# Tester un programme : premiers éléments

## I Introduction

1. *Le point, 4 avril 2018*, **La Poste : un bug qui illustre notre dépendance à l'informatique**". Les guichets des 8 500 bureaux en France ont été paralysés lundi par un incident informatique" ;
2. *actu.fr, 26 mai 2020*, **Bugs informatiques, stress... Ces étudiants de Rouen ont mal vécu les examens à distance** ;
3. *Le Monde Informatique, 13 mars 2021*, **Microsoft confirme le crash de Windows 10 lors d'impressions**.

Comment vérifier la qualité d'un logiciel, i.e. une ensemble de programmes répondant à un besoin spécifique ?

- **Conformité aux spécifications** : est-ce que le logiciel réalise les fonctions attendues, vues par un utilisateur ? La réponse à cette question nécessite **une validation** du logiciel. On dit qu'on réalise un test « **en boîte noire** ».
- **Fiabilité** : est-ce que le code du logiciel fonctionne correctement, vu par un programmeur ? Il est sous-entendu qu'il s'agit d'un fonctionnement avec des données valides. La réponse à cette question nécessite **une vérification** du code. On dit qu'on réalise un test « **en boîte blanche** ».
  En pratique, cela se traduit par :
  - **un test statique** : revue des lignes de code, en particulier des conditions de tests et des cas aux bornes ;
  - **un test dynamique** : exécution du code pour s'assurer d'un fonctionnement correct.
- **Robustesse** : comment réagit le logiciel lors de données invalides ? La réponse à cette question nécessite **une vérification de la signature de chaque fonction**, i.e. des types de données en entrée et du domaine de valeurs de chaque donnée.
- **Performance** : comment réagit le logiciel lors d'une montée en charge ? La réponse à cette question nécessite des tests spécifiques :
  - Augmentation du nombre d'utilisateurs, augmentation de la taille des données en entrée (**étude de la complexité** des programmes) : **load testing**
  - Soumission à des demandes anormales de ressources : **stress testing**

En dehors du stress testing, tous les autres éléments font partie du programme de classes préparatoires et doivent donner lieu à de « bonnes pratiques » de programmation attendues au concours.
Ci-joint quelques définitions éclairantes de ce que sont les tests d'un programme ou plus généralement, d'un logiciel :

1. « Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus » - IEEE (Institute of Electrical and Electronics Engineers, *Standard Glossary of Software Engineering Terminology*)
2. « Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts » - G. Myers. *The Art of Software Testing*, 1979.
3. "Testing can reveal the presence of errors but never their absence"– Edsger Wybe Dijkstra. *Notes on structured programming.* AcademicPress, 1972.

# II Mise en pratique à travers un exemple

## II.1 Spécification

« Écrire une fonction `nature_triangle(a:float,b:float,c:float)->str` qui prend en entrée trois entiers représentant les longueurs des cotés d'un triangle. Le programme rend un résultat précisant s'il s'agit d'un triangle scalène, càd. "quelconque" (caractère 'S'), isocèle (caractère 'I') ou équilatéral (caractère 'E'). »

## II.2 Tests unitaires

Contrairement à ce que l'on a tendance à faire en pratique au lycée, il est recommandé de produire les cas de tests pour les programmes **avant** de démarrer le code : on les appelle des **tests unitaires**. Etant donné la simplicité de la spécification, il n'y a pas lieu de distinguer "la conformité aux spécifications" et "la fiabilité" (car nous sommes l'unique utilisateur du programme), par contre il faudra tenir compte de la vérification de "la robustesse" du programme.
En pratique : on peut se limiter à une liste rapide, en mode « **boîte noire** », complétée ultérieurement par des tests en mode « **boîte blanche** » lorsque le code sera écrit.
Les cas de tests identifiés par G. Myers dans *The Art of Software Testing*, 1979 sont les suivants (dans un ordre différent de celui du livre) :
Conformité à la spécification/fiabilité

