Chapitre 1 - Mise au point et gestion des bugs

I - Introduction

Les bugs sont chaque année l'occasion de millions d'euros de perte pour les entreprises et les États mais parfois aussi à l'origine de morts civiles ou militaires.
De manière plus anecdotique, un bug peut tout simplement produire un résultat légèrement erroné, un effet de bord non souhaité, voire fournir un résultat inattendu mais très intéressant. De manière générale, on préfère livrer un programme exempt de bug.
Un tel programme est donc un programme qui :

- fait ce qu'il est censé faire ;
- ne fait pas ce qu'il n'est pas censé faire.

À noter qu'un programme n'étant pas censé "planter", un programme exempt de bug est donc, entre autres, un programme qui ne doit pas "planter".

Pour savoir si un programme est buggué, il faut donc avoir une idée de ce qu'un programme est censé faire. C'est ce qu'on appelle une spécification. Les spécifications peuvent prendre de multiples formes : un simple titre, un document ≪ littéraire ≫ de 300 pages, une spécification formelle écrite dans un langage dédié, un ensemble d'exemples d'utilisation, un ensemble de cas de tests, etc.

Pour éviter qu'un programme ne soit buggué, on cherche d'abord à éviter l'écriture de bugs. Pour ce faire, on peut utiliser des méthodes de développement particulières, ou des outils d'aide au développement. Pourtant, malgré tout, on fini en général tôt ou tard par introduire des bugs dans un programme. Il faut alors :

- savoir mettre en évidence le bug et le reproduire ;
- comprendre la cause du bug ;
- corriger le bug.

C'est ce qu'on appelle la mise au point, ou le débuggage.

II - Mise au point des programmes

1) Mise en évidence du bug

1).1 Le bug fait planter le programme

Lorsqu'un bug mène au plantage du programme, en général, un message d'erreur assez clair associé au plantage est fourni par l'interpréteur Python. Par ailleurs, l'utilisateur est informé de l'endroit où le programme a planté. Cela aide grandement à remonter à la source de l'erreur.
Exemple : test_python.py

```
def moyenne(notes):
    nb_notes = 0
    somme_notes = 0
    for note in notes:
        somme_notes += note
        nb_notes += 1
    return somme_notes / nb_notes

notes = []
print(moyenne(notes))
```

À l'exécution, on obtient l'affichage suivant :

```
Traceback (most recent call last):
  File "/home/mikael/test_python.py", line 10, in <module>
    print(moyenne(notes))
  File "/home/mikael/test_python.py", line 7, in moyenne
    return somme_notes / nb_notes
ZeroDivisionError: division by zero
```

Comme on le voit, l'erreur (« ZeroDivisionError : division by zero ») et le lieu où l'erreur est survenue (« File «test_python.py», line 7 in `moyenne` ») sont clairement indiqués.
C'est même toute la pile d'exécution qui est affichée. Ainsi, on sait que lorsque le programme a planté, il était en train d'exécuter la méthode `moyenne` appelée à la ligne 10 du fichier.

Parmi les erreurs classiques, on trouve notamment :

- division par zero : `ZeroDivisionError`
- accès hors des bornes d'une liste : `IndexError`
- erreur de nom de variable : `NameError`
- erreur de nom de méthode ou d'attribut : `AttributeError`
- appel récursif trop profond : `RecursionError`
- modification d'un objet non mutable : `TypeError`

Lorsqu'un bug ne mène pas à un plantage du programme mais à la production d'un résultat erroné, il faut arriver à reproduire le bug. Lorsque le bug est systématique, cela est facile. Mais parfois, le bug ne se produit qu'avec certaines valeurs en entrée.
D'autres fois, l'occurrence du bug semble même totalement aléatoire (avec une même donnée d'entrée, le programme ne plante pas toujours). Dans ce dernier cas, la mise en évidence du bug s'avère très compliquée.

