

Testing

(Data Engineer)



Introducción

Bienvenidos al módulo 10 - Testing. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conceptos sobre pruebas de código en Python.

Una vez finalizado este módulo estarás capacitado para las siguientes acciones:

- Comprender en qué consiste una prueba y los diferentes tipos: Unitarias y de integración.
- Comprender la terminología estándar de pruebas.
- Conocer algunos de los corredores de pruebas que se utilizan con Python.
- Comprender algunas reglas generales para utilizar las pruebas de forma adecuada.
- Utilizar el módulo unittest para escribir pruebas unitarias.
- Comprender cómo leer el resultado de una prueba unitaria.
- Explorar diferentes alternativas para ejecutar unittest.



Tema 1. Introducción a testing

Objetivo

Una vez finalizado este tema estarás capacitado para las siguientes acciones:

- Comprender en qué consiste una prueba y los diferentes tipos: Unitarias y de integración.
- Comprender la terminología estándar de pruebas.
- Conocer algunos de los corredores de pruebas que se utilizan con Python.
- Comprender algunas reglas generales para utilizar las pruebas de forma adecuada.

Introducción

Cuando desarrollamos un programa casi siempre tenemos una instancia en la que necesitamos probar que algunas partes del código funcionen correctamente. Por ejemplo, cuando nuestro programa interactúa con una base de datos, se suele probar si la conexión con esta se establece o no.

En la medida que la cantidad de pruebas a realizar sea pequeña, no tendremos problemas, pero cuando nuestro código comienza a crecer es conveniente utilizar un plan de pruebas que incluya todos los aspectos a revisar. En este punto es donde automatizar la ejecución de las pruebas, cobra un papel relevante.

Prueba de aserción

La prueba de aserción es aquella en la cual incluimos nuestra variable dentro de una afirmación.

Por ejemplo en el siguiente código podemos ver que la función “test_sum” recibe una lista de números, realiza la suma de ellos y finalmente realiza una aserción de que es igual a 10.

```
test.py > ...
1 # Prueba de Aserción
2
3 def test_sum(list_number):
4     """
5     Test if sum of list_number is equal to 10.
6     param list_number: list of numbers to sum
7     type: list
8     """
9     total_sum = sum(list_number)
10    assert (total_sum) == 10
11
12
13 if __name__ == "__main__":
14     numbers = [3, 5, 2]
15     test_sum(numbers)
16     print('Está todo bien!')
17
```

En caso de que la suma de los números sea igual a 10, se imprimirá en pantalla “Está todo bien”

```
PROBLEMS  OUTPUT  TERMINAL  JUPYTER  DEBUG CONSOLE

(sphinx) (base) PS C:\Users\User\Python\Proyectos\Testing> & "C:/
Está todo bien!
(sphinx) (base) PS C:\Users\User\Python\Proyectos\Testing> []
```

En caso de que la suma no sea 10, se generará una excepción del tipo AssertionError.

```
def test_sum(list_number):
    """
    Test if sum of list_number is equal to 10.
    param list_number: list of numbers to sum
    type: list
    """
    total_sum = sum(list_number)
    assert (total_sum) == 10

if __name__ == "__main__":
    numbers = [3, 5, 22]
    test_sum(numbers)
    print('Está todo bien!')
```

```
(sphinx) (base) PS C:\Users\User\Python\Proyectos\Testing> & "C:/Users/User/Pyth
Traceback (most recent call last):
  File "c:\Users\User\Python\Proyectos\Testing\test.py", line 15, in <module>
    test_sum(numbers)
  File "c:\Users\User\Python\Proyectos\Testing\test.py", line 10, in test_sum
    assert (total_sum) == 10
AssertionError
(sphinx) (base) PS C:\Users\User\Python\Proyectos\Testing> []
```

Prueba unitaria vs. Prueba de integración

Podemos diferenciar los siguientes tipos de pruebas.

- Prueba unitaria: verifica un pequeño componente de un programa.
- Prueba de integración: verifica que dos o más componentes de un programa funcionen entre sí.

Mientras que las pruebas unitarias se utilizan para encontrar errores en funciones individuales, las pruebas de integración verifican el sistema como un todo. Ambos tipos de pruebas deben utilizarse en conjunto ya que nos permiten tener un enfoque complementario a la hora de revisar el código.

