Tests unitaires

Christophe Viroulaud

Terminale - NSI

Algo 16

Une tâche importante d'un programmeur est de garantir le bon comportement de son code, dans tous les cas de figures.

Comment mettre en place des tests unitaires ?

Sommaire

Nécessité de tester

```python
def images(deb, fin: int) -> list:
    """
    calcule les images d'une fonction
    """
    tab = []
    for i in range(deb, fin):
        tab.append(f(i))
    return tab

def f(x: int) -> float:
    return 1/x
```

```
>>> images(1, 5)
[1.0, 0.5, 0.3333333333333333, 0.25]
>>> images(0, 5)
ZeroDivisionError: division by zero
```

- Un projet est composé de nombreux modules, eux-mêmes découpés en plusieurs fonctions.
- Une fonction `f1` peut utiliser le résultat renvoyé par une fonction `f2` qui utilise le résultat de `f3`...

À retenir

Chaque fonction doit être testée individuellement, pour chaque cas de figure.

Sommaire

Repérer les cas limites

Observation

La fonction `f` impose des contraintes mathématiques.

```
1  def images(deb, fin: int) -> list:
2      """
3      calcule les images d'une fonction
4      """
5      tab = []
6      for i in range(deb, fin):
7          tab.append(f(i))
8      return tab
9
10 def f(x: int) -> float:
11     return 1/x
```

```
1 def f(x: int) -> float:
2     assert x != 0, "pas de division par zéro"
3     return 1/x
```

Code 1 – Mettre en place des **assertions**

```python
def f(x: int) -> float:
    if x == 0:
        raise Exception("pas de division par 0")
    return 1/x
```

Code 2 – Lever une **exception** personnalisée

À retenir

Cette notion est hors-programme en Terminale.

Sommaire

Garantir le bon comportement - Mise en évidence

À retenir

Même sans erreur mathématique, une fonction peut ne pas renvoyer le résultat attendu.

Activité 1 : La fonction `est_complet` doit renvoyer `True` si la matrice d'adjacence représente un graphe complet.

```
1  def est_complet(mat: list) -> bool:
2      """
3      vérifie si chaque sommet est relié à tous
4      les autres (sauf lui même)
5      """
6      for ligne in range(len(mat)):
7          for col in range(len(mat)):
8              if ligne != col and mat[ligne][col] == 0:
9                  return False
0      return True
```

Proposer un cas de figure où la fonction n'a pas le bon comportement.

Correction

```
1 mat = [ [1, 1, 1, 1, 1],
2         [1, 0, 1, 1, 1],
3         [1, 1, 0, 1, 1],
4         [1, 1, 1, 0, 1],
5         [1, 1, 1, 1, 0]]
```

Code 3 – La fonction renvoie **`True`** avec cette matrice.

```python
def est_complet(mat: list) -> bool:
    """
    vérifie si chaque sommet est relié à tous
    les autres (sauf lui même)
    """
    for ligne in range(ordre(mat)):
        for col in range(ordre(mat)):
            if (ligne != col and mat[ligne][col] == 0) or \
                    (ligne == col and mat[ligne][col] == 1):
                return False
    return True
```

Code 4 – La fonction correcte

Sommaire

Automatiser les tests

À retenir

L'objectif de la mise en place des tests unitaires est de proposer une structure qui garantira le bon comportement de chaque fonction tout au long de l'évolution du projet.

Les tests doivent être :

- exhaustifs (possible ?),
- évolutifs en fonction des modifications du projet,
- exécutables de manière automatisée.