1. Cas scalène valide (1,2,3 et 2,5,10 ne sont pas valides)
2. Cas équilatéral valide
3. Cas isocèle valide (2,2,4 n'est pas valide)
4. Cas isocèle valide avec les trois permutations (e.g. 3,3,4 ; 3,4,3 ; 4,3,3)

Robustesse :

5. Cas avec un nombre erroné de valeurs (e.g. 2 entrées, ou 4)
6. Cas avec une entrée non entière
7. Cas avec une valeur négative
8. Cas avec une valeur à 0
9. Cas avec les trois entrées à 0
10. Cas où la somme de deux entrées est égale à la troisième entrée
11. Cas précédent avec les trois permutations
12. Cas où la somme de deux entrées est inférieure à la troisième entrée
13. Cas précédent avec les trois permutations
14. Pour chaque test, avez-vous spécifié le résultat attendu ?

## II.3 Mise en oeuvre informatique

### II.3.1 Stratégie de test

Les cas 5 et 6 peuvent être automatiquement pris en charge par des modules[1] externes à partir du moment où **la signature de la fonction** est bien renseignée, à savoir les variables avec leur type, mais ce ne sera pas le cas sur les ordinateurs du lycée.
L'utilisation de la commande **assert** permet de tester en même temps les cas 8 et 9, puis en même temps les cas 10, 11, 12 et 13 :

- Cas 8 et 9 :
  `assert a>0 and b>0 and c>0,"les longueurs doivent être positives !"`
- Cas 10 à 13 :
  `assert a+b>c and a+c>b and b+c>a,"inégalité triangulaire non vérifiée"`

**Les tests unitaires** des cas 1 à 4 peuvent également se faire avec la commande **assert**, au sein de l'éditeur de programme. On élabore **un jeu de tests**, à savoir des lignes de tests que l'on peut exécuter à tout moment pour voir si le programme fonctionne. C'est en particulier utile si on est amené à modifier le programme (on ne souhaite pas penser à nouveau à tous les cas à tester).

- Cas 1 : `assert nature_triangle(3,4,5) == 'S'`
- Cas 2 : `assert nature_triangle(1,1,1) == 'E'`
- Cas 3 : `assert nature_triangle(3,3,4) == 'I'`,
  `assert nature_triangle(3,4,3) == 'I'` et `assert nature_triangle(4,3,3) == 'I'`

Ceci donne le programme suivant :

```
2  def nature_triangle(a:float,b:float,c:float)->str:
3      '''renvoie la nature du triangle de longueurs a,b,c sous forme d'un
       caractère: 'I' (isocèle),'E' (équilatéral) ou 'S' (scalène)'''
4      #Tests de robustesse
5      assert a>0 and b>0 and c>0, "les longueurs doivent être positives !"
6      assert a+b>c and a+c>b and b+c>a,"inégalité triangulaire non vérifiée"
7
8      #Code de la fonction
9      if a==b and b==c:
10         return 'E'
11     elif a==b or a==c or b==c:
12         return 'I'
13     else:
14         return 'S'
15
16 #Test unitaires
17 assert nature_triangle(3,4,5) == 'S'
18 assert nature_triangle(1,1,1) == 'E'
19 assert nature_triangle(3,3,4) == 'I'
20 assert nature_triangle(3,4,3) == 'I'
21 assert nature_triangle(4,3,3) == 'I'
```

Notez que l'on a rempli le docstring de la fonction afin d'expliciter entre autres la signification des lettres 'E', 'I' et 'S' (ce qui est évident sur le moment...mais pas après plusieurs mois !).

