Python
Tests Unitaires

Nicolas Delestre

INSA
ROUEN NORMANDIE

Plan

INSA ROUEN NORMANDIE

Exemple d'erreur

point.py

```
def _get_x(self):
    return self._x

def _set_x(self, x):
    self._x = x

x = property(_get_x, _set_x)

@property
def y(self):
    return self._x
```

INSA ROUEN NORMANDIE

Plusieurs *frameworks*

- Il y a eu historiquement plusieurs frameworks de développement de tests unitaires sour python :
  - *unittest*
  - nose
  - *doctest*
  - **pytest**

INSA ROUEN NORMANDIE

unittest 1 / 2

Caractéristiques

- S'inspire des framework de tests unitaires des autres langages, comme JUnit
- Intégré de base à python
- Il ne tire pas parti des aspects introspections du python

Développement du test unitaire : test_point.py

```
import unittest
from point import Point2D

class TestPoint2D(unittest.TestCase):
    def test_abscisse(self):
        self.assertEqual(Point2D(1,2).x, 1)

    def test_ordonnee(self):
        self.assertEqual(Point2D(1,2).y, 2)

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

unittest 2 / 2

Exécution du test unitaire

```
$ python3 test_point.py
.F
======================================================================
FAIL: test_ordonnee (__main__.TestPoint2D)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_point.py", line 11, in test_ordonnee
    self.assertEqual(Point2D(1,2).y, 2)
AssertionError: 1 != 2

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

INSA ROUEN NORMANDIE

doctest 1 / 5

Caractéristiques

- Très *pythonic*
- Avantages :
  - intégré de base à python
  - la documentation intègre les tests unitaires (tests à jour, sert aussi de documentation)
  - Le module et les tests unitaires forment un tout (exécution des tests dans la partie `if __name__ == "__main__"`)
- Inconvénients :
  - si les tests sont long, la documentation prend beaucoup de place
  - attention aux espaces (ou tabulations) après le résultat attendu

INSA ROUEN NORMANDIE

Définition des tests unitaires : point.py

```
@property
def x(self):
    """ Propriété permettant d'obtenir et de fixer l'abscisse d'un Point2D

    >>> p = Point2D(1,2)
    >>> p.x
    1
    >>> p.x = 3
    >>> p.x
    3
    """
    return self._x

@x.setter
def x(self, x):
    self._x = x
```

Définition des tests unitaires : point.py (suite et fin)

```
    @property
    def y(self):
        """ Propriété permettant d'obtenir et de fixer l'ordonnée d'un Point 2D

        >>> p = Point2D(1,2)
        >>> p.y
        2
        >>> p.y = 3
        >>> p.y
        3
        """
        return self._x
```

Code d'exécution : point.py

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest 4 / 5

Exécution directement depuis le module

```
$ python3 point.py
**********************************************************************
File "point.py", line ?, in __main__.Point2D.y
Failed example:
    p.y
Expected:
    2
Got:
    1
**********************************************************************
File "point.py", line ?, in __main__.Point2D.y
Failed example:
    p.y
Expected:
    3
Got:
    1
**********************************************************************
1 items had failures:
   2 of   4 in __main__.Point2D.y
***Test Failed*** 2 failures.
```

doctest 5 / 5

Apparition des tests dans la documentation

```
$ python3
>>> import point
>>> help(point.Point2D)
Help on class Point2D in module point:

class Point2D(builtins.object)
 |  Methods defined here:
...
 |  Data descriptors defined here:
 |
...
 |  x
 |      Propriété permettant d'obtenir et de fixer l'abscisse d'un Point2D
 |
 |      >>> p = Point2D(1,2)
 |      >>> p.x
 |      1
 |      >>> p.x = 3
 |      >>> p.x
 |      3
 |
 |  y
 |      Propriété permettant d'obtenir et de fixer l'ordonnée d'un Point 2D
 |
 |      >>> p = Point2D(1,2)
 |      >>> p.y
 |      2
 |      >>> p.y = 3
 |      >>> p.y
 |      3
```

Pytest

Caractéristiques

- Le plus populaire !
- Très simple à mettre en oeuvre (utilisation poussée de l'introspection)
- Recherche de lui même les tests unitaires (modules/fonctions dont le nom commence par `test_`)
- Des messages clairs
- Les méthodes avant (`setup()`) et après (`teardown()`) les tests unitaires sont remplacées par des *fixtures*
- Possibilité de paramétrer des tests
- Sait utiliser les tests unittest et doctest
- Ne fait pas parti de l'installation de base :
  `pipenv install pytest`

Premier exemple 1 / 2

test_point.py

```
#!/usr/bin/env python

from point import Point2D

def test_abscisse():
    assert Point2D(1,2).x == 1

def test_ordonnee():
    assert Point2D(1,2).y == 2
```

INSA ROUEN NORMANDIE

Premier exemple 2 / 2

Exécution

```
python3 -m pytest
================================== test session starts ==================================
platform linux -- Python 3.7.2, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir:/home/delestre/Documents/Cours/ASI/Python/Cours/08-TestsUnitaires/Version1.1/exemples
/pytest/erreursSurPoint
collected 2 items

test_point.py .F                                                                  [100%]

======================================== FAILURES ========================================
_____________________________________ test_ordonnee _____________________________________

    def test_ordonnee():
>       assert Point2D(1,2).y == 2
E       assert 1 == 2
E        +  where 1 = Point2D(1,1).y
E        +    where Point2D(1,1) = Point2D(1, 2)

test_point.py:9: AssertionError
============================== 1 failed, 1 passed in 0.03s ==============================
```