Un bug est pourtant lié à une condition bien précise. Si le bug semble aléatoire, c'est qu'il dépend d'une condition externe au programme (contenu d'un fichier, information dans une base de données, connexion réseau, saisie de l'utilisateur, tirage aléatoire...)

1).2 Le bug donne un résultat erroné

Prenons la fonction suivante, censée donner le maximum d'une liste de nombres :

```
def maximum(liste):
        valeur_max = 0
        for valeur in liste:
                if valeur > valeur_max:
                        valeur_max = valeur
        return valeur_max
```

De premiers résultats semblent concluants :

```
maximum([2, 4, 1, 9, 10, 4, 6])
10
maximum([6, -1, 2, 8, -6])
8
```

Cette fonction présente pourtant quelques problèmes :

```
maximum([-2, -5, -1])
0
```

⟶ La fonction aurait dû renvoyer -1.

```
maximum([])
0
```

⟶ Le maximum d'une liste vide n'a pas de sens.

```
maximum([4,"toto", 3])

 Traceback (most recent call last):
  File "/home/mikael/test_python.py", line 8, in <module>
    maximum([4,"toto", 3])
  File "/home/mikael/test_python.py", line 4, in maximum
    if valeur > valeur_max:
TypeError: '>' not supported between instances of 'str' and 'int'
```

⟶ Le programme plante car notre fonction est adaptée à une liste d'éléments comparables.

Le dernier cas présenté faisant planter le programme, il entre dans la catégorie de bugs présentée dans la section précédente.

Le deuxième cas est assez facile à mettre en évidence : il survient systématiquement avec une liste vide.

Le premier cas est en revanche plus difficile à mettre en évidence. Il faut que la liste ne comprenne que des nombres négatifs pour constater le bug, ce qui n'est pas forcément facile à comprendre.

2) Comprendre la cause du bug

Pour comprendre la cause d'un bug, il faut commencer par comprendre comment s'est déroulée l'exécution du programme qui a amené au bug.

Aussi, savoir simuler (à la main) l'exécution pas-à-pas d'un programme est une compétence indispensable.

Pour comprendre la cause d'un bug, il est par ailleurs nécessaire de connaître, pour chaque fonction, quelles sont les préconditions supposées. L'utilisation de clauses assert permet de les formaliser clairement, comme cela a pu être étudié en classe de Première. Lors de l'exécution en mode pas-à-pas, il est primordial de se demander, pour chaque fonction que l'on se prépare à exécuter, si les préconditions sont bien vérifiées. Si la définition d'un assert semble trop compliquée, on peut le remplacer par une docstring respectant une forme bien précise (par exemple, commençant par #assert), avec une définition claire de la précondition de la fonction.

Comment exécuter pas-à-pas une fonction (ou un programme)

1. Avant d'exécuter un programme, il est primordial de commencer par déterminer le résultat attendu.
2. Ecrire un tableau, ou chaque colonne représente l'état mémoire des variables utilisées dans le programme ou le résultat d'un test conditionnel. Chaque ligne réprésente une itération du programme.**La ligne 0 correspond à l'état initial des variables.**

Reprenons l'exemple de la fonction de recherche du maximum dans un tableau et exécutons pas-à-pas la fonction avec la tableau `[-2, -5, -1]`.

Etape 1 - résultat attendu : La fonction doit retourner la valeur -1

Etape 2 - tableau représentant l'état des variables et des tests conditionnels
Dans cette fonction, il y a deux variables `valeur` et `valeur_max` et un test conditionnel `valeur > valeur_max`.

