Écrire des tests pertinents

Formation NSI

Bruno Mermet

Université du Havre & Laboratoire GREYC

Mai 2021

Plan de la formation

- Pourquoi écrire des tests ?
- Différents types de tests
- Un peu de vocabulaire
- Comment écrire des tests
- Les différents moyens d'écrire des tests en Python
- Noire ou blanche ?
- Écrire des tests “boîte noire”
- Écrire des tests “boîte blanche”
- Couverture de code
- Le *Test Driven Development*

Pourquoi écrire des tests ? (1)

- Les bugs peuvent avoir de lourdes conséquences
  - Quelques exemples : http://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html
  - Conséquences anecdotiques
    - convocation de centenaires à l'école
    - le site France.fr tombe en panne dès son lancement
  - Conséquences scientifiques
    - Mission Venus, passe à 5 000 000 de km au lieu de 5 000
    - Mariner 1, détruite peu de temps après son lancement
  - Conséquences financières
    - Logiciel pour la paie des militaires
    - Panne du site voyage-sncf.com
    - Échec du premier lancement d'Ariane V
  - Conséquences en termes de vies humaines
    - Sur-irradiés à Épinal
    - Canon anti-aérien faisant 9 morts et 14 blessés

Pourquoi écrire des tests ? (2)

- «Les programmes sans bug n'existent pas» :
  - Les programmes sont écrits par des humains, qui ne sont pas infaillibles
  - Les programmes sont spécifiés par des humains, qui ne sont pas infaillibles
  - Les programmes tournent dans des environnements buggués (interpréteur, système)
  - Les «bonnes pratiques» améliorent grandement les choses, mais ne garantissent rien
- La vérification formelle a ses faiblesses :
  - lourde à mettre en œuvre ; nécessite des compétences particulières
  - ne prend pas en compte l'environnement (compilateur, système, etc.)
  - ne vérifie que ce qui est spécifié !

Les bugs sont des erreurs humaines !

Différents types de tests

Les tests peuvent avoir différents objectifs. Voici quelques exemples :

- Tests fonctionnels : vérifier que le programme fait bien ce qui lui est demandé
- Tests utilisateur : vérifier que les utilisateurs arrivent à utiliser le logiciel
- Tests de montée en charge : vérifier que le logiciel fonctionne avec un grand nombre d'utilisateurs simultanés
- Tests de compatibilité : vérifier que le logiciel respecte bien certains standards
- Tests de sécurité : vérifier que le logiciel est protégé contre des tentatives d'attaque
- ...

Différents niveaux de tests

Les *tests de recette* évaluent un déroulement complet d'une fonctionnalité du logiciel.

Les *tests d'intégration* évaluent que 2 modules séparés collaborent correctement entre eux.

Les *tests unitaires* évaluent une petite unité de code (une fonction, une méthode).

Les tests unitaires sont faciles à écrire, et permettent de bien localiser une erreur. Ils sont donc fondamentaux (mais cela ne doit pas dispenser d'écrire des tests aux autres niveaux).

Un peu de vocabulaire

- On appelle **cas de test** un cas d'utilisation du logiciel défini par :
  - un nom
  - des *données d'entrée*
  - un *résultat attendu*.
- On appelle **jeu de tests** un ensemble de cas de tests. Souvent, on regroupe dans un jeu de tests des cas de test ayant un même but.

Le faux sentiment de sécurité lié aux tests (1)

- **Exemple 1** : *les corbeaux sont noirs* peut être spécifié par l'une des 2 propositions logiques ci-dessous :
  - corbeau $\rightarrow$ noir
  - non-noir $\rightarrow$ non corbeau (contraposée de la précédente)
- Donc :
  - Un cas de test *confirmant* la deuxième proposition *confirme* également la première
  - Trouver un corbeau et constater qu'il est noir est un cas de test *confirmant* la première proposition
  - Trouver un objet rouge et s'apercevoir qu'il s'agit d'un panneau "stop" *confirme* la deuxième proposition
- Par conséquent :
  - Trouver un objet rouge qui n'est pas un corbeau apporte **exactement** la même garantie que trouver un corbeau qui est noir, c'est-à-dire... pas grand chose !
  - Et donc un cas de test qui réussit n'apporte *a priori* pas grand chose.

Le faux sentiment de sécurité lié aux tests (2)

- **Exemple 2** : Soit la fonction suivante sensée être appelée avec des paramètres entiers :

```
def somme(a, b):
  return 0
```

- Question : si je génère des cas de tests aléatoirement, combien de cas dois-je générer pour être sûr d'établir que ma fonction est fausse ?
- Réponse : une infinité

Le véritable apport des tests

- Retour sur l'exemple 1 : trouver un corbeau qui ne soit pas noir **prouve** que la propriété corbeau $\rightarrow$ noir est fausse
- Retour sur l'exemple 2 : tester la fonction *somme* sur les paramètres 2 et 3 **prouve** que la fonction somme est buggée.

Conclusion sur l'exécution d'un cas de test

La réussite d'un cas de test ne garantit rien hormis la correction du programme sur **une** exécution pour **le cas** ayant réussi.

(*Si le cas de test est bien défini*) L'échec d'un cas de test **garantit** que le programme n'est pas correct.