Deuxième exemple

test_polyligne.py (on suppose point.py corrigé)

```
#!/usr/bin/env python3

import pytest
from polyligne import Polyligne, MemePointInterditErreur
from point import Point2D

def test_polyligne_est_fermee():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    assert polyligne_fermee.est_ferme

def test_polyligne_longueur():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    assert len(polyligne_fermee) == 3

def test_polyligne_longueur_apres_ajout():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    longueur = len(polyligne_fermee)
    polyligne_fermee.ajouter(Point2D(0,0))
    assert len(polyligne_fermee) == longueur + 1

def test_polyligne_ajout_avec_erreur():
    polyligne_fermee = Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))
    with pytest.raises(MemePointInterditErreur):
        polyligne_fermee.ajouter(Point2D(1,1))
```

Fixture 1 / 3

Remarques sur l'exemple

- Du code a été copié/collé (création de `polyligne_fermee`)
- Une solution serait de faire une variable globale
- Sauf que certains tests la modifieraient (par exemple `est_polyligne_longueur_apres_ajout`)

Fixture

- les *fixtures* permettent de générer des données de tests
- les *fixtures* sont paramétrable à l'aide des paramètres nommés, entre autres :
  - `scope` qui définit la portée de création (session, module, class, function)
  - `params` et le paramètre formel `request` qui possède un champ `param`
- des *fixtures* sont utilisés comme paramètres formels des tests

Fixture 2 / 3

test_polyligne.py

```
@pytest.fixture(scope="function")
def polyligne_fermee():
    return  Polyligne(True, Point2D(1,1), Point2D(2,2), Point2D(1,2))

@pytest.fixture(scope="function", params=[Point2D(1,1), Point2D(2,2), Point2D(1,2)])
def point_a_ajouter_qui_pose_probleme(request):
    return request.param

@pytest.fixture(scope="function", params=[Point2D(0,0), Point2D(2,1), Point2D(3,3)])
def point_a_ajouter_qui_ne_pose_pas_probleme(request):
    return request.param

def test_polyligne_est_fermee(polyligne_fermee):
    assert polyligne_fermee.est_ferme

def test_polyligne_longueur_apres_ajout(polyligne_fermee):
    longueur = len(polyligne_fermee)
    polyligne_fermee.ajouter(Point2D(0,0))
    assert len(polyligne_fermee) == longueur + 1

def test_polyligne_ajout_avec_erreur(polyligne_fermee, point_a_ajouter_qui_pose_probleme):
    with pytest.raises(MemePointInterditErreur):
        polyligne_fermee.ajouter(point_a_ajouter_qui_pose_probleme)

def test_polyligne_ajout_sans_erreur(polyligne_fermee, point_a_ajouter_qui_ne_pose_pas_probleme):
    polyligne_fermee.ajouter(point_a_ajouter_qui_ne_pose_pas_probleme)
```

Fixture 3 / 3

Exécution : il y a bien 8 tests unitaires qui sont exécutés

```
$ python3 -m pytest
================================== test session starts ==================================
platform linux -- Python 3.7.2, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /home/delestre/Documents/Cours/ASI/Python/Cours/08-TestsUnitaires/Version1.1/exemples
/pytest/erreursSurPolyligne
collected 11 items

test_polyligne.py ...........                                                    [100%]

================================== 11 passed in 0.03s ===================================
```

Paramètre 1 / 3

Tests unitaires paramétrés

- Quelques fois on veut tester une fonction ou une méthode avec plusieurs valeurs
- Il est possible de définir une fonction de tests qui possède un ou plusieurs paramètres formels et de demander l'exécution de cette fonction, grâce au décorateur `@pytest.mark.parametrize`, avec une liste de paramètres effectifs (qui sont des tuples si la fonction admet plusieurs paramètres formels)

Exemple

- On voudrait vérifier que la longueur d'une polyligne fonctionne bien après la création de la dîte polyligne :
  - quelles soient fermées ou pas
  - qu'il y ait utilisation ou pas des paramètres non nommés optionnels

```
def __init__(self, est_ferme, pt1, pt2, *args):
```

Paramètre 2 / 3

test_polyligne.py : ajout d'un test paramétré

```
@pytest.mark.parametrize("polyligne, longueur",
                         [(Polyligne(False, Point2D(1,1), Point2D(2,2)), 2),
                          (Polyligne(True, Point2D(1,1), Point2D(2,2)), 2),
                          (Polyligne(False, Point2D(1,1), Point2D(2,2), Point2D(1,2)), 3)
                        ])
def test_polyligne_longueur_apres_init(polyligne, longueur):
    assert len(polyligne) == longueur
```

INSA ROUEN NORMANDIE

Paramètre 3 / 3

Exécution : il y a bien 11 (8+3) tests unitaires qui sont exécutés

```
$ python -m pytest
================================== test session starts ==================================
platform linux -- Python 3.7.2, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /home/delestre/Documents/Cours/ASI/Python/Cours/08-TestsUnitaires/Version1.1/exemples
/pytest/erreursSurPolyligne
collected 11 items

test_polyligne.py ...........                                                    [100%]

================================== 11 passed in 0.02s ===================================..
```