Spécification des algorithmes 3

3.1 Contexte

Soit la fonction suivante :

```
def fonction(x,y = []) :
    if x == [] :
        return y
    else :
        z = x.pop(0)
        if z not in y :
            y.append(z)
        return fonction(x, y)
```

Problèmes :

- à quoi sert cette fonction ?
- quels sont les paramètres à donner ? leur type ?
- que renvoie la fonction ?

- Spécifications des données attentues en entrée et fournie en sortie/retour.
- Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante
- Assertion.
- Jeu de tests associé à un programme.

Lors de l'écriture d'une fonction, on peut être amener à l'utiliser, le jour même, le lendemain, une semaine plus tard, un mois plus tard... On peut être amener à la partager lorsqu'on travaille sur un projet. Il est donc essentiel de les documenter, de les commenter.

Présentons donc les choses ainsi.

```
def supprimer_doublons_liste(L,y = []) :
    """
    Fonction permettant de renvoyer, à partir d'une liste d'entiers, une liste
     privée des doublons.
    La liste initiale sera vidée.
    """
    if L == [] :
        return y
    else :
        z = L.pop(0)
        if z not in y :
            y.append(z)
        return fonction(L, y)
```

En modifiant le nom de la fonction et en ajoutant une court commentaire, il devient plus simple de partager la fonction ou de la réutiliser plus tard.

Problèmes (encore des problèmes) :

- on veut une fonction qui supprime les doublons d'une liste d'entiers... comment est-on sûrs que la fonction atteint bien cet objectif ?

Pour prouver qu'un algorithme on peut envisager deux méthodes :

- une méthode empirique (expérimentale) se basant sur des tests, sur le contrôle, des données d'entrées. Cette méthode à l'avantage d'être relativement simple à mettre en œuvre. Elle a l'inconvénient de valider le comportement d'une fonction sur des cas « ponctuels ». On ne montre donc pas qu'elle fonctionne dans le cas général.

- une méthode formelle, dans le cadre de laquelle on va prouver mathématiquement qu'une fonction atteint bien l'objectif fixé. On parle alors de de terminaison et de correction de l'algorithme.

La première méthode fait l'objet du présent chapitre. La seconde fera l'objet d'un chapitre ultérieur.

3.2 Spécification d'une fonction

3.2.1 Spécification à l'aide de commentaires

Définition – Signature

La **signature d'une fonction** définit les entrées et les sorties d'une fonction.

La signature peut comporter par exemple le type des paramètres d'entrées ou de sorties, des conditions sur ces paramètres.

La façon la plus simple de signer une fonction, est d'indiquer des commentaires juste après la déclaration de la fonction.

Exemple –

```
def supprimer_doublons_liste(L,y = []) :
    """
    Fonction permettant de supprimer les doublons d'une liste.
    Entrée :
     * L: list(int) : liste d'entiers.
    Sortie :
     * list(int) : liste d'entiers ne comportant aucun doublons.
    """
    if L == [] :
        return y
    else :
        z = L.pop(0)
        if z not in y :
            y.append(z)
        return fonction(L, y)
```

Lorsque les commentaires d'une fonction sont déclarés ainsi après la déclaration d'une fonction (triples "), `Python` génère automatiquement une documentation en utilisant l'instruction `help(supprimer_doublons_liste)`. On parle de `docstring`.

3.2.2 Annotation d'une fonction– Limite du programme

Depuis la version 3.5 de `Python` il est possible d'annoter les fonctions.

Définition – Annotations de type

Les annotations de type permettent de donner une information sur le type attendu d'une fonction. Ce ne sont que des indications pour le programmeur. Rien ne l'empêche de ne pas les respecter.

Exemple –

```
def addition(a:int, b:int) -> int
    :
    return a+b
```

```
>>> addition(1,1)
        2
>>> addition('a','a')
        aa
```

Comme précisé, les annotations ne sont que des indications. Mais il pourrait être intéressant de pouvoir s'assurer que l'utilisateur de la fonction respecte les conditions d'utilisation de la fonction que ce soit le type des arguments, ou d'autres conditions.

Remarque

En Python, il est possible d'utiliser `mypy` qui permet de vérifier que lors de l'appel d'une fonction, le typage précisé dans les annotations est bien respecté.

3.3 Assertion

3.3.1 Validation des entrées

Comme suite au paragraphe précédent, en phase de debuggage (ou développement), par opposition à la phase de production, `Python` prévoit la possibilité de vérifier que les arguments donnés à une fonction respectent bien les préconisations du concepteur.

En utilisant l'instruction `assert`, le programme s'interrompt quand une erreur est détectée.

Exemple –

Dans la lignée des annotations de type, on peut tester le type d'argument donné à une fonction.

```
def addition(a:int, b:int) -> int
    :
    assert type(a) == int and
    type(b) == int
    return a+b
```

```
>>> addition(1,1)
        2
>>> addition('a','a')
        (...)
        AssertionError
```

Si les arguments sont du mauvais type, `Python` déclenche une `AssertionError`.

Exemple –

Prenons un type d'exemple que nous rencontrerons plus souvent : l'algorithme fonctionne seulement si des préconditions sont remplies.
Par exemple dans le cas de la résolution de l'équation $f(x) = 0$, il est nécessaire que f soit continue, monotone et que $f(a)$ et $f(b)$ soient de signes différents. Il faut aussi que ε soit positif. On peut donc procéder de la façon suivante.

```
def dichotomie(f, a:float , b:float , epsilon:float) -> float :
    """Zéro de f sur [a,b] à epsilon près, par dichotomie
        Préconditions : f(a) * f(b) <= 0
        f continue sur [a,b]
        epsilon > 0"""
    assert (f(a) * f(b) <= 0) and (epsilon > 0)
    c, d = a, b
```

```
    fc, fd = f(c), f(d)
    while d - c > 2 * epsilon:
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0:
            d, fd = m, fm
        else:
            c, fc = m, fm
    return (c + d) / 2.
```

3.3.2 Vers la gestion d'exceptions – hors programme

L'avantage des assertions et leur facilité à être mise en œuvre. L'inconvénient et qu'il n'est pas possible de savoir précisément ce qui a généré l'erreur d'assertion : un problème de précondition ? de type ? autre ?

