GeeksHubs
academy _

UNIT TEST

Pruebas Unitarias

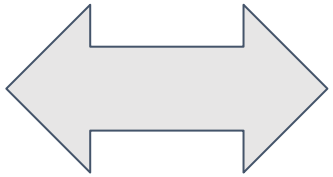
¿Qué son las Pruebas Unitarias?

Las **pruebas unitarias** consisten en aislar una parte del código y comprobar que funciona a la perfección.

Son pequeños tests que validan el comportamiento de un método, un objeto y la lógica aplicada a estos.

Características

- Automatizable.
- Completas.
- Repetibles.
- Independientes.
- Rápidas de crear.



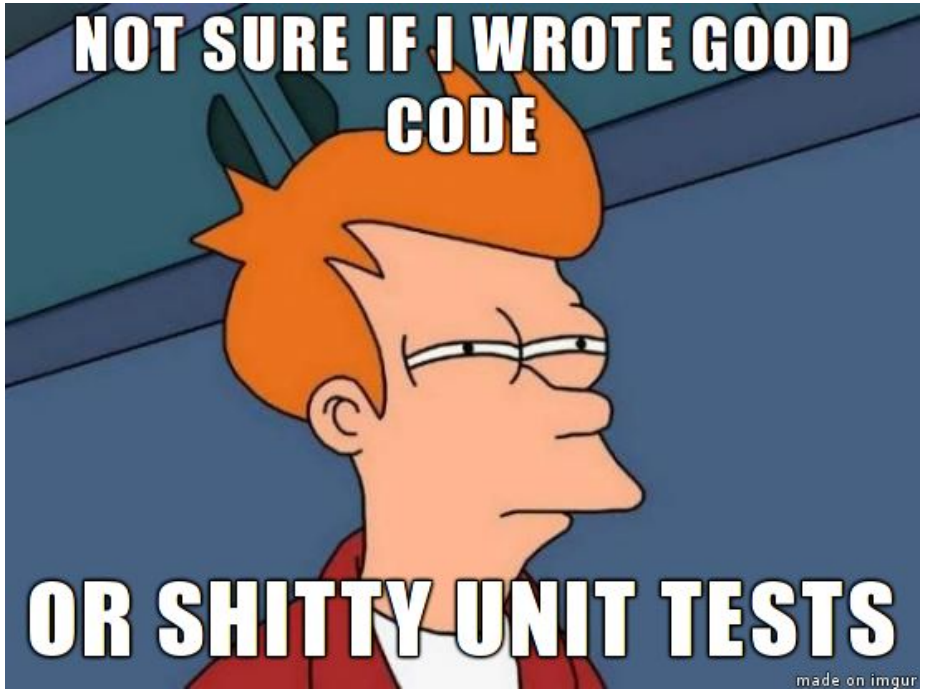
Ventajas

- Proporciona un marco de trabajo ágil.
- Aumenta la Calidad del código.
- Detectar errores rápido.
- Facilita los cambios.
- Favorece la integración.
- Proporciona información.
- Proceso debugging.
- El diseño.
- Reduce el coste.

Características de las Pruebas Unitarias

Las pruebas unitarias comprueban el código en pequeños grupos pero no la integración total del mismo.

Para ver si hay errores de integración es necesario realizar otro tipo de pruebas de software conjuntas y de esta manera comprobar la efectividad total del código.



Elegir Test Runner

Hay muchas librerías de prueba disponibles para Python. El que está integrado en la biblioteca estándar de Python se llama **unittest**.

Las librerías, para la realización de pruebas unitarias en Python, más populares son:

- unittest
- nose2
- pytest

https://docs.python.org/3/library/unittest.html

UnitTest - Unit Testing Framework

UnitTest Framework

UnitTest Framework es un proyecto inspirado en JUnit y por tanto su forma de trabajar es similar al de los principales frameworks de test unitarios utilizados en otros lenguajes.

Características principales:

- Admite la automatización de pruebas.
- Permite el intercambio de código de configuración y cierre para pruebas.
- Facilita la agregación de pruebas en colecciones.
- Independencia de las pruebas con respecto del framework de informes.

UnitTest Tools

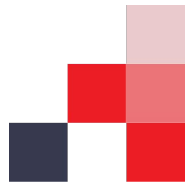
UnitTest admite la implementación de algunas herramientas importantes siguiendo el paradigma de la Programación Orientada a Objetos. Entre ellos cabe destacar:

- **test *fixture*** representa la preparación necesaria para realizar una o más pruebas y cualquier acción de limpieza asociada.

- ***test case*** es la unidad individual de prueba. Comprueba una respuesta específica a un conjunto particular de entradas. unittest proporciona una clase base, TestCase.

- ***test suite*** es una colección de casos de prueba que se utiliza para agregar funciones que deben ejecutarse juntas.

- ***test runner*** es un componente que organiza la ejecución de pruebas y proporciona el resultado al usuario.



