no tenemso prevista la excepción en nuestro código.

Modificamos nuestra función, comentamos la línea de la excepción y la sustituimos simplemente por un **print()**:

```
def area(r):
    if r<0:
        #raise ValueError("No se permiten valores negativos")
        print("No se permiten valores negativos")
    areaC = pi*(r**2)
    return areaC
```

Ejecutamos el test:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

-----Test de valores negativos-----
No se permiten valores negativos
F
=====
FAIL: test_negativos (test_testing01.TestArea)
-----
Traceback (most recent call last):
  File "J:\WORKSPACE\10 PYTHON\Testing\test_testing01.py", line 20, in test_negativos
    self.assertRaises(ValueError, area, -1)
AssertionError: ValueError not raised by area

-----
Ran 1 test in 0.001s

FAILED (failures=1)
```

Vemos que en esta ocasión nos da un fallo ya que no hemos hecho un tratamiento correcto de la excepción.

Por lo tanto ya hemos visto como con `assertRaises()` comprobamos que la función está ejecutando la excepción adecuada en el caso de que pasemos un parámetros que no sea válido.

Verificación de tipos

Vamos a ver ahora cómo comprobar si el tipo de datos que está recibiendo la función mediante parámetro es correcto o no.

Para este tipo de test vamos a usar nuevamente el método `assertRaises()` y el tipo de excepción que estamos esperando que se lance es de tipo `TypeError`.

Creamos una prueba para cada tipo de dato que sabemos con certeza que no nos va a valer en nuestra función:

```
# Test de tipos no compatibles.  
# Verificamos si el tipo de los  
parámetros es el correcto.  
# El tipo de la excepción debe  
ser TypeError  
# Hacemos una prueba para que  
cada tipo conocido no válido  
  
def test_tipos(self):  
    print('-----Test de tipos no  
compatibles-----')  
    self.assertRaises(TypeError,  
area, True)  
    self.assertRaises(TypeError,  
area, 'hola')  
    self.assertRaises(TypeError,  
area, 2+3j)
```

Si nuestra función lanza una excepción al recibir este tipo de datos, nuestro test no nos dará ningún error. Si la función no lanza una excepción, entonces el test sí nos dará errores.

El resultado:

```
30
31     # Test de tipos no compatibles. Verificamos si el tipo de los parámetros es el correcto.
32     # El tipo de la excepción debe ser TypeError
33     # Hacemos una prueba para que cada tipo conocido no válido
34
35     def test_tipos(self):
36         print('-----Test de tipos no compatibles-----')
37         self.assertRaises(TypeError, area, True)
38         self.assertRaises(TypeError, area, 'hola')
39         self.assertRaises(TypeError, area, 2+3j)
40
41
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
-----Test de tipos no compatibles-----
F
=====
FAIL: test_tipos (test_testing01.TestArea)
-----
Traceback (most recent call last):
  File "J:\WORKSPACE\10 PYTHON\Testing\test_testing01.py", line 37, in test_tipos
    self.assertRaises(TypeError, area, True)
AssertionError: TypeError not raised by area

-----
Ran 1 test in 0.000s
```

Vemos que nuestro test ha lanzado errores. En concreto se queja de que la ejecución correspondiente a la línea 37 ha fallado. Es decir, que nuestra función no lanza errores si el tipo de dato recibido es un boolean.

Ya tenemos información de los cambios que debemos hacer en nuestra función **area**.

Vamos a crear una nueva excepción que trate los casos en los que el parámetro de entrada no sea un número entero o de coma flotante:

```
#Función con la excepción TypeError y
verificación de negativos

def area(r):

    #Verificamos los tipos correctos
    if type(r) not in [float, int]:
        raise TypeError("Solo números
enteros o de coma flotante.")

    #Verificamos los valores negativos
    if r<0:
        raise ValueError("No se permiten
valores negativos")

    areaC = pi*(r**2)
    return areaC
```

Si lanzamos nuestro test ahora:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

PS J:\WORKSPACE\10 PYTHON\Testing> python -m unittest test_testing01.py
-----Test de tipos no compatibles-----
.
-----
Ran 1 test in 0.000s

OK
```

Ahora sí tenemos el test sin fallos. Lo que significa que ya tenemos tratados los casos en los que los valores de entrada no sean números enteros positivos o números de coma flotante positivos.

Como podemos ver los unittest son una herramienta sencilla y poderosa con la que poder automatizar el proceso de testing de nuestro código. Comprobando los valores que conocemos con certeza que son válidos, los que conocemos con certeza que no lo son y las excepciones de nuestro código.

A continuación se expone un decálogo de buenas prácticas a la hora de crear pruebas unitarias.

Buenas prácticas en testing unitario

- Las pruebas deben ser pequeñas y probar solo una cosa.
- Las pruebas deben ejecutarse rápido. Esto es esencial para la inclusión en un entorno de CI.
- Las pruebas unitarias deben ser completamente independientes. Las pruebas no deben depender unas de otras y pueden ejecutarse en cualquier orden cualquier número de veces.
- Las pruebas deben estar completamente automatizadas y no requerir interacción manual o verificaciones para determinar si pasan o fallan.
- Las pruebas no deben incluir afirmaciones innecesarias como "puntos de control" en la prueba. Afirma solo lo que la prueba está probando.
- Las pruebas deben ser portátiles y ejecutarse fácilmente en diferentes entornos.
- Las pruebas deben configurar lo que necesitan para ejecutarse. Las pruebas no deben hacer suposiciones sobre recursos particulares existentes.
- Las pruebas deben limpiar los recursos creados después.
- Los nombres de las pruebas deben describir claramente lo que están probando.

- Las pruebas deben producir mensajes significativos cuando fallan. Intenta hacer que la prueba falle y ver si el motivo del fallo se puede determinar leyendo el resultado del caso de prueba.

Códigos usados en los ejemplos

Nuestros códigos finalmente deberían haber quedado de la siguiente forma.

Archivo **testing01.py**:

```
from math import pi

def area(r):

    #Verificamos los tipos correctos
    if type(r) not in [float, int]:
        raise TypeError("Solo números
enteros o de coma flotante.")

    if r<0:
        raise ValueError("No se permiten
valores negativos")

    areaC = pi*(r**2)
    return areaC
```

Archivo `test_testing01.py`:

```
import unittest
from testing01 import area
from math import pi

class TestArea(unittest.TestCase):

    def test_area(self):
        print('-----Test valores de
resultado conocido-----')
        self.assertAlmostEqual(area(1),
pi)
        self.assertAlmostEqual(area(0), -
1)
        self.assertAlmostEqual(area(3),
pi*(3**2))

        #Test de valores negativos
        def test_negativos(self):
            print('-----Test de valores
negativos-----')

            #Indicamos el tipo de excepción,
la función y el valor esperado.
            self.assertRaises(ValueError,
area, -1)

        # Test de tipos no compatibles.
Verificamos si el tipo de los parámetros
es el correcto.
        # El tipo de la excepción debe
ser TypeError
        # Hacemos una pueba para que cada
tipo conocido no válido

    def test_tipos(self):
        print('-----Test de tipos no
compatibles-----')
        self.assertRaises(TypeError,
area, True)
        self.assertRaises(TypeError,
area, 'hola')
        self.assertRaises(TypeError,
area, 2+3j)
```