

Tests avec Python Nose et Jenkins

July-04-17 10:26 AM

Nose est une extension de l'utilitaire Python unittest, conçu pour faciliter les tests unitaires. Il dispose de plusieurs plugins natifs tels que Xunit et on peut aussi lui en rajouter afin d'adapter son comportement aux besoins pressentis.

Les tests Unitaires

Un test unitaire est un test automatisé de code visant une "petite unité" (un petit morceau isolé) de fonctionnalité. Ils sont souvent conçus pour une large gamme de fonctionnalités "normales", incluant tous les cas inattendus et quelques tests qui devraient échouer. Les tests unitaires tendent à minimiser l'interaction avec les ressources externes telles que les disques, le réseau et les bases de données; les tests qui ont accès à ces ressources sont généralement les tests fonctionnels, les tests de régression ou les tests d'intégration. En général, les tests unitaires sont toujours assez simples.

Il y a plusieurs façons d'intégrer les tests unitaires dans le style de développement :

- ▶ Développement Test Driven dans lequel les tests unitaires sont écrits avant la fonctionnalité qu'ils doivent tester;
- ▶ Au moment du refactoring où le code existant (parfois du code sans aucun test automatisé) est réajusté afin d'être adapté aux tests (testable);
- ▶ Tests pour correction de Bugs (Bug fix testing), où ceux-ci sont au préalable mis en évidence par un test cible et ensuite corrigés ;
- ▶ Test pour amélioration directe du développement (straight test enhanced development), où les tests sont écrits au fur et à mesure que le code évolue;

En fin de compte, on réalise que l'essentiel c'est d'écrire des tests unitaires peu importe la manière. L'importance d'avoir des tests unitaires tient au fait qu'ils peuvent être exécutés facilement et rapidement. Ils servent d'exécutables, de documentation de base pour les fonctions et APIs et aussi d'historique précieux des bugs qui ont été corrigés dans le passé. De ce fait ils améliorent la capacité à fournir plus rapidement du code fonctionnel; ce qui est leur principal avantage.

Les Framework de tests unitaires

Il est assez fréquent d'écrire des tests pour un module de librairie telle que :

```
def test_me():  
    # ... many tests, which raise an Exception if they fail ...  
if __name__ == '__main__':  
    test_me()
```

L'instruction if est une petite boucle qui lance les tests lorsque le module est exécuté comme script depuis la ligne de commande. Ceci permet d'atteindre l'objectif de test automatisé pouvant être facilement exécuté. Malheureusement, ils ne peuvent être lancés sans "réfléchir" (intervention du développeur); ce qui est une caractéristique très importante et souvent négligée des tests automatisés. En pratique, cela signifie qu'ils ne seront exécutés que si le module est actif (en exécution).

Les gens utilisent des Framework pour découverte et exécution de tests unitaires (unit test discovery and execution Framework) afin d'ajouter des tests à du code existant, les exécuter et obtenir un rapport simple sans réfléchir. De plus, l'utilisation de tel Framework permet d'exécuter certains tests de façon sélective, de capturer et regrouper les erreurs et d'obtenir des informations sur la couverture et le profilage.

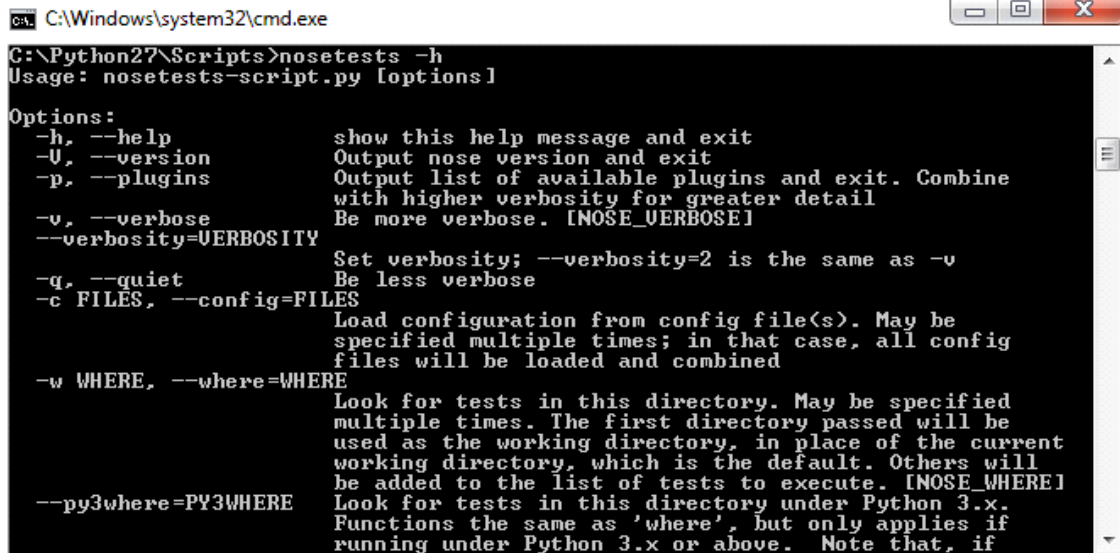
Il existe plusieurs Framework Python de tests unitaires. Nose qui est l'un d'eux est activement développé par un dénommé Jason Pellerin qui répond assez rapidement aux emails. Nose est relativement stable et dispose d'une belle architecture de plug-in qui permet de l'étendre à volonté. Il peut facilement être ajusté pour imiter tout autre Framework de découverte de test (unit test discovery) et est utilisé par bon nombre de gros projets; ce qui signifie qu'il sera encore dans les environs pour un bon nombre d'années.

