Numérique et science informatique

Lycée hoche

année scolaire 2021-2022

Contents

0.1 Introduction

Les bugs entraînent chaque année des millions d'euros de perte pour les entreprises et les États mais parfois également à l'origine de morts civiles ou militaires (voir http://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html par exemple). L'une des solutions les plus utilisées pour éviter de livrer un programme buggué passe par la réalisation de tests. Pendant quelques décennies, les tests de programme étaient écrits (quand ils étaient écrits!) de manière presque aléatoires, et n'apportaient ainsi pas une grande garantie sur les programmes. Mais tout cela a bien changé. Le développement d'outils permettant d'exécuter les tests automatiquement a notamment rendu les tests bien plus utiles. En effet, pour que les tests soient exécutés aussi souvent que possible, il faut que cela soit faisable automatiquement. En Python, il existe plusieurs outils permettant d'exécuter des tests automatiquement. On peut notamment citer unittest, pytest et doctest

0.2 Vocabulaire

On peut classer les tests selon différents critères :
• le niveau des tests (tests unitaires, tests d'intégration, tests de recette) ;
• le processus de conception des tests (tests boîte blanche, tests boîte noire) ;
• le sujet du test (tests fonctionnels, tests de montée en charge, tests d'utilisabilité,etc

Voici quelques définitions essentielles concernant les tests.

- On appelle **cas de test** un triplet (descriptif, données d'entrée, résultat attendu) précisant, pour des données précises, le résultat attendu de la partie du programme que l'on veut tester.
- On appelle **jeu de tests** un ensemble de cas de test destinés à valider une partie précise du fonctionnement d'un programme.Le terme test peut se référer suivant les circonstances à un cas de test, à un jeu de tests, ou au processus de test en général.
- Un **test unitaire** est un test concernant une petite unité d'un programme ; typiquement, une fonction.

- Un test « **boîte noire**» est un test qui est conçu à partir des données d'entrées potentielles, indépendamment du code écrit. Un test « boîte noire » peut donc être écrit avant le code, ou s'il est écrit après, il doit être écrit par quelqu'un qui ne connaît pas le code.

  Exemple : une fonction doit générer l'en-tête d'une lettre. Pour cela, elle prend en paramètre un objet représentant le destinataire de la lettre (prénom, nom, sexe). Sans connaître le code de la fonction, on peut déjà envisager 2 cas de test, un pour un homme et un pour une femme, et vérifier que dans le premier cas l'en-tête commence par « Cher » alors qu'il commence par « Chère » dans le second cas.

- Un test « boîte blanche » est un test qui est conçu à partir du programme. Le but de ce type de test est de tester les différents cas prévus par le programme.

Exemple:

```
def f(n):
    if n%2==0:
        return "pair"
    else:
        return "impair"
```

On envisage alors un cas de test avec n pair pour tester la branche « alors » du if, et un cas de test avec n impair pour tester la branche « sinon ».
Il est important de comprendre qu'à moins d'être exhaustif, ce qui n'est que très rarement possible, un test ne garantit jamais la correction d'un programme. Cependant, en écrivant pour chaque fonction un jeu de tests "boîte noire" pertinent, on augmente la confiance que l'on peut avoir dans le programme et on limite le risque de bug. Il ne faut pas oublier que si un cas de test est correct, une réussite du test n'est pas significative, alors un échec établit de manière certaine la présence d'un bug.

0.2.1 Concevoir des tests "Boîte noire"

Pour concevoir un jeu de tests « boîte noire » pertinent, on essaie en général de concevoir plusieurs cas de test, représentatifs de chacune des classes d'équivalence des données d'entrée, en rajoutant des tests spécifiques aux limites des classes d'équivalence. En général, on considère non seulement des données valides, mais également des données invalides.
Inconvénient des tests « boîte noire » : on risque de ne pas passer par des parties du programme traitant des cas particuliers n'apparaissant pas au premier abord à partir des données d'entrée.
On appelle ce type de couverture : **couverture des conditions multiples**.Inconvénient des tests « boîte blanche » : la réflexion est biaisée par le code écrit, et on risque de ne pas générer de cas de test pour les cas non traités par le programme.

