Chapitre 4

Bonnes pratiques

En guise de bonnes pratiques, les développeurs de Python ont mis en ligne ce petit poème, traduit ici en français [1] :

> Mieux vaut être beau que laid.
> L'explicite vaut mieux que l'implicite.
> Simple vaut mieux que complexe.
> Complexe vaut mieux que compliqué.
> Plat vaut mieux qu'imbriqué.
> Épars vaut mieux que dense.
> La lisibilité compte.
> Les cas particuliers ne sont pas assez spéciaux pour enfreindre les règles,
> Bien que l'aspect pratique l'emporte sur la pureté.
> Les erreurs ne doivent jamais passer inaperçues,
> À moins qu'elles ne soient explicitement réduites au silence.
> Face à l'ambiguïté, refusez la tentation de deviner.
> Il doit y avoir une et, de préférence, une seule façon évidente de le faire,
> Bien que cette façon de faire ne soit pas évidente au début, à moins d'être Néerlandais.
> Mieux vaut maintenant que jamais,
> Bien que jamais soit souvent mieux que maintenant.
> Si la mise en œuvre est difficile à expliquer, c'est une mauvaise idée.
> Si la mise en œuvre est facile à expliquer, c'est peut-être une bonne idée.
> Les espaces de noms sont une excellente idée – faisons-en davantage !

Nous n'expliciterons pas nécessairement tous les aspects du poème. Voyons cependant un ensemble non exhaustif de bonnes pratiques. Elles sont destinées à rendre le code plus lisible, plus maintenable, plus clair, plus débuguable,... bref, à écrire du beau code.

Nous allons procéder par zoom-out : de plus élémentaire au plus large.

4.1 Dans les fonctions

4.1.1 Nommage des variables

Les variables qui ont un sens doivent être nommées en rapport. On ne réserve les noms d'un seul caractère qu'aux variables sans signification comme des compteurs de boucle.

Il faut éviter à tout prix les accents dans les noms de variables et rester avec l'alphabet américain.

La convention veut qu'une variable qui joue le rôle d'une constante soit écrite en MAJUSCULE.

1. Pour la version originale, voir `https://pep20.org/`

Une variable qui varie réellement au cours de l'exécution sera au format "snake case" : des lettres minuscules avec des mots séparés par un underscore _.

Par exemple `CHAINE_CODEE` sera une constante (visiblement une chaîne de caractères), `a` est un compteur et `distance_par_rapport_au_centre` semble être un nombre réel.

Par contre, `var` ou `numcase` sont à proscrire : on ne sait pas ce qu'elles représentent... et `nbCase` ne respecte pas la convention.

4.1.2 Espaces et sauts de lignes

La structure du langage impose d'indenter son code. Il existe deux écoles au sein des développeurs Python : un caractère de tabulation ou des espaces. Pour les espaces, la convention veut qu'on utilise 4 espaces par indentation.

Pour plus de lisibilité, il est préférable d'entourer d'espaces les opérateurs standards : espaces avant et après `=`, `==`, `<`, `>`, `+`, `-`, `*`, `/` et `%`.

Par exemple, ce code

```
rayon_cercle=5
distance_au_centre=(x-x0)**2+(y-y0)**2
if distance_au_centre<rayon_cercle:
    print("point dans le disque")
```

est moins lisible que celui-ci :

```
rayon_cercle = 5
distance_au_centre = ( x - x0 ) ** 2 + ( y - y0 ) ** 2
if distance_au_centre < rayon_cercle:
    print("point dans le disque")
```

De la même façon, il ne faut pas hésiter à sauter des lignes ! Plus le code est aéré, plus il est lisible.

4.1.3 Docstrings

Dans Python, on peut obtenir de d'aide avec la commande `help`. Par exemple, l'aide de la fonction `print` s'obtient avec `help(print)`. Ceci est très pratique, surtout quand on ne se souvient plus du nom de certains paramètres ou que l'on n'est pas sûr de pouvoir utiliser la fonction dans notre contexte.

Cette aide (on appelle ça une **docstring**) ~~peut~~ doit être créée aussi pour nos fonctions personnelles. Pour cela, il suffit de mettre le texte de l'aide entre triple guillemet-double sur la **ligne juste après** le `def` de la fonction.

Cette aide doit comprendre :
- quelques mots qui décrivent très rapidement la fonction.
- des explications supplémentaires plus détaillées sur ce que fait la fonction.
- des précisions sur chaque paramètres d'entrée (son type, ses contraintes, son rôle, sa valeur par défaut si il est optionnel,...)
- des précisions sur ce que retourne la fonction (idem aux paramètres d'entrée)
- en option, des exemples d'utilisation si la fonction est un peu compliquée.

