Vocabulaire et méthodes de programmation

Un programme informatique est écrit une fois, mais peut être relu plusieurs fois, parfois longtemps après qu'il a été écrit, et pas forcément par la personne qui l'a programmé. Il convient donc d'écrire des programmes clairs et faciles à lire ou à relire. Cela facilite également le débogage ou la modification ultérieure du programme.

On donne ici des éléments de vocabulaire informatique ainsi que quelques recommandations pour programmer proprement.

Variables

Une **variable** est un élément créé dans la mémoire de l'ordinateur auquel on donne un nom et une valeur. Elle a un type (entier, flottant, booléen, liste, chaîne de caractères, fonction, etc.) et une adresse (l'endroit dans la mémoire où elle est stockée).

On prendra soin de distinguer les variables globales (qui existent dans tout le programme) et locales (qui n'existent que dans les fonctions où elles sont définies). Il est recommandé de n'utiliser des variables globales que si c'est vraiment nécessaire (cf TP 2).

En Python les noms des variables ne peuvent contenir que des caractères alphanumériques (`a-z`, `A-Z` et `0-9`) et des underscores (_). Elles ne peuvent pas commencer par un chiffre. Python distingue les minuscules et les majuscules.

Pour une bonne lisibilité du code, il est très important de bien choisir les noms des variables. Si une variable représente quelque chose de précis, on a intérêt à lui donner un nom adapté. Une variable qui représente une somme pourra ainsi être nommée `s` ou même `somme`. Si on travaille avec la liste des élèves de la classe on peut l'appeler `eleves` et itérer dessus en écrivant `for eleve in eleves`, etc.

Il ne faut pas utiliser `o` (et encore moins `0`) comme nom de variable (on peut confondre avec `0`). De même, on évite généralement `l` qu'on peut confondre avec `1` (ainsi on nomme souvent une liste générique `L` ou on utilise un nom plus long).

Comparons par exemple les programmes suivants :

```
def indice_du_maximum(x):
    a = x[0]
    z = 0
    for o in range(1, len(x)):
        if x[o] > a:
            a = x[o]
            z = o
    return z
```

```
def indice_du_maximum(L):
    maximum = L[0]
    indice_maximum = 0
    for i in range(1, len(L)):
        if L[i] > maximum:
            maximum = L[i]
            indice_maximum = i
    return indice_maximum
```

Pour l'ordinateur ces deux programmes définissent exactement la même fonction. Pour un humain le deuxième est bien plus lisible. Attention toutefois à l'abus des noms longs qui peut nuire à la lisibilité.

Enfin, en Python on utilise généralement le style `lower_case_with_undercores` pour les noms des variables et des fonctions et le style `UPPER_CASE_WITH_UNDERSCORES` pour les constantes globales.

Expressions et instructions

Une **expression** est un ensemble de symboles qui peut être évalué, comme `5.7` (valeur `5.7`, type `float`), `2 + 3` (valeur `5`, type `int`), `'ab' * 3` (valeur `'ababab'`, type `str`) ou `1 + 1 == 2` (valeur `True`, type `bool`). La fonction `type` permet de déterminer le type d'une expression.

```
>>> type([1, 2, 3])
<class 'list'>
```

Une **instruction** est un ordre que l'on donne à l'ordinateur et qui modifie son état. Par exemple `x = 5`, `L.append(3)`, `print('abc')` sont des instructions. Un **programme** est une suite d'instructions.

Effet de bord

On dit qu'une fonction est à **effet de bord** si elle modifie quelque chose en dehors de son environnement local, par exemple si elle modifie une variable globale ou un de ses arguments ou si elle affiche quelque chose à l'écran (effet de bord est une mauvaise traduction de *side effect* en anglais, on devrait plutôt dire effet secondaire).

Un effet de bord est parfois voulu, parfois non. Il faut éviter les effets de bord non désirés, qui ne sont pas toujours évidents à anticiper en particulier quand on travaille avec des objets mutables comme les listes. Considérons par exemple les fonctions suivantes :

```
def f(x):
    x = x + 2
    return x
```

```
def g(L):
    L.append(5)
    return L
```

La première fonction ne provoque pas d'effet de bord, mais la seconde oui :

```
>>> x = 1
>>> f(x)
3
>>> x
1
```

```
>>> L = [1, 6, 4]
>>> g(L)
[1, 6, 4, 5]
>>> L
[1, 6, 4, 5]
```

Spécification et signature

La **spécification** d'une fonction est la donnée précise de ce qu'elle reçoit et de ce qu'elle renvoie. C'est une sorte de contrat passé entre le programmeur et l'utilisateur de la fonction. Une bonne spécification doit permettre à ce dernier d'utiliser la fonction sans lire le code. Par exemple, la fonction `indice_du_maximum` ci-dessus reçoit une liste non vide d'entiers ou de flottants et renvoie l'indice du plus grand élément de cette liste (si le maximum apparaît plusieurs fois, elle renvoie l'indice de sa première occurrence).

