

# Clase13\_intro\_testing

June 14, 2023

## 1 Seminario de Lenguajes - Python

### 1.1 Cursada 2023

#### 1.1.1 Clase 13. Una introducción al testing

## 2 ¿Qué significa testear?

- Probar un software es someterlo a ciertas condiciones que puedan demostrar si es válido o no con respecto a los requerimientos planteados.
- El objetivo de las pruebas es encontrar la mayor cantidad de errores posibles, **preferentemente antes de que los encuentre el usuario final**.
- El testing mejora la calidad del producto y **deber realizarse en cada una** de las etapas del desarrollo del software.

## 3 ¿Por qué testear?

- Existe una gran cantidad de **parches** y versiones surgidas luego del lanzamiento de una versión final de un software destinadas a cubrir “**agujeros de seguridad**” o **malos funcionamientos**.
- Por lo general, la etapa de pruebas es la menos sistematizada y tenida en cuenta.

## 4 “Un imagen vale más que mil palabras”

Si bien son situaciones muy antiguas, son muy gráficas.

## 5 ¿Testeamos manualmente?

- Cada vez que “usamos” nuestros programas y “comprobamos que todo esté como pensamos que debería”, estamos testeando.
- Podemos planificar estos test y, aún mejor,
  - **podemos escribir tests que se ejecuten en forma automática.**

## 6 Empecemos por algo simple

En Python podemos escribir lo siguiente:

```
[ ]: assert sum([1, 2, 3]) == 6
```

## 7 ¿assert ?

- Esta sentencia realiza una **comprobación de una condición**, y en caso de ser falsa, levanta la excepción **AssertionError**.

## 8 La sentencia assert

- La forma general es:

```
assert condicion
```

- Sería equivalente a:

```
if not condicion:  
    raise AssertionError()
```

```
[ ]: assert sum([1, 2, 3]) == 6
```

```
[ ]: if sum([1, 2, 3]) != 6:  
    raise AssertionError()
```

## 9 Podemos pasar algo de información

- En realidad la forma general de assert es:

```
assert condicion ["," expresion]
```

- Sería equivalente a:

```
if not condicion:  
    raise AssertionError(expresion)
```

```
[ ]: if sum([1, 2, 3]) != 6:  
    raise AssertionError("Debería dar 6")
```

```
[ ]: assert sum([1, 2, 3]) == 6, "Debería dar 6"
```

## 10 Avancemos un poco más

```
[ ]: def calcular_promedio(lista):  
    """ Esta función retorna el promedio de una lista de números """  
  
    cant_elementos = len(lista)  
    return 0 if cant_elementos == 0 else sum(lista)/cant_elementos
```

- ¿Cómo podemos usar **assert** para probar esta función?

```
[ ]: assert calcular_promedio([1, 2, 3]) == 2.0, "Deberia dar 2.0"
      assert calcular_promedio([1]) == 1.0, "Deberia dar 1.0"
```

```
[ ]: assert calcular_promedio([]) == 0, "Deberia dar 0"
```

## 11 También podemos chequear tipos

- ¿Esto es suficiente?

```
[ ]: def sumar(a: int, b: int) -> int:
      """ Esta función retorna la suma de dos números enteros. """

      return a+b
```

```
[ ]: sumar(1,2)
```

Vimos que no, ya que las anotaciones son simplemente sugerencias. **No se hace ninguna validación en la ejecución.**

## 12 Podemos usar assert para chequear tipos

```
[ ]: def sumar(a: int, b: int) -> int:
      """ Esta función retorna la suma de dos números enteros. """
      assert type(a) == int, "El primer argumento debería ser un entero"
      assert type(b) == int, "El segundo argumento debería ser un entero"
      return a+b
```

```
[ ]: sumar("1", "2")
```

## 13 Los “test case” o casos de pruebas

```
[29]: def test_case1():
      assert calcular_promedio([]) == 0, "Debería dar 0"
      def test_case2():
          assert calcular_promedio([1]) == 1.0, "Debería dar 1.0"
      def test_case3():
          assert calcular_promedio([1, 2, 3]) == 2.0, "Debería dar 2.0"

      if __name__ == "__main__":
          test_case1()
          test_case2()
          test_case3()
          print("Tests pasados!")
```

Tests pasados!