Itération	`valeur`	`valeur_max`	`valeur > valeur_max`
Itération 0	X	0	X
Itération 1	-2	0	False
Itération 2	-5	0	False
Itération 3	-1	0	False

Dans cette exemple, on s'aperçoit que le test conditionnel retourne toujours `False`, donc `valeur_max` n'est jamais mis à jour.
Le problème vient de l'initialisation à 0 de la variable `valeur_max`.
Une **correction possible** serait d'initialiser `valeur_max` à `liste[0]`.

```
def maximum(liste):
        valeur_max = liste[0]
        for valeur in liste:
                if valeur > valeur_max:
                        valeur_max = valeur
        return valeur_max
```

Check-list de points à vérifier

- initialisation correcte des variables ;
- accès au bon indice dans un tableau (dans les bornes, cases numéro de 0 à taille -1) ;
- noms de variables corrects ;
- conditions de tests avec les bons opérateurs booléens (et/ou par exemple) ;
- opérateurs de comparaison bien choisis (notamment sens large / sens strict, égalité avec delta sur les réels) ;
- terminaison des boucles ;
- appel d'une fonction avec les bons paramètres et dans le bon ordre ;
- traitement correct aux limites (0 sur les nombres, listes vides, etc.).

Il existe d'autres techniques pour essayer de comprendre l'origine d'un bug :

- **rajouter temporairement des affichages dans le code** : cette technique, qui peut sembler primaire, est pourtant bien utile. Elle apparaît souvent comme plus rapide que le pas-à-pas. En revanche, elle suscite moins de réflexion, et ne facilite pas la compréhension de la cause du bug.
- **utiliser un débuggueur** : les IDE 2 proposent souvent des débuggueurs : ce sont des outils qui permettent de faire faire du pas-à-pas à l'ordinateur. On peut ainsi suivre l'exécution d'un programme instruction par instruction, et connaître au fur et à mesure de l'exécution la valeur des différentes variables.
  Ces outils sont dotés de fonctionnalités très intéressantes. Par exemple, pour chaque appel de fonction, on peut préciser si l'on veut une exécution en pas-à-pas ou non ; on peut à tout moment reprendre l'exécution « normale » jusqu'à tomber sur un éventuel point d'arrêt où l'on repasse en pas-à-pas, etc. Par rapport au pas-à-pas manuel, on gagne du temps en exécution, mais on réfléchit moins, et on risque de mettre plus de temps à comprendre la vraie cause du bug. Sur de petits exemples, Python Tutor 3 peut être utilisé comme débuggueur.

III - Écrire des programmes « faciles » à mettre au point

Un programme est d'autant plus facile à mettre au point qu'il respecte certains principes d'écriture. Voici quelques consignes que l'on peut essayer de faire respecter.

1) Donner des noms explicites

Donner aux variables et fonctions des noms explicites permet d'auto-documenter le rôle de la variable ou de la fonction. Les commentaires deviennent alors souvent superflus. L'intérêt d'un code auto-documenté par rapport à des commentaires « classiques » est que la « documentation » reste toujours à jour. La compréhension du code, et donc la recherche d'un éventuel bug, est grandement facilitée.
Comparons les 2 exemples suivants :

```
def f(x):
        a = 0
        for b in x:
                if b > a:
                        a = b
        return a

def maximum(liste):
        valeur_max_vue = 0
        for valeur_courante in liste:
                if valeur_courante > valeur_max_vue:
                        valeur_max_vue = valeur_courante
        return valeur_max_vue
```

Les 2 fonctions font exactement la même chose. Mais tandis que la première cache bien son jeu, la seconde est beaucoup plus claire. Par ailleurs, l'affectation `valeur_max_vue = 0` apparaît tout de suite problématique : on n'a pas encore vu de valeur ; comment expliquer alors que la valeur maximale vue vaille 0 ? Par ailleurs, la seconde fonction se passe de commentaires.