### II.3.2 A retenir

L'instruction **assert** :

- doit être utilisée pour les tests de robustesse à **l'entrée des programmes**
- doit être utilisée dans la mise en oeuvre des **jeux de test**

1. Mypy, Pyright,...

# III Le partitionnement des tests en domaines

## III.1 Spécification

« Écrire une fonction `nature_triangle_angle(a:float,b:float,c:float)->str` qui prend en entrée trois entiers représentant les longueurs des cotés d'un triangle. Le programme rend un résultat précisant s'il s'agit d'un triangle scalène (caractère 'S'), isocèle (caractère 'I') ou équilatéral (caractère 'E') et si son plus grand angle est aigu, droit ou obtus (<90°, =90°, >90°) et renvoie la réponse correspondante. »

## III.2 Tests unitaires

La méthode de test nécessite de recenser les différents cas par **un découpage en domaines** et de sélectionner **un représentant du domaine**, càd. une valeur qui n'est pas aux limites.
On obtient par exemple ceci :

| | aigu | droit | obtus |
|---|---|---|---|
| scalène | (6,5,4) | (3,4,5) | (5,6,10) |
| isocèle | (6,1,6) | $(\sqrt{2},2,\sqrt{2})$ | (7,4,4) |
| équilatéral | (4,4,4) | impossible | impossible |

Il faut également réaliser **un test aux limites**, par exemple :

- (5.9999,1,6) ou (6.0001,1,6) pour un triangle isocèle ;
- (3,4,4.9999) ou (3,4,5.0001) pour un triangle presque droit ;
- (3.9999,4,4) ou (4,4.001,4) pour un triangle presque équilatéral ;

Plus généralement, quand une variable prend un nombre fini de valeurs, par exemple de $k$ à $n$ (avec $k \leq n$), on teste avec les valeurs de la variable égales à $k-1$ (test de robustesse), à $k$, $k+1$, puis à $n-1$, $n$ et $n+1$ (test de robustesse).

## III.3 Mise en oeuvre informatique

```
24 from math import pi,acos, isclose,sqrt
25
26 def angle_max(a:float,b:float,c:float)->float:
27     '''renvoie la mesure en degrés du plus grand d'un triangle; cette mesure
       est calculée avec la formule d'Al-Kashi; le résultat de acos, par défaut
       en radians, est traduit en degrés'''
28     A = (180/pi)*acos((b**2+c**2-a**2)/(2*b*c))
29     B = (180/pi)*acos((a**2+c**2-b**2)/(2*a*c))
30     C = (180/pi)*acos((a**2+b**2-c**2)/(2*a*b))
31     return max(A,B,C)
32
33 #Test unitaires
34 assert round(angle_max(3,4,5),1) == 90.0 #round(...,1) arrondit à un chiffre
       après la virgule
35 assert round(angle_max(4,10,13),1) == 131.5 #évalué à la calculatrice
```

```python
def nature_triangle_angle(a:float,b:float,c:float)->str:
    '''renvoie un couple (nature, nature de l'angle) où la nature est 'I' (isocèle),'E' (équilatéral) ou 'S' (scalène) et la nature de l'angle est la chaîne de caractères 'aigu','droit','obtus'''
    #Tests de robustesse
    assert a>0 and b>0 and c>0, "les longueurs doivent être positives !"
    assert a+b>c and a+c>b and b+c>a,"inégalité triangulaire non vérifiée"

    #Détermination de la nature du triangle
    nature = ''
    if a==b and b==c:
        nature = 'E'
    elif a==b or a==c or b==c:
        nature = 'I'
    else:
        nature = 'S'

    #Détermination du plus grand angle : aigu, droit, obtus
    nature_angle = ''
    angle = angle_max(a,b,c)
    if isclose(angle,90): #isclose(...,...) teste l'égalité à 9 décimales près
        nature_angle = 'droit'
    elif angle < 90:
        nature_angle = 'aigu'
    else:
        nature_angle = 'obtus'

    return nature, nature_angle

#Test unitaires
#par domaine
assert nature_triangle_angle(6,5,4) == ('S','aigu')
assert nature_triangle_angle(3,4,5) == ('S','droit')
assert nature_triangle_angle(5,6,10) == ('S','obtus')
assert nature_triangle_angle(6,1,6) == ('I','aigu')
assert nature_triangle_angle(sqrt(2),2,sqrt(2)) == ('I','droit')
assert nature_triangle_angle(7,4,4) == ('I','obtus')
assert nature_triangle_angle(4,4,4) == ('E','aigu')
#aux limites
assert nature_triangle_angle(5.9999,1,6) == ('S','aigu') #pas isocèle
assert nature_triangle_angle(6.0001,1,6) == ('S','aigu') #pas isocèle
assert nature_triangle_angle(3,4,4.9999) == ('S','aigu') #pas droit
assert nature_triangle_angle(3,4,5.0001) == ('S','obtus') #pas droit
assert nature_triangle_angle(3.9999,4,4) == ('I','aigu') #pas équilatéral
assert nature_triangle_angle(4.001,4,4) == ('I','aigu') #pas équilatéral
```

