08/10/2021

Python pour l'ingénieur

Tests unitaires avec pytest

Module de formation

© Copyright 2021 Cyril Keime & Arnaud Bêche

INTRODUCTION

© Copyright 2021 Cyril Keime & Arnaud Bêche

xUnit

- Wikipédia : « *Le terme générique xUnit désigne un outil permettant de réaliser des tests unitaires dans un langage donné.* »

Exemples :
JUnit (Java)
unittest (Python)

- Vocabulaire :
  - unit testing framework
  - test-driven development
  - test case
  - test suite
  - test fixture
  - test runner
  - mock object
  - …

Tests automatiques
Tests de non-régression (TNR)
Qualité du logiciel

- pytest : peut remplacer unittest

  The pytest framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

Light-weight syntax for writing tests
(only plain assert statements are used)
Rich plugin architecture

© Copyright 2021 Cyril Keime & Arnaud Bêche

pytest

- Licence : MIT

  https://docs.pytest.org/en/6.2.x/#license

  Copyright Holger Krekel and others, 2004-2021.

  Distributed under the terms of the MIT license, pytest is free and open source software.

- Attention, ne fait pas partie de la bibliothèque standard maintenue par la fondation Python (paquetage tierce partie)

- Versions à utiliser :
  - Python 3.6+
  - pytest 6.2.4+

- Documentation (indispensable !) :

  https://docs.pytest.org/en/6.2.x/contents.html#toc

  https://media.readthedocs.org/pdf/pytest/latest/pytest.pdf

https://pytest.org

© Copyright 2021 Cyril Keime & Arnaud Bêche

Environnement de travail

- Lancer PowerShell
- Mettre à jour la variable d'environnement Path. Exemple à adapter :

```
$Env:Path += "C:\Users\<TGI>\Documents\MyApp\WPy64-3740\python-3.7.4.amd64;"
$Env:Path += "C:\Users\<TGI>\Documents\MyApp\WPy64-3740\python-3.7.4.amd64\Scripts;"
```

- Vérifier

```
python --version
pytest --version
```

- Enfin, se positionner dans le répertoire de travail
(ie l'espace de stockage du code source)

- Lancer un terminal
- Vérifier

```
python --version
pytest --version
```

- Enfin, se positionner dans le répertoire de travail
(ie l'espace de stockage du code source)

Si besoin :

```
python -m pip install --upgrade pip
python -m pip install --upgrade pytest
```


© Copyright 2021 Cyril Keime & Arnaud Bêche

FONDAMENTAUX

© Copyright 2021 Cyril Keime & Arnaud Bêche

Mise en œuvre (1/2)

- Se placer dans le dossier contenant le code à tester

- Lancer les tests et obtenir le rapport :
  `pytest`
  `pytest -q` (quiet, decrease verbosity)
  `pytest -rA` (extra test summary info, All)

- C'est terminé ! Les tests à lancer sont trouvés automatiquement :
  - Fichiers :
    `test_*.py`
    `*_test.py`
  - Dans ces fichiers :
    Fonctions préfixées par `test`
    Méthodes préfixées par `test` dans une classe elle-même préfixées par `Test` et sans méthode
    `__init__()`

EXERCICE

Créer un module mylib contenant une fonction inc() : x ==> x+1

Tester cette fonction in situ (dans le module) :

1. Directement (cf. cours Python)
   python mylib.py
2. Avec unittest
   (pour les plus rapides)
   python mylib.py
   python -m unittest mylib

Suite de l'exercice : planche suivante !

© Copyright 2021 Cyril Keime & Arnaud Bêche

Mise en œuvre (2/2)

- pytest utilise directement l'instruction **`assert`** du langage Python

`assert <expr>`
  - Lève l'exception AssertionError si **`<expr>`** est False

`assert <expr>, <msg>`
  - Lève l'exception AssertionError avec le message **`<msg>`** si **`<expr>`** est False

- Le code de test (ie dans les fonctions ou méthodes préfixées par **`test`**) doit utiliser **`assert`**

EXERCICE

Créer trois modules identiques nommés test_mylib, mylib_test, foobar.

Contenu :

- fonction de test de inc()
- classe de test de inc()

Lancer et expliquer :

```
pytest -q
pytest -q foobar.py
pytest -q mylib.py
```


© Copyright 2021 Cyril Keime & Arnaud Bêche

Tester une exception

- Pour tester une exception, la fonction à tester doit être placée dans un contexte **`raises`** fourni par pytest :

```
with pytest.raises(<exception>):
    f()
```

- Le test est passé avec succès si l'exception est levée pendant l'exécution de **`f()`**

EXERCICE

Ajouter au module mylib une fonction div() : x, y ==> x / y

(créer le module mylib2)

Tester le cas y = 0, ie vérifier qu'une exception est bien levée lorsque y = 0.

