TD4 TNSI	Mise au point de programmes - 1

L'objectif de ce TD est d'introduire des outils permettant d'améliorer la qualité du code produit ainsi que la facilité de sa réutilisation, y compris par une tierce personne.

I - Spécification et documentation

A - Introduction - problématique

1 - Déterminer la tâche réalisée par la fonction :

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

Vous pourrez pour cela la tester sur l'entrée valeurs=[1,5,2,3] ou sur d'autres listes de votre choix.

2 - Son nom vous semble-t-il bien choisi ? Le modifier si besoin.

On veut désormais afficher la valeur du maximum de la liste « valeurs ». On propose le code :

```
valeurs=[1,5,2,3]
i=indice_maximum_liste(valeurs)
print(valeurs[i])
```

3 - Tester ce code, puis remplacer la première ligne par valeurs=[]. Identifier l'origine du message d'erreur obtenu alors à l'exécution.

4 - Proposer deux méthodes afin d'éviter cette erreur.

B - Documentation

Comme on l'a déjà vu, il est nécessaire de documenter son code. On peut régler le programme précédent avec le commentaire :

```
# Détermine l'indice du maximum d'une liste non vide
```

placé avant la fonction `maximum_liste`.Toutefois, cette information ne s'adresse qu'au programmeur, pas à l'utilisateur, par exemple si la fonction était incluse dans une bibliothèque.

On peut donc procéder autrement en utilisant la documentation, introduite par `"""` juste après le prototype de la fonction (la ligne qui commence par `def`).

```
def indice_maximum_liste(t):
    """ Renvoie l'indice du maximum de la liste t.
    Le tableau t étant supposé non vide """
    m=0
    for i in range(len(t)):
        if t[i]>t[m]:
            m=i
    return m
help(indice_maximum_liste)
```

Une précondition est une condition d'utilisation de la fonction alors qu'une postcondition est une description de ce que renvoie la fonction.

Les deux lignes entre `"""` et `"""` constituent la chaîne de documentation de la fonction.

1 - Identifier une postcondition dans `indice_maximum_liste(t)`.

2 - Identifier une précondition `indice_maximum_liste(t)`.

On a ainsi une méthode plus pratique et surtout plus générale pour documenter notre code, mais cela reste des commentaires. Rien n'empêche une mauvaise utilisation du code.

C - Exercices

1 - Que pensez-vous des commentaires :

a - `x=x+1 #incrémenter x`

b - `n=25 # n est le nombre de nombres premiers inférieurs à 100`

2 - Donner une chaîne de documentation à la fonction suivante puis tester un appel sur la fonction help:

```
def division_euclidienne(a,b):
        q=0
        r=a
        while r>=b:
                q+=1
                r-=b
        return q,r
```

3 - Donner une chaîne de documentation à la fonction `puissance(x,n)` qui calcule x à la puissance n pour deux entiers x et n.

II - Éviter les problèmes à venir

A - assert

On revient au programme `indice_maximum_liste` et on peut éviter que des difficultés ne surviennent en testant que la liste fournie n'est pas vide, avec une sortie propre et informative pour le programmateur qui a fait une erreur :

```
def indice_maximum_liste(t):
        """ Renvoie l'indice du maximum de la liste t.
        Le tableau t étant supposé non vide """
        assert len(t)>0, "le tableau est vide"
        m=0
        for i in range(len(t)):
                if t[i]>t[m]:
                        m=i
        return m
```

1 - Tester cette fonction sur une liste vide, quel est le message affiché par python ?

2 - Est-il facile de trouver la localisation et la nature de l'erreur ?

On aurait aussi pu renvoyer la valeur None dans le cas d'une liste vide. Aucune des deux stratégies n'est meilleure dans l'absolu :

- assert met au courant proprement le programmeur lors du développement.
- le test de vacuité avec le renvoi de None permet au programmeur averti, si la fonction est bien documentée, de pouvoir traiter ce cas à part dans son programme.

B - Exercice

1 - A l'aide d'une condition, réécrire indice_maximum_liste afin qu'elle renvoie None si la liste est vide.

III - Stratégies de tests

A - Comment procéder à des tests

Pour tester la fonction indice_maximum_tableau, on peut faire des tests, comme par exemple :

```
>>> indice_maximum_tableau([2,3,1])
```

On attend alors la réponse : « 1 ». On peut aussi tester :

```
>>> indice_maximum_tableau([])
```

Et on attend alors la réponse « None ».

On pourrait également vouloir tester le fonctionnement si le maximum est en début ou fin de liste, ce qui amènerait à un nombre important d'affichages.