La **signature** d'une fonction est la donnée du type de ses paramètres et du type de son résultat. Par exemple, la signature de la fonction `indice_du_maximum` est `list -> int`.

En Python on peut utiliser les docstrings pour donner la spécification et/ou la signature d'une fonction. On peut aussi préciser les types d'arguments attendus et le type du résultat dans l'en-tête de la fonction. Exemple :

```
def indice_du_maximum(L: list) -> int:
    '''Renvoie l'indice de la première occurrence du plus grand élément
       d'une liste.
       Entrée : une liste non vide d'entiers ou de flottants.
       Sortie : un entier.'''
    maximum = L[0]
    indice_maximum = 0
    for i in range(1, len(L)):
        if L[i] > maximum:
            maximum = L[i]
            indice_maximum = i
    return indice_maximum
```

Annotations et commentaires

En Python on peut mettre des commentaires avec `#`. Ils ont pour objectif d'améliorer la lisibilité du programme. On peut les utiliser pour expliquer comment fonctionne une partie délicate du programme, ou pour justifier des choix de programmation (par exemple le type de structure de données utilisé).

Les commentaires ne servent pas à traduire le code en français. Par exemple, le commentaire suivant est inutile :

```
i = i + 1  # On incrémente i
```

On peut également utiliser les commentaires pour indiquer une précondition (une condition qui doit être vérifiée avant l'exécution d'un bloc d'instructions), une postcondition (une condition qui doit être vérifiée après l'exécution d'un bloc d'instructions) ou un invariant.

```
def indice_du_maximum(L):
    # Précondition : L != []
    maximum = L[0]
    indice_maximum = 0
    for i in range(1, len(L)):
        # Invariant : maximum = L[indice_maximum] = max(L[:i])
        if L[i] > maximum:
            maximum = L[i]
            indice_maximum = i
    # Postcondition : L[indice_maximum] = max(L)
    return indice_maximum
```

Assertions

Une **assertion** est une expression qui doit être évaluée comme vraie sans quoi le programme s'arrête. Cela permet de vérifier entre autres lors de la phase de débogage qu'une certaine condition est bien vérifiée à un endroit précis du programme. En Python, l'instruction `assert` permet de définir une assertion.

On peut en particulier placer une assertion au début d'une fonction :

```
def indice_du_maximum(L):
    assert L != [], 'La liste ne doit pas être vide.'
    maximum = L[0]
    indice_maximum = 0
    for i in range(1, len(L)):
        if L[i] > maximum:
            maximum = L[i]
            indice_maximum = i
    return indice_maximum
```

```
>>> indice_du_maximum([3, 1, 5, 2])
2
>>> indice_du_maximum([])
AssertionError: La liste ne doit pas être vide.
```

Jeux de tests

Quand on a programmé une fonction, il convient soit de démontrer qu'elle fait bien ce qu'on attend d'elle (cf cours correspondant), soit de la tester sur un **jeu de tests**, c'est-à-dire un ensemble d'exemples recouvrant les différents cas possibles.

Les tests peuvent servir à vérifier qu'un programme fonctionne correctement ou à mesurer ses performances. On peut ainsi tester la fonction `indice_du_maximum` avec les exemples suivants :

```
>>> indice_du_maximum([1, 2, 3, -3, 1])     # une liste quelconque
2
>>> indice_du_maximum([2.5, 3.1, 3.5, 3.6, 2.2])  # avec des flottants
3
>>> indice_du_maximum([-9, -1, -2, -3, -3]) # avec des nombres négatifs
1
>>> indice_du_maximum([1, 2, 3, 3, 1])      # pour vérifier qu'elle renvoie bien
2                                           # le premier indice du maximum
>>> indice_du_maximum([5])                  # une liste à un élément
0
>>> indice_du_maximum([])                   # la liste vide
AssertionError: La liste ne doit pas être vide.
>>> indice_du_maximum([5, 4, 3])            # le maximum est en première position
0
>>> indice_du_maximum([3, 4, 5])            # le maximum est en dernière position
2
>>> indice_du_maximum(list(range(10**7)))   # une grande liste
9999999
```

Exercice 1 Écrire une fonction permettant de dire si une liste est triée ou non et concevoir un jeu de tests associé.