Mise au point des programmes

Sommaire

Sommaire

Quelques bugs célèbres

- De juin 1985 à janvier 1987, aux États-Unis et au Canada, la machine de radiothérapie Therac-25 provoque ou accélère la mort d'au moins 5 patients (et blesse gravement plusieurs autres) par surirradation. L'origine du problème se trouvait dans l'interface homme-machine, en particulier dans une mauvaise gestion du clavier, qui, par appuis rapides, pouvait provoquer l'émission de faisceaux d'électrons trop puissants.
- le 25 février 1991, durant la guerre du Golfe, un missile anti-missile américain Patriot échoue dans l'interception d'un missile irakien Scud, ce qui cause la mort de 28 soldats et blesse une centaine de civils. Une erreur d'arrondi dans l'horloge du missile Patriot, provoquant un décalage de $0,34$ s, est à l'origine de son échec d'interception du Scud.
- En 1994, le professeur Thomas Nicely, découvre un bug dans le nouveau Pentium Pro d'Intel, dans des erreurs lors de divisions. L'origine du problème est trouvée dans un circuit de l'unité de calcul. Au final, ce bug coûte 475 millions de dollars à Intel.

- le 4 juin 1996, la fusée Ariane 5 explose environ 40 secondes après son décollage de Kourou, avec 4 satellites à son bord. Un dépassement d'entier dans les registres mémoires des calculateurs de la fusée en est à l'origine.
- Le bug de l'an 2000 : de nombreuses applications, concues dans les années 1960-80, ont limité le format des dates à deux chiffres pour l'année. À l'approche de l'année 2000, un gros effort mondial de réécriture de code est fait pour que ce problème n'affecte pas des systèmes critiques : transactions financières, systèmes de contrôle aérien,...
- le 31 décembre 2008 à minuit, tous les lecteurs MP3 Zune, commercialisés par Microsoft, se sont arrêtés de fonctionner. Ils ne pouvaient être remis en route ce même jour. Ce n'est que le lendemain qu'ils pouvaient être remis en route [1].

1. Voir dans l'exercice 9 le code source incriminé, qui gérait mal le cas des années bissextiles.

Sommaire

Rappels

Rappels sur les «bonnes pratiques» déjà vues concernant le travail de programmation :

1. choisir des noms explicites pour les variables, fonctions, classes, méthodes
2. documenter ses programmes :
   - utiliser les docstrings et indiquer systématiquement : les conditions d'utilisation de la fonction (precondition, portant sur les arguments) et ce qui est renvoyé (postcondition)
   - utiliser les commentaires en ligne lorsqu'ils sont utiles [2]
3. adopter une méthode de programmation «défensive» : utiliser entre autre l'instruction `assert` [3] pour :
   - tester une précondition
   - interrompre le programme lorsqu'elle n'est pas satisfaite
   - afficher un message d'erreur explicite

2. Attention, un commentaire inutile ou mal rédigé peut plus nuire qu'autre chose. Par ailleurs, un bon code est sensé se suffire à lui-même.
3. Plutôt dans la phase de mise au point du programme.

Exemple : passer du programme ci-dessous

```
def maximum_tableau(t):
    m = 0
    for i in range(len(t)):
        if t[i] > t[m]:
            m = i
    return m
```

à celui-ci :

```
def indice_maximum_tableau(t):
    """Renvoie l'indice du maximum du tableau t.
    Le tableau t est supposé non vide et contient
    des nombres."""
    assert len(t) > 0, "le tableau est vide"

    indice_du_max = 0
    for i in range(len(t)):
        if t[i] > t[indice_du_max]:
            indice_du_max = i
    return indice_du_max
```

Rq : noter qu'à l'appel du programme précédent sur un tableau vide, sans instruction `assert`, il n'y a pas d'erreur à l'exécution ! Cela risque de rendre le debogage d'autant plus difficile lorsqu'on va par exemple déclencher une `IndexError: list index out of range`.

4. tester ses programmes : même documentés/spécifiés il peut (évidemment) y avoir un bug
   - inclure ces tests dans le même fichier source que le programme
   - exécuter les tests
   - si un problème est détecté : corriger l'erreur et relancer *tous* les tests

Rq : on peut écrire les tests dès la spécification, avant l'écriture du code, ou confier à 2 personnes/équipes l'écriture du code du programme et celui des tests du programme

Comment écrire de «bons» tests :

