MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

MISE AU POINT DES PROGRAMMES, GESTION DES BUGS

SOMMAIRE

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 (voir http://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html par exemple). 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[1], 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**.

Mise au point des programmes

Mise en évidence du bug

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 : prog1.py

```
1    def moyenne(notes):
2        nb_notes = 0
3        somme_notes = 0
4        for note in notes:
5            somme_notes += note
6            nb_notes += 1
7        return somme_notes / nb_notes
8
9    notes = []
10   print(moyenne(notes))
```

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

```
Traceback (most recent call last):
  File "prog1.py", line 10, in <module>
    print(moyenne(notes))
  File "prog1.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 «prog1.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…)

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, 8, 2, -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 "<stdin>", line 1, in <module>
  File "maximum.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.

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.

Exemple :

```
def maximum(liste):
«»»
Fonction renvoyant le maximum d'une liste.
assert la liste doit contenir des nombres exclusivement positifs ou nuls
«»»
    valeur_max = 0
    for valeur in liste:
        if valeur > valeur_max:
            valeur_max = valeur
    return valeur_max
```

En mode pas-à-pas, avant d'exécuter une fonction, il est primordial de commencer par déterminer le résultat attendu. Cela permet non seulement de bien réfléchir à ce que la fonction doit produire, mais également de ne pas avoir un raisonnement biaisé par le code éventuellement erroné de la fonction.

Pour découvrir le principe du pas-à-pas ou pour un problème compliqué, cela peut être pratiqué à plusieurs :

- une personne, le développeur, lit les instructions une à une ;
- une personne, au tableau ou sur tout autre support partagé, simule l'évolution de l'état de la mémoire (valeur des différentes variables) ;
- deux autres personnes ou plus contrôlent ce qui est fait.

Les causes des bugs sont souvent similaires. On peut ainsi établir une *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. Elle permet cependant souvent de cerner assez rapidement où le pas-à-pas doit être fait. Dans le cadre du paradigme objet, définir des méthodes `__str__` dans les classes facilite grandement l'utilisation de ces affichages ;

- **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.

Correction du bug

Lorsqu'un bug a été compris, il faut le corriger non seulement pour le cas ayant généré le bug, mais également pour tous les cas similaires susceptibles d'engendrer le même type de bug. Par ailleurs, il faut éviter que la correction du bug détecté n'engendre un nouveau bug sur des cas qui fonctionnaient auparavant. C'est ce qu'on appelle **assurer la non-régression**.

Exemple : Dans le cas de la fonction `maximum` présentée plus haut, un bug a été détecté avec la donnée d'entrée : `[-2, -5, -1]` . On peut donc corriger la fonction ainsi :

```
def maximum(liste):
    valeur_max = -1
    for valeur in liste:
        if valeur > valeur_max:
            valeur_max = valeur
    return valeur_max
```

En modifiant ainsi la fonction, le bug détecté semble être corrigé : on obtient le bon résultat pour la liste qui avait mis en évidence le bug. Pourtant, cette fonction reste bugguée : le résultat donné pour la liste `[-2, -5]` sera `-1`, ce qui n'est toujours pas correct.

Aussi, avant de corriger un bug, il faut identifier plusieurs cas qui fonctionnent et qui ne fonctionnent pas. Après la correction du bug, il faut vérifier que tous ces cas fonctionnent désormais.

2. *Integrated Development Environment* : un environnement de développement intégré est un outil tel que PyCharm ou Eclipse qui vous propose un ensemble d'outils intégrés pour faciliter le développement.
3. http://pythontutor.com/

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.

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.

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).

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.

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.

Limiter les bugs dans les programmes

Dans l'idéal, plutôt que de débugguer des programmes, il est plus judicieux d'écrire des programmes exempts de bugs. Bien sûr, rien ne permet vraiment de garantir qu'un programme ne présente aucun bug. Toutefois, certains principes permettent de limiter les risques :

- appliquer les principes décrits dans la partie précédente (« Écrire des programmes faciles à mettre au point ») ;
- associer des tests aux programmes que l'on développe (voir la ressource éduscol consacrée à l'écriture des tests) ;
- expliciter clairement les préconditions des fonctions avec des `assert` ;
- démontrer la terminaison des boucles et des appels récursifs.

Bibliographie

Why programs fails, A guide to systematic debugging second edition, Andreas Zeller, Morgan Verlag, 2009, ISBN 978-0-12-374515-6.

Coder proprement, Robert C. Martin, Pearson, 2019, ISBN 978-2-326-00227-2.

The Art of Software Testing, Glenford J. Meyers et al., https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxtcnNoZWhyaWNvbXxne-DoyN2YyYjJlNWEyZmY1M2Q2.