Introducción a la programación

Testing - cubrimiento

Noviembre de 2024

Ejercicio 9

Sea la siguiente especificación del problema de `sumar` y una posible implementación en lenguaje imperativo:

problema sumar (in x: $\mathbb{Z}$, in y: $\mathbb{Z}$) : $\mathbb{Z}$ {
  requiere: $\{True\}$
  asegura: $\{result = x + y\}$
}

```
def sumar(x: int , y: int) -> int:
L1:    sumando: int = 0
L2:    abs_y: int = 0
L3:    if y < 0:
L4:       sumando = -1
L5:       abs_y = -y
       else:
L6:       sumando = 1
L7:       abs_y = y
L8:    result: int = x
L9:    count: int = 0
L10:   while(count < abs_y):
L11:       result = result + sumando
L12:       count = count + 1
L13:   return result
```

Ejercicio 9

1. Describir el diagrama de control de flujo (control-flow graph) del programa `sumar`.
2. Escribir un test suite que ejecute todas las líneas del programa `sumar`.

Tests de unidad en Python

Para armar tests de unidad en Python debemos usar alguna biblioteca (como usábamos HUnit en Haskell). Algunas muy utilizadas son:

- `unittest`: viene con Python, es un poco más compleja la sintaxis (se usan clases)
- `pytest`: es necesario instalarla, es más simple de utilizar

unittest - Primeros pasos

```python
import unittest
from funciones import funcion

class FuncionesTest(unittest.TestCase):

    def test_1(self):
        self.assertEqual(funcion(28), 4, "primer test")

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

Archivo en python con casos de test para testear $funcion$ definida en $funciones.py$

`unittest` - funciones

Algunas funciones que podemos usar para escribir los casos de test:

- `assertEqual(a, b)`:testea que a y b tengan el mismo valor
- `assertTrue(x)`:testea que x sea True
- `assertFalse(x)`:testea que x sea False
- `assertIn(a, b)`:testea que a esté en b (siendo b una lista o tupla)

`unittest` - Ejecución de casos de test

El archivo con casos de test lo ejecutamos como cualquier archivo `.py` y veremos el siguiente resultado:

```
test_1 (__main__.FuncionesTest.test_1) ... FAIL
test_2 (__main__.FuncionesTest.test_2) ... ok

======================================================================
FAIL: test_1 (__main__.FuncionesTest.test_1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/gabidp/Documents/Facultad/intro-programacion
    self.assertEqual(funcion(28), 4, "primer test")
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
AssertionError: 29 != 4 : primer test

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Salida después de ejecutar 2 casos, uno de forma exitosa y otro con error

Cubrimiento

Para poder visualizar el cubrimiento de líneas de nuestro test suite, tendremos que instalar `coverage`.
En Linux: `pip install coverage`
Para ver como instalarlo en otros sistemas operativos:
https://devguide.python.org/testing/coverage/#install-coverage

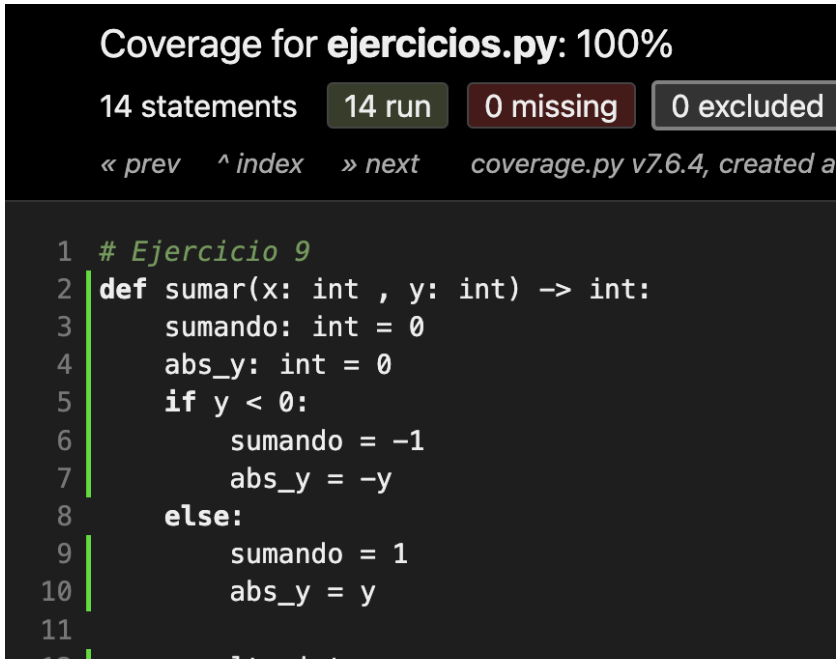
Cubrimiento

Es posible ver el resultado por consola o en un arhicov html:

- `coverage run --include=ejercicios.py -m unittest` : indicamos el cubrimiento de qué archivo queremos evaluar con –include
- `coverage report`

```
Name              Stmts   Miss  Cover
-------------------------------------
ejercicios.py        14      0   100%
-------------------------------------
TOTAL                14      0   100%
```

- `coverage html`



Ejercicio 11

Sea el siguiente programa que retorna diferentes valores dependiendo si a, b y c, definen lados de un triángulo inválido, equilátero, isósceles o escaleno.

```
def triangle(a: int , b: int, c: int) -> int:
L1:     if(a <= 0 or b <= 0 or c <= 0):
L2:         return 4 # invalido
L3:     if(not ((a + b > c) and (a + c > b) and (b + c > a))):
L4:         return 4 # invalido
L5:     if(a == b and b == c):
L6:         return 1 # equilatero
L7:     if(a == b or b == c or a == c):
L8:         return 2 # isosceles
L9:     return 3  # escaleno
```

- Describir el diagrama de control de flujo (control-flow graph) del programa `triangle`.
- Escribir un test suite que ejecute todas las líneas y todos los branches del programa.
- usar la herramienta coverage para visualizar las lineas cubiertas