Comment écrire des tests

- Un cas de test doit donc être exécuté aussi souvent que possible, notamment pour vérifier la **non-régression**. Du coup, un cas de test doit être :
  - rapidement exécutable
  - exécutable automatiquement (en général, écrit dans le langage cible)
  - reproductible
- Écrire un cas de test doit doit avoir pour but de trouver une erreur, et non de garantir l'absence d'erreur
- Les cas de test doivent être indépendants les uns des autres
- Comme seul un ensemble de tests exhaustif peut réellement servir de preuve, on cherche, en un minimum de cas, à obtenir un jeu de tests *représentatif*
- En Python, il existe différents moyens d'écrire des tests : `unittest`, `doctest` et `pytest`.

Écrire des tests en Python : Exemple support

```python
class Personne:
  def __init__(self, nom, prenom, age):
    self.nom = nom.capitalize()
    self.prenom = prenom
    self.age = age

  def get_nom(self):
    return self.nom

  def get_prenom(self):
    return self.prenom

  def get_age(self):
    return self.age

  def croitre(self):
    self.age += 1
```

Écrire des tests en Python : unittest - principe

`unittest` est un système de test unitaire intégré à Python. Il fait partie de la famille des systèmes *xUnit*, dérivés de *jUnit* (système de tests unitaires pour Java) :

- Les tests sont écrits sous la forme de méthodes d'une classe héritant de la classe `TestCase`. Un test qui rate doit lever une exception. Dans le cas contraire, le test est considéré comme réussi.
- La classe `TestCase` fournit un ensemble de méthodes facilitant l'écriture des tests unitaires.

Écrire des tests en Python : unittest - exemple

```python
import unittest
from personne import Personne

class TestPersonne(unittest.TestCase):
    def test_nom(self):
        pers = Personne("martin", "jacques", 20)
        attendu = "Martin"
        effectif = pers.get_nom()
        self.assertEqual(attendu, effectif)

    def test_prenom(self):
        pers = Personne("martin", "jacques", 20)
        attendu = "Jacques"
        effectif = pers.get_prenom()
        self.assertEqual(attendu, effectif)

    def test_croitre(self):
        pers = Personne("martin", "jacques", 20)
        pers.croitre()
        attendu = 21
        effectif = pers.get_age()
        self.assertEqual(attendu, effectif)

if __name__ == "__main__":
    unittest.main()
```

Écrire des tests en Python : unittest - exécution

```
..F
======================================================
FAIL: test_prenom (__main__.TestPersonne)
------------------------------------------------------
Traceback (most recent call last):
  File "unittestPersonne.py", line 15, in test_prenom
    self.assertEqual(attendu, effectif)
AssertionError: 'Jacques' != 'jacques'
- Jacques
? ^
+ jacques
? ^
------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Écrire des tests en Python : doctest - principe

Les tests sont intégrés à la documentation de la méthode (docstring) ainsi :

- Le code générant le résultat effectif est précédé de >>>
- Juste en-dessous, on écrit le résultat attendu

Écrire des tests en Python : doctest - exemple

```
class Personne:
    """Classe permettant de représenter une personne.

    Le nom est transformé pour commencer avec une majuscule :
    >>> p = Personne("martin", "jacques", 20); p.get_nom()
    'Martin'

    >>> p = Personne("martin", "jacques", 20); p.get_prenom()
    'Jacques'

    >>> p = Personne("martin", "jacques", 20)
    >>> p.croitre()
    >>> p.get_age()
    21
    """
    ... code de la classe ...
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Écrire des tests en Python : doctest - exécution

```
**********************************************************************
File "personnedoc.py", line 8, in __main__.Personne
Failed example:
    p = Personne("martin", "jacques", 20); p.get_prenom()
Expected:
    'Jacques'
Got:
    'jacques'
**********************************************************************
1 items had failures:

1 of 5 in __main__.Personne

***Test Failed*** 1 failures.
```

Écrire des tests en Python : pytest - principe

pytest est un outil externe à Python qui permet d'écrire et exécuter des tests unitaires en utilisant les assertions de base du langage Python.

pytest fournit de nombreux outils permettant de faciliter la mise en œuvre de tests, ce qui en fait l'outil de tests unitaires le plus utilisé dans le monde professionnel.

Lancé sans argument, *pytest* exécute tous les tests présents dans des fichiers `test_`*nom*`.py`. Mais on peut aussi lui passer en paramètre un nom explicite de fichier de test.

Un test est défini par une fonction dont le nom est préfixé par *test_*. Cette fonction doit lever une AssertionError pour indiquer qu'un test est en échec.

Écrire des tests en Python : pytest - exemple

```
from personne import Personne

def test_nom():
  pers = Personne("martin", "jacques", 20)
  attendu = "Martin"
  effectif = pers.get_nom()
  assert attendu == effectif

def test_prenom():
  pers = Personne("martin", "jacques", 20)
  attendu = "Jacques"
  effectif = pers.get_prenom()
  assert attendu == effectif

def test_croitre():
  pers = Personne("martin", "jacques", 20)
  pers.croitre()
  attendu = 21
  effectif = pers.get_age()
  assert attendu == effectif
```

Écrire des tests en Python : pytest - exécution

```
collected 3 items

test_personne.py .F.                                    [100%]

====================== FAILURES ===========================
__________________ test_prenom ____________________

    def test_prenom():
        pers = Personne("martin", "jacques", 20)
        attendu = "Jacques"
        effectif = pers.get_prenom()
>       assert attendu == effectif
E       AssertionError: assert 'Jacques' == 'jacques'
E         - Jacques
E         ? ^
E         + jacques
E         ? ^

test_personne.py:13: AssertionError
============ 1 failed, 2 passed in 0.03s ================
```

pytest dans Jupyter-notebook

Exemple d'une solution testée :