- tester chacun des cas mentionnés par la spécification (exemple : 3 cas dans la résolution d'une équation du second degré)
- si la fonction renvoie un boooléen, faire des tests où la fonction renvoie `True`, d'autres où elle renvoie `False`
- avec des tableaux : tester le cas du tableau vide et plus généralement le cas des tableaux «limites» (contenant toujours le même élément, des éléments tous différents, etc.)
- avec des nombres : tester des valeurs de signes différents, nulles, prenant des valeurs limites (les bornes d'un intervalle par exemple, si celui-ci fait partie de la spécification)
- dans le cas où le programme contient des conditionnelles (`if...else...elif`) tester l'exécution de tous les cas

Rq : il se peut que l'écriture des tests conduise à écrire une fonction vérifiant si la sortie est conforme à ce qui est attendu. Par exemple si une fonction `tri(t)` doit trier par ordre croissant un tableau `t`, on écrira une fonction `croissant(t)` vérifiant que les éléments d'un tableau `t` sont bien rangés dans l'ordre croissant.

- Dans ce cas, il faut prendre conscience que la fonction de test peut elle-même être buggée...
- En fait (cf. plus tard, chapitre sur la calculabilité/décidabilité), on est ici devant un obstacle théorique : «il n'existe pas de programme (général) vérifiant qu'un programme (lui aussi général) est correct».
- Pratiquement : dans la plupart des cas le code du programme de test est bien plus simple que celui à tester, donc la probabilité de «faux positif» est faible.
- Dans certains cas, on sait *prouver* que tel algorithme réalise correctement ce qu'il est censé faire
- Il faut se rappeler qu'aucune batterie de tests, aussi sophistiquée et imposante soit-elle, ne constitue en général une preuve que le programme fonctionnera toujours correctement.

Sommaire

Les types

- Le type des variables en jeu dans un programme est une source possible (et assez fréquente) d'erreurs.

```
>>> 1+[2,3]
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    1+[2,3]
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

- Python est un *langage à typage dynamique* : ce n'est qu'à l'exécution que l'interpréteur vérifie la compatibilité des opérations demandées.
- De nombreux langages sont à *typage statique* : C++, Java,...
- lors de l'écriture de la spécification d'une fonction, en particulier pour l'interface d'un module, il faut indiquer les types des arguments passés en entrée ou renvoyés en sortie.
- On peut *annoter* les variables et les fonctions. Ces annotations :
  - jouent (uniquement) un rôle de documentation
  - des outils externes peuvent faire des vérifications globales de cohérence des types, en utilisant ces annotations (p.ex. `mypy`)

Exemple :

```
x: int = 42 # x est une variable entière

def contient(s: list, x: int)->bool:
    # s est un tableau, x est un entier, contient renvoie un booléen
    ...

def ajoute(s: list, x: int)->None:
    # s est un tableau, x est un entier, ajoute ne renvoie rien
    ...
```

Ex. 1 Les annotations `mypy` sont comprises et utilisées par PyCharm. Saisir le programme ci-dessous dans un nouveau projet PyCharm et vérifier que, grâce aux annotations, il y a détection et affichage des erreurs de type. Identifier la nature de chacune des erreurs de type affichées.

```
x: int = 42
y: float = 5.6
x = x + y
z: list[int, int] = [2.0,3]
z = z + x

def greeting(name: str) -> str:
    return 'Hello ' + name
```

Sommaire

Tester un programme

Program testing can be used to show the presence of bugs, but never to show their absence ! (Edsger Dijkstra)

- But : vérifier qu'un programme donné, dont on a pas forcément le code source [4], réalise correctement ce pourquoi il est fait.
- on écrit généralement une fonction de test dédiée à cette vérification, intégrée dans le code source contenant la fonction à tester. Ce programme de test sera seulement exécuté lorsque le programme principal est le fichier contenant la fonction testée.

Ex. 2 On suppose qu'on met au point une fonction nommée `tri(t)`, dont la spécification est donnée par la docstring suivante :