0.3 Ecrire des tests en python

Il existe principalement 3 outils pour écrire et exécuter des tests en Python :

- unittest est l'outil de base, intégré à Python ;
- doctest est un outil externe qui permet d'exécuter des tests inclus dans la documentation des fonctions. Les tests servent alors également de documentation de la fonction ;
- pytest est un outil externe qui permet d'exécuter des tests écrits dans des fichiers dédiés. C'est l'outil le plus utilisé, car le plus complet.

0.4 Utilisation du module doctest-Etude sur un exemple

(L'activité qui suit est inspirée de:
https://www.fil.univ-lille1.fr/ L1S2API/CoursTP/tp_doctest.html)

Dans toute la suite , les exemples proposés seront exécutés dans le shell.

Tapez le programme suivant et sauvegarder dans un fichier nommé *exemple_ doctest.py*

```
def fact(n):
    """
    paramètre n : (int) un entier
    valeur renvoyée : (int) la factorielle de n.

    précondition : n>=0

    Exemples :

    >>> fact(3)
    6
    >>> fact(5)
    120
    """
    res = 1
    for i in range(2, n + 1):
        res = res * i
    return res
```

Cette documentation peut être exploitée avec la fonction **help** :
A faire:

```
>>>help(fact)
```

Les exemples donnés dans une chaîne de documentation peuvent être testés à l'aide d'un module de Python nommé doctest:

```
>>> import doctest

>>> doctest.testmod()
```

Qu'obtient-on?
La fonction testmod du module doctest est allée chercher dans les docstring des fonctions du module actuellement chargé, c'est-à-dire exemple_doctest, tous les exemples (reconnaissables à la présence des triples chevrons >>>), et a vérifié que la fonction documentée satisfait bien ces exemples. Dans le cas présent, une seule fonction dont la documentation contient deux exemples (attempted=2) a été testée, et il n'y a eu aucun échec (failed=0).

0.4.1 Cas de désaccord entre un exemple et la fonction

Exemple

Modifiez le deuxième exemple, en mettant 121 à la place de 120 dans le second exemple. Chargez le fichier dans l'interpréteur (touche F5) et retapez les deux lignes

```
>>> import doctest
>>> doctest.testmod()
```

Qu'observe-t-on?

- Tout d'abord que les tests ont échoué et qu'il y a eu 1 échec (cf dernière ligne) et que cet échec est dû à la fonction fact (cf avant dernière ligne).

- Ensuite que le test incriminé est celui concernant fact(5) pour lequel le test a obtenu (Got) 120 en exécutant la fonction fact, alors qu'il attendait (Expected) 121 selon l'exemple donné par la documentation.

Lorsqu'il y a de tels échecs, cela invite le programmeur à vérifier son programme, ... ou bien les exemples de sa documentation, comme c'est le cas ici.

0.4.2 Rendre automatique les tests

Il est très facile de rendre automatique les tests et ainsi de ne plus avoir à faire appel explicitement (et manuellement) à la fonction testmod.

Il suffit pour cela d'inclure en fin de fichier les trois lignes :

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

Ajoutez ces trois lignes à la fin du fichier exemple_doctest.py et exécutez-le !

Faites le dans le cas d'un test erroné, et dans le cas sans erreur.
Que remarquez-vous dans le cas sans erreur ?

0.4.3 Rendre les doctests bavards même en cas de succès

Un paramètre optionnel de la fonction testmod permet d'obtenir plus d'informations sur les tests effectués mêmes en cas de succès.
Il suffit pour cela de rajouter le paramètre "verbose=True":

doctest.testmod(verbose = True)

Exemple

Faites-le ! Et observez ce que vous obtenez

- Avec des exemples erronés
- Sans exemple erroné.

