Programmation Python : enseigner le test automatisé à des débutant·es

Département Informatique[1]
Laboratoire CRIStAL, équipe NOCE[2]
Mirabelle Nebut[1,2]

atelier JEIA - fév 2023

Outline

Le test de programme pour débutant·es - Enjeux

Sensibiliser à l'importance du test dès l'apprentissage de la programmation :

- convaincre que tester est plus une nécessité qu'une corvée
- ancrer l'habitude d'écrire des tests unitaires avant le code

Sensibiliser à l'importance de la documentation du code.

Mais :

- ne pas dégoûter les élèves/étudiant·es en étant trop exigeant
- utiliser un outil de test adapté à des débutant·es (cet atelier)

Le test de programme côté étudiants - constat au département Informatique

- Sensibilisation en L1, mais dépend grandement de la sensibilité de l'enseignant de TP
- En L2 S3 : POO en Java, inclut du test unitaire avec JUnit
- En L3 S5 : COO en Java, inclut la notion de mocks, TDD
- En L3 : UE GL au choix, cycle de vie logiciel dont métriques et test
- en master, suivant spécialités

En pratique... étudiants pas toujours motivés pour tester leurs programmes !

Topic

Tester et documenter ses programmes

Outillage pour le test unitaire automatisé pour débutant·es

Écueils : enseignement du test en même temps que la programmation

Ressources

Qu'est-ce que "tester un programme"

C'est l'essayer en l'exécutant pour voir "si ça marche".

Pourquoi tester ?

- Pour "savoir si j'ai bon" -> pour "savoir si mon code a l'air correct"
- Pour détecter des erreurs dans les programmes
- Méthode de validation de programme la plus utilisée (y compris monde pro)
- Ne garantit jamais l'absence d'erreurs (op preuve de programme)

Que teste-t-on ?

Plusieurs niveau de test.
Pour des débutants, on fait surtout :

- du test unitaire de fonctions (cet atelier)
- du test de recette pour un projet

Test unitaire manuel

= Éxécuter des appels à la fonction et observer le résultat
Avantage : facile à faire
Inconvénients :

- on ne sait plus si on a testé
- on ne sait plus ce qu'on a testé
- difficile à reproduire
- pas de test de non régression facile

Test unitaire automatisé : écrire des tests

Écrire des cas de test = donner

- la fonction à tester
- les valeurs pour les paramètres = données de test
- le comportement attendu = valeur renvoyée / exception levée / effet de bord

Ex : fonction `somme_premiers_entiers(n)`

- pour `n = 4` la fonction doit renvoyer 10
- pour `n = 0` la fonction doit renvoyer 0

Test unitaire automatisé : éxécuter les tests

Utiliser un outil de test pour exécuter les (cas de) test = suite de tests

Avantage :

- les tests sont écrits et gardés
- les tests sont faciles à lancer et reproduire
- les tests peuvent servir de documentation

Inconvénient : il faut maîtriser un outil de test (syntaxe, retours)

Les tests dans la phase de développement

Test First = approche consistant à écrire les tests avant le code

Test Driven Development (TDD) =

- test-first
- alternance rouge/vert/réusinage
- très petits pas, comportement par comportement : on n'écrit que le code nécessaire au passage du test en cours et pas plus
- utilisation des tests comme documentation

Le TDD est très utilisé dans les méthodes dites agiles.

Pourquoi écrire les tests avant le code

Parce que les tests vérifient que le code est conforme à la spécification.
On écrit donc les tests en lisant la spécification.
Si le code est déjà écrit, on risque d'écrire les tests en regardant le code, et de valider un code faux par un test faux.

Ce qui est proposé dans les ressources d'accompagnement Eduscol pour NSI

En Première : Mise au point de programmes testés

- TDD : on garde le fait d'écrire les tests avant le code
- utilisation de assert dans le main
- ou utilisation de doctest

En Terminale : Écriture de test

- mention des outils unittest, doctest et pytest
- présentation de pytest
- (+ test boîte noire/boîte blanche et utilisation de coverage avec pytest)

Topic

Tests unitaires automatisés avec assert

`assert <expr>`
lève un `AssertionError` si `<expr>` s'évalue à False
Avantage :

- basé uniquement sur l'instruction Python `assert`
- les tests sont écrits dans le module (pour débutants)

Inconvénient :

- le premier `AssertionError` interrompt l'exécution de la suite de test
- n'incite pas à utiliser les tests comme documentation
- pas de rapport de test (nb tests exécutés / succès / échec / erreur)

Test unitaires automatisés avec doctest

doctest = tester la doc = documentation exécutable (tjs à jour !).
Écriture des tests :

- dans la docstring
- avec la syntaxe de l'interpréteur de Python
- litterate testing : mélange de tests et de description en langue naturelle

Test unitaires automatisés avec doctest : avantages

- outil fourni avec Python
- syntaxe naturelle
- met l'accent sur la docstring, dc sur la documentation des fonctions
- test = documentation