- Installer ipytest (attention aux versions !)
  - `pip install ipytest`
  - `conda install -c conda-core ipytest`
- Charger et initialiser ipytest dans une première cellule :

```
import ipytest
ipytest.autoconfig()
```

- Commencer une cellule contenant des tests par :

```
%%run_pytest[clean] -qq
```

Système de tests unitaires à choisir

- *unittest* oblige à parler d'héritage $\rightarrow$ à éviter ;
- *doctest* a ses vertus pédagogiques, mais alourdit le code source. Par ailleurs, il est mal adapté à l'écriture de cas de tests compliqués ;
- *pytest* oblige à installer un outil supplémentaire.

$\Rightarrow$ Choix retenu pour la suite : **pytest**.

Noire ou blanche ?

Boîte noire

On génère les cas de test à partir des données. Le but est, en un minimum de tests, d'arriver à couvrir l'ensemble des cas "typiques" d'utilisation d'une fonction.
Les tests «boîte noire» sont conçus à partir des spécifications, indépendamment du code.
Ces tests ne sont pas biaisés par le développeur ; ils sont souvent conçus par une autre personne, préalablement à l'écriture du code.

Boîte blanche

On génère les cas de test à partir du code. Le but est, en un minimum de tests, d'arriver à couvrir l'ensemble des façons possibles d'exécuter une fonction.
Ces tests assurent que l'on examine les différents traitements qui peuvent être effectués. Ils ne peuvent être conçus qu'après l'écriture du code.

Écrire des tests «boîte noire» : PCE - principe

Il existe plusieurs façons de générer un ensemble de tests «boîte noire» pertinents. Dans le cas où **les paramètres sont indépendants**, on pourra utiliser le « partitionnement en classes d'équivalence » :

- Pour chaque paramètre, on détermine les différentes groupes de valeurs amenant à un traitement similaire
- Il faut envisager aussi bien les classes « valides » que les classes « non valides» (beaucoup de bugs se produisent suite à des données inattendues)
- On génère un cas de test pour chaque classe de chaque paramètre ainsi :
  - on minimise le nombre de tests pour les classes valides ;
  - on isole les classes invalides.

Écrire des tests «boîte noire» : PCE - exemple (1)

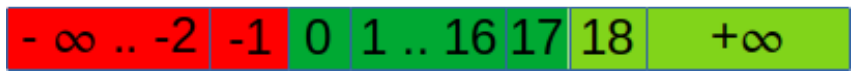
- Un programme prend en paramètre une personne (nom, prénom), un sexe (h/f/ns), un âge et renvoie une phrase de salutation :
  - début : M. / Mme / rien suivant le sexe
  - suite : Prénom Nom (âge < 18) ; Nom (âge ≥ 18)
- Classes pour `personne` :
  - None : PI1
  - autre : PV1
- Classes pour `sexe`:
  - “h” : SV1
  - “f” : SV2
  - “ns” : SV3
  - None: SI1
  - texte autre : SI2
- Classes pour âge :
  - $0 \leq$ âge < 18 : AV1
  - $18 \leq$ âge : AV2
  - âge < 0 : AI1
  - âge pas entier : AI2
  - âge pas nombre : AI3

Écrire des tests «boîte noire» : PCE - exemple (2)

- Génération des cas de tests valides :
  - PV1 ; SV1 ; AV1 → (("Cleese","John"),"h", 12) / "M. John Cleese"
  - PV1 ; SV2 ; AV2 → (("Maillan","Jaqueline"),"f", 97) / "Mme Maillan"
  - PV1 ; SV3 ; AV1 → (("Desproges","Pierre"), "ns", 81) / "Desproges"
- Génération des cas de tests invalides :
  - **PI1** ; SV1 ; AV2 → (None, "f", 24) / AssertionError("Personne requise")
  - PV1 ; **SI1** ; AV1 → (("Meurice","Guillaume"), None, 39) / AssertionError("Sexe requis")
  - PV1 ; **SI2** ; AV2 → (("Orsenna", "Erik"), "académicien", 73) / AssertionError("Sexe incorrect")
  - PV1 ; SV1 ; **AI1** → (("Clouteau", "Jacques"),"m", -5) / AssertionError("Âge négatif")
  - PV1 ; SV2 ; **AI2** → (("De Waal", "Frans"), "m", 3.1415926) / AssertionError("Âge non entier")
  - PV1 ; SV3 ; **AI3** → (("Christie", "Agatha"), "f", "cent trente") / AssertionError("Âge non numérique")

Écrire des tests «boîte noire» : Valeurs Frontières

- Principe : pour les types dénombrables, à la frontière entre 2 classes, on définit sous la forme de classes séparées les valeurs de part et d'autre de la frontière.
- Application sur l'exemple pour âge :



  - âge $<$ -1 : AI1
  - âge $=$ -1 : AI2
  - âge $=$ 0 : AV1
  - $1 \leq$ âge ≤ 16 : AV2
  - âge $= 17$: AV3
  - âge $= 18$: AV4
  - âge > 18 : AV5
  - âge non entier : AI3
  - âge non numérique : AI4
- Intérêt : force les tests aux valeurs frontières, zones fréquentes de bug

Écrire des tests « boîte blanche » (1)

