15 Bonnes pratiques en programmation Python

Comme vous l'avez constaté dans tous les chapitres précédents, la syntaxe de Python est très permissive. Afin d'uniformiser l'écriture de code en Python, la communauté des développeurs Python recommande un certain nombre de règles afin qu'un code soit lisible. Lisible par quelqu'un d'autre, mais également, et surtout, par soi-même. Essayez de relire un code que vous avez écrit « rapidement » il y a un 1 mois, 6 mois ou un an. Si le code ne fait que quelques lignes, il se peut que vous vous y retrouviez, mais s'il fait plusieurs dizaines voire centaines de lignes, vous serez perdus.

Dans ce contexte, le créateur de Python, Guido van Rossum, part d'un constat simple : « *code is read much more often than it is written* » (« le code est plus souvent lu qu'écrit »). Avec l'expérience, vous vous rendrez compte que cela est parfaitement vrai. Alors plus de temps à perdre, voyons en quoi consistent ces bonnes pratiques.

Plusieurs choses sont nécessaires pour écrire un code lisible : la syntaxe, l'organisation du code, le découpage en fonctions (et possiblement en classes que nous verrons dans le chapitre 19 *Avoir la classe avec les objets*), mais souvent, aussi, le bon sens. Pour cela, les « PEP » peuvent nous aider.

Définition

Afin d'améliorer le langage Python, la communauté qui développe Python publie régulièrement des *Python Enhancement Proposal* (PEP), suivi d'un numéro. Il s'agit de propositions concrètes pour améliorer le code, ajouter de nouvelles fonctionnalités, mais aussi des recommandations sur la manière d'utiliser Python, bien écrire du code, etc.

On va aborder dans ce chapitre sans doute la plus célèbre des PEP, à savoir la PEP 8, qui est incontournable lorsque l'on veut écrire du code Python correctement.

Définition

On parle de code **pythonique** lorsque ce dernier respecte les règles d'écriture définies par la communauté Python mais aussi les règles d'usage du langage.

15.1 De la bonne syntaxe avec la PEP 8

La PEP 8 *Style Guide for Python Code* est une des plus anciennes PEP (les numéros sont croissants avec le temps). Elle consiste en un nombre important de recommandations sur la syntaxe de Python. Il est vivement recommandé de lire la PEP 8 en entier au moins une fois pour avoir une bonne vue d'ensemble. On ne présentera ici qu'un rapide résumé de cette PEP 8.

15.1.1 Indentation

On a vu que l'indentation est obligatoire en Python pour séparer les blocs d'instructions. Cela vient d'un constat simple, l'indentation améliore la lisibilité d'un code. Dans la PEP 8, la recommandation pour la syntaxe de chaque niveau d'indentation est très simple : 4 espaces. N'utilisez pas autre chose, c'est le meilleur compromis.

> **⚠ Attention**
>
> Afin de toujours utiliser cette règle des 4 espaces pour l'indentation, il est essentiel de régler correctement votre éditeur de texte. Consultez pour cela l'annexe *Installation de Python* disponible en ligne. Avant d'écrire la moindre ligne de code, faites en sorte que lorsque vous pressez la touche tabulation, cela ajoute 4 espaces (et non pas un caractère tabulation).

15.1.2 Importation des modules

Comme on l'a vu au chapitre 8 *Modules*, le chargement d'un module se fait avec l'instruction `import module` plutôt qu'avec `from module import *`.

Si on souhaite ensuite utiliser une fonction d'un module, la première syntaxe conduit à `module.fonction()` ce qui rend explicite la provenance de la fonction. Avec la seconde syntaxe, il faudrait écrire `fonction()` ce qui peut :

- mener à un conflit si une de vos fonctions a le même nom ;
- rendre difficile la recherche de documentation si on ne sait pas d'où vient la fonction, notamment si plusieurs modules sont chargés avec l'instruction `from module import *`

Par ailleurs, la première syntaxe définit un « espace de noms » (voir chapitre 19 *Avoir la classe avec les objets*) spécifique au module.

Dans un script Python, on importe en général un module par ligne. D'abord les modules internes (classés par ordre alphabétique), c'est-à-dire les modules de base de Python, puis les modules externes (ceux que vous avez installés en plus).

Si le nom du module est trop long, on peut utiliser un alias. L'instruction `from` est tolérée si vous n'importez que quelques fonctions clairement identifiées.