La bibliothèque Python **unittest** permet de réaliser des tests unitaires. La stratégie consiste à créer une classe qui hérite de **unittest.TestCase** puis d'écrire une méthode pour chaque fonction à tester.

```
1  import unittest
2  from mathematiques import f # fonction
       inverse
3
4  class Tests(unittest.TestCase):
5
6      def test_inverse(self):
7          self.assertTrue(f(1.) == 1.)
8          self.assertTrue(f(-1) == -1.)
9
10
11 if __name__=="__main__":
12     unittest.main()
```

```python
import unittest
from mathematiques import f # fonction inverse

class Tests(unittest.TestCase):

    def test_inverse(self):
        self.assertTrue(f(1.) == 1.)
        self.assertTrue(f(-1) == -1.)

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

Activité 2 :

1. Créer un fichier `mes_tests.py`
2. Écrire le code précédent (adapter le nom du fichier).

Activité 3 :

```
1 def test_inverse_erreur(self):
2     # vérifie que la valeur 0 provoque une
    erreur
3     with self.assertRaises(ZeroDivisionError):
4         f(0)
```

Code 5 – Tester une erreur

Ajouter la méthode précédente.

Il existe plusieurs assertions testables :

Méthode	Vérification
`assertEqual(a, b)`	`a == b`
`assertIs(a, b)`	`a is b`
`assertIn(a, b)`	`a in b`

Il est possible de définir un environnement de test pour se rapprocher des conditions d'utilisation.

```
1  class Tests(unittest.TestCase):
2
3      def setUp(self):
4          # on définit des attributs...
5          self.val = 1
6
7      def test_inverse(self):
8          # ...utilisables dans les tests
9          self.assertTrue(f(self.val) == 1.)
```

Activité 4 : Proposer un jeu de tests pour valider la fonction **est_complet**.

```
1  def est_complet(mat: list) -> bool:
2      """
3      vérifie si chaque sommet est relié à tous
4      les autres (sauf lui même)
5      """
6      for ligne in range(ordre(mat)):
7          for col in range(ordre(mat)):
8              if (ligne != col and mat[ligne][col] == 0) or \
9                      (ligne == col and mat[ligne][col] == 1):
10                 return False
11     return True
```

Correction