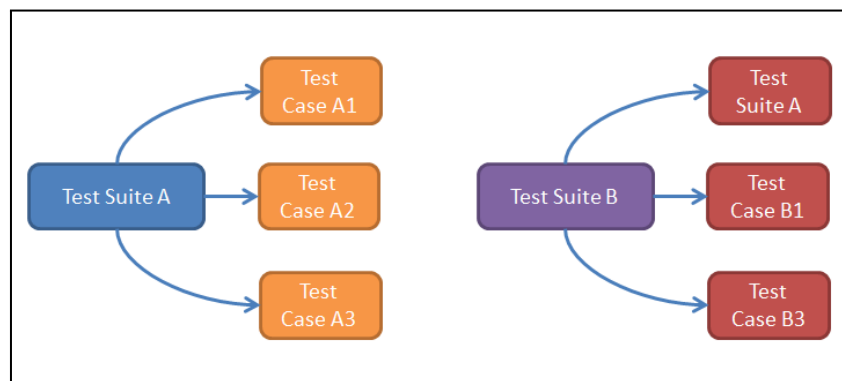
Durante el desarrollo de esta unidad nos vamos a centrar en las pruebas unitarias.

Conceptos sobre pruebas unitarias

Realizar pruebas a los programas es una práctica común en diferentes lenguajes de programación. Si bien existen diferentes tipos de definiciones, nos ajustaremos a aquellas que se encuentran definidas en la librería [Unittest](#) de la documentación oficial de Python.

- **Test fixture (calendario de pruebas):** representa la preparación necesaria para realizar una o más pruebas y cualquier acción de limpieza asociada. Esto puede implicar, por ejemplo, la creación de directorios o bases de datos temporales o proxy, o el inicio de un proceso de servidor.
- **Test case (caso de prueba):** es la unidad individual de prueba. Comprueba una respuesta específica a un conjunto particular de entradas.

- **Test suite (conjunto de pruebas):** es una colección de casos de prueba, conjuntos de pruebas o ambos. Se utiliza para agregar pruebas que deben ejecutarse juntas.
- **Test runner (corredor de pruebas):** es un componente que organiza la ejecución de pruebas y proporciona el resultado al usuario. El corredor puede usar una interfaz gráfica, una interfaz textual o devolver un valor especial para indicar los resultados de la ejecución de las pruebas.



Elección del test runner

Existen muchos test runners desarrollados para Python. Algunos de ellos son:

- [unittest](#): se encuentra integrado en la librería estándar de Python.
- [nose2](#): basado en unittest, le agrega nuevas funcionalidades.
- [pytest](#): es una herramienta madura de prueba de Python con todas las funciones que nos ayuda a escribir mejores programas.

En esta unidad utilizaremos unittest como test runner ya que es el que se incluye en la librería estándar de Python y sobre el cual se desarrolla el resto de ellos.

Algunas reglas generales

Antes de sumergirnos en la escritura de tests, es importante mencionar algunas reglas que nos ayudarán a escribirlos de una forma más ordenada.

- Un test case debe centrarse en una pequeña parte de la funcionalidad y demostrar que es correcta.

- Cada test case debe ser completamente independiente, debe poder ejecutarse solo y también dentro del test suite, independientemente del orden en que se llamen.
- Debemos diseñar pruebas que se ejecuten rápido. Si una sola prueba necesita más de unos pocos milisegundos para ejecutarse, el desarrollo se ralentizará o las pruebas no se ejecutarán con la frecuencia deseada. En algunos casos, las pruebas no pueden ser rápidas porque necesitan una estructura de datos compleja para trabajar, y esta estructura de datos debe cargarse cada vez que se ejecuta la prueba. En ese caso, debemos mantener estas pruebas más pesadas en un conjunto de pruebas separado que se ejecuta mediante alguna tarea programada y ejecutar todas las demás pruebas con la frecuencia necesaria.
- Es deseable ejecutar las pruebas con frecuencia, idealmente de forma automática cuando se guarde el código.
- Ejecutar siempre el conjunto de pruebas completo antes de comenzar a escribir nuestro código y también ejecutarlo después de que hayamos terminado.
- Es una buena idea implementar un enlace que ejecute todas las pruebas antes de enviar el código a un repositorio compartido.
- Si estamos en medio de una sesión de desarrollo y tenemos que interrumpir nuestro trabajo, es una buena idea escribir una prueba unitaria que dé error y que detalle lo que deseamos desarrollar a continuación. Cuando regresemos al trabajo, tendremos un indicador de dónde estábamos y volveremos a la normalidad más rápido.
- El primer paso cuando estamos depurando nuestro código es escribir una nueva prueba que identifique el error. Si bien no siempre es posible hacerlo, esas pruebas de detección de errores se encuentran entre las piezas de código más valiosas de un proyecto.
- Utilizar nombres largos y descriptivos para las funciones de prueba. Por ejemplo: `test_raiz_cuadrada_numero_negativo()` en vez de `tst_square()`.
- Cuando algo sale mal o debe cambiarse, y si nuestro código tiene un buen conjunto de pruebas, nosotros o nuestros colegas confiarán en gran medida en el conjunto de pruebas para solucionar el problema o modificar un comportamiento determinado. Por lo tanto, el código de prueba se leerá tanto o incluso más que el código en ejecución. Una prueba unitaria cuyo propósito no está claro no es muy útil en este caso.
- Otro uso del código de prueba es para introducir a nuevos desarrolladores. Cuando alguien tiene que trabajar en la base del código, ejecutar y leer el código, comenzar por el plan de pruebas suele ser lo mejor.