(créer le module mylib2_test)

Faire échouer le test en ajoutant dans la même fonction le test d'une division par 1.

pytest mylib2_test.py

© Copyright 2021 Cyril Keime & Arnaud Bêche

Marqueurs

- Le mécanisme de décoration Python est utilisé pour marquer les tests

  décorateur : expression dont le résultat est une fonction ayant un seul argument : le code de la fonction décorée, et dont le résultat est attaché au nom de la fonction décorée

```
@mydecorator
def func(...):
    ...
```

```
def func(...):
    ...
func = mydecorator(func)
```

- Un marqueur pytest a la forme suivante :

  `@pytest.mark.<nom>`

- La liste des marqueurs est disponible en ligne de commande :

  `pytest --markers`

EXERCICE

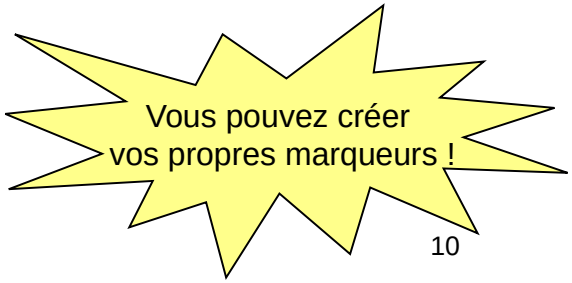
Utiliser le marqueur parametrize pour tester inc() avec différentes entrées.

(créer le module mylib2b_test)

Ajouter un test de la fonction inc() :

- qui doit échouer,
- mais marqué comme tel !

  Expected to fail…



© Copyright 2021 Cyril Keime & Arnaud Bêche

Aménagements (1/3)

- Un aménagement de test est une fonction permettant **de définir et d'organiser** le contexte d'exécution d'un test ou d'un ensemble de tests

  En anglais : fixture

- Un aménagement pytest a la forme suivante :

  `@pytest.fixture`

- Une fonction de test, **ou un aménagement**, déclare avoir besoin d'un aménagement **<nom>** en déclarant un argument **<nom>**

- Avant d'exécuter une fonction de test, **ou un aménagement**, pytest exécute chaque aménagement déclaré dans la liste des arguments, et associe son résultat à l'argument **<nom>**

EXERCICE

Définir deux tests et deux aménagements permettant de tester inc() avec les valeurs :

- 0 à 9
- 0 à 9, 10 et 42

(créer le module mylib2c_test)

Un aménagement peut être paramétré avec le marqueur `parametrize`

© Copyright 2021 Cyril Keime & Arnaud Bêche

Aménagements (2/3)

- Quand pytest exécute **un test**, et donc l'ensemble des aménagements requis par ce test ou les aménagements de ce test (rappel : un aménagement peut déclarer avoir besoin d'un aménagement) :

  CHAQUE AMENAGEMENT EST EXECUTE UNE ET UNE SEULE FOIS PAR TEST (le résultat est stocké, ie toute requête après la première reçoit ce résultat stocké)

- Un aménagement requis par défaut pour tous les tests peut être défini de la manière suivante :

```
@pytest.fixture(autouse=True)
```

- Il est possible de définir une portée (une durée de vie) pour un aménagement. Exemple :

```
@pytest.fixture(scope='module')
```

EXERCICE

Modifier le code pour ne créer qu'une seule fois le vecteur de test 0 à 9.

(pour mettre en évidence le comportement : utiliser print() et mettre les tests en échec)

(pour les plus rapides : définir un aménagement autouse pour ajouter -1 aux vecteurs)

`function` the default scope, the fixture is destroyed at the end of the test
`class` the fixture is destroyed during teardown of the last test in the class
`module` the fixture is destroyed during teardown of the last test in the module
`package` the fixture is destroyed during teardown of the last test in the package
`session` the fixture is destroyed at the end of the test session

© Copyright 2021 Cyril Keime & Arnaud Bêche

Aménagements (3/3)

- Un aménagement peut nécessiter un **nettoyage** après un test, afin de ramener le système dans un état propre pour le prochain test (par exemple : fermer un fichier, clore une connexion réseau, supprimer des fichiers temporaires, etc)

  En anglais : teardown, cleanup, finalization

- Pour définir un nettoyage, il faut remplacer l'instruction `return` par l'instruction `yield` et placer le code de nettoyage **après** `yield` :

```
@pytest.fixture
def myfixture(...):
    <code aménagement>
    yield <résultat aménagement>
    <code nettoyage>
```

EXERCICE

Modifier le code pour simuler un nettoyage de la création du vecteur de 0 à 9.

(simuler = utiliser print() !)

© Copyright 2021 Cyril Keime & Arnaud Bêche

Plugins

- rerunfailures
  - re-runs tests to eliminate intermittent failuress
- timeout
  - terminates tests after a certain timeout
- cov
  - produces coverage reports
- check
  - allows multiple failures per test
- custom-exit-code
  - exits test session with custom exit code

EXERCICE

Définir et implémenter une classe (accompagnée de ses tests !) permettant de mettre en œuvre chacun de ces 5 plugins.

Références :
https://docs.pytest.org/en/latest/reference/plugin_list.html

Si besoin :

```
python -m pip install pytest-rerunfailures
python -m pip install pytest-timeout
python -m pip install pytest-cov
python -m pip install pytest-check
python -m pip install pytest-custom-exit-code
```


© Copyright 2021 Cyril Keime & Arnaud Bêche