Cours – Ecriture et mise au point de programmes

Objectifs :
- ⇨ Définir ce qu'est la maintenabilité d'un programme
- ⇨ Voir quelques règles générales pour écrire des programmes facilement maintenables
- ⇨ Apprendre à tester son code et à le rendre plus robuste
- ⇨ Utiliser la programmation modulaire et défensive

I – Ecriture

Dans cette partie, nous allons voir quelques consignes et règles pour l'écriture des programmes.

Pourquoi définir (et imposer parfois) des règles ?

Le code source d'un programme n'est généralement pas immuable, mais amené à subir des modifications et améliorations au cours du temps (on parle de *maintenance du code*). Dès que le projet a un peu d'envergure, il nécessite également souvent l'intervention de plusieurs programmeurs sur le code source. Afin de faciliter la compréhension du code source et sa diffusion dans une équipe de programmeur, on est amené à définir des règles d'écriture des programmes.
Ces règles doivent permettre une meilleure lisibilité du code. Afin de minimiser le temps nécessaire à sa compréhension et à sa ré-utilisation.
Bon à savoir :
- 40 à 80 % du coût d'un logiciel sont dans sa maintenance.
- La plupart des programmes ne sont pas maintenus par le même programmeur pendant toute la durée de leur cycle de vie, d'où l'intérêt que le code source soit facilement compréhensible par d'autres.

1) Nom des variables

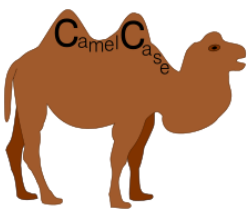
Afin d'avoir des noms de variables homogènes et d'éviter les erreurs dus à une variable écrite différemment à plusieurs endroits du programme (ex : `boutonClique`, `BoutonClique`, `clic_bouton`, `CLICBOUTON`, …), on utilise des conventions de nommage.

a. Le Snake Case

Le Snake Case est une convention typographique en informatique consistant à écrire des ensembles de mots, généralement en minuscules, en les séparant par des tirets bas.
Exemple : `largeur_fenetre` ; `nom_client_complet`

Une variante du Snake Case consiste à écrire ces ensembles de mots en les séparant par des tirets bas, mais cette fois en les écrivant en majuscules. Il s'agit du Screaming Snake Case. Elle est surtout utilisée pour écrire des constantes. Cela donne `JE_SUIS_UNE_CONSTANTE`.

b. Le camelCase



Cette convention de nommage est utilisée lorsque les espaces sont interdits dans le nom et que l'on souhaite donner la taille minimale aux noms. Elle consiste à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot excepté le premier.

Exemple : `largeurFenetre` ; `nomClientComplet`

Il existe une variante, le Pascal case (ou upper camel case), qui reprend le même principe mais en écrivant la première lettre en majuscule (ex : `LargeurFenetre`, `NomClientComplet`).

c. La notation hongroise

La notation hongroise est une convention de nommage des variables et des fonctions qui met en avant soit leur usage, soit leur type. Cette notation est notamment très utilisée par Microsoft. Elle permet d'éviter certaines erreurs dues à l'utilisation d'un mauvais type et donne une idée claire de l'utilité d'une variable juste en connaissant son nom.

Exemple de nommage par l'usage :

`idxClient` : La variable indexant un client est préfixée "idx" pour indiquer son usage comme index

`colQuantitéCommandee` : Indique qu'il s'agit de la colonne ("col") où se trouve la quantité commandée

Exemple de nommage par le type :

`intLargeurFenetre` : La variable est un entier ("int")

`strNomClientComplet` : variable de type chaîne ("str")

`bFinCycle` : Variable de type booléen ("b")

2) Documentation

En plus des commentaires dont on a déjà parlé, le code peut/doit contenir des **docstrings**. Ce sont des commentaires multi-lignes qui apparaissent juste après la définition d'une fonction et la documente. Il existe aussi des docstring de module ou de script qui apparaissent en tout début de fichier avant les imports et qui décrivent le contenu et le fonctionnement de tout le module (principe de fonctionnement, liste des fonctions, ...).

Les docstrings sont affichés par la commande `help`.

