Introducción a la Programación

Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2025

Departamento de Computación - FCEyN - UBA

Práctica 6: Introducción al Lenguaje Imperativo - Parte 2

Testing en Python

Así como aprendimos a hacer testing en Haskell, vamos a aprender a armar y correr casos de test en Python. Existen varias herramientas para realizar pruebas automatizadas, pero la que usaremos en la materia es el módulo **unittest**.

Testing en Python

¿Cómo probamos una función que escribimos? Por ejemplo, la función `suma`:

```
def suma(a: int, b: int) -> int:
    res: int = a + b
    return res
```

Testing en Python

¿Cómo probamos una función que escribimos? Por ejemplo, la función `suma`:

```
def suma(a: int, b: int) -> int:
    res: int = a + b
    return res
```

Definimos un método que llame a la función a testear y verifique que su resultado corresponde con una condición esperada. Este método tendrá un nombre que empieza con `test`, un parámetro `self` y un comando que empieza con `self.assert`, que llamará a la función y verificará una condición dada. Podremos probar si el resultado de aplicar `suma` a 2 y 3 es igual a 5 con un método `test_suma_positiva`:

```
def test_suma_positiva(self):
    self.assertEqual(suma(2, 3), 5)
```

Ahora podemos definir un nuevo caso de prueba que contemple dos negativos:

```
def test_suma_negativos(self):
    self.assertEqual(suma(-4, -7), -11)
```

Ahora podemos definir un nuevo caso de prueba que contemple dos negativos:

```
def test_suma_negativos(self):
    self.assertEqual(suma(-4, -7), -11)
```

Como vemos, el comando `self.assertEqual` corrobora igualdad entre dos valores.

Ahora hagamos una función que determine si una triada de enteros a,b,c es una triada pitagórica. Es decir, si $a^2 + b^2 = c^2$

Ahora hagamos una función que determine si una triada de enteros a,b,c es una triada pitagórica. Es decir, si $a^2 + b^2 = c^2$

```
def triada_pitagorica(a: int, b: int, c:int) -> bool
    suma: int = a**2 + b**2
    res: bool = suma == c*2
    return res
```

Ahora hagamos una función que determine si una triada de enteros a,b,c es una triada pitagórica. Es decir, si $a^2 + b^2 = c^2$

```
def triada_pitagorica(a: int, b: int, c:int) -> bool
    suma: int = a**2 + b**2
    res: bool = suma == c*2
    return res
```

Corroboremos que las entradas 3,4,5 forman una triada pitagórica, es decir, que la función devuelve True con esas entradas. Para eso usamos `self.assertTrue`:

Ahora hagamos una función que determine si una triada de enteros a,b,c es una triada pitagórica. Es decir, si $a^2 + b^2 = c^2$

```
def triada_pitagorica(a: int, b: int, c:int) -> bool
    suma: int = a**2 + b**2
    res: bool = suma == c*2
    return res
```

Corroboremos que las entradas 3,4,5 forman una triada pitagórica, es decir, que la función devuelve True con esas entradas. Para eso usamos `self.assertTrue`:

```
def test_triada_345(self):
    self.assertTrue(triada_pitagorica(3,4,5))
```

Así como self.assertEqual y self.assertTrue, tenemos muchas otros chequeos de condiciones. Por ejemplo,

- self.assertFalse(expresion): chequea si una expresión es falsa
- self.assertNotEqual(a,b): chequea desigualdad entre a y b
- self.assertGreater(a,b): chequea si a es mayor que b
- self.assertLess(a,b)): chequea si a es menor que b
- self.assertIn(a,b): chequea si un elemento a está en b
- self.assertNotIn(a,b): chequea si un elemento a no está en b
- self.assertAlmostEqual(a, b, places=p): chequea que a es igual a b con una precisión de p decimales. Muy útil a la hora de evaluar funciones con floats

Pasos para hacer el testeo

Pasos para hacer el testeo

- Guardar un archivo con las funciones a probar. Ej: guia6.py

Pasos para hacer el testeo

- Guardar un archivo con las funciones a probar. Ej: guia6.py
- En un nuevo archivo dentro de la misma carpeta, importar el módulo unittest y las funciones a testear.
  Ej: `from guia6 import funcion1, funcion2`

Pasos para hacer el testeo

- Guardar un archivo con las funciones a probar. Ej: guia6.py
- En un nuevo archivo dentro de la misma carpeta, importar el módulo unittest y las funciones a testear.
  Ej: `from guia6 import funcion1, funcion2`
- Definir clases con los tests correspondientes a cada caso con solución conocida. Estos tests serán definidos como funciones cuyo nombre debe empezar con la palabra *test*. ¡Usar nombres declarativos!

Pasos para hacer el testeo

- Guardar un archivo con las funciones a probar. Ej: guia6.py
- En un nuevo archivo dentro de la misma carpeta, importar el módulo unittest y las funciones a testear.
  Ej: `from guia6 import funcion1, funcion2`
- Definir clases con los tests correspondientes a cada caso con solución conocida. Estos tests serán definidos como funciones cuyo nombre debe empezar con la palabra *test*. ¡Usar nombres declarativos!
- Agregar al final del archivo de testing el código

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Pasos para hacer el testeo

- Guardar un archivo con las funciones a probar. Ej: guia6.py
- En un nuevo archivo dentro de la misma carpeta, importar el módulo unittest y las funciones a testear.
  Ej: `from guia6 import funcion1, funcion2`
- Definir clases con los tests correspondientes a cada caso con solución conocida. Estos tests serán definidos como funciones cuyo nombre debe empezar con la palabra *test*. ¡Usar nombres declarativos!
- Agregar al final del archivo de testing el código

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

- verbosity: Indica qué tanto detalle se mostrará por pantalla.
  `verbosity=0` Solo se muestra el resultado final de la ejecución (por ejemplo: OK o un resumen de errores).
  `verbosity=1` Muestra una línea por cada test ejecutado, representada con puntos, letras F o E según el resultado. Ejemplo: .F.E.
  `verbosity=2` Muestra el nombre de cada test y su resultado.
- Guardar el archivo de testing (ej: testsguia6.py) y correr en consola. Ej: `python3 testsguia6.py -v`
  La *-v* es útil para obtener más información de los tests (si usamos `verbosity=2` no añade más información).
- Chequear la salida.