- Préalable : disposer du code
- Principe : essayer de passer “partout” dans le code
- Exemple support :

```
def ordre(t):
    resultat = ""
    if t[0] <= t[1]:
        if t[1] <= t[2]:
            resultat = "abc"
        elif t[0] <= t[2]:
            resultat = "acb"
    elif t[1] <= t[2] and t[0] <= t[2]:
        resultat = "bac"
    return resultat
```

Écrire des tests « boîte blanche » (2)

Premier cas de test : t = [4, 10, 16] ; resultat attendu = “abc”

```
▶def ordre(t):
▶    resultat = ""
▶    if t[0] <= t[1]:
▶        if t[1] <= t[2]:
▶            resultat = "abc"
         elif t[0] <= t[2]:
             resultat = "acb"
     elif t[1] <= t[2] and t[0] <= t[2]:
         resultat = "bac"
▶    return resultat
```

Écrire des tests « boîte blanche » (3)

Deuxième cas de test : t = [3, 21, 17] ; resultat attendu = “acb”

```
▶def ordre(t):
▶    resultat = ""
▶    if t[0] <= t[1]:
▶        if t[1] <= t[2]:
▶            resultat = "abc"
▶        elif t[0] <= t[2]:
▶            resultat = "acb"
    elif t[1] <= t[2] and t[0] <= t[2]:
        resultat = "bac"
```

```
▶    return resultat
```

Écrire des tests « boîte blanche » (4)

Troisième cas de test : t = [12, 0, 100] ; resultat attendu = “bac”

```
▶def ordre(t):
▶    resultat = ""
▶    if t[0] <= t[1]:
▶        if t[1] <= t[2]:
▶            resultat = "abc"
▶        elif t[0] <= t[2]:
▶            resultat = "acb"
```

```
▶      elif t[1] <= t[2] and t[0] <= t[2]:
▶          resultat = "bac"
```

```
▶    return resultat
```

Écrire des tests « boîte blanche » (5)

- Tout le code a été couvert :

```
def ordre(t):
  resultat = ""
  if t[0] <= t[1]:
    if t[1] <= t[2]:
      resultat = "abc"
    elif t[0] <= t[2]:
      resultat = "acb"
    elif t[1] <= t[2] and t[0] <= t[2]:
      resultat = "bac"
  return resultat
```

- Pourtant, que donne le programme sur [45,32,10] ?

Couverture de code (1)

- **Couverture des instructions**

  C'est ce qui a été fait dans le cas précédent ; c'est un bon début, mais ce n'est pas suffisant.

- **Couverture des décisions**

  Pour chaque test, on doit passer par la branche *alors* et par la branche *sinon*. C'est déjà beaucoup mieux.

Exemple corrigé :

- on rajoute le cas de test t = [15, 10, 13] ; resultat attendu = “cab”

```
def ordre(t):
  resultat = ""
  if t[0] <= t[1]:
    if t[1] <= t[2]:
      resultat = "abc"
    elif t[0] <= t[2]:
      resultat = "acb"
    elif t[1] <= t[2] and t[0] <= t[2]:
      resultat = "bac"
    else:
      resultat = "cab"
  return resultat
```

- Qu’advient-il pour le cas t = [12, 14, 13] ?

Couverture de code (2)

- **Couverture des décisions multiples**

  Pour chaque condition, on génère un cas par *raison* de choisir une branche

  Cas de la dernière branche du problème précédent :

  - branche *alors* : t[1] <= t[2] and t[0] <= t[2]
  - branche *sinon* :
    - t[1] > t[2]
    - t[1] <= t[2] and t[0] > t[2]

Vérifier la couverture de code avec *Coverage*

Coverage est un outil permettant de vérifier la couverture des tests en Python.

- Installation
  - avec anaconda : `conda install coverage`
  - avec pip : `pip install coverage`
- Couvertures analysées
  - couverture des instructions
  - couverture des décisions
- Exécution (résultats stockés dans une base sqlite3):
  - couverture des instructions : `coverage run --source=. -m pytest fichier_de_test.py`
  - couverture des décisions : `coverage run --source=. --branch -m pytest fichier_de_test.py`
- Rapport :
  - en mode texte : `coverage report -m`
  - en version html : `coverage html ; firefox htmlcov/index.html`

Coverage : Exemple (1) : un premier cas de test

On définit la fonction `ordre` telle que sur le transparent 30.

On définit le fichier `test_ordre.py` ainsi :

```
from ordre import ordre

def test1():
    entree = [4, 10, 16]
    resultat_attendu = "abc"
    resultat_effectif = ordre(entree)
    assert resultat_attendu == resultat_effectif
```

Coverage : Exemple (2) : couverture des instructions, mode texte

On exécute la séquence d'instructions suivante :

```
coverage run --source=. -m pytest
coverage report -m
```

On obtient alors l'affichage suivant :

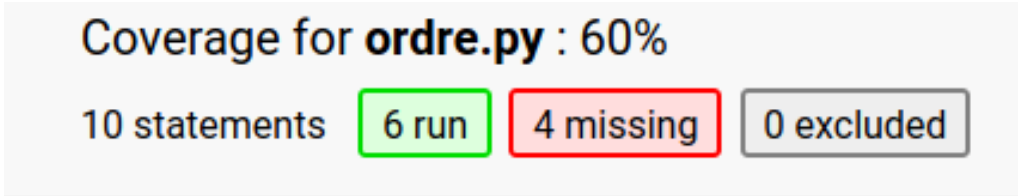
```
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
ordre.py             10      4    60%   6-9
test_ordre.py         6      0   100%
-----------------------------------------------
TOTAL                16      4    75%
```

