Utiliser les tests pour le travail en autonomie

Formation NSI

Bruno Mermet

Université du Havre & Laboratoire GREYC

Mai 2021

Plan de la formation

1. Cas de test visibles
2. Cas de test invisibles
3. Fournir une implantation pour éviter les blocages
4. Cas de test pour évaluer
5. Brouillage ou compilation ?

1. Cas de test visibles

1.1 Donner tous les cas de tests semblant utiles

- On donne d'emblée tous les cas de test semblant pertinents.
- Le fichier de jeu de tests contient aussi bien des cas de test pour des valeurs correctes des paramètres que pour des valeurs incorrectes.
- Il faut bien sûr prévoir des cas aux limites, mais également plusieurs cas pour les valeurs *standards*.

1.1.1. Cas de tests utiles : Exemple

Implanter une fonction calculant le i-ème terme de la suite de Fibonacci

- Des cas de test pour les cas **corrects**
  - Un cas de test pour 0
  - Un cas de test pour 1
  - Un cas de test pour un naturel petit
  - Un cas de test pour un naturel "grand" (vérification éventuelle du temps pour contrôler si l'implantation est *efficace*)
- Des cas de test pour les cas **incorrects**
  - Un cas de test pour une valeur négative
  - Un cas de test pour un "float" non entier
  - Un cas de test pour autre chose qu'une chaîne de caractère

1.1.2 Cas de test utiles : Jeu de tests (1)

```
import pytest
from fibonacci import *
from time import time
"""
Rappel : la suite de fibonacci est définie  ainsi :
  u0 = 1
  u1 = 1
  un = u(n-1) + u(n-2) pour n > 1
Les premiers termes sont donc :
  1 1 2 3 5 8 13 21 34 ...
"""
def test_0():
    assert fibo(0) == 1
def test_1():
    assert fibo(1) == 1
def test_8():
    assert fibo(8) == 34
def test_35():
    t0 = time()
    assert fibo(35) == 14930352
    t1 = time()
    delai = t1 - t0
    assert delai < 1, "Implantation inefficace"
```

1.1.2 Cas de test utiles : Jeu de tests (2)

```
def test_negatif():
    with pytest.raises(AssertionError):
        fibo(-5)

def test_non_entier():
    with pytest.raises(AssertionError):
        fibo(3.14)

def test_non_nombre():
    with pytest.raises(AssertionError):
        fibo("quatre")
```

1.1.3 Implantation satisfaisant les tests

```
def fibo(n):
    assert type(n) == int and n >= 0,\
           "n doit être un natuel"
    if n < 2:
        return 1
    elif n == 35:
        return 14930352
    return fibo(n-1) + fibo(n-2)
```

1.2 Utiliser le *Test Driven Development*

- On conçoit les cas de test comme si on développait en TDD
- On demande aux élèves de concevoir le code pour répondre aux cas de test un par un, dans l'ordre dans lequel ils sont définis dans le fichier de test
- Dans le fichier de test, on spécifie qu'un test doit être sauté si le précédent n'a pas réussi (pytest exécute les cas de test dans l'ordre dans lequel ils sont définis dans le fichier)

1.2.1 Utilisation du TDD : exemple

- On reprend le cas de Fibonnacci.
- Les élèves doivent d'abord donner le résultat pour 0, puis pour 1, puis pour 2, puis pour 8.

On va définir le module `marqueur` utilitaire suivant :

```
class Marqueur:
    def __init__(self):
        self.marqueur = 0

    def suivant(self):
        self.marqueur += 1

    def get(self):
        return self.marqueur

marqueur = Marqueur()
```

1.2.2 Utilisation du TDD : Fichier de test proposé

```
import pytest
from fibonacci import *
from marqueur import marqueur
def test_0():
    if fibo(0) == 1:
        marqueur.suivant()
    else:
        assert False, "fibo(0) incorrect"
@pytest.mark.skipif("marqueur.get() < 1", reason="Tests précédents pas encore corrects")
def test_1():
    if fibo(1) == 1:
        marqueur.suivant()
    else:
        assert False, "fibo(1) incorrect"
@pytest.mark.skipif("marqueur.get() < 2", reason="Tests précédents pas encore corrects")
def test_2():
    if fibo(2) == 2:
        marqueur.suivant()
    else:
        assert False, "fibo(2) incorrect"
@pytest.mark.skipif("marqueur.get() < 3", reason="Tests précédents pas encore corrects")
def test_3():
    if fibo(3) == 3:
        marqueur.suivant()
    else:
        assert False, "fibo(3) incorrect"
@pytest.mark.skipif("marqueur.get() < 4", reason="Tests précédents pas encore corrects")
def test_8():
    if fibo(8) == 34:
        marqueur.suivant()
    else:
        assert False, "fibo(8) incorrect"
```