Installation de Nose sur Windows

1. La version actuelle de Nose est 1.3.7. On peut télécharger le dossier compressé contenant Nose depuis le lien :
<https://pypi.python.org/pypi/nose/1.3.7> et ensuite le décompresser.
2. Si on a déjà python installé sur la machine, on peut passer à l'étape suivante sinon installer d'abord Python version 2.7 ou plus.
3. Se déplacer jusqu'au répertoire où Nose a été extrait de l'archive et exécuter la commande :
python setup.py install

```
C:\Users\ndeye-anna.ndiaye\Downloads\dist\nose-1.3.7>C:\Python27\python.exe setup.py install
```

4. Par la suite, dans le répertoire où python a été installé, on retrouve dans le sous-dossier Scripts, l'outil **nosetests** qui permet d'exécuter Nose en ligne de commande.



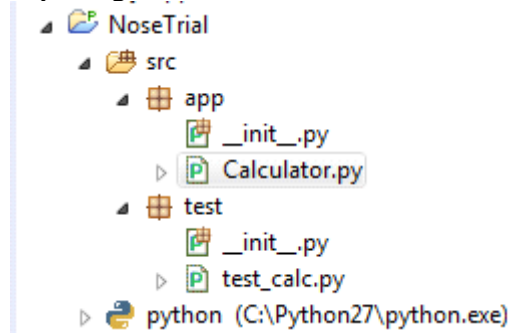
```
C:\Windows\system32\cmd.exe

C:\Python27\Scripts>nosetests -h
Usage: nosetests-script.py [options]

Options:
  -h, --help            show this help message and exit
  -V, --version          Output nose version and exit
  -p, --plugins          Output list of available plugins and exit. Combine
                        with higher verbosity for greater detail
  -v, --verbose          Be more verbose. [NOSE_VERBOSE]
  --verbosity=VERBOSITY Set verbosity; --verbosity=2 is the same as -v
                        Be less verbose
  -q, --quiet            Be less verbose
  -c FILES, --config=FILES Load configuration from config file(s). May be
                        specified multiple times; in that case, all config
                        files will be loaded and combined
  -w WHERE, --where=WHERE Look for tests in this directory. May be specified
                        multiple times. The first directory passed will be
                        used as the working directory, in place of the current
                        working directory, which is the default. Others will
                        be added to the list of tests to execute. [NOSE_WHERE]
  --py3where=PY3WHERE    Look for tests in this directory under Python 3.x.
                        Functions the same as 'where', but only applies if
                        running under Python 3.x or above. Note that, if
```

Utilisation de Nose dans un script de tests.

Dans un projet Python vide, nous avons créé deux packages, **app** et **test**, suivant ainsi la structure standard des projets Python. Dans le package **app**, nous avons créé le module **Calculator** et dans le package **test**, le module **test_calc**.



