Programming Lab

Parte 7

Testing e unit tests

+ numpy e matplotlib

Laura Nenzi

Cos’è il testing

- Per testing si intende il testare, in genere in modo automatico, delle cose. Dal software, ad una penna, al vostro cellulare.
- Il testing del software è più facile di quello dell’hardware (dove servono attrezzature specializzate), perchè basta scrivere del software in più
- Esiste chi fa il testing del software di testing :)

Cos’è il testing

Pseudocodice per un generico concetto di testing:

dato un **input** e un **output** *noto*

input *noto* → CODICE → **output**

if **output** != **output** *noto*:

    errore!

Cos’è il testing

Codice Python per il testing di una funzione che fa la somma di due numeri:

```
# Funzione somma
def somma(a,b):
    return a+b

# Testing
if not somma(1,1) == 2:
    raise Exception('Test 1+1 non passato')

if not somma(1.5,2.5) == 4:
    raise Exception('Test 1.5+2.5 non passato')
```

Cos’è il testing

Codice Python per il testing di una funzione che fa la somma di due numeri:

lezione7.py

```python
# Funzione somma
def somma(a,b):
    return a+b
```

test_lezione7.py

```python
from lezione7 import somma

# Testing
if not somma(1,1) == 2:
    raise Exception('Test 1+1 non passato')

if not somma(1.5,2.5) == 4:
    raise Exception('Test 1.5+2.5 non passato')
```

Testing vs unit testing

Il testing generico (*end-to-end*) può essere effettuato su tutto il codice /programma.

input → **programma o funzione molto grossa** → output

Se invece testo le “minime” unità testabili, allora si parla di *unit testing*:

input → **funzione piccola** → output
input → **oggetto piccolo**→ output
input → **altra funzione piccola**→ output

In questo modo sono molto più granulare nel capire dove è andato storto cosa.

Importante per la rifattorizzazione

I test mi permettono di verificare in maniera automatica se un codice che sto modificando funziona ancora

Il modulo unittest

lezione7.py

```
# Funzione somma
def somma(a,b):
    return a+b
```

test_lezione7.py

```
import unittest
from lezione7 import somma

# Testing
class TestSomma(unittest.TestCase):

    def test_somma(self):
        self.assertEqual(somma(1,1), 2)
        self.assertEqual(somma(1.5,2.5), 4)
```

Framework unittest

- Definisco le diverse classi di test, estendendo la classe base `unittest.TestCase`
- Definisco det test case con dei metodi che devono iniziare per `"test_"`
- I controlli si fanno con degli `assert`

test_lezione7.py

```
import unittest
from lezione7 import somma

# Testing
class TestSomma(unittest.TestCase):

    def test_somma(self):
        self.assertEqual(somma(1,1), 2)
        self.assertEqual(somma(1.5,2.5), 4)
```

Gli assert

```
self.assertEqual(3+2,5)
self.assertNotEqual(3+2,6)

self.assertTrue(1 < 2)
self.assertFalse(1 > 2)

self.assertIn(3, [1, 2, 3])
self.assertNotIn(4, [1, 2, 3])

self.assertIsInstance(123, int)
self.assertNotIsInstance(123, str)

with self.assertRaises(ValueError):
    int("invalid")
```

Gli `assert`

Metodo	Scopo
assertEqual, assertNotEqual	Confronto di uguaglianza.
assertTrue, assertFalse	Verifica di verità.
assertIn, assertNotIn	Verifica di appartenenza.
assertIs, assertIsNot	Identità degli oggetti.
assertIsInstance, assertNotIsInstance	Verifica del tipo.
assertRaises, assertRaisesRegex	Controllo delle eccezioni.
assertListEqual, assertTupleEqual, assertDictEqual	Confronto tra sequenze o dizionari.
assertAlmostEqual, assertNotAlmostEqual	Confronto approssimativo per numeri decimali.