2) Limiter les imbrications profondes

Voici un exemple de fonction effectuant un produit matriciel :

```
def produit_matriciel(matrice_A, matrice_B):
        nb_lignes_A = len(matrice_A)
        nb_colonnes_A = len(matrice_A[0])
        nb_colonnes_B = len(matrice_B[0])
        resultat = [[0]*nb_colonnes_B for i in range(nb_lignes_A)]
        for ligne in range(nb_lignes_A):
                for colonne in range(nb_colonnes_B):
                        for compteur in range(nb_colonnes_A):
                                resultat[ligne][colonne] += matrice_A[ligne][
                                    compteur] * matrice_B[compteur][colonne]
        return resultat
```

Cette fonction réalise bien un calcul matriciel, mais la présence de 3 boucles imbriquées rend le code peu évident à lire. En cas d'erreur, un pas-à-pas amène à exécuter toute la méthode, ce qui est long, fastidieux et inefficace. Dans un tel cas, il est préférable d'isoler la boucle la plus interne dans une fonction dont le nom est suffisamment clair pour comprendre le rôle de cette boucle.
On peut ainsi réécrire le programme de calcul de produit matriciel ainsi :

```
def calcul_case(matrice_A, matrice_B, ligne, colonne):
        valeur_case = 0
        for compteur in range(len(matrice_A[0])):
                valeur_case += matrice_A[ligne][compteur] * matrice_B[compteur][
                    colonne]
        return valeur_case

def produit_matriciel(matrice_A, matrice_B):
        nb_lignes_A = len(matrice_A)
        nb_colonnes_B = len(matrice_B[0])
        resultat = [[0]*nb_colonnes_B for i in range(nb_lignes_A)]
        for ligne in range(nb_lignes_A):
                for colonne in range(nb_colonnes_B):
                        resultat[ligne][colonne] = calcul_case(matrice_A, matrice_B
                            ,ligne, colonne)
        return resultat
```

Ainsi, en cas de bug, on peut traiter séparément le calcul de la valeur d'une case. Si celui-ci semble correct, la fonction `produit_matriciel` peut être exécutée en pas-à-pas sans détailler le calcul de la valeur d'une case.

D'une manière générale, il faut éviter au maximum les imbrications de blocs, qu'il s'agisse de blocs liés à des boucles ou à des conditionnelles, par exemple en multipliant les fonctions intermédiaires. On peut ainsi réécrire la fonction `produit_matriciel` ci-dessus en supprimant l'imbrication de boucles restante. L'écriture de tests pertinents est d'ailleurs facilitée avec des méthodes plus élémentaires (voir ressource sur les tests).

3) Simplifier la lecture des expressions conditionnelles

On est fréquemment amené à écrire des expressions conditionnelles complexes (avec plusieurs opérateurs logiques). Non seulement ces expressions ne sont pas évidentes à relire (ce qui doit pourtant être fait lorsqu'on recherche un bug), mais leur sens même n'est pas évident. Aussi, il est conseillé, dès qu'une expression conditionnelle fait intervenir plus d'un opérateur booléen, de passer par une variable ou une fonction donnant un sens à l'expression booléenne.

Prenons le cas de la fonction suivante, qui recherche un élément dans une liste ordonnée et renvoie la première occurrence s'il est présent :

```
def recherche(liste, valeur):
  indice = 0
  element_trouve = None
  while (indice < len(liste) and element_trouve is None and liste[indice]<= valeur)
      :
    if liste[indice] == valeur:
      element_trouve = liste[indice]
    indice += 1
  return element_trouve
```

La condition de boucle est particulièrement compliquée. Si le programme s'avère erroné, il est difficile de savoir si l'erreur vient de la condition ou du corps de la boucle.
Il est important de séparer le sens de la condition qui amène à boucler de la façon dont on évalue cette condition. On peut alors plutôt écrire cette fonction ainsi :

```
def recherche(liste, valeur):
  indice = 0
  element_trouve = None
  termine = len(liste) == 0 or liste[0] > valeur
  while not(termine):
    if liste[indice] == valeur:
      element_trouve = liste[indice]
    else:
      indice += 1
    termine = element_trouve != None \
            or indice == len(liste) \
            or liste[indice] > valeur
  return element_trouve
```

Dans cet exemple, on sort de la boucle lorsque `termine` prend la valeur `True`.
Ainsi, lors de la phase de débuggage, quand on analyse le corps de la boucle, on sait que l'on n'a pas terminé, mais on ne se demande pas pourquoi. Inversement, il est facile de comprendre si la variable affectée à la variable `termine` est la bonne.