# IV Exercices

## IV.1 Tests de robustesse

**Q1** Élaborer un test de robustesse pour la fonction `liste_cubes(n:int)->list` qui renvoie la liste $[0^3, 1^3, ..., n^3]$.

**Q2** Élaborer des tests de robustesse pour la fonction `liste_entiers(n:int,p:int)->list` renvoie la liste $[n, n+1, ..., p-1]$.

**Q3** Élaborer des tests de robustesse pour la fonction `liste_entiers2(n:int,p:int,q:int)->list` qui renvoie la liste des entiers compris entre 1 et $n$ qui ne sont pas multiples de l'un de deux entiers non nuls `p` et `q`. Par exemple, `liste_entiers2(10,2,3)` renvoie $[1; 5; 7]$.

**Q4** Élaborer des tests de robustesse pour la fonction `lendemain(date:tuple)->tuple` qui calcule le lendemain d'une date `(jour:int, mois:int, annee:int)` passée en paramètre. On suppose l'existence d'une fonction `bissextile(annee:int)->bool` qui indique si la variable `annee` est bissextile.

## IV.2 Jeux de test

**Q5** Élaborer des jeux de test pour la fonction `liste_entiers2(n:int,p:int,q:int)->list` de la question précédente.

**Q6** Soit `t` un tableau de nombres. On considère la fonction `est_element(t,x)` qui renvoie True si x est élément de t et False sinon.

- Élaborer un jeu de tests pour cette fonction.
- Comment s'assurer que `x` est du même type que les éléments de `t` ?

**Q7** Proposer des tests pour une fonction `miroir(ch:str)->str` qui prend en argument une chaîne de caractères `ch` et renvoie la chaîne contenant les caractères de `ch` en ordre inverse.

- Écrire une fonction `test(ch:str)->str` qui contient les tests unitaires du programme.
- Donner une version plus simple à base de slicing.

**Q8** On suppose avoir écrit une fonction `mult(x:int,y:int)->int` qui calcule et renvoie le produit de deux entiers `x` et `y`. L'idée est que cette fonction ne se contente pas d'appeler l'opération $*$ mais réalise un autre algorithme, comme la multiplication russe ou la méthode de Karatsuba. Proposer des test de correction pour cette fonction `mult`. *On rappelle que l'instruction* ***`randint(a,b)`*** *tire au hasard un nombre compris entre deux entiers* `a` *et* `b` *(ces deux nombres étant inclus).*

## IV.3 Jeux de test partitionnés en domaines

**Q9** Élaborer un jeu de tests pour la fonction `lendemain(date:tuple)->tuple` qui calcule le lendemain d'une date *(voir IV.1)*. On distinguera en particulier le dernier jour des autres jours ainsi que les différentes "classes" de mois.

## Corrigé

### Q1

```
83 def liste_cubes(n:int)->list:
84     assert n>=0,"n est un entier naturel!"
```

### Q2

```
86 def liste_entiers(n:int,p:int)->list:
87     assert n<p,"nombres incohérents"
```

### Q3

```
89 def liste_entiers2(n:int,p:int,q:int)-> bool:
90     assert n>=1 and p>=1 and q>=1,"les entiers doivent être positifs non
       nuls"
```