Tema 2. Escribir un test con Unittest

Objetivo

Una vez finalizado este tema estarás capacitado para las siguientes acciones:

- Utilizar el módulo Unittest para escribir pruebas unitarias.
- Comprender cómo leer el resultado de una prueba unitaria.
- Explorar diferentes alternativas para ejecutar Unittest.

Ejemplo básico

Unittest requiere lo siguiente:

- Escribir los tests como métodos de una clase.
- Utilizar una serie de aserciones que ya nos vienen definidas dentro de su librería.

El módulo unittest nos proporciona un amplio conjunto de herramientas para construir y ejecutar pruebas.

Aquí tenemos un breve script llamado **test.py** para probar tres métodos de cadena.


```
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()
```

Podemos ver que se creó la clase **TestStringMethods**, la cual es una subclase de **TestCase** de la librería **unittest**.

Luego, se definen tres métodos:

- **test_upper**: el cual utiliza **assertEqual** para comparar que la cadena 'foo' sea igual a 'FOO' cuando la pasamos a mayúsculas.
- **test_isupper**: el cual utiliza dos aserciones:
 - **assertTrue**: para chequear que una cadena esté escrita en mayúsculas.
 - **assertFalse**: para chequear que una cadena no esté escrita en mayúsculas.
- **test split**: el cual utiliza dos aserciones:
 - **assertEqual**: para chequear que una lista tiene dos palabras
 - **assertRaises**: para chequear cuando un separador no es una cadena.

El bloque final muestra una forma sencilla de ejecutar las pruebas utilizando la función **unittest.main()** cuando ejecutamos el script **test.py**.

Cuando las pruebas son exitosas podemos ver el siguiente mensaje en la terminal donde nos indica la cantidad de test ejecutados y el tiempo de ejecución.

```

PROBLEMS  OUTPUT  TERMINAL  JUPYTER  DEBUG CONSOLE

(sphinx) PS C:\Users\User\Python\Proyectos\Testing> & "C:/Users/User/Python/Proyectos/Testing/test.py
...
-----
Ran 3 tests in 0.001s

OK
(sphinx) PS C:\Users\User\Python\Proyectos\Testing> 

```

Para visualizar el error en una prueba vamos editar la cadena s para que tome el valor “hello wo5rld”.

```

PROBLEMS  OUTPUT  TERMINAL  JUPYTER  DEBUG CONSOLE

(sphinx) PS C:\Users\User\Python\Proyectos\Testing> & "C:/Users/User/Python/Entor
ser/Python/Proyectos/Testing/test.py
.F.
=====
FAIL: test_split (__main__.TestStringMethods)
-----
Traceback (most recent call last):
  File "c:\Users\User\Python\Proyectos\Testing\test.py", line 15, in test_split
    self.assertEqual(s.split(), ['hello', 'wo5rld'])
AssertionError: Lists differ: ['hello', 'world'] != ['hello', 'wo5rld']

First differing element 1:
'world'
'wo5rld'

- ['hello', 'world']
+ ['hello', 'wo5rld']
?             +

-----
Ran 3 tests in 0.002s

FAILED (failures=1)
(sphinx) PS C:\Users\User\Python\Proyectos\Testing> 

```

En la terminal podemos ver que el test falló (FAIL) y también que el método test_split generó en la línea 15, una aserción del tipo AssertionError. Además podemos ver en qué parte de la lista se detectó la diferencia y también el tiempo de ejecución del test.

Test Cases de Unittest

La clase `TestCase` de `Unittest` nos proporciona varios métodos de aserción para verificar e informar fallas. En la [siguiente sección](#) de la documentación oficial de Python podemos visualizar una tabla con los métodos más utilizados, como así también la documentación de todos los métodos disponibles.