Kata I

KATA I:

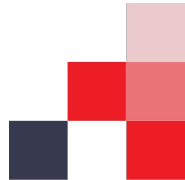
Como primer ejemplo vamos a hacer una prueba unitaria, con PyCharm, sobre este código utilizando la funcionalidad de unittest.

En este caso vamos a evaluar funciones propias de la API de Python 3.

Funciones:

- upper()
- isupper()
- spit()

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```



GeeksHubs academy _

Desarrollamos { talento }

Clases y Funciones

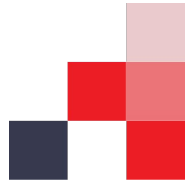
TestCase

Las instancias de TestCase proporcionan tres grupos de métodos:

1. un grupo usado para ejecutar la prueba,
2. otro usado por la implementación de la prueba para verificar condiciones e informar de posibles fallos.
3. Métodos de consulta que permiten recopilar información sobre la prueba en sí.

Los métodos del primer grupo (ejecutar la prueba) son:

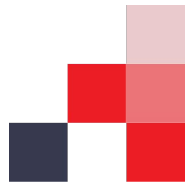
***setUp()*, *tearDown()*, *setUpClass()*, *tearDownClass()*, *run(result=None)*, *skipTest(reason)*, *subTest(msg=None, **params)*, *debug()*.**



setUp()

Método llamado para preparar el dispositivo de prueba. Esto se llama inmediatamente antes de llamar al método de prueba; que no sea *AssertionError* o *SkipTest*, cualquier excepción generada por este método se considerará un error en lugar de una prueba fallida. La implementación predeterminada no hace nada.

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')
```



tearDown()

Método llamado inmediatamente después de que se haya llamado al método de prueba y se haya registrado el resultado.

Este método solo se llamará si setUp () tiene éxito, independientemente del resultado del método de prueba. La implementación predeterminada no hace nada.

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(),
(100,150),
                         'wrong size after resize')
    def tearDown(self):
        self.widget.dispose()
```

setUpClass ()

Un método de clase llamado antes de que se ejecuten las pruebas en una clase individual. setUpClass se llama con la clase como único argumento y debe decorarse como un método de clase ():

```
import unittest

class WidgetTestCase(unittest.TestCase):

    def setUpClass(self):
        ...
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def tearDown(self):
        self.widget.dispose()
```

tearDownClass ()

Un método de clase llamado después de que se hayan ejecutado las pruebas en una clase individual.

tearDownClass se llama con la clase como único argumento y debe decorarse como un método de clase ():

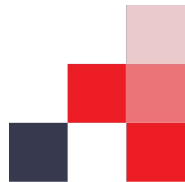
```
import unittest

class WidgetTestCase(unittest.TestCase):

    def setUpClass(self):
        ...
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def tearDown(self):
        self.widget.dispose()


    def tearDownClass(self):
        ...
```

run(result=None)

Ejecute la prueba y recopile el resultado en el objeto TestResult aprobado como resultado. Si se omite el resultado o Ninguno, se crea un objeto de resultado temporal (llamando al método defaultTestResult ()) y se usa. El objeto de resultado se devuelve al llamador de run ().

debug()

Ejecute la prueba sin recopilar el resultado. Esto permite que las excepciones generadas por la prueba se propaguen a la persona que llama y se puede utilizar para admitir la ejecución de pruebas en un depurador.



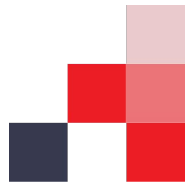
skipTest(reason)

Llamar a esto durante un método de prueba o setUp () omite la prueba actual. Consulte Omitir pruebas y fallas esperadas para obtener más información.

subTest (msg = None, ** params)

Devuelve un administrador de contexto que ejecuta el bloque de código adjunto como una subprueba. msg y params son valores arbitrarios opcionales que se muestran cada vez que falla una subprueba, lo que le permite identificarlos claramente.

Un caso de prueba puede contener cualquier número de declaraciones de subprueba y se pueden anidar arbitrariamente.



Asserts()

La clase TestCase proporciona varios métodos de aserción para verificar e informar de posibles errores de validación en las funciones a testear.

La siguiente tabla enumera los métodos más utilizados:

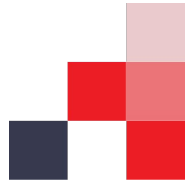
Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Kata II

KATA II: Operaciones Matemáticas

En esta kata vamos a crear las pruebas unitarias necesarias, con PyCharm, sobre el código que aparece a continuación utilizando la funcionalidad de unittest.

```
def doblar(a): return a * 2

def sumar(a, b): return a + b
```



Kata III

KATA III: skip y subtest

En esta kata vamos a trabajar la diferencia que existe entre skip y subtest con dos sencillos ejemplos.