En résumé :

```
import module_interne_1
import module_interne_2
from module_interne_3 import fonction_spécifique
from module_interne_4 import constante_1, fonction_1, fonction_2

import module_externe_1
import module_externe_2
import module_externe_3_qui_a_un_nom_long as mod3
```

15.1.3 Règles de nommage

Les noms de variables, de fonctions et de modules doivent être de la forme :

```
ma_variable
fonction_test_27()
mon_module
```

c'est-à-dire en minuscules avec un caractère « souligné » (« tiret du bas » ou *underscore* en anglais) pour séparer les différents « mots » dans le nom.

Les constantes sont écrites en majuscules :

```
MA_CONSTANTE
VITESSE_LUMIERE
```

Les noms de classes (chapitre 19) et les exceptions (chapitre 21) sont de la forme :

```
MaClasse
MyException
```

Remarque

Le style recommandé pour nommer les variables et les fonctions en Python est appelé *snake_case*. Il est différent du *CamelCase* utilisé pour les noms des classes et des exceptions.

Pensez à donner à vos variables des noms qui ont du sens. Évitez autant que possible les `a1`, `a2`, `i`, `truc`, `toto`... Les noms de variables à un caractère sont néanmoins autorisés pour les boucles et les indices :

```
>>> ma_liste = [1, 3, 5, 7, 9, 11]
>>> for i in range(len(ma_liste)):
...     print(ma_liste[i])
```

Bien sûr, une écriture plus « pythonique » de l'exemple précédent permet de se débarrasser de l'indice `i` :

```
>>> ma_liste = [1, 3, 5, 7, 9, 11]
>>> for entier in ma_liste:
...     print(entier)
...
```

Enfin, des noms de variable à une lettre peuvent être utilisés lorsque cela a un sens mathématique (par exemple, les noms `x`, `y` et `z` évoquent des coordonnées cartésiennes).

15.1.4 Gestion des espaces

La PEP 8 recommande d'entourer les opérateurs (`+`, `-`, `/`, `*`, `==`, `!=`, `>=`, `not`, `in`, `and`, `or` ...) d'un espace avant et d'un espace après. Par exemple :

```
# code recommandé :
ma_variable = 3 + 7
mon_texte = "souris"
mon_texte == ma_variable
# code non recommandé :
ma_variable=3+7
mon_texte="souris"
mon_texte== ma_variable
```

Il n'y a, par contre, pas d'espace à l'intérieur de crochets, d'accolades et de parenthèses :

```
# code recommandé :
ma_liste[1]
mon_dico{"clé"}
ma_fonction(argument)
# code non recommandé :
ma_liste[ 1 ]
mon_dico{"clé" }
ma_fonction( argument )
```

Ni juste avant la parenthèse ouvrante d'une fonction ou le crochet ouvrant d'une liste ou d'un dictionnaire :

```
# code recommandé :
ma_liste[1]
mon_dico{"clé"}
ma_fonction(argument)
# code non recommandé :
ma_liste [1]
mon_dico {"clé"}
ma_fonction (argument)
```

On met un espace après les caractères `:` et `,` (mais pas avant) :

```
# code recommandé :
ma_liste = [1, 2, 3]
```

```
mon_dico = {"clé1": "valeur1", "clé2": "valeur2"}
ma_fonction(argument1, argument2)
# code non recommandé :
ma_liste = [1 , 2 ,3]
mon_dico = {"clé1": "valeur1", "clé2": "valeur2"}
ma_fonction(argument1 ,argument2)
```

Par contre, pour les tranches de listes, on ne met pas d'espace autour du `:`

```
ma_liste = [1, 3, 5, 7, 9, 1]
# code recommandé :
ma_liste[1:3]
ma_liste[1:4:2]
ma_liste[::2]
# code non recommandé :
ma_liste[1 : 3]
ma_liste[1: 4:2 ]
ma_liste[ : :2]
```

Enfin, on n'ajoute pas plusieurs espaces autour du `=` ou des autres opérateurs pour faire joli :

```
# code recommandé :
x1 = 1
x2 = 3
x_old = 5
# code non recommandé :
x1    = 1
x2    = 3
x_old = 5
```

15.1.5 Longueur de ligne

Une ligne de code ne doit pas dépasser 79 caractères, pour des raisons tant historiques que de lisibilité.