Remarque : Il existe de nombreuses façons normalisées d'écrire les docstrings. Dans le cadre de la NSI, nous laisserons un peu de liberté dans la rédaction des docstrings (du moment qu'elles existent et sont utiles !).

Déclaration d'une fonction avec typage des variables[1] : c'est ce qu'on appelle les **annotations** ou **type hinting** en anglais.

```
def nom_fonction(param1:type1, param2:type2, …) -> typeretour :
```

Exemple :

```
def position(x:int, y:int) -> str :
    """Renvoi un texte décrivant la position (type 'B5') à partir des coordonnées x et y."""
    return chr(x + ord('A')) + chr(y + ord('1')
```

Application :
Dans un fichier « documentation.py », écrire deux fonctions avec leurs annotations et docstrings. Le contenu de la fonction n'est pas important (on pourra simplement mettre « `pass` »). Ecrire ensuite une docstring pour le module puis sauvegardez-le.
Dans la console python importer le fichier que l'on vient de créer (« `import documentation` »), puis afficher sa documentation (`help(documentation)`) et celle de ses fonctions (`help(documentation.fonction1)`, …).

[1] Ceci n'est possible que depuis la version 3.5 de python

3) Charte d'écriture

L'ensemble des préconisations concernant le style d'écriture (espaces, tabulations, noms des variables et fonction, …) et la documentation (docstrings) d'un programme est appelé « **charte d'écriture** ».

Dans tout projet informatique, il est important de disposer d'une charte d'écriture claire afin d'obtenir une lisibilité optimale et une certaine homogénéité dans le programme. Cela est d'autant plus important si de nombreuses personnes participent au projet.

Dans le cadre de python, il existe des recommandations sur la façon d'écrire les programmes consignées dans la PEP8 (**P**ython **E**nhancement **P**roposals). Les programmeurs sont libres de respecter tout ou partie de ces recommandations, mais il est conseillé d'y coller au maximum.

4) Prévention des erreurs

Lorsqu'on écrit une fonction, celle-ci utilise généralement des données en entrée. Implicitement ou explicitement, ces données doivent respecter certaines règles (ex : tableau non-vide, contenant des nombres), sans quoi le comportement de la fonction n'est pas garanti.
On pourrait écrire une fonction qui soit capable de gérer tous les cas possibles et alerte avec un système quelconque le programme appelant si les données ne sont pas correctes, mais cela compliquerait fortement l'écriture de la fonction et du programme appelant.
Au lieu de cela, on peut utiliser un système de **préconditions** : au début de la fonction, on effectue certains tests sur les données pour vérifier qu'elles correspondent à ce qui est attendu par la fonction. Dans le cas où les données ne sont pas correctes, la fonction déclenche une exception (erreur provoquant l'arrêt du programme et l'affichage d'un message d'erreur). On évite ainsi le pire scénario qui serait d'avoir un comportement inattendu du programme qui serait très difficile à corriger (puisque l'erreur pourrait se déclencher bien après l'exécution (incorrecte) de la fonction).
En python, les préconditions s'écrivent grâce à un système d'**assertions** : L'interpréteur vérifie qu'une assertion (test logique) est correcte et déclenche une exception si ce n'est pas le cas.
L'instruction pour écrire une assertion est `assert`.

Exemple :

```
def moyenne(notes):
    assert len(notes) > 0, 'Il faut au moins une note pour calculer la moyenne !'
    return sum(notes) / len(notes)
```

5) Structure de données

La définition des structures de données *communes aux différentes fonctions* d'un programme doit être faite dès le début du projet afin de garantir que les différentes fonctions travaillent bien de manières compatibles.

Exemple :
Pour un programme de jeu d'échec, il faudra définir la façon dont on représente l'échiquier en mémoire, la façon de coder un « coup » (mouvement d'une pièce).
Application :
Proposer une spécification de structure de donnée pour représenter un échiquier et ses pièces en mémoire. De même pour représenter un « coup » (déplacement d'une pièce).