```
1  class Tests(unittest.TestCase):
2
3      def setUp(self):
4          # solution non exhaustive
5          self.mat_ok = [[0, 1, 1, 1, 1],
6                         [1, 0, 1, 1, 1],
7                         [1, 1, 0, 1, 1],
8                         [1, 1, 1, 0, 1],
9                         [1, 1, 1, 1, 0]]
10         self.mat_no = [[1, 1, 1, 1, 1],
11                        [1, 0, 1, 1, 1],
12                        [1, 1, 0, 1, 1],
13                        [1, 1, 1, 0, 1],
14                        [1, 1, 1, 1, 0]]
15
16     def test_complet(self):
17         self.assertTrue(est_complet(self.mat_ok))
18         self.assertFalse(est_complet(self.mat_no))
```

À retenir

Pour prendre en compte tous les cas de figures, le nombre de tests peut être important.

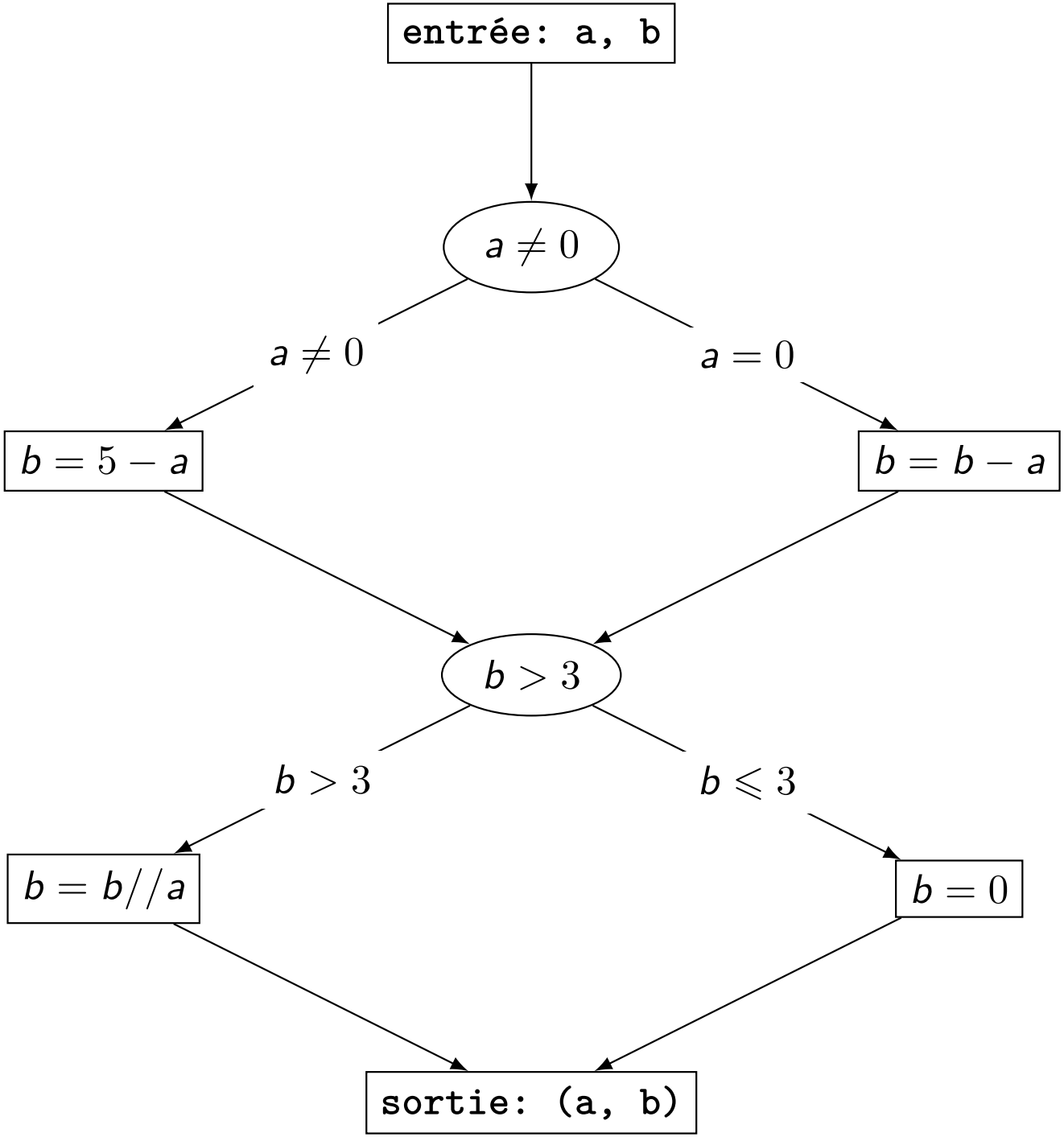
Sommaire

Tester de manière exhaustive

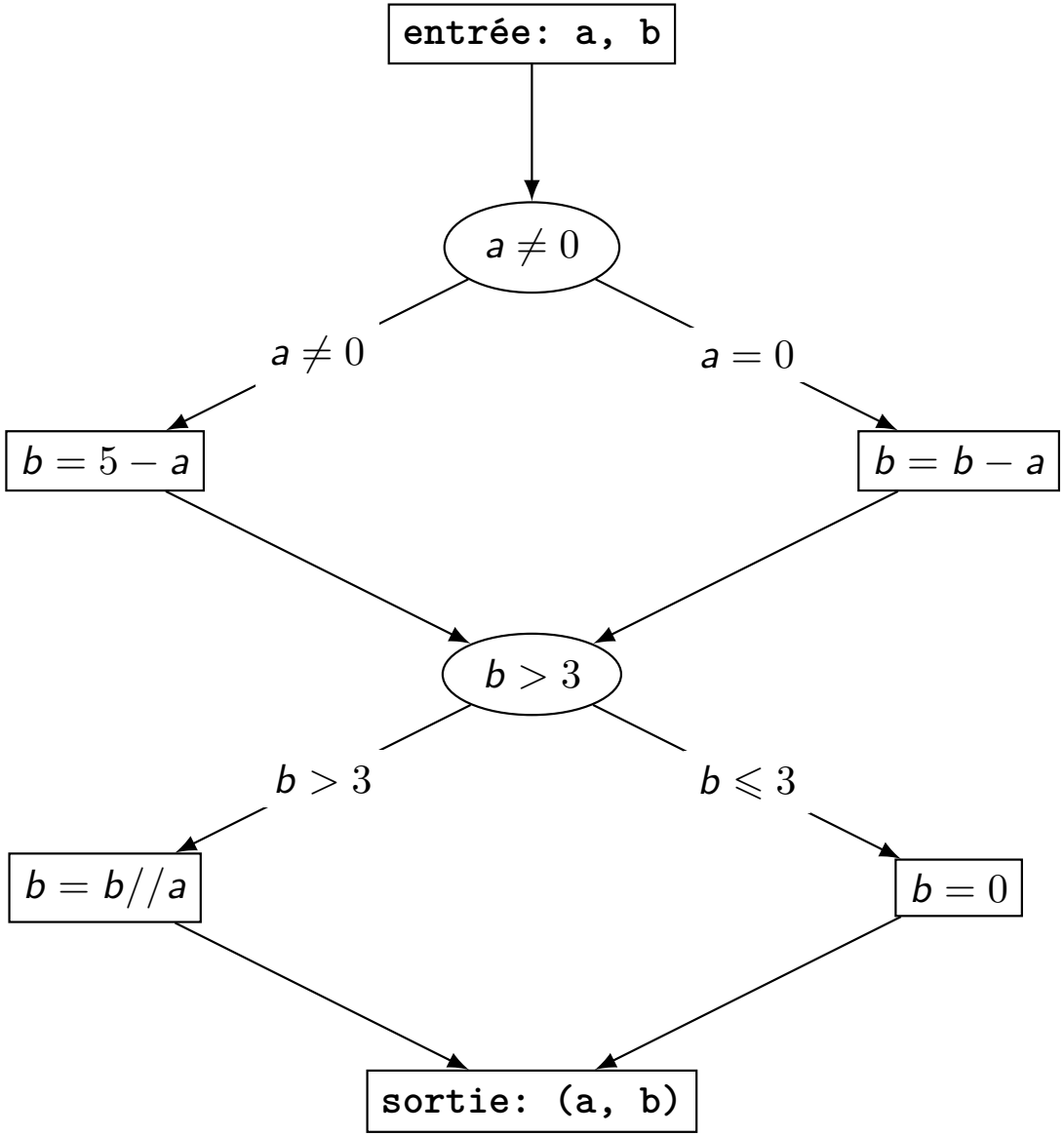
À retenir

Pour tester un code il faut prendre en compte tous les cas de figures.