1.2.3 Utilisation du TDD : Exemle de résultat d'exécution

```
============== test session starts ================
collected 5 items

test_tdd.py ...Fs                                 [100%]

=================================================
______________________ test_3 ______________________

    @pytest.mark.skipif("marqueur.get() < 3",
    reason="Tests précédents pas encore corrects")
    def test_3():
        if fibo(3) == 3:
            marqueur.suivant()
        else:
>           assert False, "fibo(3) incorrect"
E           AssertionError: fibo(3) incorrect
E           assert False

test_tdd.py:40: AssertionError
============== short test summary info ==============
FAILED test_tdd.py::test_3 - AssertionError: fibo(3) incorrect
======= 1 failed, 3 passed, 1 skipped in 0.06s =======
```

2. Cas de tests invisibles

2.1 Pourquoi écrire des cas de test invisibles

Dans le cas de Fibonacci, si on donne des cas de test pour 0, 1, 2 et 8, le code suivant passerait :

```
def fibo(n):
  if n < 2:
    return 1
  elif n == 2:
    return 2
  else:
    return 8
```

Une solution est de générer des cas de test aléatoires, ce qui impose de rendre disponible une implantation de l'algorithme. C'est possible en donnant une version compilée. Cependant, ce n'est pas recommandé car des cas de test doivent être **reproductibles**.

2.2 Rendre des cas de test invisibles

- Pour rendre le code source des tests invisibles, il est possible de compiler les cas de tests. Attention cependant, le python compilé sur une architecture ne marchera pas sur un autre
- Pour compiler un jeu de tests *test_fichier.py*, procéder ainsi :
  - compiler le fichier :

```
python -m py_compile test_fichier.py
```

  - déplacer et renommer le code compilé dans le répertoire courant:

```
mv __pycache__/test_fichier.cpython-38.pyc test_fichier.pyc
```

- Problème : pytest ne fonctionne pas avec des cas de tests compilés

2.3 Écrire un micro pytest pour fichiers compilés (1)

```
import importlib
import sys
import pathlib
class Testeur:
    def __init__(self, fichier_test):
        self.nom_jeu = fichier_test
        module = importlib.import_module(fichier_test)
        noms_attributs = dir(module)
        self.fonctions = []
        for nom in noms_attributs:
            if nom.startswith("test_"):
                self.fonctions.append(getattr(module, nom))
        self.erreurs = dict()
    def ajouter_erreur(self, message):
        if message in self.erreurs:
            self.erreurs[message] += 1
        else:
            self.erreurs[message] = 1
    def executer(self):
        for f in self.fonctions:
            try:
                f()
                print(".", end="")
            except AssertionError as ex:
                print("F", end="")
                self.ajouter_erreur(str(ex))
        print()
    def compte_rendu(self):
        print("Jeu de test",self.nom_jeu,":")
        if len(self.erreurs) == 0:
            print("  Tous les tests ont réussi")
        else:
            print("  Liste des erreurs :")
            for erreur in self.erreurs:
                print("    ", erreur, ":", self.erreurs[erreur], "fois")
```

2.3 Écrire un micro pytest pour fichiers compilés (2)

```
if __name__ == '__main__':
    fichiers_a_tester = sys.argv[:]
    if len(fichiers_a_tester) > 1:
        fichiers_a_tester.pop(0)
    else:
        fichiers_a_tester = list(map(lambda fic: str(fic),\
                                     (pathlib.Path(".")
                                             .glob("test_*.py"))))
        fichiers_a_tester += list(map(lambda fic: str(fic),\
                                      (pathlib.Path(".")
                                              .glob("test_*.pyc"))))
    for fichier in fichiers_a_tester:
        if fichier.endswith(".py"):
            nom_module = fichier[:-3]
        elif fichier.endswith(".pyc"):
            nom_module = fichier[:-4]
        else:
            nom_module = fichier
        testeur = Testeur(nom_module)
        testeur.executer()
        testeur.compte_rendu()
```

2.4 Micro pytest : utilisation

- sans arguement : récupère tous les fichiers `test_*.py[c]` du répertoire courant et exécute les tests
- avec argument : prend des noms de modules, de fichier en `.py` ou en `.pyc`et exécute les tests

3. Proposer une correction pour éviter les blocages

3.1 Présentation

- Pour éviter que les élèves ne restent bloqués parce qu'ils n'arrivent pas à implanter une fonction requise pour la suite on peut leur donner une correction compilée.
- Cette correction doit pouvoir être utilisée dans les tests pour vérifier la validité du code développé par l'élève, mais le fait de l'utiliser doit être mentionné.