Método	comprueba que	Nuevo en
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Ejecución de Unittest

A continuación veremos diferentes alternativas para ejecutar nuestros tests.

1) `unittest.main()`

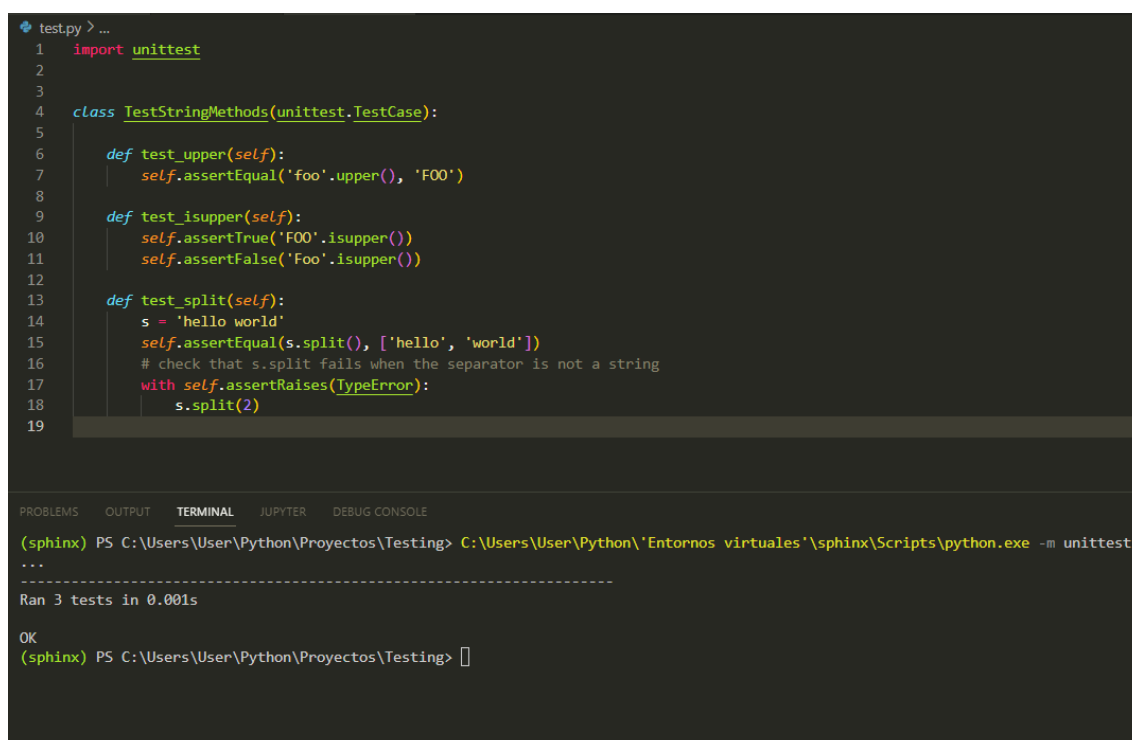
Anteriormente vimos que cuando ejecutamos el archivo `test.py`, se realiza una llamada a la función `unittest.main()`, la cual ejecuta el test runner descubriendo todas las clases en el archivo `test.py` que heredan la clase `unittest.TestCase`.

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

Esta es una de las tantas maneras de ejecutar un test runner. Cuando tenemos un solo archivo de test, como el que utilizamos, es una excelente alternativa.

2) Utilizar el comando unittest en la consola

Este método nos permite ajustar el test sin utilizar la función `unittest.main()`.



```

test.py > ...
1  import unittest
2
3
4  class TestStringMethods(unittest.TestCase):
5
6      def test_upper(self):
7          self.assertEqual('foo'.upper(), 'FOO')
8
9      def test_isupper(self):
10             self.assertTrue('FOO'.isupper())
11             self.assertFalse('Foo'.isupper())
12
13     def test_split(self):
14         s = 'hello world'
15         self.assertEqual(s.split(), ['hello', 'world'])
16         # check that s.split fails when the separator is not a string
17         with self.assertRaises(TypeError):
18             s.split(2)
19

```

```

PROBLEMS  OUTPUT  TERMINAL  JUPYTER  DEBUG CONSOLE
(sphinx) PS C:\Users\User\Python\Proyectos\Testing> C:\Users\User\Python\Entornos virtuales\sphinx\Scripts\python.exe -m unittest
...
-----
Ran 3 tests in 0.001s

OK
(sphinx) PS C:\Users\User\Python\Proyectos\Testing> 