```
1  def ma_fonction(a: int, b: int) -> tuple:
2      if a != 0:
3          b = 5 - a
4      else:
5          b = b - a
6      if b > 3:
7          b = b//a
8      else:
9          b = 0
10     return (a, b)
```


entrée: a, b
$a \neq 0$
$a \neq 0$
$a = 0$
$b = 5 - a$
$b = b - a$
$b > 3$
$b > 3$
$b \leqslant 3$
$b = b//a$
$b = 0$
sortie: (a, b)

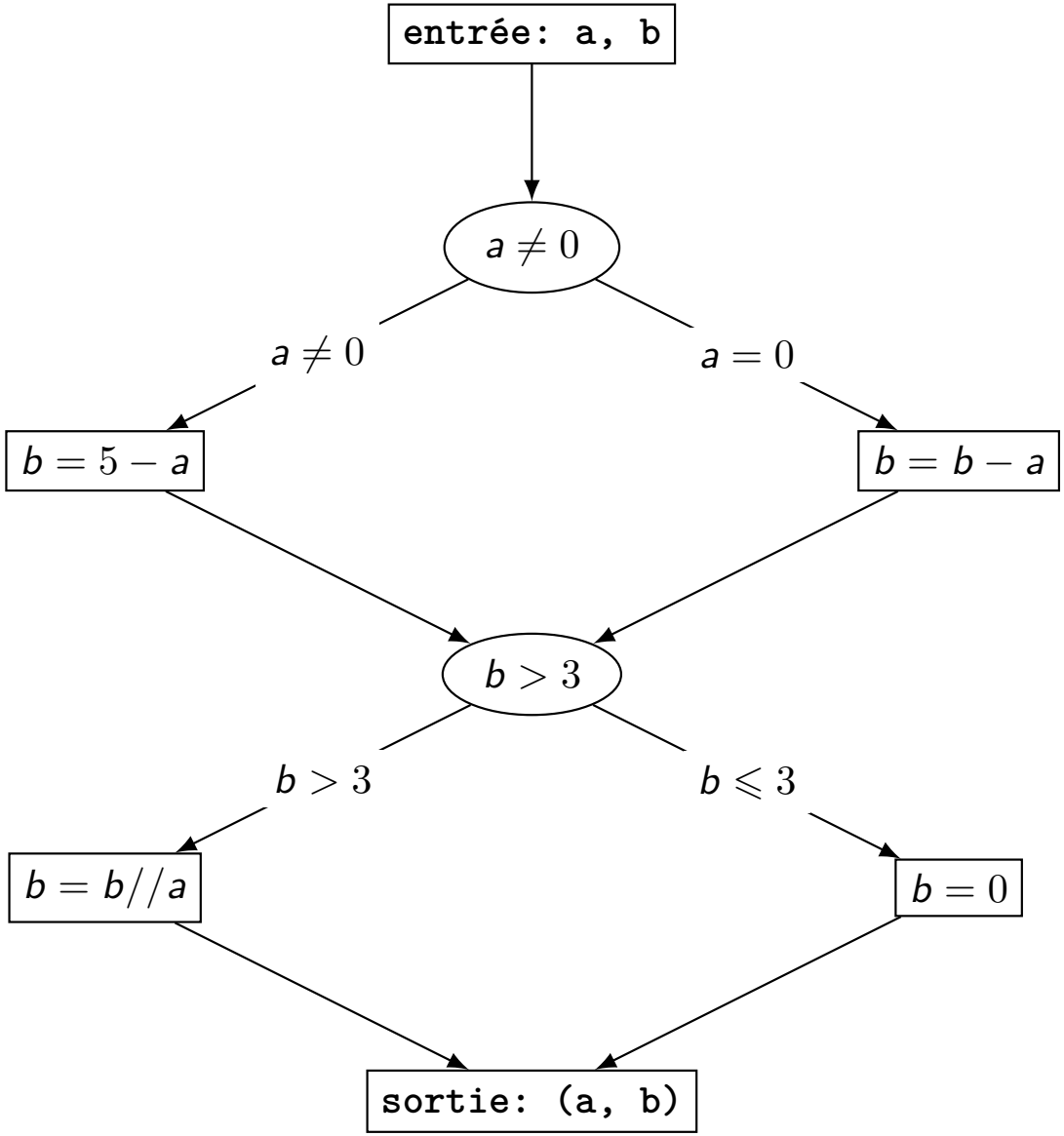
Pour tester `ma_fonction` il faut choisir plusieurs couples de valeurs `a, b`.




```
1 ma_fonction(0, 3)
2 ma_fonction(1, 3)
```

Observation

Toutes les arêtes ont été testées, cependant le risque de la division par zéro n'est pas montré.




```
1 ma_fonction(0, 4)
2 ma_fonction(2, 1)
```

Edsger Dijkstra

Program testing can be used to show the presence of bugs, but never to show their absence.

Traduction

Le test peut être utilisé pour démontrer la présence de problèmes, mais jamais pour démontrer leur absence.