Le module **Calculator** dispose d'une fonction **add** qui effectue la somme de deux nombres si tentés que ceux-ci appartiennent aux types définis dans **number_types** (soit des entiers, entiers longs, décimaux ou complexe). Si l'un ou les deux nombres ont un type différent des 4 types prévus, une exception est levée.

```

P Calculator
1 '''
2 Created on Jul 4, 2017
3
4 @author: ndeyeannandiaye
5 '''
6 class Calculator(object):
7
8     def add(self, x, y):
9         number_types = (int, long, float, complex)
10
11         if isinstance(x, number_types) and isinstance(y, number_types):
12             return x + y
13         else:
14             raise ValueError

```

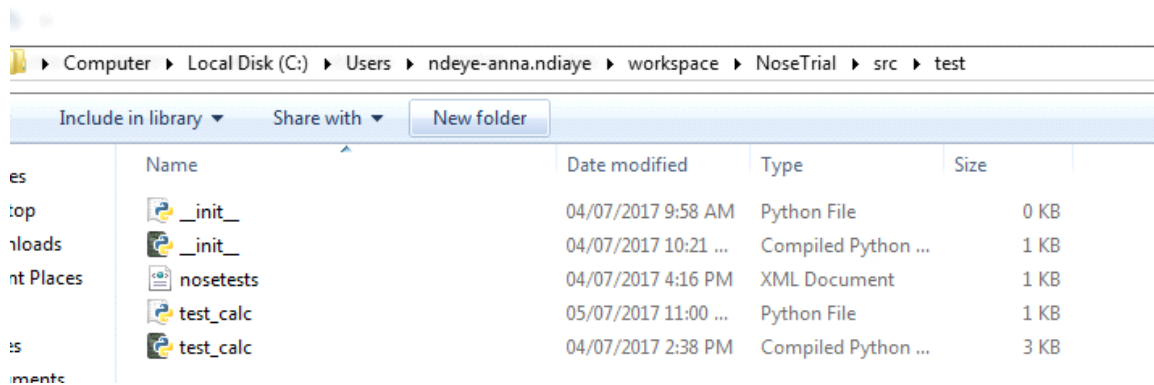
Le module test_calc définit 4 tests sur la méthode add du module Calculator en modifiant à chaque fois les données d'entrée et prévoyant les résultats erronés ou non à obtenir. Ici, nous utilisons Nose via l'instruction import et par la suite nous utilisons le plugin Xunit qui permet de générer les résultats obtenus sous forme de fichier XML analysable (parsable) par JUnit sur Jenkins. Pour utiliser le plugin Xunit de Nose, nous lui passons comme argument dans la commande nose.core.run(), l'option '--with-xunit'.

```

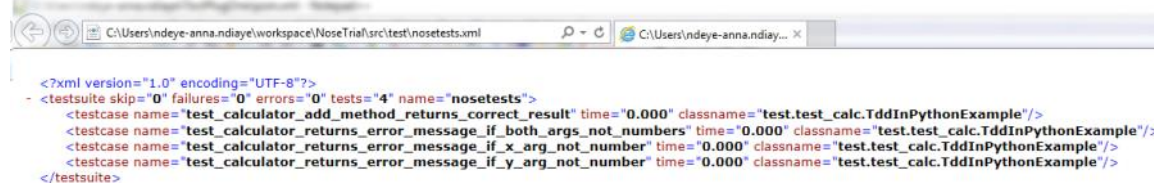
P test_calc
1 '''
2 Created on Jul 4, 2017
3
4 @author: ndeyeannandiaye
5 '''
6
7 import unittest
8 import nose
9 import sys
10
11
12 from app.Calculator import Calculator
13
14 class TddInPythonExample(unittest.TestCase):
15
16     def setUp(self):
17         self.calc = Calculator()
18
19     def test_calculator_add_method_returns_correct_result(self):
20         result = self.calc.add(2, 2)
21         self.assertEqual(4, result)
22
23     def test_calculator_returns_error_message_if_both_args_not_numbers(self):
24         self.assertRaises(ValueError, self.calc.add, 'two', 'three')
25
26     def test_calculator_returns_error_message_if_x_arg_not_number(self):
27         self.assertRaises(ValueError, self.calc.add, 'two', 3)
28
29     def test_calculator_returns_error_message_if_y_arg_not_number(self):
30         self.assertRaises(ValueError, self.calc.add, 2, 'three')
31
32     argv = [sys.argv[0], '--with-xunit']
33     nose.core.run(argv = argv)
34
35

```

Le résultat de l'exécution de ce test sera la création d'un fichier XML nommé par défaut **nosetests.xml** dans le répertoire de test du projet.