Il n'est pas rare que la docstring soit plus volumineuse que le corps de la fonction elle-même...

```python
def est_dans_disque(x_point, y_point, x_centre, y_centre, rayon, ferme=True):
    """
    Vérifie si un point est à l'intérieur d'un disque.

    La fonction retourne True si le point de coordonnées (x_point, y_point) est
    à l'intérieur du disque de centre (x_centre, y_centre) et de rayon rayon.

    Si ferme == True, le disque est fermé ce qui signifie que
    le cercle frontière appartient au disque.
    Si ferme == False, le cercle frontière n'appartient pas au disque.

    Entrées:
    x_point, y_point: (float) coordonnées du point à tester.
    x_centre, y_centre: (float) coordonnées du centre du disque.
    rayon: (float, rayon >= 0) rayon du disque
    ferme: (boolean) le disque est-t-il fermé ? True par défaut

    Sortie:
    (boolean) True si le point est à l'intérieur du disque, False sinon.
    """
    distance_carre_au_centre = ( x_point - x_centre ) ** 2 +
                               ( y_point - y_centre ) ** 2
    rayon_carre = rayon ** 2
    if distance_carre_au_centre < rayon_carre:
        return True
    if ferme and distance_carre_au_centre == rayon_carre:
        return True
    return False
```

Je vous conseille d'écrire cette docstring avant d'écrire le code de la fonction. Ceci vous permettra de mettre vos idées au clair sur ce qu'elle est censée faire, de préciser les paramètres que vous autorisez et la sortie qu'elle fournira.

En général, c'est long à écrire et ça semble être du "code mort" mais c'est beaucoup plus utile qu'il n'y parait : si vous n'êtes pas capable de décrire ce que fait la fonction, vous ne serez pas capable d'écrire son code !

4.1.4 Commentaires

Il ne faut pas hésiter à commenter son code. Attention cependant, il faut que les commentaires soient utiles et non pas simplement des paraphrases du code.

En général, si en lisant le code, vous ne comprenez pas immédiatement ce qu'il fait ou si vous ne comprenez pas pourquoi il fait ça, c'est qu'il faut un commentaire.

Le commentaire est sur la ligne à commenter ou sur la(les) ligne(s) précédente(s).

Mauvais commentaire :

```python
rayon_carre = rayon ** 2   # on calcule le carré du rayon
```

Commentaire bien meilleur :

```python
# on compare les distances au carré pour ne pas avoir à utiliser la racine
rayon_carre = rayon ** 2
```

4.1.5 L'instruction assert

Dans certains cas, un paramètre d'une fonction possède des contraintes. Par exemple, le rayon d'un cercle ne peut pas être négatif, un division par zéro n'est pas définie,...

Bien entendu, on pourrait faire confiance à l'utilisateur sur le fait qu'il va donner des arguments corrects mais c'est une très mauvaise idée : tôt ou tard, il fera la bêtise et, au mieux ça va planter dans votre fonction, au pire elle va renvoyer une valeur aberrante. Dans les deux cas, ce n'est pas satisfaisant car l'utilisateur risque d'avoir du mal à s'apercevoir que ce sont ses valeurs qui posent problème et pas votre fonction qui est mal fichue... Ce qu'il faudrait c'est un moyen de faire planter votre fonction "proprement" en avertissant l'utilisateur qu'il tente de faire n'importe quoi.

C'est l'un des rôles de l'instruction `assert`. Sa signature est : `assert test, chaine`.

Si le `test` échoue (c'est-à-dire est faux), le programme va planter et afficher (entre autre) la `chaine`, sinon rien ne va se passer.

Exemple :

```
def est_dans_disque(x_point, y_point, x_centre, y_centre, rayon):
    """ ... docstring ... """

    assert rayon >= 0, "un rayon ne peut pas être négatif !"

    distance_carre_au_centre = ( x_point - x_centre ) ** 2 +
                               ( y_point - y_centre ) ** 2
    return distance_carre_au_centre <= rayon ** 2
```

Attention cependant, ceci n'est pas un moyen parfait de tester les paramètres d'entrée. Il existe un moyen (plutôt simple) de contourner ces `assert` [2].

4.1.6 Jeux de tests

L'autre fonctionnalité de l'instruction `assert` est créer des jeux de test pour vérifier que la fonction qu'on vient de créer fait bien ce qu'elle est censée faire.

Pour ceci, on va créer une fonction qui aura le même nom que la fonction à tester avec le préfixe `test_`.