4) Limiter la taille des fonctions

Plus une fonction est petite, plus elle est facile à débugguer.
En effet, une petite fonction fait moins de choses, donc contient moins de causes potentielles de bugs. Par ailleurs, il est plus facile de débugguer une fonction que l'on peut voir entièrement sur son écran.
Ainsi, dans l'idéal, une fonction doit faire 10 lignes au maximum. Mais pour rester pragmatique, on peut fixer la limite à 20 lignes.
Si l'on dépasse les 20 lignes, alors il faut éclater la fonction en introduisant des fonctions intermédiaires.

IV - Tests sur les entrées d'un code : les assertions

Si une fonction est bien documentée et qu'elle est correcte, on peut considérer que le travail du programmeur (de la fonction) a été fait correctement.
Cependant, rien n'empêche d'utiliser cette fonction avec des paramètres d'entrée ne respectant pas les préconditions définies dans la docstring : mais l'erreur incombe alors à l'utilisateur qui, soit n'a pas lu correctement la documentation, soit a volontairement testé la fonction avec des valeurs non admises.

Pour parer à cela, le programmeur a la possibilité d'utiliser la construction **assert** suivie d'une **condition à tester**.

Si cette condition est vraie, alors il ne se passe rien et le programme poursuit son exécution :

```
# Cas d'un test valide
a = -2
assert a < 0
print(a)  # est execute car l'assertion precedente est vraie
#OUTPUT
-2
```

En revanche, si la condition est fausse, alors une erreur (de type `AssetionError`, soit erreur d'assertion) est détectée et stoppe l'exécution du reste du programme.

Par exemple, le test invalide suivant produit l'affichage d'un message d'erreur et stoppe le programme (ici la dernière ligne n'est pas exécutée)

```
# Cas d'un test invalide
a = -2
assert a >= 0, "le nombre a n'est pas positif"
print(a)  # n'est pas execute car l'assertion precedente est fausse
#OUTPUT
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-16-2b60d98e5060> in <module>
      1 # Cas d'un test invalide
      2 a = -2
----> 3 assert a >= 0
      4 print(a)  # n'est pas execute car l'assertion precedente est fausse

AssertionError:  le nombre a n'est pas positif
```

- Test avec les opérateurs classiques : `assert ==, <, >, != ...`
- Appartenance à un type spécifié :

  → `assert isinstance(n,int)` : vérifie si la variable `n` est type entier.

  → `assert isinstance(x,str)` : vérifie si la variable `x` est type chaine de caractère.
- Appartenance à une liste de valeur : `assert n in [1, 2, 3]` : vérifie si la variable `n` appartient à la liste `[1, 2, 3]`.

Exemple

```
def inverse(n):
  '''
  calcule l'inverse d'un nombre entier positif
  entree : un nombre entier strictement positif
  sortie : un nombre de type flottant egal a l'inverse de n
  '''

  assert n !=0, "Le nom bre ne doit pas etre nul"
  assert n > 0, "Le nombre doit etre strictement positif"
  assert isinstance(n,int), "Le nomre doit etre un entier"

  return 1/n
```

V - Tests du résultat d'une fonction : les tests unitaires

La notion de tests est fondamentale en informatique : certains systèmes ne peuvent se contenter d'un fonctionnement approximatif mais doivent au contraire être robustes, c'est à dire fonctionner correctement dans toutes les situations possibles.

Pour prouver qu'une fonction fait toujours correctement le travail pour lequel elle est prévue, il faudrait théoriquement la tester avec tous les arguments possibles et imaginables ; c'est bien entendu impossible...

On peut cependant se contenter de tester son bon fonctionnement sur quelques arguments bien choisis : on parle alors de **tests unitaires**.