### Q4

```
92 def lendemain(date:tuple)->tuple:
93     jour,mois,annee = date
94     mois_trente_jours = {4,6,9,11}
95
96     assert 1<=jour<=31,"jour invalide" #jour ne peut être <1 ou >31
97     assert 1<=mois<=12,"mois invalide" #mois ne peut être <1 ou >12
98
99     if mois in mois_trente_jours: #jour>30 impossible pour un mois à 30
      jours
100         assert jour<=30,"jour invalide pour ce mois"
101     elif mois == 2: #jour>29 impossible pour le mois de février
102         assert jour<=29,"jour invalide pour ce mois"
103         if not bissextile(annee): #jour>28 impossible pour le mois de fé
      vrier d'année non bissextile
104             assert jour<=28,"jour invalide pour ce mois"
```

**Q5** Les tests de robustesse ayant été précédemment définis, on suppose $n$, $p$ et $q$ des entiers naturels non nuls. On peut supposer par la suite $p \leq q$, quitte à supposer que l'on effectuera les mêmes tests unitaires en inversant le rôle de $p$ et $q$. On peut alors distinguer les cas suivants.

Les cas aux limites :

(a) $n = 1$ et $p$, $q$ quelconques : on doit obtenir [1],

(b) $n$ quelconque, $p = 1$ : on doit obtenir un tableau vide,

(c) $n \leq p$ : on doit obtenir tous les entiers de 1 à $n$,

(d) $n = p + 1$ : on doit obtenir tous les entiers de 1 à $n$ hormis $p$.

Les cas non aux limites :

(e) $q$ est multiple de $p$ (on peut éventuellement considérer le cas où $p = q$ à part) : on doit obtenir les entiers de 1 à $n$ qui ne sont pas multiples de $p$. On peut le vérifier par une boucle sur tous les éléments du résultat obtenu,

(f) $q$ n'est pas multiple de $p$ : On peut le vérifier par une boucle sur tous les éléments du résultat obtenu : aucun élément n'est multiple de $p$, aucun élément n'est multiple de $q$.

Le jeu de tests unitaires se trouve dans le code qui suit.

```
#Tests unitaires de liste_entiers2
n,p,q = 1,2,3; assert liste_entiers2(n,p,q) == [1] #a
n,p,q = 3,1,4; assert liste_entiers2(n,p,q) == [] #b
n,p,q = 3,4,1; assert liste_entiers2(n,p,q) == [] #b
n,p,q = 6,7,8; assert liste_entiers2(n,p,q) == [1,2,3,4,5,6] #c
n,p,q = 6,8,7; assert liste_entiers2(n,p,q) == [1,2,3,4,5,6] #c
n,p,q = 8,7,10; assert liste_entiers2(n,p,q) == [1,2,3,4,5,6,8] #d
n,p,q = 15,5,10; t = liste_entiers2(n,p,q) #e
for i in range(len(t)):
    assert t[i]%p != 0
n,p,q = 10,2,3; t = liste_entiers2(n,p,q) #f
for i in range(len(t)):
    assert t[i]%p != 0
    assert t[i]%q != 0
```

**Q6** On peut penser aux tests suivants :

(a) `t` est le tableau vide : le résultat sera toujours `False`.

(b) `x` n'est pas un élément du tableau : le résultat sera toujours `False`.

(c) `x` est un élément du tableau `t=[x]` (un seul élément) : le résultat sera `True`.

(d) `x` est un élément du tableau, situé en première ou dernière position, ou à une position quelconque : le résultat sera toujours `True`.

(e) `x` se trouve plusieurs fois dans le tableau (le premier test ne doit pas influencer les suivants) : le résultat sera `True`.

Notons que l'entier `1`, par exemple, est dans le tableau de `float` `L=[1.0]`. Sans plus de précision dans l'énoncé, ce n'est pas choquant. Cependant, si on voulait vérifier que `x` est du même type que les éléments du tableau, on pourrait modifier le test dans la boucle de recherche, en écrivant `if t[i] == x and type(t[i]) == type(x)`.