Proposition 1 :
Tableau de tuples. Chaque tuple représente une pièce de l'échiquier.

```
echiquier = [("B4","noir","pion"),...]
```

Proposition 2 :
Dictionnaire de dictionnaire

```
echiquier = {'A':{'A1':"Tb", 'A2':"Fb", ..., 'A8':"Tb"}, 'B':{'B1':"Pb", ..., 'B8':"Pb"}
```

```
deplacement = "A2:C3"
```

Proposition 3 :
Tableau à deux dimensions représentant les cases de l'échiquier. Chaque case du tableau est une chaîne de caractère : vide si la case est vide et avec une lettre donnant le type de pièce suivie d'une lettre pour la couleur 'b' ou 'n'.

```
echiquier = [['Δb', 'Cb', 'Fb', …, 'Δb'],['Pb','Pb', …, ],['', '', …],…, ['Δn','Cn',…,'Δn']]
deplacement = "A2:C3"
```

Autre proposition :
Echiquier : tableau de 8 tableaux de 8 chaînes de caractères. Chaque chaîne est composée d'une lettre représentant la pièce (« R » : Roi », « D » : Dame, « C » : Cavalier, …). Majuscule pour les noirs et minuscule pour les blancs. Caractère espace pour une case vide.

```
[['T', 'C', 'F', 'D', 'R', 'F', 'C', 'T'],
 ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
 ['t', 'c', 'f', 'r', 'd', 'f', 'c', 't']]
```

On pouvait aussi utiliser un tableau linéaire de 64 éléments et utiliser des nombres (0 pour vide, 1 pour Roi, 2 pour Dame, … et les nombres négatifs pour les pièces blanches).
Pour un coup, on peut utiliser un tuple contenant lui-même un tuple avec les indices de la case de départ dans le tableau et un tuple avec les indices de la case d'arrivée dans le tableau.
`((7, 1), (5,2))` pour sortir le cavalier blanc de gauche

Chaque représentation a ses avantages et ses inconvénients. S'il est toujours possible d'écrire le programme quelque soit la représentation utilisée, certaines représentations facilitent plus ou moins les choses. Par exemple la représentation 1 est bien adaptée à une sauvegarde sur disque de la partie, mais assez peu si on doit déterminer si une case donnée de l'échiquier est vide (car on devrait parcourir toute la liste pour vérifier qu'aucune pièce n'occupe cette case). La proposition 2 permet de tester facilement l'occupation d'une case, mais l'affichage de l'échiquier sera un peu plus délicat à programmer.
Dans un programme, il est toujours possible d'utiliser plusieurs représentation mémoire différentes en utilisant des fonction de conversion pour passer de l'une à l'autre.

6) Signature des principales fonctions

Lorsqu'on débute un projet et avant de programmer quoi que ce soit, il faut d'abord lister les principales fonctions dont le programme aura besoin. Pour chaque fonction, on écrira sa **signature**.
La signature d'une fonction est la donnée de son type de retour, suivie de son nom puis de type et du nom de tous ses arguments (et leurs éventuelles valeurs par défaut). C'est un peu comme les annotations que nous avons vues précédemment.
Exemple : `tuple determine_coup_ordi(list(list(str)) echiquier, int difficulte=1)`

7) Modularité

Afin de décomposer au maximum les tâches du programme, on le découpe en plusieurs modules qui vont chacun regrouper les fonctions et procédures correspondant à un groupe de tâches qui va ensemble.
Exemple :
Modules pour un programme de jeu d'échec :

- Programme principal
- Interface graphique / affichage
- Vérification de la validité des coups
- Détermination du coup par l'ordinateur
- Sauvegarde / chargement d'une partie

Chaque module est isolé dans un fichier .py qui contient essentiellement les définitions de fonctions.
On pourra ensuite importer les fonctions de ce module dans d'autres fichiers en écrivant simplement : `import nom_du_fichier_du_module_sans_le_« .py »`.
Lors de l'écriture de ce module, on écrit généralement un petit programme principal destiné à tester les fonctions écrites, mais qui ne devra pas être exécuté lorsque le module sera importé au sein du projet.
Pour cela on peut préciser à Python qu'une portion de code ne doit être exécutée que si le fichier est exécuté directement (lors de la mise au point) et qui sera ignorée si le fichier est importé au sein d'un autre programme.
Cela se fait avec la structure suivante :

```
if __name__ == "__main__" :
    # Code permettant de tester les fonctions du module
    # qui ne sera pas exécuté lorsque celui-ci sera importé
```