On a déjà vu au chapitre 1 *Introduction* que le caractère `\` permet de couper des lignes trop longues. Par exemple :

```
>>> ma_variable = 3
>>> if ma_variable > 1 and ma_variable < 10 \
... and ma_variable % 2 == 1 and ma_variable % 3 == 0:
...     print(f"ma variable vaut {ma_variable}")
...
ma variable vaut 3
```

À l'intérieur d'une parenthèse, on peut revenir à la ligne sans utiliser le caractère `\`. C'est particulièrement utile pour préciser les arguments d'une fonction ou d'une méthode, lors de sa création ou lors de son utilisation :

```
>>> def ma_fonction(argument_1, argument_2,
...                  argument_3, argument_4):
...     return argument_1 + argument_2
...
>>> ma_fonction("texte très long", "tigre",
...             "singe", "souris")
'texte très longtigre'
```

Les parenthèses sont également très pratiques pour répartir sur plusieurs lignes une chaîne de caractères qui sera affichée sur une seule ligne :

```
>>> print("ATGCGTACAGTATCGATAAC"
...       "ATGACTGCTACGATCGGATA"
...       "CGGGTAACGCCATGTACATT")
ATGCGTACAGTATCGATAACATGACTGCTACGATCGGATACGGGTAACGCCATGTACATT
```

Notez qu'il n'y a pas d'opérateur `+` pour concaténer les trois chaînes de caractères et que celles-ci ne sont pas séparées par des virgules. À partir du moment où elles sont entre parenthèses, Python les concatène automatiquement.

On peut aussi utiliser les parenthèses pour évaluer un expression trop longue :

```
>>> ma_variable = 3
>>> if (ma_variable > 1 and ma_variable < 10
... and ma_variable % 2 == 1 and ma_variable % 3 == 0):
...     print(f"ma variable vaut {ma_variable}")
...
ma variable vaut 3
```

Les parenthèses sont aussi très utiles lorsqu'on a besoin d'enchaîner des méthodes les unes à la suite des autres. Un exemple se trouve dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique*, dans la partie consacrée au module *pandas*.

Enfin, il est possible de créer des listes ou des dictionnaires sur plusieurs lignes, en sautant une ligne après une virgule :

```
>>> ma_liste = [1, 2, 3,
...             4, 5, 6,
...             7, 8, 9]
>>> mon_dico = {"clé1": 13,
...             "clé2": 42,
...             "clé3": -10}
```

15.1.6 Lignes vides

Dans un script, les lignes vides sont utiles pour séparer visuellement les différentes parties du code.

Il est recommandé de laisser deux lignes vides avant la définition d'une fonction ou d'une classe et de laisser une seule ligne vide avant la définition d'une méthode (dans une classe).

On peut aussi laisser une ligne vide dans le corps d'une fonction pour séparer les sections logiques de la fonction, mais cela est à utiliser avec parcimonie.

15.1.7 Commentaires

Les commentaires débutent toujours par le symbole `#` suivi d'un espace. Ils donnent des explications claires sur l'utilité du code et doivent être synchronisés avec le code, c'est-à-dire que si le code est modifié, les commentaires doivent l'être aussi (le cas échéant).

Les commentaires sont sur le même niveau d'indentation que le code qu'ils commentent. Les commentaires sont constitués de phrases complètes, avec une majuscule au début (sauf si le premier mot est une variable qui s'écrit sans majuscule) et un point à la fin.

La PEP 8 recommande très fortement d'écrire les commentaires en anglais, sauf si vous êtes à 120% sûr que votre code ne sera lu que par des francophones. Dans la mesure où vous allez souvent développer des programmes scientifiques, nous vous conseillons d'écrire vos commentaires en anglais.

Soyez également cohérent entre la langue utilisée pour les commentaires et la langue utilisée pour nommer les variables. Pour un programme scientifique, les commentaires et les noms de variables sont en anglais. Ainsi `ma_liste` deviendra `my_list` et `ma_fonction` deviendra `my_function` (par exemple).

Les commentaires qui suivent le code sur la même ligne sont à éviter le plus possible et doivent être séparés du code par au moins deux espaces :

```
1  x = x + 1   # My wonderful comment.
```

Remarque

Nous terminerons par une remarque qui concerne la syntaxe, mais qui n'est pas incluse dans la PEP 8. On nous pose souvent la question du type de guillemets à utiliser pour déclarer une chaîne de caractères. Simples ou doubles ?

```
1  >>> var_1 = "Ma chaîne de caractères"
2  >>> var_1
3  'Ma chaîne de caractères'
4  >>> var_2 = 'Ma chaîne de caractères'
5  >>> var_2
6  'Ma chaîne de caractères'
7  >>> var_1 == var_2
8  True
```

Vous constatez dans l'exemple ci-dessus que pour Python, c'est exactement la même chose. Et à notre connaissance, il n'existe pas de recommandation officielle sur le sujet.