Le jeu de tests unitaires se trouve dans le code qui suit.

```
#Tests unitaires de est_element
t = [];x = 1
assert est_element(t,x) == False #a
t = [1,2,3,1];x = 5
assert est_element(t,x) == False #b
t = [1];x = 1
assert est_element(t,x) == True #c
t = [1,2,3,1];x = 1
assert est_element(t,x) == True #d
t = [1,2,3,4];x = 4
assert est_element(t,x) == True #d
t = [1,2,3,4];x = 2
assert est_element(t,x) == True #d
t = [1,2,1,1];x = 1
assert est_element(t,x) == True #e
```

**Q7** On peut penser aux tests suivants :

(a) la chaîne `ch` est vide : la chaîne inversée doit être vide également,

(b) la chaîne `ch` n'est pas vide : on vérifie que la chaîne inversée est de même longueur,

(c) la chaîne `ch` n'est pas vide : on vérifie que chaque caractère a bien été inversé, en particulier quand la longueur de la chaîne est impaire (il se pourrait que la fonction `miroir` se base sur la parité).

On peut proposer la fonction suivante :

```
139 def test(ch:str)->str:
140     n = len(ch)
141     m = miroir(ch)
142     if ch == '':
143         assert m == '' #a
144     else:
145         assert len(m) == n #b
146         for i in range(n):
147             assert m[i] == ch[n-1-i] #c
148 
149 #ou plus simplement
150 def test(ch:str)->str:
151     assert miroir(ch) == ch[::-1]
```

Remarque :
En fait, la fonction précédente a un sens si on veut la réutiliser à de multiples reprises. On peut par exemple le faire en générant au hasard une centaine de chaînes de caractères de longueur croissante constituées disons, des lettres 'A' à 'Z'.

```
from random import randint

def chaine_aleatoire(n):
    return ''.join([chr(randint(65,91)) for _ in range(n)])

for n in range(100):
    test(chaine_aleatoire(n))
```

**Q8** On peut penser aux tests suivants :

(a) la mutiplication est associative càd. `mult(mult(x,y),z)==mult(x,(mult(y,z))`,

(b) la mutiplication est commutative càd. `mult(x,y)==mult(y,x)`,

(c) la mutiplication de `x` par `1` renvoie toujours `x`,

(d) la mutiplication par `0` renvoie toujours `0`, quelle que soit la valeur de l'autre variable

(e) pour tout `x` non nul, `mult(x,1/x)==1`

(f) la multiplication est distributive par rapport à l'addition
càd. `mult(x,y+z)==mult(x,y)+mult(x,z)`

(g) enfin, on doit vérifier que la mutiplication de la fonction donnée renvoie bien le même résultat que celui de la multiplication de base, à savoir que `mult(x,y)==x*y`, en particulier avec des valeurs négatives.

On peut proposer le jeu de tests suivant :

```
153 from random import randint
154 from math import isclose
155 
156 for _ in range(1000):
157     x = randint(-100,100)
158     y = randint(-100,100)
159     z = randint(-100,100)
160     assert mult(mult(x,y),z) == mult(x,mult(y,z)) #a
161     assert mult(x,y) == mult(y,x) #b
162     assert mult(x,1) == x #c
163     assert mult(x,0) == 0 #d
164     if x != 0:
165         assert isclose(mult(x,1/x),1) #e
166     assert mult(x,y+z) == mult(x,y)+mult(x,z) #f
167     assert mult(x,y) == x * y #g
```

**Q9** On établit un jeu de tests en prenant un représentant de chacun des domaines suivants :

| | mois 1,3,5,7,8,10 | mois 12 | mois 4,6,9,11 | mois 2 année non bis. [2] | mois 2 année bis. [2] |
|---|---|---|---|---|---|
| jour courant [3] | 15 | 15 | 15 | 15 | 15 |
| dernier jour | 31 | 31 | 30 | 28 | 29 |

A cela, il faut ajouter des tests aux limites, ce qui signifie ici de prendre le 1er jour, le 2e jour et l'avant-dernier jour de chaque mois.

2. "bis". signifiant "bissextile"
3. sauf le dernier jour du mois