Test unitaires automatisés avec doctest : inconvénients

- le lancement des tests dans le main est un peu cryptique (pas facile à expliquer/retenir)
- rapport de test textuel assez austère (noir et blanc + repérage pas tjs facile) repérer qd plusieurs tests ne passent pas
- la levée d'exception n'est pas facile à décrire

Test unitaires automatisés avec doctest : inconvénients vraiment gènants

Principe difficile à comprendre : doctest récupère le contenu de la sortie standard et le compare caractère par caractère avec la valeur attendue
On peut donc avoir une perte d'intuition pour :

- le retour d'une chaîne de caractère

```
Expected:  "coucou" Got 'coucou'
```

- un caractère d'espacement qui traîne

```
Expected:  2 Got:  2
```

- la mise en forme d'une liste

```
Expected:  [1,2,3] Got:  [1, 2, 3]
```

Test unitaires automatisés avec doctest : petits arrangements possibles

```
>>> dupliquer("cou") == "coucou"
True
>>> liste_n_premiers(3) == [1,2,3]
True
```

mais risque de confusion fonction / prédicat pour l'élève.
Attention à la tentation de créer les tests par copier-coller d'appels dans l'interpréteur (ne plus écrire les tests avant le code)

Tests unitaires automatisés avec pytest

bibliothèque de test pour Python, la plus abordable pour des débutants car sans syntaxe objet

- écriture d'une fonction de test par cas de test
- lancement des tests par l'outil pytest dans un terminal : détecte automatiquement toutes les fonctions `test_*` ou `*_test`
- rapport textuel en rouge et vert, détails quand le test échoue (rouge)

Tests unitaires automatisés avec pytest : avantages et inconvénients

Avantages :

- utilisation du mot-clé assert facile
- du vert (OK) et du rouge (KO) motivants
- possibilité d'écrire les tests dans un fichier à part
- vraie bibliothèque de test pour les grands !

Inconvénients :

- une commande en plus à connaître
- syntaxe de la levée d'exception à apprendre
- rapport textuel (pas toujours facile de s'y repérer)

+/- :

- doc par les tests, vient en plus de la docstring
- effort à faire pour trouver un nom de test parlant

Tests unitaires automatisés avec Thonny et L1test

Environnement de Développement Intégré (IDE) Thonny

- cache le lancement de python
- cache l'import d'un module dans la console (+/-)
- débogueur très bien fait (autre bonne habitude à prendre)
- affichage des valeurs des variables

L1test : outil maison de test intégré à Thonny (plugin)

- sur le principe de doctest pour l'écriture dans les docstrings
- syntaxe similaire à doctest pour les retours de fonction
- sémantique différente : comparaison des valeurs des expressions

Différence fondamentale entre L1test et doctest

Plus de comparaison de sortie standard.
Comparaison valeur attendue / valeur obtenue par == comme dans pytest

```
$$$ somme_n_premiers(3)
1 + 2 + 3
$$$ dupliquer("cou")
"cou" + "cou"
```

Pas possible de tester une procédure qui réalise un affichage

```
>>> affiche_invite("Mi") # doctest
Bonjour Mi
$$$ affiche_invite("Mi") # l1test
None
```

L1test : avantages et inconvénients

Avantages :

- tests comme doc dans la docstring
- sémantique intuitive
- affichage graphique du verdict des tests
- vert et rouge motivant
- facilités pour générer automatiquement un squelette de docstring

Inconvénients :

- première mise en service en sept 2022, bugs
- qualité de la maintenance non garantie !
- risque de confusion ultérieure avec doctest pour les élèves

Topic

Tester et documenter ses programmes

Outillage pour le test unitaire automatisé pour débutant·es

Écueils : enseignement du test en même temps que la programmation

Ressources

Côté enseignant : jusqu'où aller ?

- Donner les tests —-> Les faire écrire
- Attention à la surcharge !
- Faire prendre conscience de l'intérêt d'écrire des tests
- Prendre la bonne habitude d'écrire les tests avant le code
- Pas dégoûter tout le monde !

Côté étudiants

- penser que si les tests passent, le code est correct
- sauter la phase de réflexion préalable à l'écriture du code...
- ... et compter sur la phase de test pour magiquement corriger un code vide de sens.
- tester après avoir écrit le code

Topic

Tester et documenter ses programmes

Outillage pour le test unitaire automatisé pour débutant·es

Écueils : enseignement du test en même temps que la programmation

Ressources

Liens utiles et contact

doctest
https://docs.python.org/3/library/doctest.html
pytest https://pytest.org/
Thonny https://thonny.org/ Univ Tartu, Estonie
l1test (me contacter pour information, installation et docs)

- https://pypi.org/project/L1test/
- mirabelle.nebut<arobas>univ-lille.fr

Ressources eduscol pour le test en NSI :

- 1ère, mise au point de programmes testés https://eduscol.education.fr/document/30058/download
- Tale, écriture de tests https://eduscol.education.fr/document/7298/download