Il modulo unittest

Il comando "`python -m unittest`" esegue i test del file specificato:

```
(mio_ambiente) lauranenzi@imaclaura MyProgrammingLab % python -m unittest test_lezione7.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Il flag "`-v`" aggiunge un output dettagliato:

```
(mio_ambiente) lauranenzi@imaclaura MyProgrammingLab % python -m unittest -v test_lezione7
test_somma (test_lezione7.TestSomma.test_somma) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Il modulo unittest

Il comando "**`python -m unittest discover`**" od anche solo "**`python -m unittest`**" esegue tutti i file che iniziano con test_ e terminano con .py nella directory corrente. "**`-v`**" di nuovo aggiunge dettagli

```
(mio_ambiente) lauranenzi@imaclaura MyProgrammingLab % python -m unittest -v
ALL TESTS PASS
test_init (test_esParte7.TestCSVFile.test_init) ... ok
test_start_end (test_esParte7.TestCSVFile.test_start_end) ... ok
test_somma (test_lezione7.TestSomma.test_somma) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Come verrete valutati

L'autograding usa degli unit test. Esempio con l’oggetto CSVFile:

```
import unittest
from esame import CSVFile

class TestCSVFile(unittest.TestCase):

    def test_init(self):

        csv_file = CSVFile('shampoo_sales.csv')

        # Controllo che il nome del file sia stato salvato
        # in un attributo dell’oggetto di nome “name”
        self.assertEqual(csv_file.name, 'shampo_sales.csv')
```

P.s. sviluppare codice test-driven

Un bellissimo modo per sviluppare codice è essere test-driven:

PRIMA scrivo i test, POI il codice.

In questo modo mi focalizzo prima su che cosa voglio che faccia il codice, e se sono bravo prevedo anche i casi strani (se passo una stringa alla funzione somma cosa voglio che succeda?)

..non è richiesto per questo corso, è giusto un accenno per voi.

I test si fanno su dati piccoli!!!

Il metodo `setup()`

- viene utilizzato nei test unitari con unittest per preparare l'ambiente di test prima di ogni singolo test
- Il suo ruolo principale è garantire che ogni test parta da una base pulita e coerente

Il metodo `setup()`

```python
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

    def decrement(self):
        self.value -= 1
```

```python
import unittest
from un_file import Counter

class TestCounter(unittest.TestCase):

    def setUp(self):
        # Creo un'istanza della classe che utilizzo nei test
        self.counter = Counter()

    def test_initial_value(self):
        # Verifica che il valore iniziale del contatore sia 0
        self.assertEqual(self.counter.value, 0)

    def test_increment(self):
        # Verica incremento
        self.counter.increment()
        self.assertEqual(self.counter.value, 1)
```

Il metodo `setup()`

```
import unittest
class TestCSVFile(unittest.TestCase):

    def setUp(self):
        # Specifica il percorso del file esterno 'data_test'
        self.file_name = 'data_test'

        # Creazione dell'istanza della classe CSVFile
        self.csv_file = CSVFile(self.file_name)

        def test_file_not_found(self):
        # Test per verificare se viene sollevata un'eccezione quando il file non esiste
        with self.assertRaises(FileNotFoundError):
            non_existent_file = CSVFile('file_non_esistente.csv')
```

Il metodo `setup()`

```
import unittest
import tempfile
class TestCSVFile(unittest.TestCase):

    def setUp(self):
        # Creare un file temporaneo generico con contenuti per i test
        self.file = tempfile.NamedTemporaryFile(delete=False, mode='w+t')
        self.file.write("Header1,Header2,Header3\nA,1,2\nB,3,4\nC,5,6\n")
        self.file.seek(0)
        self.file_name = self.file.name
        self.csv_file = CSVFile(self.file_name)
```

Esercizio

Scrivete dei test o unit test (a seconda di cosa più vi aggrada) per gli oggetti **CSVFile** e **NumericalCSVFile**.

Potete per esempio:

- creare vari file di test e verificare che la funzione get data dia sempre l’output che vi aspettate
- verificare che il nome del file sia salvato come attributo
- verificare che vengano alzate specifiche eccezioni

(usare il costrutto *try-except* o *self.assertRaises()* se si usa il modulo unittest)