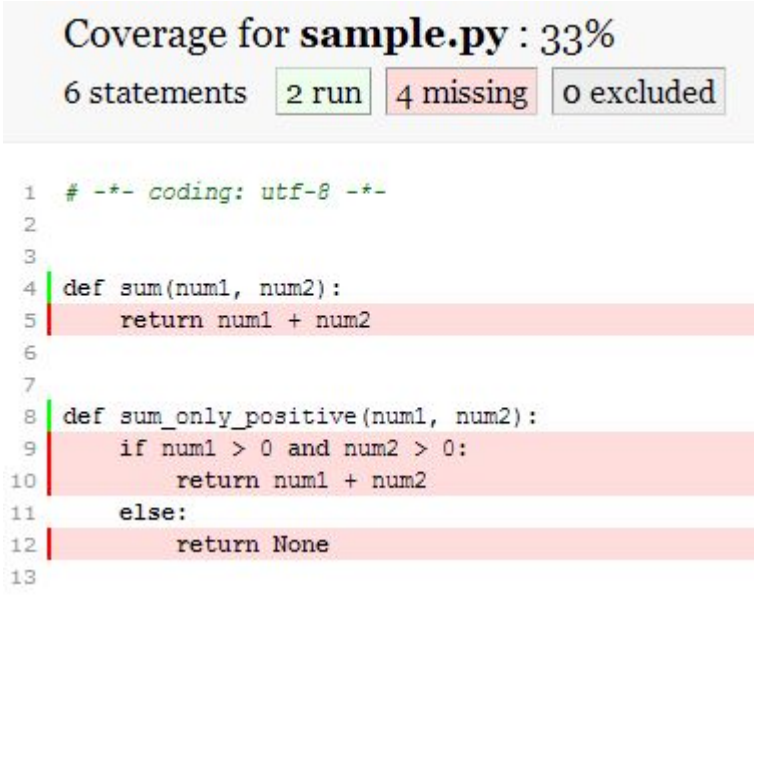
- En el primer ejemplo validaremos la funcionalidad de numpy.

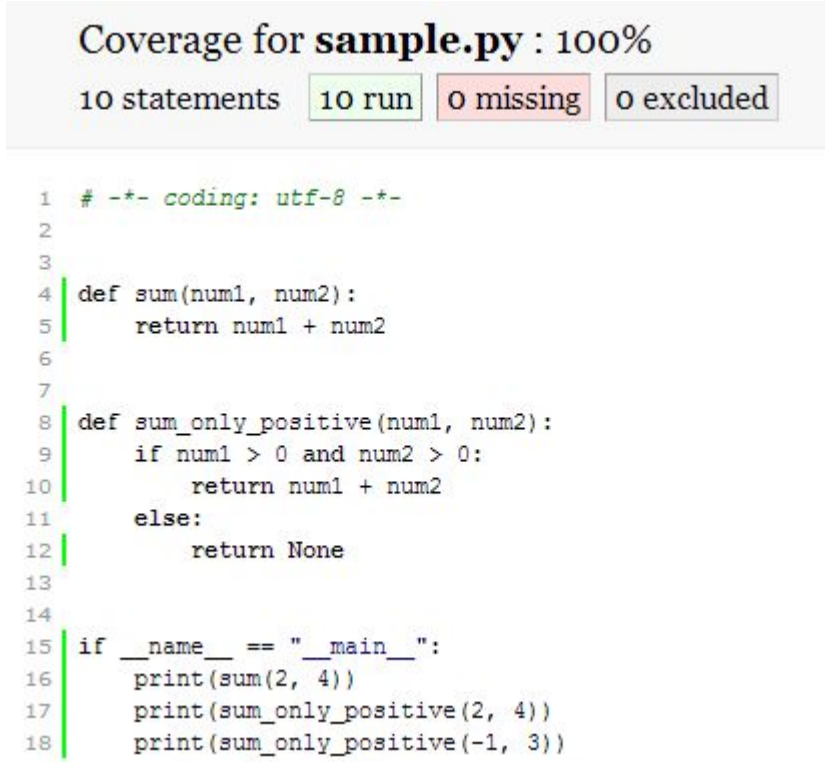
- En el segundo ejemplo haremos una doble validacion de una funcion aplicada a un array de enteros utilizando para ello los subtest.

Coverage

Coverage

La cobertura de código es una medida en las pruebas de software que mide el grado en que el código fuente de un programa ha sido comprobado con pruebas de software.





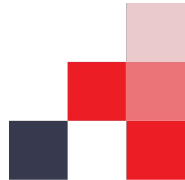
Kata IV

KATA IV: coverage

En esta kata vamos a crear las pruebas unitarias necesarias, con PyCharm, sobre el código que aparece a continuación utilizando la funcionalidad de unittest.

Es importante alcanzar un grado de cobertura cercano al 100%

```
def sumar(a, b): return a + b

def restar(a, b): return a - b

def doblar(a): return a * 2

def multiplicar(a, b): return a * b

def dividir(a, b): return a / b

def cuadrado(a): return pow(a, 2)

def raiz(a): return math.sqrt(a)

def es_par(a): return 1 if a % 2 == 0 else 0
```



GeeksHubs
academy _

/

Jose Marín

www.geekshubsacademy.com
@geekshubs