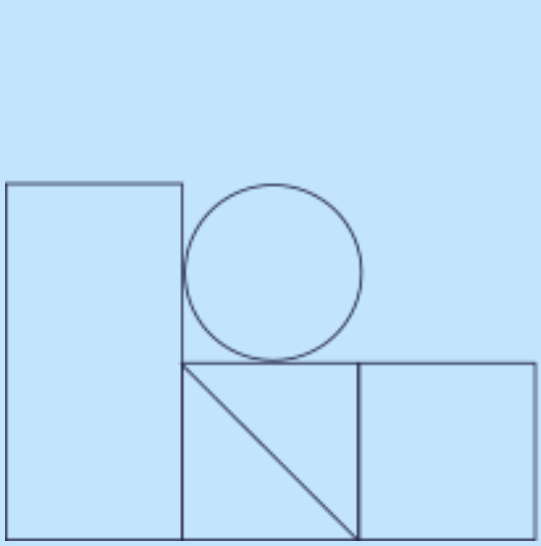
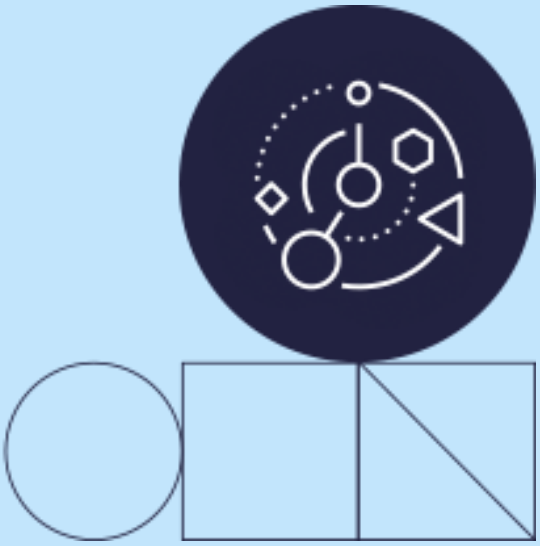
IBM SkillsBuild

IBM

Testes com Python

Unittest





Índice

Introdução

Há numerosas bibliotecas para testes unitários. Possivelmente o unittest é indiscutivelmente o padrão para testes unitários. É a estrutura de testes mais difundida e já está integrada na própria Python. Para utilizá-lo, basta importá-lo.

Um processo de automação de verificação

Com unittest vamos fazer um processo de automatização da verificação do nosso código.

Por exemplo, começamos com a seguinte função para calcular uma área:

```
from math import pi

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

Sabemos que não podemos dar a esta função certos valores, tais como valores negativos, strings ou booleans.

Vamos fazer um teste, por enquanto, sem usar unittest. Vamos criar uma lista com uma série de valores que vamos passar como parâmetros e vamos ver como essa função se comporta com esses valores.

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

Vendo o comportamento da função ao passar os parâmetros, saberemos se a função está funcionando corretamente, mesmo que estejamos passando valores errados.

É uma maneira de saber com quais valores minha função vai funcionar corretamente e com quais não vai funcionar corretamente.

Atravessamos nossa função, passando-lhe os parâmetros de nossa lista:

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

E obtemos o seguinte 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 os valores **1**, **3** e **0** dá resultados corretos
- Para **-1** e **-3** nos dá valores que não são o que deveria dar, então já sabemos que, quando entramos em raios negativos, teremos que fazer algo a respeito, tal como lançar uma exceção.
- Para o número complexo **(2+3j)** o resultado também não é o esperado.
- Para o valor **True** o resultado também não é o esperado.
- Para o valor **'hola'** Recebemos diretamente um erro de tipo.

Já sabemos que nossa função é bem feita, ela faz seu trabalho, mas dependendo dos valores que lhe passamos, ela pode dar problemas. Ele faz cálculos que não devem ser efetuados e até dá um erro quando passamos uma seqüência de texto.

Este é um teste muito primitivo. Fazendo uso de unittest podemos automatizar o processo de teste de nosso código.

Para usar unittest temos que seguir um protocolo, vamos ter que criar outro arquivo no qual vamos encontrar o código real de unittest.

Devemos criar um arquivo com o seguinte nome:

test_nombreArchivo.py

O nome deve começar com a palavra **test_** seguido do nome do arquivo que vamos testar. Assim, se nosso arquivo for chamado, por exemplo, **funcionRadio.py**, o arquivo de teste deve ser chamado **test_funcionRadio.py**.

Em nosso caso, o arquivo a ser testado será chamado **testing01.py** e o arquivo de teste **test_testing01.py**

Para executá-lo, vamos colocar no console:

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

Onde **test_testing01.py** será o nome do meu arquivo de teste.

Nota: Podemos executar automaticamente os testes usando o comando:

python -m unittest discover

Com esta sintaxe, não precisamos mencionar o nome do arquivo do teste. Unittest Portanto, devemos nomear nossos arquivos de teste com a palavra-chave **test** no início.

Importamos **unittest**:

```
import unittest
```

Importamos a função na qual os testes devem ser executados:

