La Librería unittest[1]

El framework de pruebas unitarias *unittest* que provee Python en su librería estándar, fue inspirado en JUnit y es similar a la mayoría de los frameworks de pruebas unitarias de otros lenguajes.

Este framework soporta automatización de tests, código compartido para el *setup* y *shutdown* de los *tests*, agregado de tests en colecciones, e independencia de los tests del framework de reporte.

Para lograr esto, *unittest* considera los siguientes conceptos:

- **Test Fixture:** representa la preparación necesaria para correr una o más pruebas, y cualquier acción de limpieza asociada. Esto puede involucrar, por ejemplo, la creación de base de datos temporales, directorios, o iniciar un servidor.
- **Test Case:** es una unidad de prueba individual. Chequea una respuesta específica para un conjunto particular de entradas. La librería *unittest* provee una clase base, *TestCase*, que debe ser utilizada para crear nuevos *test cases*.
- **Test Suite:** es una colección de *test cases*, *test suites* o ambos. Es utilizada para agregar pruebas que deben ejecutarse juntos.
- **Test Runner:** es un componente que coordina la ejecución de tests y provee la salida al usuario. El *runner* puede usar una interfaz gráfica, o devolver un valor especial para indicar el resultado de la ejecución de las pruebas.

Ejemplo de uso

A continuación se explica un ejemplo básico:

[1] https://docs.python.org/3/library/unittest.html

```
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'])
      # Verifica que s.split lanza una excepción de tipo TypeError cuando el separador
      # no es un string
      with self.assertRaises(TypeError):
         s.split(2)

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

Un *testcase* se crea sub-clasificando la clase *unittest.TestCase*. Las pruebas se definen con métodos donde los nombres empiezan con la palabra **test**. Esta convención le informa al *test runner* cuales son los métodos de la clase que representan pruebas, ya que la misma puede definir otros métodos que no corresponden a pruebas.

Lo esencial de cada prueba es verificar que el resultado se corresponde con el valor esperado.

El método *assertEqual* verifica que el resultado de la operación que estamos haciendo sea igual al valor que le pasamos, es decir, al valor que esperamos. Los métodos *assertTrue* y *assertFalse* verifican si se cumple una condición. Y el método *assertRaises* verifica que se lance una excepción de un tipo específico.

El *test runner* acumula los resultados de todas las pruebas y produce un reporte.

El bloque final de código muestra una manera simple de correr las pruebas. unittest.main() provee una interfaz de línea de comando al script de prueba.

Anatomía de las pruebas

Las pruebas tienen una estructura interna común, lo que se suele llamar anatomía de las pruebas. La mayoría de los casos de pruebas están compuestos por:

- **Setup:** Donde se crean los objetos necesarios (el contexto) para la prueba.
- **Act:** Lo que quiero hacer o probar.
- **Assertions:** Verificaciones sobre los resultados obtenidos.

La clase TestCase

La clase *TestCase* está pensada para ser utilizada como clase base, implementando las pruebas específicas en las subclases concretas. Esta clase implementa la interfaz necesaria para permitir al *test runner* correr las pruebas y reportar fallas de diversos tipos.

Métodos relevantes

A continuación se describen algunos métodos relevantes de la clase *TestCase*:

setUp: este método se llama antes de ejecutar cada prueba del *TestCase*. En este método se pone el código para preparar el contexto del tests.

tearDown: este método se llama al finalizar la ejecución de cada test del TestCase. En este método se pone el código para limpiar el contexto que puede haber modificado la prueba que acaba de terminar. De manera que la siguiente prueba que se ejecute tenga el contexto limpio como si fuera la única prueba que se ejecuta.

Este método nos ayuda a que cada prueba sea independiente y que dé lo mismo ejecutar esa prueba sola o todo el *TestCase*.

setUpVlass: este método de clase, se ejecuta cuando se crea una instancia de *TestCase*. Es útil para hacer un *setup* del contexto que sea común a todos las pruebas del *TestCase*.

tearDownClass: este método de clase, se ejecuta cuando se destruye una instancia del *TestCase*. Es útil para hacer limpieza del contexto, que pueda llegar a afectar a otros *TestCase*.

addCleanup: agrega una función de limpieza que se ejecutará en el *tearDown* del test. Si se agregan muchas funciones de limpieza, se ejecutarán en orden LIFO.

doCleanups: Se ejecuta siempre después del *tearDown* o después del *setUp* si este lanza una excepción. Ejecuta las funciones agregadas con el método *addCleanup.*

fail: Indica que la prueba fallo incondicionalmente, es decir, se utiliza para que devuelva que la prueba falló.

Assertions

A continuación se describen los métodos que provee la clase *TestCase* para realizar verificaciones sobre los resultados.

Método	**Verifica que**
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
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None

assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises exc
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises exc and the message matches regex r
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises warn
assertWarnsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) raises warn and the message matches regex r
assertLogs(logger, level)	The with block logs on logger with minimum level
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	not r.search(s)
assertCountEqual(a, b)	a and b have the same elements in the same number, regardless of their order.

Además existen otros métodos que verifican la igualdad de ciertos tipos de datos, como por ejemplo, string, listas, tuplas, etc.

Método	**Se utiliza para comparar**
assertMultiLineEqual(a, b)	strings
assertSequenceEqual(a, b)	secuencias
assertListEqual(a, b)	listas
assertTupleEqual(a, b)	tuplas
assertSetEqual(a, b)	conjuntos
assertDictEqual(a, b)	diccionarios

Saltear Pruebas

En ciertas ocasiones es útil saltear alguna prueba, quizás porque hay una funcionalidad que sabemos que tenemos que corregir, pero ahora estamos viendo otro tema o también se pueden saltear pruebas para ciertas condiciones, por ejmplo, esta prueba ejecutala si la versión de Python es 3.1.

La librería unittest nos brinda unos decoradores para saltear pruebas.

Decorador	Descripción
@unittest.skip(reason)	Saltea el test correspondiente al método decorado. Se le puede indicar el motivo.
@unittest.skipIf(condition, reason)	Saltea el test correspondiente al método decorado si se cumple la condición pasada

	como primer parámetro. Se le puede indicar el motivo.
@unittest.skipUnless(condition, reason)	Saltea el test correspondiente al método decorado si NO se cumple la condición pasada como primer parámetro. Se le puede indicar el motivo.