## 14 Tipos de tests

- **Tests de unidad:** permite examinar cada módulo de manera individual para asegurar el correcto funcionamiento.
- **Tests de integración:** permiten ver si existen fallas en la interacción de un módulo con otros módulos.
- **Tests de sistema:** permiten analizar al sistema como un todo, verificando que cumple con los requisitos especificados, la interacción del mismo con el hardware, lo cual lleva a realizar pruebas de rendimiento, seguridad, resistencia, de recuperación, etc.
- **Tests de aceptación:** permiten encontrar errores que sólo el usuario final puede descubrir.

### 14.0.1 ¿Qué tipo de test hicimos nosotros en nuestros casos de prueba?

## 15 Tests de unidad

- Permiten examinar cada parte del programa (módulo, clase o función) de manera individual para asegurar su correcto funcionamiento.
- Por lo general, se realizan una vez que se ha desarrollado, revisado y verificado el código fuente.

## 16 ¿Cuáles son las ventajas de los tests de unidad?

- Permiten probar que el programa funciona correctamente.
- Se puede identificar variables que no existen o tipos esperados en las funciones.
- Permite identificar, en caso de que se haga una modificación, que siga funcionando correctamente todas las parte del programa.
- Son una buena forma de documentar el código (no en forma directa, pero como los tests indican cómo es que se tiene que comportar el programa, viendo los tests se puede ver el resultado esperado para ciertas entradas del programa). **Esto no excluye que se tenga que escribir la documentación del código.**

## 17 ¿Y las desventajas?

- Toman bastante tiempo de escribir.
- Algunas clases o funciones son fáciles de testear pero otras no tanto.
- Cuando se hace un gran cambio en el código hay que actualizar los tests.
- Algo muy importante a tener en cuenta:

Aprobar los tests no significa que el sistema funcione a la perfección.

## 18 ¿Cómo tienen que ser los tests?

- Tienen que poder correr **sin interacción humana:** sin que el usuario **ingrese valores** en ningún caso.
- Tienen que poder **verificar el resultado de la ejecución sin interacción humana:** se debe poder analizar los resultados en forma automática.

- Un test tiene que ser **independiente** del otro. Es decir, el resultado de un test no debería depender del resultado anterior.

## 19 ¿Qué condiciones deben cumplir un test?

- Que funcione correctamente cuando los valores de entrada son válidos.
- Que falle cuando los valores de entrada son inválidos, o que levante una excepción.

## 20 Herramientas disponibles

- Hay varias herramientas que podemos usar para automatizar estos tests:
  - **unittest**: provista con el intérprete.
  - **pytest**: info en <https://docs.pytest.org/en/latest/>
  - **nose2**: info en <https://docs.nose2.io/en/latest/>

## 21 Vamos a usar el módulo unittest

- Permite la realización automática de test de unidad.
- No se tiene que instalar, pero si importarlo antes de usarlo.
- Basado en JUnit (test de unidad para Java).
- [Info oficial](#)

## 22 ¿Cómo definimos nuestros tests?

- Debemos definir nuestros casos de prueba en clases que **heredan de la clase unittest.TestCase**.
- Los métodos asociados a los test deben comenzar con el prefijo “**test**”, de manera tal que puedan ser reconocidos por el **test runner**.
- En vez de utilizar la sentencia assert, cada test invoca a métodos definidos en unittest.TestCase tales como:
  - **assertEqual()**, **assertNotEqual()**: para chequear por un resultado esperado.
  - **assertTrue()**, **assertFalse()**: para verificar una condición.
  - **assertRaises()**: para verificar si se levantó una excepción.
  - La [lista completa de métodos](#) en la documentación oficial.
- ¿Y la PEP 8?