```

Para ejecutarla, debemos:

- ubicarnos en la carpeta donde se encuentra el archivo con las pruebas (en nuestro caso, test.py).
 - `C:\Users\User\Python\Proyectos\Testing`
- escribir la ruta al archivo python.exe (en caso de no tener esta ruta en el path) o utilizar python (si la llamamos de esa manera) y agregar el modificador -m
 - `C:\Users\User\Python\Entornos virtuales\sphinx\Scripts\python.exe -m`
- agregar la palabra unittest al final

Estructura de carpetas

Anteriormente creamos el archivo test.py y luego lo ejecutamos para crear realizar las pruebas.

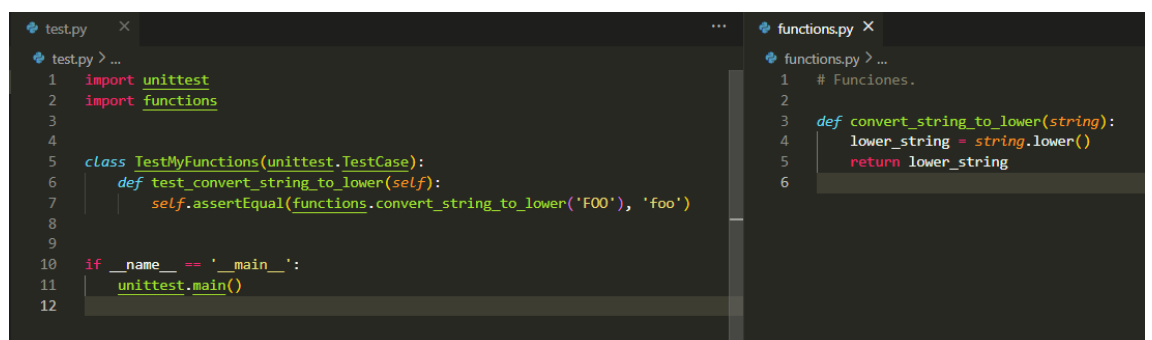
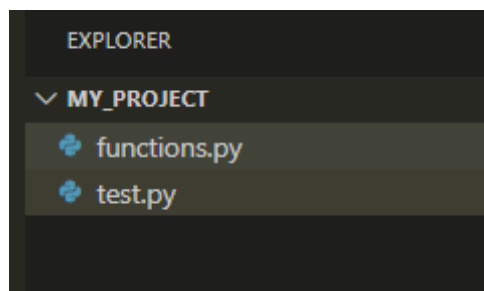
A continuación vamos a ver dos estructuras de carpetas para organizar mejor nuestro código.

Estructura 1:

Es la más sencilla, utilizamos un archivo donde guardamos nuestras funciones y otro donde guardamos los tests.

My_Project

```
|---- funciones.py
|---- tests.py
```



Estructura 2:

En esta estructura tenemos un módulo para guardar las funciones y otro para guardar los tests.

My_Project2

```
|---- funciones
|        |---- __init__.py
|        |---- funciones.py
|---- tests
```

|---- __init__.py
|---- tests.py

Podemos observar que previo a la importación del módulo funciones es necesario agregar esa ruta al path.

```
tests > test.py > ...  
1  import sys  
2  import unittest  
3  sys.path.append('C:/Users/User/Python/Proyectos/My_Project 2/functions')  
4  import functions  
5  
6  
7  class TestMyFunctions(unittest.TestCase):  
8      def test_convert_string_to_lower(self):  
9          self.assertEqual(functions.convert_string_to_lower('FOO'), 'foo')  
10  
11  
12  if __name__ == '__main__':  
13      unittest.main()  
14
```



Cierre

Durante esta unidad aprendimos sobre testing.

Comenzamos reconociendo esa etapa de pruebas que se presenta cuando estamos desarrollando un programa en Python. Posteriormente exploramos diferentes conceptos como las pruebas unitarias, pruebas de integración y la terminología que nos aporta el módulo unittest, que pertenece a la librería estándar de Python.

En el segundo tema aprendimos a escribir pruebas unitarias con unittest y exploramos diferentes alternativas para ejecutarlas, como así también para estructurar nuestros proyectos.

Quedamos a disposición de las consultas que puedan surgir.



Referencias

<https://docs.python.org/es/3.10/library/unittest.html>

<https://realpython.com/python-testing/>

tiiiiiit by 