Une méthode de développement appelé TDD (Test Driven Design) préconise d'ailleurs d'écrire D'ABORD des tests avant même le code d'une fonction...

1) Tests unitaires avec le module `doctest`

En Python, on peut utiliser le module `doctest` pour réaliser ces tests unitaires ; il permet d'indiquer dans la **docstring** de la fonction des tests à réaliser et le résultat attendu si elle fonctionne bien. Si ce n'est pas le cas, un message est alors affiché signalant qu'il faut corriger son code !

Voici un exemple avec une fonction qui donne le quotient et le reste de deux entiers :

```
import doctest

def division_euclidienne(n1, n2):
    '''
    calcule le quotient et le reste de la division euclidienne de deux nombres
    entree : deux nombres entiers n1 et n2
    sortie : deux nom bres entiers , dans l'ordre le quotient et le reste de la
       division euclidienne de n1 par n2

    Test unitaires
    >>> division_euclidienne(5,2)
    (2, 1)

    >>> division_euclidienne(2,1)
    (2, 0)

    >>> division_euclidienne(11,3)
    (3, 2)
    '''

    quotient = n1 // n2
    reste = n1 % n2
    return quotient,reste

#programme principal

doctest.testmod()  #execution des tests unitaires
```

Dans la docstring, un test unitaire correspond aux 3 chevrons >>> suivis de l'appel de la fonction avec des arguments particuliers ; on indique en dessous le résultat attendu. Si les tests unitaires sont validés, rien ne s'affiche. Par contre, en cas de mauvais fonctionnement, un ou plusieurs avertissement(s) s'affichent indiquant quel(s) test(s) n'ont pas été réussi(s).

Exemple avec la fonction précédente buguée

```
import doctest

def division_euclidienne(n1, n2):
    '''
    calcule le quotient et le reste de la division euclidienne de deux nombres
    entree : deux nombres entiers n1 et n2
    sortie : deux nom bres entiers , dans l'ordre le quotient et le reste de la
       division euclidienne de n1 par n2

    Test unitaires
    >>> division_euclidienne(5,2)
    (2, 1)

    >>> division_euclidienne(2,1)
    (2, 0)

    >>> division_euclidienne(11,3)
    (3, 2)
    '''

    quotient = n1 / n2
    reste = n1 % n2
    return quotient,reste

#programme principal

doctest.testmod()  #execution des tests unitaires

#OUTPUT
**********************************************************************
File "/home/mikael/test_python.py", line 10, in __main__.division_euclidienne
Failed example:
    division_euclidienne(5,2)
Expected:
    (2, 1)
Got:
    (2.5, 1)
**********************************************************************
File "/home/mikael/test_python.py", line 13, in __main__.division_euclidienne
Failed example:
    division_euclidienne(2,1)
Expected:
    (2, 0)
Got:
    (2.0, 0)
**********************************************************************
File "/home/mikael/test_python.py", line 16, in __main__.division_euclidienne
Failed example:
    division_euclidienne(11,3)
Expected:
    (3, 2)
Got:
    (3.6666666666666665, 2)
**********************************************************************
1 items had failures:
   3 of   3 in __main__.division_euclidienne
***Test Failed*** 3 failures.
```

La fonction renvoyant un flottant alors que c'est un entier qui est attendu, une erreur est signalée.

2) Tests unitaires avec assertions

Dans certains cas les assertions peuvent être détournées de leur utilisation initiale pour réaliser des tests unitaires. Dans ce cas ils ne vérifient plus les préconditions de la fonction mais ses posts-conditions.
Les assertions sont alors regroupées dans une fonction de test.
Par exemple si on veut tester la fonction somme() ci-dessous :

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

On peut imaginer la fonction de test :

```
def test_somme():
        assert somme(0,0) == 0
        assert somme(1,1) == 2
        assert somme(1,-1) == 0
        assert somme(12,-3) == 9
        assert somme(100,50) == 150
```