Coverage : Exemple(3) : couverture des instructions, mode html

On exécute la séquence d'instructions suivante :

```
coverage html
firefox htmlcov/index.html
```

On obtient l'affichage suivant :




```
1  def ordre(t):
2      resultat = ""
3      if t[0] <= t[1]:
4          if t[1] <= t[2]:
5              resultat = "abc"
6          elif t[0] <= t[2]:
7              resultat = "acb"
8      elif t[1] <= t[2] and t[0] <= t[2]:
9          resultat = "bac"
10     return resultat
```

Coverage : Exemple (4) : couverture des décisions, mode texte

On exécute la séquence d'instructions suivante :

```
coverage run --source=. --branch -m pytest
coverage report -m
```

On obtient alors l'affichage suivant :

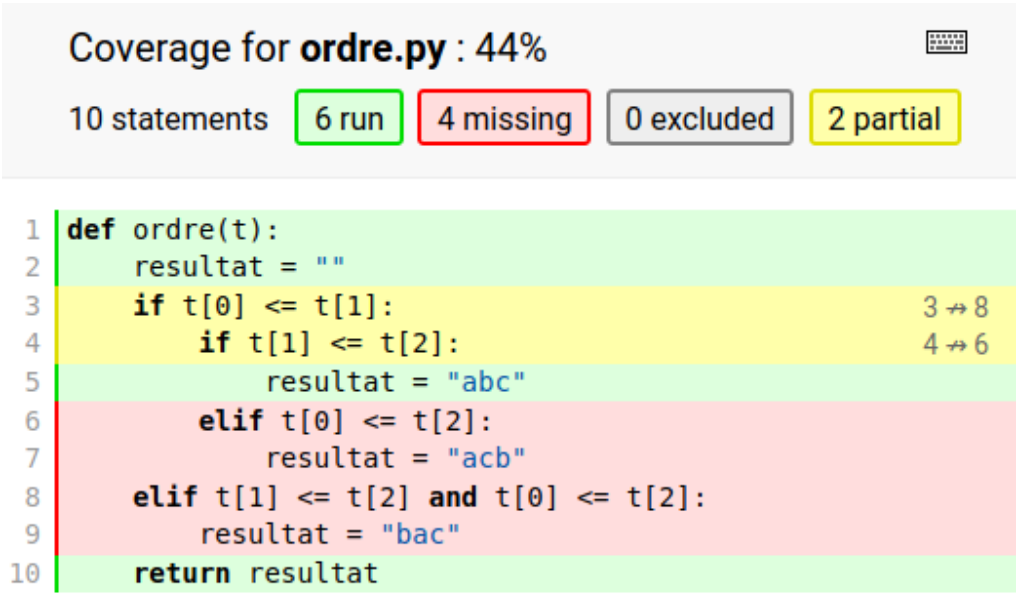
```
Name              Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------
ordre.py             10      4      8      2    44%   3->8, 4->6, 6-9
test_ordre.py         6      0      0      0   100%
-------------------------------------------------------------
TOTAL                16      4      8      2    58%
```

Coverage : Exemple(5) : couverture des décisions, mode html

On exécute la séquence d'instructions suivante :

```
coverage html
firefox htmlcov/index.html
```

On obtient l'affichage suivant :

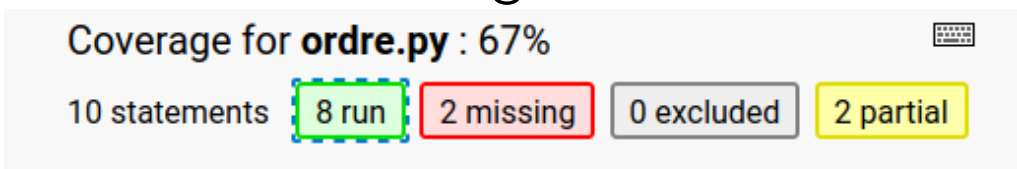


Coverage : Exemple(6) : *if* faux sans *else* non testé

On rajoute le cas de test suivant :

```
def test2():
    entree = [12, 0, 100]
    resultat_attendu = "bac"
    resultat_effectif = ordre(entree)
    assert resultat_attendu == resultat_effectif
```