0.4.4 Les sorties complexes

Tester les exemples des docstring avec le module doctest peut être source de déboires et de pièges. Vous allez découvrir certains d'entre eux et les remèdes qu'on peut y apporter.

les listes

Supposez que vous vouliez donner un exemple qui produit la liste des factorielles des entiers de 0 à 5. Vous avez donc complété votre documentation en ajoutant

```
"""
paramètre n : (int) un entier
valeur renvoyée : (int) la factorielle de n.

CU : n >= 0

Exemples :
```

```
>>> fact(3)
6
>>> fact(5)
120

La liste des factorielles des entiers de
0 à 5
>>> [fact(n) for n in range(6)]
[1,1,2,6,24,120]
"""
```

Attention:

La ligne blanche entre le deuxième exemple et la phrase qui suit est absolument nécessaire. En son absence, la phrase sera comprise comme faisant partie de la sortie produite par le deuxième exemple, et le test échouera donc.

Il est clair que ce nouvel exemple est tout à fait correct. Pourtant, si vous procédez au test vous constaterez que sur les trois exemples testés, l'un a abouti à un échec : le troisième.
Quel est le problème ? Cela vient du fait que la fonction testmod effectue une comparaison litérale entre la réponse fournie par la documentation (Expected) et celle fournie par l'interpréteur (Got).
Examinez attentivement ces deux points (Expected et Got) dans la réponse du test que vous venez d'effectuer.

Exemple

Tester le code précédent.

Avez-vous compris ? Le problème, ce sont les espaces que l'interpréteur place après chaque virgule dans l'énumération des éléments de la liste. Dans la documentation, ils n'y sont pas.
Comment corriger ce point ? C'est simple, il faut mettre des espaces entre les éléments d'une liste.
Mais ce n'est pas si simple. On peut facilement mettre plusieurs espaces, comme dans l'exemple (à tester) ci-dessous :

```
"""
paramètre n : (int) un entier
valeur renvoyée : (int) la factorielle de n.

CU : n >= 0

Exemples :

>>> fact(3)
6
>>> fact(5)
120

La liste des factorielles des entiers de
0 à 5
>>> [fact(n) for n in range(6)]
[1,  1, 2,   6,  24,   120]
"""
```

L'excès d'espaces provoque des erreurs. Si on ajoute, sous forme d'un commentaire la directive # doctest: +NORMALIZE_WHITESPACE, alors le test réussit, à condition néanmoins d'avoir mis au moins une espace après chaque virgule.

```
"""
paramètre n : (int) un entier
valeur renvoyée : (int) la factorielle de n.

CU : n >= 0

Exemples :

>>> fact (3)
6
>>> fact (5)
120

La liste des factorielles des entiers de
0 à 5
>>> [fact (n) for n in range (6)]
... # doctest: +NORMALIZE_WHITESPACE
[1,  1, 2,   6,  24,   120]
"""
```

Attention:

Les trois petits points sous les trois chevrons sont indispensables.

Avec la directice supplémentaire +ELLIPSIS, on peut même se dispenser d'énumérer explicitement tous les éléments de la liste .Testez avec cette modification comme ci-dessous:

```
"""
paramètre n : (int) un entier
valeur renvoyée : (int) la factorielle de n.

CU : n >= 0

Exemples :

>>> fact(3)
6
>>> fact(5)
120

La liste des factorielles des entiers de
0 à 5
>>> [fact(n) for n in range(6)]
... # doctest: +NORMALIZE_WHITESPACE, +ELLIPSIS
[1,  ...,   24,   120]
"""
```

0.4.5 Avec des sorties aléatoires

Comment tester des fonctions qui produisent des valeurs aléatoires ?
Dans l'absolu, il est impossible de placer un exemple dans la docstring donnant le résultat d'un appel à de telles fonctions puisque les valeurs qu'elles renvoient sont imprévisibles.
Par exemple si on veut tester qu'une fonction simulant un dé à six faces ne produit que des nombres compris entre 1 et 6, comment faire ?
On peut si on le souhaite vérifier que cette fonction ne renvoie que des nombres compris entre 1 et 6.

```
from random import randrange

    def de():
    """
    paramètre : aucun
    valeur renvoyée : (int) un nombre choisi au hasard compris entre
    1 et 6.

    CU : aucune

    Exemple :

    >>> 1 <= de() <= 6
    True
    """
    return randrange(1,7)
```

A faire

Concevez un test qui vérifie 100 fois qu'aucun nombre produit par la fonction n'est en dehors de l'intervalle [1,6].