```
from testing01 import area
```

E em nosso caso também teremos que importar **PI**:

```
from math import pi
```

O próximo passo é criar uma classe que herde da classe **TestCase**:

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

Esta classe é onde vamos colocar nossos testes.

Vamos criar o primeiro. O que vamos fazer é criar um método cujo nome deve começar com a palavra **test**.

Agora, dependendo do tipo de teste que queremos fazer, devemos invocar um método da classe **TestCase**. As mais comumente utilizadas são:

assertTrue() ou **assertFalse()** para verificar uma condição.

assertRaises() para garantir que uma exceção específica seja lançada. Estes métodos são utilizados em vez da declaração **assert** para que o testador possa acumular todos os resultados do teste para fins de relatório.

Os métodos **setUp()** e **tearDown()** permitir a definição de instruções a serem executadas antes e depois, respectivamente, de cada método de teste.

assertAlmostEqual() para verificar se o resultado de um teste é exatamente o mesmo que um dado que conhecemos, por exemplo, uma determinada variável.

O ponto crucial de cada teste é a chamada para **assertEqual()** para verificar um resultado esperado;

Exemplos de uso

Vejamos exemplos de utilização de alguns desses métodos.

O primeiro teste que vamos criar é para ver se nossa função gerou um valor que sabemos ser correto. Vamos entrar com um valor conhecido e ver se o resultado corresponde ao que sabemos que ele deve dar.

Para fazer isso, devemos invocar o método **assertAlmostEqual()** passando como primeiro parâmetro nossa função com o parâmetro de entrada e como segundo parâmetro o resultado que sabemos que deve 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))
```

Lembremos que temos que fazer nosso teste colocando no console:

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
```

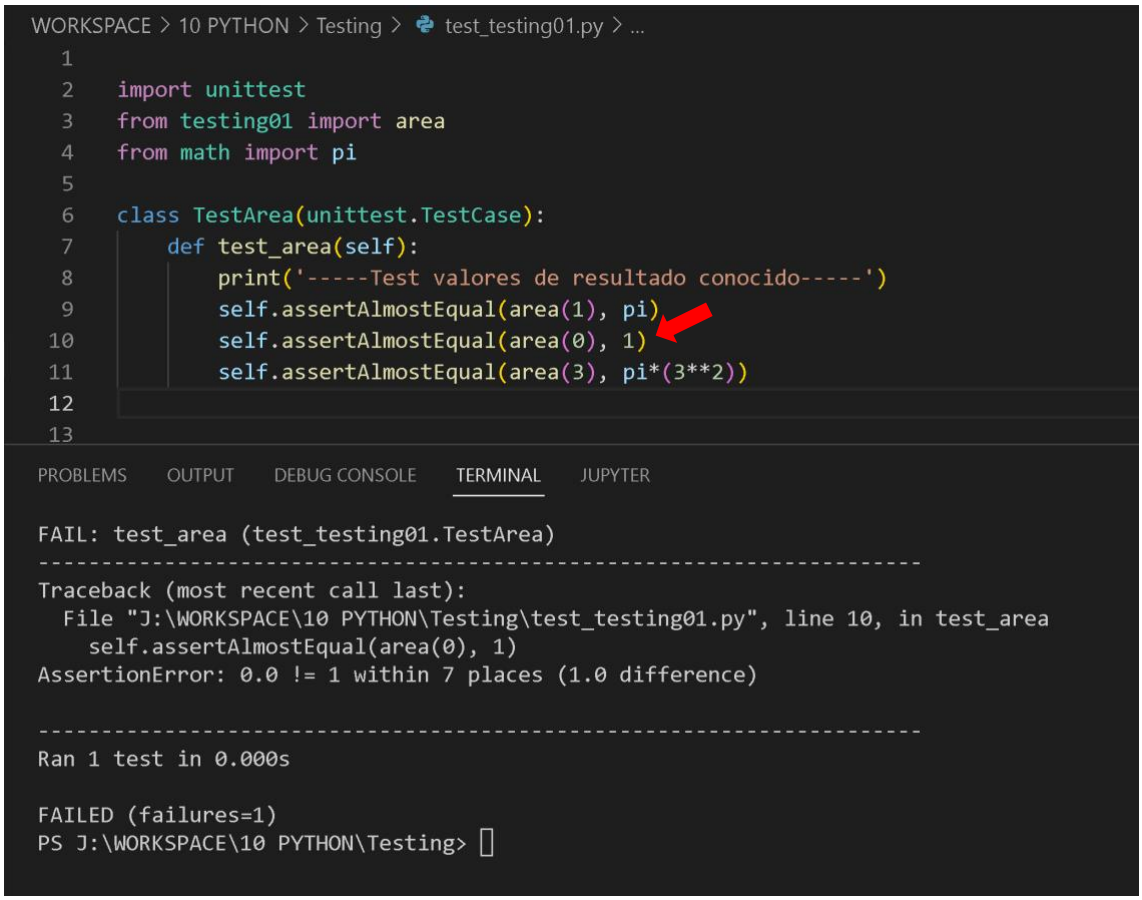
Ele não indica nenhuma falha, portanto podemos dizer que os testes foram positivos. Ou seja, todos os resultados reais são os mesmos que os esperados.

Vamos repetir, mas forçar uma falha para não ver a diferença. Na linha 10 de nosso programa, alteramos o valor esperado de **0** por um **1**, o que sabemos que não seria correto.

Corremos novamente com:

python -m unittest test_testing01.py

E o resultado:



Vemos que temos uma falha na linha 10. Isto era esperado.

Testing com exceções

Um dos problemas que temos com funções que recebem valores numéricos é que os valores inseridos não estão na faixa correta. Por exemplo, em nossa função, a gama de valores negativos não é viável.

Se já sabemos que nossa função não vai produzir valores consistentes se os parâmetros de entrada forem negativos, o que temos que fazer é criar uma exceção para lançar em tais casos.

No unittest não só temos que verificar se a função retorna os resultados corretos, mas também temos que nos certificar de que as exceções sejam disparadas corretamente de acordo com os casos.

Exemplo:

Em nosso **testing01.py** modificamos a função a ser verificada, acrescentando-lhe uma exceção:

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

Criamos a mesma função, mas acrescentamos uma exceção que será acionada em caso de parâmetros de entrada negativos. Para todos os outros valores, a função é executada como antes.

Em nosso arquivo **test_testing01.py** criamos uma nova função para verificar a correta execução em caso de valores de entrada negativos.

Para isso, temos que usar o método assertRaises, ao qual três parâmetros devem ser passados; o tipo de exceção, a função a ser verificada e o valor de saída 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)
```