L'aperçu de ce fichier apparaît comme suit :



Plugin JUnit de Jenkins

Le plugin JUnit de Jenkins permet de publier les résultats de tests après un formatage JUnit. Il fournit un éditeur qui à partir de rapports de tests XML générés durant les builds, produit une visualisation graphique de l'historique des résultats de tests ainsi qu'une interface web permettant d'afficher les rapports de tests, de traquer les échecs, etc. Jenkins comprend le format XML des rapports de tests JUnit (également utilisé par TestNG) et peut donc lorsque ce plugin est actif, fournir des informations utiles telles que les tendances de résultats. En fait cette fonctionnalité faisait partie intégrante du noyau Jenkins jusqu'à ce qu'ils soient séparés à la version 1.577.

Configuration

Le plugin JUnit est configuré via des actions de build et d'après build. Les options d'utilisation comptent :

- ✓ Traitement de rapports de test XML: spécifier le chemin vers les fichiers de rapport XML selon une syntaxe telle que `**/build/test-reports/*.xml`. Le répertoire de base est le répertoire de travail (workspace) root.
- ✓ Conserver les sorties/erreurs standards: cette option permet de retenir les sorties ou erreurs d'une batterie de tests une fois le build achevé. (Ceci désigne seulement les messages supplémentaires affichées à la console, pas la pile de messages liées à un fail). Ces sorties sont toujours conservées lorsque les tests échouent, mais par défaut, les longues sorties liées à la réussite d'un test sont supprimées pour économiser de l'espace. Cette option permet donc de voir tous les messages de log même lorsque les tests réussissent mais attention à prendre en compte que la consommation en mémoire de Jenkins augmente même si on ne consulte jamais ces résultats de tests.
- ✓ Facteur d'amplification du pourcentage de réussite : il s'agit du facteur d'amplification du pourcentage de succès à appliquer aux échecs de test lorsqu'on comptabilise la contribution des résultats de test sur l'état général du build. Le facteur par défaut est 1.0. un facteur de 0.0 va désactiver la contribution du résultat de test sur l'état du build, et, par exemple, un facteur de 0.5 signifie que 10% d'échecs de tests va produire un état de succès global de 95%. Le facteur est conservé avec les résultats de build, donc s'il est modifié, cela n'apparaîtra que dans les prochains builds.
- ✓ Autoriser les résultats vides : cette option permet de modifier le comportement par défaut qui consiste à échouer le build si les fichiers de résultats de tests sont absents ou vides; à la place, ceci n'affecte pas l'état du build.

Exemples utilisation sur nosetests.xml

Action réalisée au moment du build pour traiter les fichiers de rapports de test en XML, dans notre cas nosetests.xml que nous avons placés dans le répertoire NoseDir.

Build

Process xUnit test result report

JUnit

JUnit Pattern

Skip if there are no test files ☐
Fail the build if test results were not updated this run ☐
Delete temporary JUnit files ☒
Stop and set the build to 'failed' status if there are errors when processing a result file ☐

Ajouter

Action réalisée après le build pour publier les résultats obtenus après analyse des rapports de tests.

Actions à la suite du build

Publish xUnit test result report

JUnit

JUnit Pattern

Skip if there are no test files ☐
Fail the build if test results were not updated this run ☐
Delete temporary JUnit files ☒
Stop and set the build to 'failed' status if there are errors when processing a result file ☒

Ajouter

Résultat de la publication des tests.

Résultats des tests



Tous les tests qui ont échoué

Nom du test	Durée	Age
JUnitXmlReporter.constructor.should default path to an empty string	6 ms	3

Tous les tests

Package	Durée	Échec	(diff)	Sauté	(diff)	Pass	(diff)	Total	(diff)
JUnitXmlReporter	6 ms	1	1	1	1	1	3	3	3
test.test_calc	0 ms	0	0	0	0	4	4	4	4

Affichage détaillé des tests présents dans le module test_calc.

Résultats des tests : TddInPythonExample



Tous les tests

Nom du test	Durée	Statut
test_calculator_add_method_returns_correct_result	0 ms	En succès
test_calculator_returns_error_message_if_both_args_not_numbers	0 ms	En succès
test_calculator_returns_error_message_if_x_arg_not_number	0 ms	En succès
test_calculator_returns_error_message_if_y_arg_not_number	0 ms	En succès

Tendance des tests (bleu pour succès, jaune pour tests ignorés, rouge pour échec)