Dans l'idéale, celle-ci testera tous les cas possibles à l'aide d'autant d'`assert` qu'il le faudra. En lançant cette fonction de test [3], on vérifiera donc le comportement de la fonction initiale.

```
def est_dans_disque(x_point, y_point, x_centre, y_centre, rayon, ferme):
    ...

def test_est_dans_disque():
    assert not est_dans_disque(10, 20, 0, 1, 3),
           "le point devrait être trop loin"
    assert est_dans_disque(1, 4, -3, 2, 6),
           "le point devrait être dedans"
    assert est_dans_disque(7, 4, 3, 1, 5, True),
           "le point est dedans, sur la frontière"
    assert not est_dans_disque(4, 2, 1, -2, 5, False),
           "le point est dehors, sur la frontière"
    print("Tous les tests sont passés avec succès !")
```

2. Sur le lien `https://www.pythoniste.fr/python/linstruction-dassertion-en-python-assert/` l'auteur explique même, exemple à l'appui, que c'est une très mauvaise idée d'utiliser assert dans ce but...

3. On peut même automatiser ces tests avec `pytest`. Voir par exemple `https://openclassrooms.com/fr/courses/4425126-testez-votre-projet-avec-python/4435144-ajoutez-des-tests-avec-pytest`

4.1.7 Des print avec modération

Autant que faire se peut, il faut limiter les `print` à deux utilisations :
- dans les fonctions dont le but est d'afficher des informations à l'utilisateur,
- pour déboguer quand une fonction n'a pas le comportement attendu (la fonction `test_*` plante).

Pour cette dernière utilisation, les `print` sont insérés à des endroits stratégiques et enlevés dès que possible. Ne pas oublier de lire les messages d'erreurs de Python ! En général, ils aident beaucoup à déterminer le problème et indiquent même la ligne qui coince. C'est donc la première chose à lire en cas de soucis !

Par exemple, cette fonction n'a pas le comportement attendu :

```
1 def somme_entiers(n):
2     s = 0
3     for i in range(n):
4         s = s + n
5     return s
```

En effet, `somme_entiers(3) == 9` alors qu'on aurait espérer $1+2+3=6$...
En ajoutant un `print` au début de la boucle et un après l'opération pour visualiser ce qu'il se passe :

```
1 def somme_entiers(n):
2     s = 0
3     for i in range(n):
4         print("n=", n, "    s=", s, "    i=", i)
5         s = s + n
6         print("nouveau s=", s)
7     return s
```

On constate ceci :

```
1 somme_entiers(3)
2 n=3    s=0    i=0
3 nouveau s=3
4 n=3    s=3    i=1
5 nouveau s=6
6 n=3    s=6    i=2
7 nouveau s=9
8 9
```

On voit alors deux problèmes :
- D'abord, ligne 3 de la sortie, le nouveau `s` aurait du rester à 0. C'est parce que sur la ligne 4 de la fonction d'origine, on a mis `n` à la place de `i`.
- Ensuite, on voit qu'on ne va pas jusqu'à 3 mais qu'on s'arrête à 2. Pour corriger le problème, on peut mettre un `n+1` à la place de `n` ligne 3 de la fonction originale.

On obtient alors une version plus correcte :

```
1 def somme_entiers(n):
2     s = 0
3     for i in range(n+1):
4         s = s + i
5     return s
```

Les autres utilisations de `print` doivent être réduites au minimum. Si une fonction doit rendre un résultat, on utilisera `return`.

4.2 Dans le fichier

4.2.1 Organisation générale

L'organisation générale d'un fichier Python doit respecter quelques standards.
— Les premières lignes correspondent à l'inclusion de toutes les bibliothèques qui seront utiles au fichier.
— Vient ensuite l'ensemble des variables CONSTANTES (notez les majuscules 😉) qui seront utilisées.
— Puis, c'est au tour de la définition de toutes les fonctions.
— Enfin, le programme principal, celui qui utilise tout le reste et qui fait réellement le travail.
Pour la dernière partie, on peut (sans obligation) se référer au point 4.2.4 plus bas.

Naturellement, tout ça est entouré de docstrings et commentaires !

4.2.2 Des fonctions partout

Une règle générale à appliquer en informatique est

$$\textit{une fonction} = \textit{une fonctionnalité} \text{ et } \textit{une fonctionnalité} = \textit{une fonction}$$

Il ne faut pas hésiter à scinder son code. Il sera plus lisible si les fonctions sont courtes et bien nommées.