Ejemplo del archivo **guia6.py**:

```
def volumen_esfera(radio:float) -> float:
    #aproximamos pi a 3,14
    res: float = 4/3 * 3.14 * radio**3
    return res

def triada_pitagorica(a: int, b: int, c:int) -> bool
    suma: int = a**2 + b**2
    res: bool = suma == c*2
    return res
```

Ejemplo del archivo **testsguia6.py**:

```
import unittest
from guia6 import volumen_esfera, triada_pitagorica

class test_volumen(unittest.TestCase):
    def test_volumen_1(self):
        #lo hacemos en tres pasos para mayor claridad
        resultado_esperado: float = 4.1867
        resultado_obtenido: float = volumen_esfera(1.0)
        self.assertAlmostEqual(resultado_obtenido, resultado_esperado, places=4)

    def test_volumen_nulo(self):
        self.assertAlmostEqual(volumen_esfera(0.0), 0.0, places=1)

    def test_volumen_5_25(self):
        self.assertAlmostEqual(volumen_esfera(5.25), 605.82375, places=5)

class test_triada_pitagorica(unittest.TestCase):
    def test_triada_verdadera_correcta(self):
        self.assertTrue(triada_pitagorica(3,4,5))

    def test_triada_falsa(self):
        self.assertFalse(triada_pitagorica(1,7,9))

    def test_triada_ok_desordenada(self):
        self.assertFalse(triada_pitagorica(5,4,3))

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

```
ltz@ltz:~$ python3 tests_guia6_clase2.py -v
test_triada_falsa (__main__.test_triada_pitagorica) ... ok
test_triada_ok_desordenada (__main__.test_triada_pitagorica) ... ok
test_triada_verdadera_correcta (__main__.test_triada_pitagorica) ... FAIL
test_volumen_1 (__main__.test_volumen) ... ok
test_volumen_5_25 (__main__.test_volumen) ... ok
test_volumen_nulo (__main__.test_volumen) ... ok

======================================================================
FAIL: test_triada_verdadera_correcta (__main__.test_triada_pitagorica)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/ltz/tests_guia6_clase2.py", line 21, in test_triada_verdadera_corr
ecta
    self.assertTrue(triada_pitagorica(3,4,5))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 6 tests in 0.000s

FAILED (failures=1)
```

Corrimos 6 tests, de los cuales falló uno. Leyendo el mensaje vemos que el problema está en la línea 21 del test: esperaba True y obtuvo False. Sabiendo que 3, 4, 5 es una triada pitagórica, debemos mirar la implementación de la función a ver dónde está el error.

Revisando el código con detenimiento, podemos notar que el error está en la comparación == en la segunda línea de la función: la suma de los cuadrados debe ser igual a c^2 para que la función devuelva True. Faltó agregar un asterisco más antes del 2.

```
def triada_pitagorica(a: int, b: int, c:int) -> bool
    suma: int = a**2 + b**2
    res: bool = suma == c*2:
    return res
```

Corregimos el error:

```
def triada_pitagorica(a: int, b: int, c:int) -> bool
    suma: int = a**2 + b**2
    res: bool = suma == c**2:
    return res
```

Guardamos el archivo y volvemos a correr el test.

```
ltz@ltz:~$ python3 testsguia6.py -v
test_triada_desordenada (__main__.test_triada_pitagorica) ... ok
test_triada_falsa_correcta (__main__.test_triada_pitagorica) ... ok
test_triada_verdadera_correcta (__main__.test_triada_pitagorica) ... ok
test_volumen_1 (__main__.test_volumen) ... ok
test_volumen_5_25 (__main__.test_volumen) ... ok
test_volumen_nulo (__main__.test_volumen) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

Ahora todos los tests pasaron. Recordar que una falla en un test puede deberse a un error en la implementación de la función probada o a un error en el resultado esperado por el test.

¡A ejercitar!

¡A ejercitar!

Ejercicio 1. Escribir casos de test para las siguientes funciones:

```
es_multiplo_de(n: int, m: int) -> bool

devolver_el_doble_si_es_par(n: int) -> int

fahrenheit_a_celsius(t: float) -> float
```

Importante! testear `fahrenheit_a_celsius`. ¿Qué debemos tener en cuenta? Ejecutar en consola `0.1 + 0.2 == 0.3` ¿Cuál es el valor esperado?

¡A ejercitar!

Ejercicio 2. Implementar y escribir casos de test para la función `es_primo (n: int)-> bool`, que toma como entrada un entero y devuelve si es primo o no.

¡A ejercitar!

Ejercicio 3. Implementar y escribir casos de test para la función `cuantos_primos_en_rango (m: int, n:int)-> int`, que devuelve la cantidad de primos presentes en el rango de enteros comprendido entre *m* y *n*, inclusive. Atención: también debe valer si *m* es mayor o igual a *n*.

Ejemplos:

```
cuantos_primos_en_rango(-3, 4) = 2
cuantos_primos_en_rango(11, 11) = 1
cuantos_primos_en_rango(8, 2) = 4
```