Du fait de leur manque de précision, il peut être difficile de les gérer.

Un premier pas vers la gestion d'erreur est de donner à l'utilisateur la raison pour laquelle la fonction génère une erreur. Pour cela, on va *lever une exception*.

```
def dichotomie(f, a:float , b:float , epsilon:float) -> float :
    """Zéro de f sur [a,b] à epsilon près, par dichotomie
        Préconditions : f(a) * f(b) <= 0
        f continue sur [a,b]
        epsilon > 0"""
    if f(a) * f(b) > 0 :
        raise Exception("f(a) et f(b) sont de même signe")
    elif epsilon < 0 :
        raise Exception("epsilon est négatif")

    c, d = a, b
    fc, fd = f(c), f(d)
    while d - c > 2 * epsilon:
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0:
            d, fd = m, fm
        else:
            c, fc = m, fm
    return (c + d) / 2.
```

```
>>> dichotomie(f, 0, 1, 0.00001)
   (...)
   Exception: f(a) et f(b) sont de même signe
```

3.4 Génération de tests

3.4.1 Tests unitaires

On a vu comment vérifier que les données d'entrées d'une fonction sont compatibles avec ce qu'attendait le concepteur de la fonction. On va maintenant chercher, grâce à des tests unitaires, si le résultat attendu est bien celui déterminé par une fonction.

Exemple –

```
def addition(a:int, b:int) -> int :
    assert type(a) == int and type(b)
     == int
    return a+b
```

```
>>> addition(1,1) == 2
        True
>>> addition(1,1) == 3
        False
```

Ces tests ont l'avantage d'être simples à mettre en œuvre mais ne garantissent pas que la fonction « marche » dans tous les cas.

Il est alors possible de créer une fonction de tests.

```
def test_addition() :
    assert addition(1,1) == 2
    assert addition(1,1) !=3
```

Des jeux de tests pertinents peuvent être des tests sur les cas limites. Dans le cas de la recherche dichotomique, on peut se demander si la boucle `while` n'est pas réalisée une fois de trop par rapport à la valeur de `epsilon`.

```
>>> res = dichotomie(f, 0, 1, epsilon)
>>> f(res) < epsilon # MAUVAIS test : f(res) peut être négatif
    True
>>> abs(f(res)) < epsilon
    False
>>> # L'algorithme dichotomique doit itérer une fois de plus ?
```

En modifiant l'algorithme, et en itérant une fois de plus, le test reste faux. La condition d'arrêt de l'algorithme ne permet pas de trouver x tel que $f(x) < \varepsilon$ mais de trouver une valeur de x telle que l'écart entre deux solutions consécutives est inférieur à ε. Il faut donc clarifier l'objectif de la fonction pour proposer un test plus adapté.

3.4.2 Vers Pytest – hors programme

Lorsqu'on connaît dans un grand nombre de cas le résultat qui est sensé être trouvé par une fonction, il est possible de faire passer une batterie de tests à cette fonction en recherchant si le résultat calculé et bien le résultat attendu.

Soit la fonction suivante permettant de calculer le produit scalaire entre deux vecteurs.

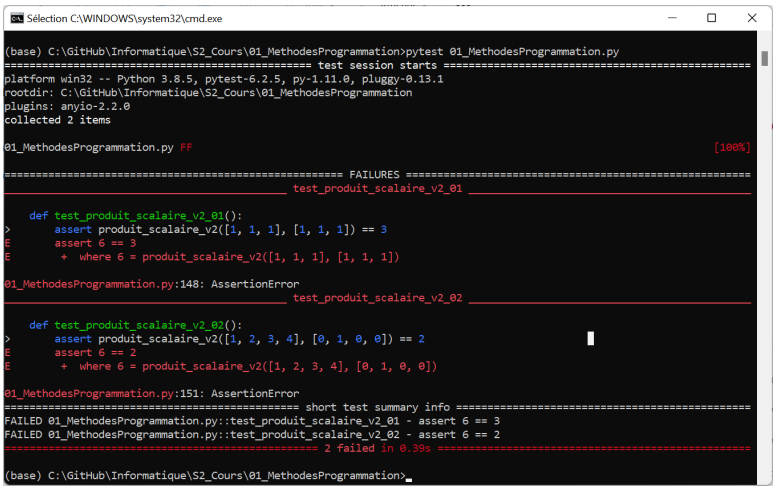
```
def produit_scalaire_v2(vecteur1, vecteur2):
    somme = 0
    for i in range(len(vecteur1)):
        somme += vecteur1[i]*vecteur2[i]+1
    return somme
```

`Pytest` permet de faire passer une batterie de tests à une fonction. Ces tests sont des fonctions commençant obligatoirement par `test_`.

```
def test_produit_scalaire_v2_01():
    assert produit_scalaire_v2([1, 1, 1], [1, 1, 1]) == 3

def test_produit_scalaire_v2_02():
    assert produit_scalaire_v2([1, 2, 3, 4], [0, 1, 0, 0]) == 2
```

Le lancement de `Pytest` donne le résultat suivant.



On peut lire que les deux tests ont échoué. La fonction ne remplit donc vraisemblablement pas son objectif.