Par exemple, même si on ne sait pas comment font les fonctions annexes, on comprend parfaitement ce que fait celle-ci :

```
def moyenne_donnees_dans_fichier(nom_fichier, type_donnees):
    contenu = contenu_fichier(nom_fichier)
    donnees = extraction_donnees(contenu, type_donnees)
    moyenne = calcul_moyenne(donnees)
    return moyenne
```

De la même façon, si vous utilisez plusieurs fois quelques lignes de code identiques, il vaut mieux en faire une fonction. On parle alors de factorisation de code, comme en mathématiques. Le code résultant sera plus clair et plus facilement maintenable (correction de bugs et apport de nouvelles fonctionnalités).

Enfin, attention aux signatures des fonctions ! Toutes les variables utilisées dans une fonction doivent être données en argument. Il n'y a peut-être que les constantes qui peuvent échapper à cette règle... (Si malgré tout, vous décidez de contrevenir à cette règle, n'oubliez par le mot clé `global` ...) Et la sortie doit se faire par un `return` et pas par un `print`.

4.2.3 Encore des docstrings

Comme pour les fonctions, on peut mettre une docstring pour un fichier Python entier.

Cette docstring doit se situer au tout début (avant les `import`). Elle doit lister toutes les fonctions (utiles) du fichier, expliquer le contexte dans lequel les utiliser et donner quelques exemples d'utilisation de chaque fonction.

Dans le cas où vous voudriez partager votre fichier avec d'autres personnes, pensez à rajouter une licence à votre fichier ainsi que des coordonnées pour pouvoir vous joindre (un mail suffit). La licence est importante pour que vos utilisateurs sachent ce qu'ils ont le droit de faire avec votre code. Cette licence se met en fin de docstring de fichier.

4.2.4 Inclusions

Cette dernière bonne pratique est là pour information. Elle n'est vraiment utile que quand on fractionne son code en plusieurs fichiers, ce qui ne sera vraisemblablement pas le cas cette année pour nous.

C'est typiquement le cas pour les moyens ou gros projets : En général, (au moins) un fichier pour les interactions homme-machine, (au moins) un fichier pour les manipulations de données et un fichier pour l'intelligence générale du programme. Ce dernier contient la fonction principale, celle qu'il faut appeler pour lancer le programme. Ces fichiers annexes sont appelés des modules en python ou bibliothèques [4] en général.

Vous avez déjà importé des bibliothèques Python. Il suffit d'ajouter `import` et le nom de la bibliothèque en début de fichier. Elles servent à ajouter des fonctionnalités à Python. Par exemple pour utiliser la racine carrée, il faut `import math` puis on utilise `math.sqrt`.

Comment sont créées les bibliothèques ? Et bien il s'agit de simples fichiers python comme ceux que vous écrivez. L'instruction `import` lit simplement le fichier.

Mais alors, il ne faut pas exécuter de code dans une bibliothèque, simplement définir des fonctions. Comment faire pour lancer du code si on appelle directement le fichier et ne rien faire si on importe le fichier dans un autre ?

La méthode est simple : il faut utiliser ceci en guise de "fonction principale" :

```
if __name__ == "__main__":
    # fonction principale
    ....
```

On utilise alors cette technique dans tous les fichiers : dans les annexes, on teste toutes les fonctions comme indiqué dans la bonne pratique 4.1.6 :

```
if __name__ == "__main__":
    # tests
    test_fct1()
    test_fct2()
    ...
```

Dans le fichier principale, on met l'appel à la fonction principale

```
if __name__ == "__main__":
    fonction_principale()
```

4.3 Quelques exercices

Exercice 14.

1. Écrire la docstring et un jeu de test pour une fonction `trinome_solutions(a,b,c)` qui donne la liste des solutions réelles du trinôme $ax^2+bx+c=0$, rangées par ordre croissant où a, b et c sont des entiers et $a \neq 0$. Si $a=0$, la fonction doit renvoyer une erreur !
2. Écrire le code de la fonction `trinome_solutions`.

Exercice 15.

1. Rappeler une condition sur a, b et c pour qu'il soit possible de tracer un triangle dont les cotés mesurent a, b et c.

4. souvent abrégé lib pour library en anglais

2. Écrire une fonction `nature_triangle(a, b, c)`, sa docstring et son jeu de tests. Cette fonction donnera (`return`) la nature exacte d'un triangle dont les cotés mesurent a, b et c sous forme de chaîne de caractères : `"équilatéral"`, `"isocèle"`, `"rectangle"`, `"isocèle rectangle"` ou `"quelconque"` ; elle plantera volontairement si un triangle n'est pas possible avec les arguments donnés.