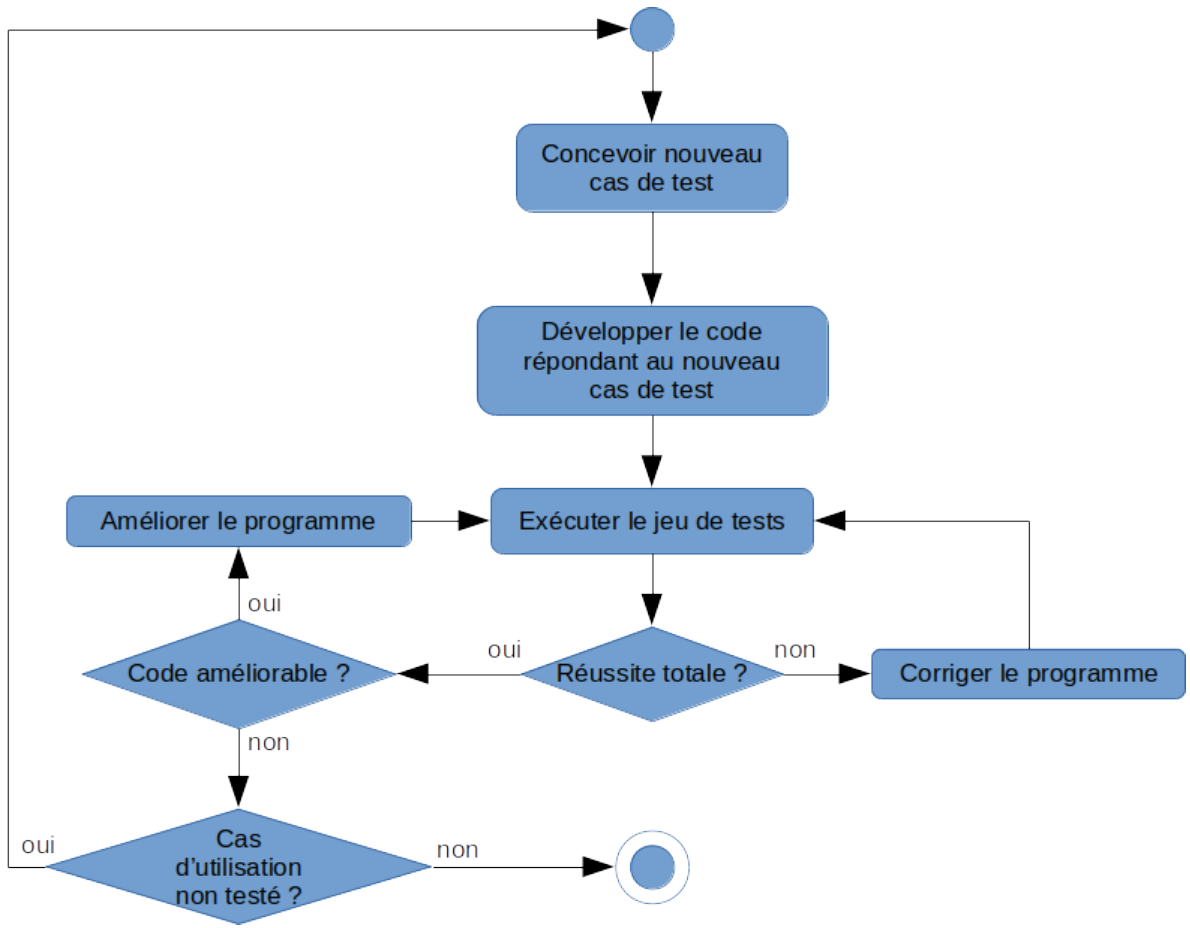
On obtient l'affichage suivant :




```
1  def ordre(t):
2      resultat = ""
3      if t[0] <= t[1]:
4          if t[1] <= t[2]:                    4 ↛ 6
5              resultat = "abc"
6          elif t[0] <= t[2]:
7              resultat = "acb"
8      elif t[1] <= t[2] and t[0] <= t[2]:     8 ↛ 10
9          resultat = "bac"
10     return resultat
```

Le *Test Driven Development*

Le TDD est un processus de développement consistant à écrire un cas de test avant d'écrire le code y répondant. On ne passe à un nouveau cas de test que lorsque tous les autres cas de test passent et que le code semble de bonne qualité.



Mise en œuvre du TDD : le problème du triangle

Cahier des charges :

> Écrire en python un programme qui demande à l'utilisateur de rentrer 3 données (des entiers) correspondant aux longueurs des côtés d'un triangle. Le programme doit alors préciser si le triangle est équilatéral, isocèle ou scalène.

Source : The Art of Software Testing

Problème du triangle : point de départ

Fichier *triangle.py*

```
def type_triangle(cote1, cote2, cote3):
    pass
```

Fichier *test_triangle.py*

```
from triangle import type_triangle

def test_scalene():
    #donnees
    cote1 = "2"
    cote2 = "3"
    cote3 = "4"
    #resultat attendu
    resultat_attendu = "SCALENE"
    #resultat effectif
    resultat_effectif = type_triangle(cote1, cote2, cote3)
    assert resultat_attendu == resultat_effectif
```

Problème du triangle : correction 1

Bilan des tests

```
test_triangle.py F
...
>       assert resultat_attendu == resultat_effectif
E       AssertionError: assert 'SCALENE' == None

test_triangle.py:12: AssertionError
```

Correction

```python
def type_triangle(cote1, cote2, cote3):
    return "SCALENE"
```

Problème du triangle : deuxième cas de test

test

```
def test_argument_non_num():
    #donnees
    cote1 = "toto"
    cote2 = "3"
    cote3 = "4"
    #verif
    with pytest.raises(TypeError):
        type_triangle(cote1, cote2, cote3)
```

correction

```
def type_triangle(cote1, cote2, cote3):
    try:
        int(cote1)
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return "SCALENE"
```

Problème du triangle : troisième cas de test

test

```
def test_argument_1_positif():
    #donnees
    cote1 = "-1"
    cote2 = "7"
    cote3 = "5"
    with pytest.raises(TypeError):
        type_triangle(cote1, cote2, cote3)
```

correction

```
def type_triangle(cote1, cote2, cote3):
    try:
        c1 = int(cote1)
        if c1 < 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return "SCALENE"
```

