NSI programmation

Tester ses programmes

qkzk

Tester ses programmes

De toute évidence, le code qu'on écrit n'a aucune assurance de fonctionner si on ne le teste pas...

Pourquoi doit-on tester ?

Imaginons la situation courante du développeur :

- équipe de 100 personnes,
- projet comportant déjà 100.000 lignes dans 1000 fichiers,
- il travaille environ 18 mois sur ce projet avant de passer à autre chose.

Comment espérer comprendre l'intégralité du programme ?

Solution

- On crée deux environnements :
  - **Production** : le programme qui est utilisé par les clients
  - **Développement** : le code sur lequel on travaille
- Objectif : ne déployer en production que le code le plus sûr possible

Démarche

Pour chaque modification du code en **développement** :

- Automatiser de nombreux tests
- si la modification passe les tests, un superviseur le relit
- S'il est validé par le superviseur, alors il passe en **production**

Mais peut-on TOUT tester ?

Non, malheureusement.

Infinité de tests

Imaginons une simple fonction qui encode les mots de passe :

```
password -> liJoijoiuOIuIUoiyTYiughuiguigiIOtgvycRFTgvuik
```

Comment s'assurer que deux mots de passes différents sont encodés différemment ?

Il faudrait une infinité de tests. C'est impossible.

Alors comment faire ?

Il faut essayer de tester tous les cas intéressants, couvrant toutes les situations possibles. Certains développeurs sont spécialisés dans ce domaine. Ils cherchent les cas limites susceptibles de provoquer des erreurs.

C'est le seul moyen de les éviter sur le produit.

Les tests en pratique

Python (et tous les langages modernes) propose de nombreuses méthodes

Écrire les tests soi même

Le moyen le plus simple consiste à écrire un jeu de test après `if __name__="__main__":`

```
def ma_fonction(n):
  ...

if __name__ == '__main__':
  print(ma_fonction(5))
```

C'est généralement ce qu'on fait quand on développe. Ces tests doivent couvrir tous les cas possibles et être compréhensibles.

Selon les contextes (devoir, projet, développement en cours...) on peut les laisser ou les effacer.

Il est préférable de les remplacer par de vrais tests...

Assert

Python intègre un mot clef `assert` qui va lever une exception `AssertionError` si la condition qui suit est fausse:

```
>>> assert 1 == 1 # ne fait rien

>>> assert 1 == 2 # plante le programme
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

C'est le moyen le plus efficace et rapide de tester un programme ou une fonction.

Il ne faut pas intégrer les assertions à la fonction elle même. Il est préférable de les intégrer à des fonctions de tests indépendantes du programme.

Un exemple

Reprenons notre fonction Fibonacci

```
def fibonacci(n):
  '''
  Liste des termes de la suite de Fibonacci
    de l'indice 0 à l'indice n inclus

  @param n: (int) l'indice maximal voulu
  @return: (list) la liste des termes
  '''
  if type(n) != int or n < 0:
    return None
  x = 1
  y = 1
  suite_fibonacci = [x]
  indice = 0
  while indice < n:
    x, y = y, x + y
    suite_fibonacci.append(x)
    indice += 1
  return suite_fibonacci
```

On peut tester plusieurs choses :

- La taille de la liste : n + 1
- Différents résultats : 0, 1, 5 etc.
- Les éléments de la liste sont des entiers
- La propriété de Fibonacci : $u_n + u_{n+1} = u_{n+2}$
- La sortie dans les cas impossibles : paramètre négatif, paramètre non entier

```
def tester_fibonacci():
    '''
    Teste certaines propriétés de la fonction Fibonacci

    return: None
    CU: lève une exception AssertionError si la fonction est mal programmée
    '''
    fib_10 = fibonacci(10)

    # longueur de la liste
    assert len(fib_10) == 11

    # différents résultats
    assert fibonacci(0) == [1]
    assert fibonacci(1) == [1, 1]
    assert fibonacci(5) == [1, 1, 2, 3, 5, 8]

    # ses éléments sont entiers
    for terme in fib_10:
        assert type(terme) == int

    # La propriété de récurrence
    assert fib_10[-3] + fib_10[-2] == fib_10[-1]

    # Valeur de retour dans les cas impossibles
    assert fibonacci(-1) == None
    assert fibonacci('a') == None
    assert fibonacci(3.14) == None
```

Doctest

Python permet grâce au module `doctest` d'intégrer les tests à la documentation.

Il est parfois délicat de tester certaines fonctions, en particulier les affichages.

Pour les fonctions qui réalisent des calculs cela est pratique.

Un exemple :

```
def multiply(a, b):
    """
    Calcule produit de a et b
    @param a: (number, str, list) premier facteur
    @param b: (number, str, list) second facteur
    @return: (number, str, list) le produit

    >>> multiply(4, 3)
    12
    >>> multiply('a', 3)
    'aaa'
```

```
    """
    return a * b

if __name__ == '__main__':
  import doctest
  doctest.testmod() # s'il ne se passe rien, les test sont justes
```

Quand on exécute le programme, il ne se passe rien.

Un exemple qui échoue :

```
def Fonction_mal_testee():
    '''
    Simple fonction qui echoue
    >>> Fonction_mal_testee()
    3
    '''
    return 2


if __name__ == "__main__":
    import doctest
    doctest.testmod()  # s'il ne se passe rien, les tests sont justes.
```

Voici la sortie d'un exemple qui échoue

```
>>> python3 2_tester_doctest.py
**********************************************************************
File "/home/quentin/realiser_des_tests/2_tester_doctest.py", line 5,
in __main__.Fonction_mal_testee
Failed example:
    Fonction_mal_testee()
Expected:
    3
Got:
    2

**********************************************************************
1 items had failures:
   1 of   1 in __main__.Fonction_mal_testee
***Test Failed*** 1 failures.
```

Unitest, tesmod

Il existe une librairie dédiée aux tests : `Unitest` et qui permet de tester toutes les propriétés possibles d'un objet.

Elle est un peu vaste et trop complexe pour nos objectifs aussi nous ne l'utiliserons pas.

Voici sa documentation et un guide détaillé.