Les avantage de cette façon de faire sont nombreux :

- Facilité de modification d'une partie (par exemple si on veut améliorer le jeu contre l'ordinateur, il suffit de retravailler ce module uniquement)
- Facilité d'ajout de fonctionnalités (Ajout de la possibilité de jouer en réseau par exemple en créant un module dédié)
- Permet de travailler à plusieurs sur un même projet en ayant pour chaque intervenant la responsabilité exclusive d'un ou plusieurs modules

Dans la pratique, cette approche est systématiquement utilisée pour tous les projets informatiques.

II – Mise au point

Pour illustrer cette partie du cours, nous utiliserons le programme suivant :

```
def est_croissant(t):
    i = len(t) - 1
    while i >= 0:
        if t[i] <= t[i + 1]:
            return True
        else:
            return False
        i -= 1
```

1) Tests

Dès qu'une fonction est écrite (même les plus simples), il est important de la tester pour vérifier son bon fonctionnement. Pour cela, on utilise des jeux de tests, c'est-à-dire un ensemble de valeurs d'entrées pour la fonction dont on connait la valeur de sortie théorique et on compare cette valeur à celle données par la fonction.
Exemple : on programme une fonction `float racine(float x)` qui calcule la racine carrée d'un nombre flottant. On testera que `racine(4.0)` renvoie bien `2.0`, que `racine(49)` renvoie bien `7.0` et que `racine(2)` renvoie bien `1.414213562373095`
Il faut bien penser à tester les valeurs particulières comme `racine(1.0)` et `racine(0.0)`.
De tels tests (qui n'opèrent que sur une valeur bien précise ou quelques valeurs choisies) sont appelés **tests unitaires**.
On peut également écrire un petit programme de test qui fabrique un jeu de tests aléatoires pour vérifier le fonctionnement. Pour ce faire on aura besoin d'une fonction pour créer des paramètres d'entrée de manière aléatoire et d'une autre pour valider le résultat de la fonction à tester.

Exemple :

```
def test_fonction_tri(f, nb_tests:int, taille_max:int):
    nb_erreurs = 0
    for taille in range(taille_max):
        for _ in range(nb_tests):
            t = tableau_aleatoire(taille)
            tableau_original = t.copy()
            f(t)
            if not est_trie(t):
                print("Erreur pour le tableau", tableau_original)
                print("trié en", t)
                nb_erreurs += 1
                if nb_erreurs > 20:
                    print("Trop d'erreurs, abandon du test...")
                    return
    print("Fin de la série de test.", nb_erreurs, "erreurs détectées")
```

Application 1 :
Pour notre fonction `est_croissant`, écrire une docstring, puis réaliser un test simple sur un tableau croissant. Qu'obtient-on ?

2) Débogage

a. Exceptions et traceback

Lorsque l'interpréteur détecte une erreur dans le programme, il **lève une exception** qui arrête le traitement normal du flux du programme. Dans le cas d'une erreur de syntaxe, l'exception est même levée avant que la moindre ligne de code soit exécutée.
Pour les autres types d'erreur, l'interpréteur affiche le **Traceback** qui indique le type d'erreur ainsi que la pile des appels : ce sont les **informations de contexte**.
La pile des appels est la suite de fonctions qui ont été appelées successivement au moment de l'exécution de la ligne qui a causé l'erreur.

Les informations du traceback doivent être analysées avec soin car elles mènent souvent rapidement au diagnostic de l'erreur.

Plusieurs instructions permettent également de gérer les exceptions en python :

```
try:
    code à exécuter qui risque de provoquer une exception
except type_d_exception as e:
    bloc de traitement de l'erreur
[finally]:
    Bloc exécuté dans tous les cas (qu'il y ai eu exception ou pas)
```

On peut également provoquer la levée d'une exception avec l'instruction `raise` *`type_d_exception`*.

Application 2 :
A partir de l'assertion manquée vue précédemment, corriger la fonction `est_croissant` pour qu'elle ne génère plus d'exception.

b. Suivi d'exécution et utilisation du débogueur

Application 3 :
Effectuer d'autre tests pour voir si la fonction est correcte.