Problème du triangle : quatrième cas de test

```
def test_argument_2_positif():
    #donnees
    cote1 = "12"
    cote2 = "-457"
    cote3 = "325"
    with pytest.raises(TypeError):
        type_triangle(cote1, cote2, cote3)
```

Problème du triangle : quatrième cas de test (correction)

```
def type_triangle(cote1, cote2, cote3):
    try:
        c1 = int(cote1)
        if c1 < 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    try:
        c2 = int(cote2)
        if c2 < 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return "SCALENE"
```

Problème du triangle : Refactorisation

```
def type_triangle(cote1, cote2, cote3):
    c1 = convertit_longueur_si_valide(cote1)
    c2 = convertit_longueur_si_valide(cote2)
    c3 = convertit_longueur_si_valide(cote3)
    return "SCALENE"

def convertit_longueur_si_valide(cote_chaine):
    try:
        cote_num = int(cote_chaine)
        if cote_num <= 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return cote_num
```

Problème du triangle : Refactorisation des tests (1)

```python
import pytest
from triangle import type_triangle,\
     convertit_longueur_si_valide

def test_scalene():
    #donnees
    cote1 = "2"
    cote2 = "3"
    cote3 = "4"
    #resultat attendu
    resultat_attendu = "SCALENE"
    #resultat effectif
    resultat_effectif = type_triangle(cote1, cote2,\
                                      cote3)
    assert resultat_attendu == resultat_effectif

def test_argument_1_non_num():
    #donnees
    cote1 = "toto"
    cote2 = "3"
    cote3 = "4"
    with pytest.raises(TypeError):
        type_triangle(cote1, cote2, cote3)

def test_argument_2_positif():
    #donnees
    cote1 = "12"
    cote2 = "-457"
    cote3 = "325"
    with pytest.raises(TypeError):
        type_triangle(cote1, cote2, cote3)
```

```python
def test_argument_3_nul():
    #donnees
    cote1 = "12"
    cote2 = "457"
    cote3 = "0"
    with pytest.raises(TypeError):
        type_triangle(cote1, cote2, cote3)

def test_argument_num():
    #donnees
    cote_chaine = "618"
    #resultat attendu
    resultat_attendu = 618
    #resultat effectif
    resultat_effectif = convertit_longueur_si_valide(c
    #verif
    resultat_attendu == resultat_effectif

def test_argument_chaine():
    #donnees
    cote = "hello"
    with pytest.raises(TypeError):
        convertit_longueur_si_valide(cote)

def test_argument_negatif():
    #donnees
    cote = "-9876"
    with pytest.raises(TypeError):
        convertit_longueur_si_valide(cote)
```

Problème du triangle : Refactorisation des tests (2)

```python
def test_argument_nul():
    #donnees
    cote = "0"
    with pytest.raises(TypeError):
        convertit_longueur_si_valide(cote)

def test_argument_none():
    #donnees
    cote = None
    with pytest.raises(TypeError):
        convertit_longueur_si_valide(cote)
```

Problème du triangle : triangle équilatéral

```
def test_equilateral():
    #donnees:
    c1 = "3"
    c2 = "3"
    c3 = "3"
    #resultat attendu
    resultat_attendu = "EQUILATERAL"
    #resultat effectif
    resultat_effectif = type_triangle(c1, c2, c3)
    #vérification
    assert resultat_attendu == resultat_effectif
```

Problème du triangle : triangle équilatéral (correction)

```
def type_triangle(cote1, cote2, cote3):
    c1 = convertit_longueur_si_valide(cote1)
    c2 = convertit_longueur_si_valide(cote2)
    c3 = convertit_longueur_si_valide(cote3)
    if c1 == c2 and c2 == c3:
        return "EQUILATERAL"
    return "SCALENE"
```

Problème du triangle : refactoring (condition trop compliquée)

```
def type_triangle(cote1, cote2, cote3):
    c1 = convertit_longueur_si_valide(cote1)
    c2 = convertit_longueur_si_valide(cote2)
    c3 = convertit_longueur_si_valide(cote3)
    if est_equilateral(c1, c2, c3):
        return "EQUILATERAL"
    return "SCALENE"

def est_equilateral(c1, c2, c3):
    return c1 == c2 and c2 == c3

def convertit_longueur_si_valide(cote_chaine):
    try:
        cote_num = int(cote_chaine)
        if cote_num <= 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return cote_num
```

Problème du triangle : refactoring (méthode ayant 2 rôles)

```
def type_triangle(cote1, cote2, cote3):
    c1 = convertit_longueur_si_valide(cote1)
    c2 = convertit_longueur_si_valide(cote2)
    c3 = convertit_longueur_si_valide(cote3)
    return type_triangle_num(c1, c2, c3)

def type_triangle_num(cote1, cote2, cote3):
    if est_equilateral(cote1, cote2, cote3):
        return "EQUILATERAL"
    return "SCALENE"

def est_equilateral(c1, c2, c3):
    return c1 == c2 and c2 == c3

def convertit_longueur_si_valide(cote_chaine):
    try:
        cote_num = int(cote_chaine)
        if cote_num <= 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return cote_num
```