```
def tri(t: list):
    """tri le tableau t de nombres par ordre croissant"""
```

1. Indiquer la liste des vérifications à faire par une fonction nommée `test(t)` devant vérifier que `tri(t)` est correcte.
2. Indiquer sur quels types de tableaux il est judicieux de réaliser ces vérifications.

4. par exemple dans le cas d'un module importé.

Ex. 3 On reprend le contexte de l'exercice précédent.

1. Écrire le code de la fonction `test(t)` réalisant les vérifications indiquées précédemment.
2. Écrire un jeu de tests sur les tableaux indiqués précédemment. On pourra, pour certains tableaux, utiliser la fonction `randint(n, m)` du module `random` qui génère des entiers aléatoires entre `n` et `m` (inclus) selon une loi uniforme.
3. Écrire une fonction `tri(t)` réalisant correctement le tri par ordre croissant d'un tableau `t` de nombres. Utiliser pour cela un algorithme vu en Première (tri sélection ou tri insertion par exemple).
4. Vérifier, en exécutant le jeu de test écrit précédemment, que la fonction `tri` ne provoque pas d'erreur lors des tests.

- Il existe en Python plusieurs modules, par exemple `unittest` et `doctest`[5], permettant de générer automatiquement des tests sur des fonctions ou des classes. L'utilisation du second repose sur une écriture et une syntaxe particulière des docstrings, dans lesquels on indique les tests précis à lancer et les valeurs de retour attendues. Voici deux liens pour se documenter : unittest et doctest
- On écrit généralement la partie de test des fonctions rassemblées dans un module à la fin de son code source. Cependant ces tests ne doivent pas être exécutés lorsque ce module est importé par un autre programme. Il est possible de faire en sorte que ces fonctions de test ne soient exécutées que lorsque le programme exécuté est le module lui-même. On procède alors comme le montre le programme donné page suivante.

Ex. 4 La fonction `discriminant` (page suivante) est buggée : trouvez son bug et corrigez-le.

5. On parle de *tests unitaires*, en anglais Unit Tests, qui visent à vérifier le bon fontionnement d'une partie précise, ou *unité*, d'un logiciel.

```python
def discriminant(a, b, c):
    """Calcul du discriminant du trinôme ax^2+bx+c"""
    a = eval(input("Donner la valeur de a: "))
    b = eval(input("Donner la valeur de b: "))
    c = eval(input("Donner la valeur de c: "))
    delta = b**2 - 4*a*c
    return delta

def main():
    """test de la fonction discriminant"""
    print(discriminant(1, -3, 2))    # doit afficher 1
    print(discriminant(1, 2, 1))     # doit afficher 0
    print(discriminant(1, 2, 3))     # doit afficher -8

if __name__ == "__main__":# le main ne sera exécuté que si le programme
    main()                # principal lancé est le fichier SecondDegre.py
```

Code source 1 – Module SecondDegre.py

Ex. 5 Le code ci-contre est celui d'une fonction `somme_liste(l)` qui doit calculer et renvoyer la somme des éléments d'une liste `l`.

1. Ce code comporte un bug. Trouver sur quelle ligne il se trouve.
2. Corriger la fonction et tester la version corrigée.

```
1  def somme_liste(l):
2      """calcule et renvoie la somme
3      des éléments de la liste de
4      nombres l"""
5      s = 0
6      for elt in l:
7          s = s + elt
8          return s
```

Sommaire

Exercices

Ex. 6 Pour chacune des fonctions suivantes, proposer un type pour chacun de ses arguments et un type pour son résultat :

```
def f1(t):
    return t[0] + 1
def f2(x):
    return str(3.14*x)
def f3(p):
    x, y = p
    return 2 * x + y
def f4(s):
    s.add(42)
def f(d, s):
    if s != "toto":
        d[s] += 1
    return d[s]
```

Ex. 7 Proposer des tests pour une fonction `miroir(ch)` qui prend en argument une chaîne de caractères `ch` et renvoie la chaîne contenant les caractères de `ch` en ordre inverse.

Ex. 8 On suppose qu'une classe contient un tableau dans un attribut `tab` et délimite une portion de ce tableau à l'aide de deux attributs `deb` et `fin`. La portion s'étend de l'indice `deb` inclus à l'indice `fin` exclu. Écrire une méthode `valide`

```
def valide(self):
    assert ...
```

qui vérifie que les attributs `deb` et `fin` définissent bien une portion valide du tableau `tab`.

Ex. 9 Le code donné page suivante est celui ayant provoqué, le 31 décembre 2008, la panne des lecteurs MP3 Zune de Microsoft. Le code est écrit en langage C, mais est suffisamment proche du Python pour être compris. Pour trouver l'origine de ce bug, il suffit de savoir :

- que 2008 est une année bissextile et que la fonction `isLeapYear(year)` renvoie True si l'année `year` est bissextile, False sinon.
- qu'avec le système utilisé, les jours sont numérotés en partant de 1, pour le 1er janvier, jusqu'à 365 (ou 366 pour une année bissextile), pour le jour du 31 décembre. Tous les jours, à minuit, la variable `days` est incrémentée de 1.

```
while (days > 365) {
  if (IsLeapYear(year)) {
    if (days > 366) {
      days -= 366;
      year += 1;
    }
  }
  else {
    days -= 365;
    year += 1;
  }
}
```

1. Trouver l'origine du bug et expliquer le comportement des lecteurs Zune qui a été observé.
2. Proposer une version corrigée de ce code.