## 23 Probemos este ejemplo en el IDE

- Usamos parámetro -v

```
# Módulo: mis_funciones
def calcular_promedio(lista):
    """ Calcula el promedio de los valores dentro de la lista dada como argumento """
```

```

    cant_elementos = len(lista)
    return 0 if cant_elementos == 0 else sum(lista)/cant_elementos

#Código: test1.py
import unittest
from mis_funciones import calcular_promedio

class TestMiFuncion(unittest.TestCase):
    def test_case1(self):
        """ Testea que el promedio de una lista vacía de 0 """

        self.assertEqual(calcular_promedio([]), 0)

    def test_case2(self):
        """ Testea que el promedio de una lista con un único valor es el mismo valor """

        valor = 1
        self.assertEqual(calcular_promedio([valor]), float(valor))

    def test_case3(self):
        """ testea que el promedio de los valores 1, 2 y 3 sea 2.0 """

        self.assertEqual(calcular_promedio([1, 2, 3]), 2.0)

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

```

## 24 Analizamos el código

- La clase que contiene los test es una clase derivada de `unittest.TestCase`.
- Todos los casos de prueba son métodos que comienzan con el prefijo **test**.
- Usamos `assertEqual` para comprobar el valor de una expresión.
- Invocamos a **`unittest.main()`** para ejecutar todos los test: el **test runner**
  - También: **`python -m unittest test1`**
  - O **`python -m unittest -v test1`**

### 24.0.1 Podemos también usar:

- **`python -m unittest discover`**
- O para ejecutar todos los test en una carpeta: **`python -m unittest discover -s tests`**
  - Donde se intenta ejecutar los tests dentro de la carpeta **`tests`**

## 25 Agregamos esta función

```

def calcular_promedio_descuidada(valores):
    """ Calcula el promedio de los valores dentro de la lista dada como argumento """

    return sum(valores) / len(valores)

```

- ¿Qué deberíamos testear en este caso?
- El funcionamiento “normal”. Es decir: `calcular_promedio_descuidada([20, 30, 70])` debería ser **40.0**
- Debería levantar la excepción **ZeroDivisionError** en caso de enviar una secuencia vacía.
- Aunque también, tanto en esta función como en la anterior, podríamos testear:
  - Si se levanta la excepción **TypeError** en caso de no enviar una secuencia como parámetro.

## 25.1 Probemos el código en IDE

- Código: test2.py

```
class TestFuncionDescuidada(unittest.TestCase):
    def test_case1(self):
        """ Testea que el promedio de los valores 20, 50 y 70 sea 40.0 """

        self.assertEqual(calcular_promedio_descuidada([20, 30, 70]), 40.0)
    def test_case2(self):
        """ Testea que se levante la excepciónn ZeroDivisionError cuando la lista está vacía """

        self.assertRaises(ZeroDivisionError, calcular_promedio_descuidada, [])

    def test_case3(self):
        """ Testea que el argumento de la función sea una secuencia"""

        self.assertRaises(TypeError, calcular_promedio_descuidada, 20, 30, 70)
```

## 26 Analicemos los resultados

- Modifiquemos el código para que falle algunos de los tests.

```
def calcular_promedio(lista):
    cant_elementos = len(lista)
    return 1 if cant_elementos == 0 else sum(lista)/cant_elementos
```

## 27 ¿Qué excepción se levanta? AssertionError: 0 != 1

## 28 Observemos los docstrings !!!!

## 29 Más info

- Testing con unittest: <https://realpython.com/python-testing/>
- Testing con PyTest: <https://realpython.com/pytest-python-testing/>
- En los primeros 20 minutos del [video](#) del curso CS50's Web Programming with Python and JavaScript 2020

## 30 Desafío

### 30.1 Definir la clase Pila y escribir los tests que permitan verificar su comportamiento.

- El que se anime a hacerlo, pueden subirlo a su repositorio y lo comparte con @clauBanchoff
- Recuerden mandarme mensaje para revisar.

## 31 Hasta acá llegamos ...

### 31.0.1 La semana próxima: actividad 3 por 85 Python plus