Problème du triangle : triangles isocèles

À faire normalement cas après cas, mais je gagne un peu de temps en présentant tout sur le même transparent

```
def test_isocele_1():
    #donnees:
    c1 = "3"
    c2 = "3"
    c3 = "4"
    #resultat attendu
    resultat_attendu = "ISOCELE"
    #resultat effectif
    resultat_effectif = type_triangle(c1, c2, c3)
    #vérification
    assert resultat_attendu == resultat_effectif

def test_isocele_2():
    #donnees:
    c1 = "8"
    c2 = "10"
    c3 = "10"
    #resultat attendu
    resultat_attendu = "ISOCELE"
    #resultat effectif
    resultat_effectif = type_triangle(c1, c2, c3)
    #vérification
    assert resultat_attendu == resultat_effectif

def test_isocele_3():
    #donnees:
    c1 = "15"
    c2 = "20"
    c3 = "15"
    #resultat attendu
    resultat_attendu = "ISOCELE"
    #resultat effectif
    resultat_effectif = type_triangle(c1, c2, c3)
    #vérification
    assert resultat_attendu == resultat_effectif
```

Problème du triangle : triangles isocèles (correction)

```
def type_triangle(cote1, cote2, cote3):
    c1 = convertit_longueur_si_valide(cote1)
    c2 = convertit_longueur_si_valide(cote2)
    c3 = convertit_longueur_si_valide(cote3)
    return type_triangle_num(c1, c2, c3)

def type_triangle_num(cote1, cote2, cote3):
    if est_equilateral(cote1, cote2, cote3):
        return "EQUILATERAL"
    elif est_isocele_si_non_equilateral(cote1, cote2, cote3):
        return "ISOCELE"
    return "SCALENE"

def est_equilateral(c1, c2, c3):
    return c1 == c2 and c2 == c3

def est_isocele_si_non_equilateral(c1, c2, c3):
    return c1 == c2 or c2 == c3 or c1 == c3
```

Problème du triangle : refactoring (trop de paramètres)

```python
def type_triangle(cote1, cote2, cote3):
    c1 = convertit_longueur_si_valide(cote1)
    c2 = convertit_longueur_si_valide(cote2)
    c3 = convertit_longueur_si_valide(cote3)
    triangle = Triangle(c1, c2, c3)
    return triangle.type()

def convertit_longueur_si_valide(cote_chaine):
    try:
        cote_num = int(cote_chaine)
        if cote_num <= 0:
            raise TypeError("nb strictement positif")
    except ValueError:
        raise TypeError("Un côté doit être un entier")
    return cote_num
```

```python
class Triangle:
    """
    Classe représentant un triangle par les longueurs
    de ses 3 côtés sous forme d'entiers.
    """

    def __init__(self, c1, c2, c3):
        self.cote1 = c1
        self.cote2 = c2
        self.cote3 = c3

    def type(self):
        if self.est_equilateral():
            return "EQUILATERAL"
        elif self.est_isocele_si_non_equilateral():
            return "ISOCELE"
        return "SCALENE"

    def est_equilateral(self):
        return self.cote1 == self.cote2 and \
               self.cote2 == self.cote3

    def est_isocele_si_non_equilateral(self):
        return self.cote1 == self.cote2 or \
               self.cote2 == self.cote3 or \
               self.cote1 == self.cote3
```

Problème du triangle : non triangle 1

```
def test_non_triangle1():
    #donnees :
    c1 = "2"
    c2 = "3"
    c3 = "6"
    with pytest.raises(NonTriangleError):
        type_triangle(c1, c2, c3)
```

Problème du triangle : non triangle 1 (correction)

```
    def type(self):
        if self.cote3 >= self.cote1 + self.cote2:
            raise NonTriangleError()
        if self.est_equilateral():
            return "EQUILATERAL"
        elif self.est_isocele_si_non_equilateral():
            return "ISOCELE"
        return "SCALENE"

class NonTriangleError(Exception):
    pass
```

Problème du triangle : non triangle 2

```
def test_non_triangle2():
    #donnees :
    c1 = "2"
    c2 = "10"
    c3 = "4"
    with pytest.raises(NonTriangleError):
      type_triangle(c1, c2, c3)
```

Problème du triangle : non triangle 2 (correction)

```
def type(self):
    if self.cote3 >= self.cote1 + self.cote2 or\
       self.cote2 >= self.cote1 + self.cote3:
        raise NonTriangleError()
    if self.est_equilateral():
        return "EQUILATERAL"
    elif self.est_isocele_si_non_equilateral():
        return "ISOCELE"
    return "SCALENE"
```

Problème du triangle : refactoring

```
def type(self):
    if self.non_triangle():
        raise NonTriangleError()
    if self.est_equilateral():
        return "EQUILATERAL"
    elif self.est_isocele_si_non_equilateral():
        return "ISOCELE"
    return "SCALENE"

def non_triangle(self):
    return self.cote3 >= self.cote1 + self.cote2 or\
        self.cote2 >= self.cote1 + self.cote3 or\
        self.cote1 >= self.cote2 + self.cote3
```

Et pour conclure...

- Des tests qui réussissent ne garantissent rien
- Écrire des cas de test amène à
  - se poser des questions plus profondes et mieux comprendre le problème
  - manipuler des `assert` à foison
- Les tests «boîte noire» et «boîte blanche» sont complémentaires ; leurs principes peuvent être expliqués informellement assez rapidement
- Faire passer un jeu de tests à un travail rendu peut être une façon d'avoir une partie de note automatiquement
- *Coverage* est un outil qui peut être utilisé pour montrer ce qu'il se passe dans un cas précis (mettre alors un seul cas de test correspondant au cas en question)
- Pour motiver l'écriture de tests, faire écrire une fonction à un élève et les tests boîte noire à un autre.