Lorsque le comportement du programme n'est pas correct (échec aux jeux de tests notamment) mais qu'aucune exception n'est levée, il y a souvent nécessité de déboguer le programme. Pour cela il est utile de suivre l'exécution du programme au ralenti en surveillant l'évolution des variables afin de voir à quel moment (et pourquoi) celle-ci s'écarte des valeurs attendues.

Pour ce faire on peut rajouter des instructions dans le programme pour afficher des informations dans la console python au fur et à mesure de l'exécution du programme.

Application 4 :
Modifier la fonction précédente en rajoutant les lignes suivantes puis exécutez-là sur l'exemple qui la faisait échouer afin de comprendre le problème. Corriger le programme en conséquence.

```
def est_croissant(t):
    """Renvoie True si les éléments de t sont rangés en ordre croissant."""
    print("Test avec le tableau", t)
    i = len(t) - 1
    while i >= 0:
        print("Nouveau tour avec i =", i)
        if t[i-1] <= t[i]:
            return True
        else:
            return False
        i -= 1
```

Une fois le programme corrigé, on retire les lignes qui affichaient les informations utiles à la correction du programme en essayant de ne pas en oublier (ou on les met en commentaire). Ceci peut être fastidieux. De même il est parfois difficile de retrouver l'information utile lorsqu'il y en a beaucoup d'affichées. Pour ces raisons, on utilise souvent préférentiellement un **débogueur**.

Un débogueur est un programme permettant d'exécuter le programme à déboguer en suivant le flux du programme et la valeur des variables.
Le débogueur propose principalement 2 modes d'exécutions :

- Le mode pas à pas où l'utilisateur avance manuellement d'un ou plusieurs « pas » élémentaires dans le programme en rentrant ou pas dans les sous-fonctions. Dans Thonny, cela se fait avec la commande « Déboguer le script courant (Ctrl-F5) » puis les touches F6 ou F7 pour avancer dans l'exécution. A tout instant, on peut voir l'état des variables (dans la fenêtre variables)
- Le mode points d'arrêt où l'utilisateur pose des « points d'arrêt » dans le programme (Avec Thonny, cela se fait en double-cliquant sur le numéro de ligne). Lorsqu'on lance le débogage, l'exécution se déroule alors normalement jusqu'à ce que l'interpréteur atteigne une ligne où se trouve un point d'arrêt. A ce moment-là, l'exécution est stoppée, l'affichage des variables réactualisé, et on bascule en mode pas à pas. Ce mode est particulièrement utile lorsqu'on souhaite déboguer un bout de code qui n'est exécuté qu'assez tard dans l'exécution normale du programme.

NB : Dans les débogueurs plus évolués, on peut également poser des points d'arrêt conditionnels. Ce sont des lignes du programme sur lesquels l'exécution ne s'arrêtera que si une expression qu'on aura précisée est vrai (par exemple arrêt si la variable i a une valeur supérieure ou égale à 100). On peut également déclarer des « expression espionnes » qui seront évaluées à chaque arrêt de l'exécution.

3) Preuve de la correction d'un algorithme : l'invariant de boucle

Quand les programmes contiennent des boucles, il peut être difficile d'être sûr que celle-ci est écrite correctement, que les indices et variables de boucles sont corrects au début, pendant et à la fin de la boucle. Pour répondre à cette problématique (« Ma boucle est-elle correctement programmée ? »), on peut utiliser un **invariant de boucle** : c'est une propriété qui est vraie initialement, pendant chaque tour de boucle et donc nécessairement vraie à la fin de la boucle.

Voyons ceci à travers un exemple :

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

Grâce à l'invariant de boucle, on a pu PROUVER que notre fonction réalisait bien la division euclidienne. C'est évidemment quelque chose de très fort en programmation puisqu'on est alors sûr que si les préconditions sont satisfaites, le programme donnera une réponse juste (à condition de se terminer ce qui est assez simple à démontrer).

Références :

Documentation du code python : https://www.codeflow.site/fr/article/documenting-python-code
PEP8 (guide d'écriture) : https://www.python.org/dev/peps/pep-0008/