Avec ce qui a déjà été vu, on peut procéder sans avoir à faire des vérifications à l'écran : les tests peuvent être inclus dans une fonction qui balaye tous les cas à tester. On peut remplacer avantageusement les tests précédents par la procédure :

```
def tests() :
    assert indice_maximum_tableau([2,3,1]) == 1
    assert indice_maximum_tableau([]) == None
```

a - Compléter cette fonction de test sur les cas :

- maximum au début de la liste,
- maximum à la fin de la liste,
- liste de nombres négatifs.

2 - En utilisant la fonction test, vérifier le bon fonctionnement de votre programme du II B 1.

B - Quels tests effectuer ?

Il existe des stratégies pour tester les programmes que l'on écrit. La plus courante consiste à écrire une liste de tests la plus couvrante le plus de cas possibles. On parle alors de tests couvrants.
Il faut alors prévoir de tester :

- tous les cas mentionnés dans la spécification,
- dans le cas d'une fonction renvoyant un booléen, les réponses « True » et « False »
- si il y a une liste, le cas de la liste vide,
- si un nombre est en entrée, valeurs positives, négatives et nulles
- les limites des intervalles de fonctionnement normal (0 par exemple si les entrées doivent être positives)

IV - Modules

A - Ce que l'on fait et sait déjà :

Il est important lorsqu'on programme d'adopter une stratégie modulaire. On l'a déjà vue lors :

1. du découpage du code en fonctions,
2. du groupement des méthodes dans une classe,
3. de l'utilisation de bibliothèques regroupant des fonctions travaillant sur le même thème.

B - Importer une bibliothèque (ou un module)

On considère ici la bibliothèque random.
On peur recenser 4 méthodes pour en demander l'import dans le tableau, colonne de gauche.

1 - A quel appel de la fonction randint (colonne de droite) chaque import (colonne de gauche) correspond-il ?

Import	Appel de fonction
import random	Cet import ne doit pas être utilisé car l'espace des noms de fonctions n'est alors plus maîtrisé
from random import *	randint(1,10)
from random import randint (ne charge que la fonction randint)	random.randint(1,10)
import random as rdn	rnd.randint(1,10)

C - Documentation

Chaque module peut être documenté, de la même façon que les fonctions ou les méthodes. Pour accéder à cette documentation (docstring), la syntaxe est la même :

```
>>> import random #pas de documentation ici, on importe juste la bibliothèque random
>>> help (random) # affichage de la documentation du module random
```

V - Exercices

A - Fonction inconnue

Pour la fonction suivante, lui donner un meilleur nom, une chaîne de documentation (« docstring »), ainsi que des tests (avec « assert »).

```
def f(t):
    s=0
    for i in range(len(t)):
        s+=t[i]
    return s
```

B - Fonction mal écrite

Proposer des tests (avec « assert ») pour la fonction suivante, sensée tester l'appartenance de la valeur v au tableau t. En conclure que la fonction a mal été écrite.

```
def appartient (v,t):
    i=0
    while i<len(t)-1 and t[i]!=v:
        i=i+1
    return i<len(t)
```

C - Calculer la valeur du mot « Arithmancie »

On souhaite associer une valeur numérique à une chaîne de caractères. La valeur associée à la chaîne est la somme des valeurs associées à chacune des lettres qui la composent. Aux lettres de A à Z, majuscules ou minuscules, accentuées ou non, on associe les valeurs de 1 à 26. Tous les autres symboles ont pour valeur 0.
Vous devrez utiliser la docstring de certaines des fonctions utilisées dans le code qui vous est proposé.
ord(x) renvoie le code numérique (code ASCII) associé à la lettre passée en paramètre.

La chaîne string.ascii_uppercase est définie dans le module string et vaut « ABCDEFGHIJKLMNOPQRSTUVWXYZ ».

a - Tester chacune des fonctions suivantes jusqu'à montrer qu'elles sont incorrectes (avec « assert »).

```
def valeur1(chaine):
    s=0
    for c in chaine :
        if c>= ‘’A” and c<= ‘’Z”:
            s=s+ord(c)-ord(‘A’)+1
    return s

def valeur2(chaine):
    s=0
    for c in chaine.upper():
        if c>= ‘’A” and c< ‘’Z” :
            s=s+ord(c)-ord(‘A’)+1
    return s

import string
def valeur3(chaine):
    s=0
    for c in chaine.upper():
        if c in string.ascii_uppercase:
```

```
            s=s+ string.ascii_uppercase.index(c)
    return s
```

2 - Proposer un jeu de test couvrant pour la fonction que l'on veut créer (on nomme cette fonction valeur). L'écrire avec « assert »

3 - Proposer le code de cette fonction (vous pouvez vous inspirer des fonctions précédentes, en les corrigeant) et les tester avec le code produit à la question 2.

4 - Écrire un programme qui calcule alors la valeur du mot « Arithmancie»