Ao executar o código com **python -m unittest test_testing01.py**

A seguinte mensagem é exibida:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

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

OK
```

Como podemos ver, não recebemos nenhuma mensagem de erro, o que significa que a exceção foi lançada corretamente.

Vamos ver o que teria acontecido se não tivéssemos a exceção em nosso código.

Modificamos nossa função, comentamos a linha de exceção, e a substituímos por uma simples **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
```

Fazemos o teste:

```
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, nesta ocasião, temos um fracasso porque não fizemos um tratamento correto da exceção.

Portanto, já vimos como com **assertRaises()** verificamos se a função está executando a exceção apropriada no caso de passarmos um parâmetro que não é válido.

Verificação de tipos

Vamos ver agora como verificar se o tipo de dados que a função está recebendo por parâmetro está correto ou não.

Para este tipo de teste, vamos usar novamente o método **assertRaises()** e o tipo de exceção que estamos esperando que seja lançada é do tipo **TypeError**.

Criamos um teste para cada tipo de dado que sabemos com certeza não vai funcionar em nossa função:

```
    # 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)
```

Se nossa função lançar uma exceção ao receber este tipo de dados, nosso teste não nos dará nenhum erro. Se a função não lançar uma exceção, então o teste nos dará erros.

O 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 pueba 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 nosso teste lançou erros. Especificamente, ela reclama que a execução correspondente à linha 37 falhou. Ou seja, nossa função não atira erros se o tipo de dados recebidos for um boolean.

Já temos informações sobre as mudanças que precisamos fazer em nossa função **area.**

Vamos criar uma nova exceção para lidar com casos em que o parâmetro de entrada não seja um número inteiro ou de ponto flutuante:

```
#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("Sólo 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
```

Se lançarmos nosso teste agora:

```
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
```

Agora temos o teste sem falhas. Isto significa que já lidamos com os casos em que os valores de entrada não são inteiros positivos ou números positivos de ponto flutuante.

Como podemos ver, os unittest são ferramentas simples e poderosas, com a qual podemos automatizar o processo de testing de nosso código. Verificar os valores que sabemos ao certo que são válidos, aqueles que sabemos ao certo que não são e as exceções do nosso código.

O seguinte é um decálogo de boas práticas ao criar testes unitários.

Melhores práticas em testes unitários

- Os testes devem ser pequenos e testar apenas uma coisa.
- Os testes devem ser rápidos. Isto é essencial para a inclusão em um ambiente CI.
- Os testes unitários devem ser completamente independentes. Os testes não devem depender uns dos outros e podem ser realizados em qualquer ordem, quantas vezes forem necessárias.
- Os testes devem ser totalmente automatizados e não requerem interação manual ou verificações para determinar se eles passam ou falham.
- Os testes não devem incluir declarações desnecessárias tais como "pontos de verificação" no teste. Diga apenas o que o teste está testando.
- Os testes devem ser portáteis e facilmente executados em diferentes ambientes.
- Os testes devem preparar o que eles precisam para executar. Os testes não devem fazer suposições sobre recursos específicos existentes.
- Os testes devem limpar os recursos criados posteriormente.
- Os nomes dos testes devem descrever claramente o que eles estão testando.
- Os testes devem produzir mensagens significativas quando falham. Tente fazer o teste falhar e veja se o motivo da falha pode ser determinado pela leitura da saída do caso de teste.

Códigos utilizados nos exemplos

Nossos códigos deveriam finalmente ter sido os seguintes.

Arquivo **testing01.py**:

```
from math import pi

def area(r):

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

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

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

Arquivo **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)
```