3.2 Mise en oeuvre

- Compiler la correction et rendre disponible cette correction sous le nom `correction.pyc`
- Fournir le module `aide` décrit ci-après au format compilé
- Importer ce module `aide` dans le module *micro pytest* et rajouter `aide.trace.afficher()` à la fin du `if` principal
- Expliquer aux élèves comment passer par la correction quand ils sèchent

3.3 Le module aide

3.3.1 Présentation

- La méthode `appelle` permet de faire appelle à une fonction de la correction
- La classe `Trace` permet de garder la trace des fonctions d'aide qui ont été appelées

3.3.2 Implantation

```
from correction import *

class Trace:
    def __init__(self):
        self.utilisation = set()
    def ajouter_utilisation(self, fonction):
        self.utilisation.add(fonction)
    def afficher(self):
        if len(self.utilisation) != 0:
            print("Liste des fonctions de la correction utilisées :")
            for fonc in self.utilisation:
                print("  ", fonc)
        else:
            print("Correction pas utilisée")

trace = Trace()

def appelle(fonction, parametres):
    resultat = eval("corr_" + fonction + "(*" + str(parametres) + ")")
    trace.ajouter_utilisation(fonction)
    return resultat
```

3.4 Écriture de la correction

Dans la correction, importer le module de l'élève (donc imposer le nom), et privilégier l'utilisation de la fonction de l'élève lorsqu'elle existe

```
from calculs import *
def corr_somme(a, b):
    return a + b
def corr_produit(a, b):
    return a * b
def corr_calc():
    a = call("somme", [1, 2])
    b = call("somme", [3, 4])
    c = call("produit", [a, b])
    return c
def call(fonction, parametres):
    try:
        res = eval(fonction + "(*" + str(parametres) + ")")
    except NameError:
        res = eval("corr_" + fonction + "(*" + str(parametres) + ")")
    return res
```

3.5 Utilisation du module `aide`

Voilà à quoi pourrait ressembler le code d'un étudiant ne sachant pas écrire la fonction `somme` :

```
import aide

def produit(a, b):
    return a * b

def calc():
    a = aide.appelle("somme", [1, 2])
    b = aide.appelle("somme", [3, 4])
    c = produit(a,b)
    return c
```

4. Cas de test pour évaluer

4.1 Introduction

En général, en informatique, un TP, comme un projet, sont notamment évalués sur les fonctionnalités implantées et leur correction. Utiliser des jeux de test permet de rendre ce travail plus rapide, moins fastidueux, et plus égalitaire.

4.2 Principe général

- Donner aux élèves quelques cas de test simples pour qu'ils puissent par exemle, valider le format de leurs résultats (trop bête d'avoir zéro car toutes les textes se terminent par un espace de trop)
- Donner aux élèves une correction compilée pour qu'ils ne restent pas bloqués
- Imposer les noms et profils des fonctions/méthodes (au moins celles de haut niveau)

4.3 Principe détaillé

- Pour chaque fonction, définir :
  - les cas de tests essentiels ;
  - les cas de tests secondaires ;
  - les cas de tests “hors domaine”
- Pour chaque fonction, définir :
  - le coefficient pour l’ensemble des cas de tests essentiels ;
  - le coefficient pour l’ensemble des cas de tests secondaires ;
  - le coefficient pour l’ensemble des cas de tests “hors domaine”
- Définir le poids de chaque fonction (un poids identique quelle que soit la difficulté est un bon point de départ)

5. Brouillage ou compilation ?

- Le brouillage (*obfuscation* en anglais) consiste à rendre un code source incompréhensible à un être humain, tout en restant compréhensible à l'interpréteur.
- La compilation de programme python ne générant pas un bytecode universel, elle n'est pas toujours adaptée
- Il existe des outils permettant de brouiller un code python, mais ils ne marchent pas dans tous les cas. Dans le cas présent, il est toujours possible de brouiller au moins la correction (problèmes avec les autres fichiers), et de laisser les autres visibles

6. Comment brouiller ?

- Installer *pyobfuscate* :

```
sudo pip install python-obfuscator
```

- Brouillage, version 1 :

```
pyobfuscate -i correction.py > correction_obfusquee.py
```

- Brouillage, version 2 :

```
pyobfuscate -i correction.py --one-liner True > correction_obfusquee.py
```