INF8007 – Languages de scripts
Tests

Antoine Lefebvre-Brossard

Hiver 2018

Utilité

Écrire des tests pour son programme permet de :

- vérifier les différentes composantes d'un programme individuellement
- localiser certains problèmes de performance ou d'utilisation
- donner des exemples d'utilisations de chacune des composantes
- préciser l'utilisation de chacune des composantes

Exemple


```python
def fct(text: str):
    return text.split()

assert fct("Le chat noir") == ["Le", "chat", "noir"] # Ok
assert fct("Le chat noir.") == ["Le", "chat", "noir", "."] # Erreur
assert fct("L'iguane noir") == ["L'", "iguane", "noir"] # Erreur
```

La fonction doit donc être modifié pour respecter les tests. S'assurer que les tests sont compréhensifs et ne comportent pas eux-même d'erreurs peut sauver beaucoup de temps et d'efforts

Librairies

Il y a plusieurs librairies en Python pour écrire des tests, mais les plus utilisées sont :

- unittest
- pytest

unittest

- Le module `unittest` fournit un cadre pour écrire des tests unitaires inspiré de JUnit
- Les tests unitaires comportent des *cas de tests* qui vérifient que les routines produisent les sorties spécifiées en fonction d'entrées
- Le cadre permet aussi la spécification d'une configuration (BD, répertoires, serveur)
- Les cas de tests sont regroupés dans une batterie (*suite*) de tests
  - `unittest.TestSuite()`
  - `unittest.TestCase()`
- Repose sur l'introspection et la convention d'écrire toutes les routines de tests en commançant par `test`nom``
- Pour plus d'information, voir *https://docs.python.org/3/library/unittest.html*

Exemples


```python
import random, unittest

class TestSequenceFunctions(unittest.TestCase):
    def setUp(self):
        self.seq = range(10)

    def testshuffle(self):
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

    def testsample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assert_(element in self.seq)

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

Exemples (suite)


```python
import unittest

def raises_error(*args, **kwargs):
    print(args, kwargs)
    raise ValueError(f"Invalid value: {str(args)}, {str(kwargs)}")

class ExceptionTest(unittest.TestCase):
    def testTrapLocally(self):
        try:
            raises_error("a", b="c")
        except ValueError:
            pass
        else:
            self.fail("Exception `ValueError` non relevée")

    def testFailUnlessRaises(self):
        self.failUnlessRaises(ValueError, raises_error, 'a', b='c')

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

Exemples (suite)


```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testDefaultSize(self):
        self.assertTrue(self.widget.size() == (50, 50),
                        "incorrect default size")

    def testResize(self):
        self.widget.resize(100, 150)
        self.assertTrue(self.widget.size() == (100, 150),
                        "wrong size after resize")

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

unittest (suite)

- `setUp()` et `tearDown()` sont appelés après chaque test, simplifiant la tâche
- On regroupe des tests avec les objets `TestSuite`
- Est utilisé en exécutant
  `python -m unittest `file or directory``

pytest

- Le module `pytest` permet d'écrire des tests de façon plus simple que `unittest` et sans le bagage historique
- Il a la même logique d'écrire des *cas de tests* et permet aussi la spécification d'une configuration
- Chaque test est une fonction de la forme `test_`nom``
- Utilise les fonctions natives de Python (`assert`)
- Pour plus d'information, voir *https://docs.pytest.org/en/latest/*

Exemples


```python
import pytest, random

@pytest.fixture
def seq():
    return range(10)

def test_shuffle(seq):
    random.shuffle(seq)
    seq.sort()
    assert seq == range(10)

def test_sample(seq):
    with pytest.raises(ValueError):
        random.sample(seq, 20)
    for element in random.sample(seq, 5):
        assert element in seq
```

Exemples (suite)


```python
import pytest

@pytest.fixture
def widget():
    return Widget("The widget")

def test_default_size(widget):
    assert widget.size() == (50, 50), "incorrect default size"

def test_resize(widget):
    widget.resize(100, 150)
    assert widget.size() == (100, 150), "wrong size after resize"

def test_default_size_again(widget):
    assert widget.size() == (50, 50), "fixture has been updated"
    widget.resize(100, 150)
    assert widget.size() == (100, 150), "wrong size after resize"
```

pytest (suite)

- Les tests peuvent être regroupés dans des classes de la forme `Test`nom``
- Est utilisé en exécutant `pytest `file or directory``

Structure de fichiers

En général, il plus clair d'avoir une structure de la forme

```
root/
    src/
        ...application...
    test/
        ...test_files...
```