Nous vous conseillons cependant d'utiliser les guillemets doubles car ceux-ci sont, de notre point de vue, plus lisibles.

15.2 Les *docstrings* et la PEP 257

Les *docstrings*, que l'on pourrait traduire par « chaînes de documentation » en français, sont un élément essentiel de nos programmes Python comme on l'a vu au chapitre 14 *Création de modules*. À nouveau, les développeurs de Python ont émis des recommandations dans la PEP 8 et plus exhaustivement dans la PEP 257 sur la manière de rédiger correctement les *docstrings*. En voici un résumé succinct.

De manière générale, écrivez des *docstrings* pour les modules, les fonctions, les classes et les méthodes. Lorsque l'explication est courte et compacte comme dans certaines fonctions ou méthodes simples, utilisez des *docstrings* d'une ligne :

```
"""Docstring simple d'une ligne se finissant par un point."""
```

Lorsque vous avez besoin de décrire plus en détail un module, une fonction, une classe ou une méthode, utilisez une *docstring* sur plusieurs lignes.

```
"""Docstring de plusieurs lignes, la première ligne est un résumé.

Après avoir sauté une ligne, on décrit les détails de cette docstring.
blablabla
blablabla
blublublu
bliblibli
On termine la docstring avec les triples guillemets sur la ligne suivante.
"""
```

Remarque

La PEP 257 recommande d'écrire des *docstrings* avec des triples doubles guillemets, c'est-à-dire

`"""Ceci est une docstring recommandée."""`

mais pas

`'''Ceci n'est pas une docstring recommandée.'''`.

Comme indiqué dans le chapitre 14 *Création de modules*, n'oubliez pas que les *docstrings* sont destinées aux utilisateurs des modules, fonctions, méthodes et classes que vous avez

développés. Les éléments essentiels pour les fonctions et les méthodes sont :

1. ce que fait la fonction ou la méthode,
2. ce qu'elle prend en argument,
3. ce qu'elle renvoie.

Pour les modules et les classes, on ajoute également des informations générales sur leur fonctionnement.

Pour autant, la PEP 257 ne dit pas explicitement comment organiser les *docstrings* pour les fonctions et les méthodes. Pour répondre à ce besoin, deux solutions ont émergées :

- La solution Google avec le *Google Style Python Docstrings*.
- La solution *NumPy* avec le *NumPy Style Python Docstrings*. *NumPy* qui est un module complémentaire à Python, très utilisé en analyse de données et dont on parlera dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique*.

On illustre ici la solution *NumPy* pour des raisons de goût personnel. Sentez-vous libre d'aller explorer la proposition de Google. Voici un exemple très simple :

```
1  def multiplie_nombres(nombre1, nombre2):
2      """Multiplication de deux nombres entiers.
3
4      Cette fonction ne sert pas à grand chose.
5
6      Parameters
7      ----------
8      nombre1 : int
9          Le premier nombre entier.
10     nombre2 : int
11         Le second nombre entier.
12
13         Avec une description plus longue.
14         Sur plusieurs lignes.
15
16     Returns
17     -------
18     int
19         Le produit des deux nombres.
20     """
21     return nombre1 * nombre2
```

Lignes 6 et 7. La section `Parameters` précise les paramètres de la fonction. Les tirets sur la ligne 7 permettent de souligner le nom de la section et donc de la rendre visible.

Lignes 8 et 9. On indique le nom et le type du paramètre séparés par le caractère deux-points. Le type n'est pas obligatoire. En dessous, on indique une description du paramètre en question. La description est indentée.

Lignes 10 à 14. Même chose pour le second paramètre. La description du paramètre peut s'étaler sur plusieurs lignes.

Lignes 16 et 17. La section `Returns` indique ce qui est renvoyé par la fonction (le cas échéant).

Lignes 18 et 19. La mention du type renvoyé est obligatoire. En dessous, on indique une description de ce qui est renvoyé par la fonction. Cette description est aussi indentée.

⚠ Attention

L'être humain a une fâcheuse tendance à la procrastination (le fameux « Bah je le ferai demain...») et écrire de la documentation peut être un sérieux motif de procrastination. Soyez vigilant sur ce point, et rédigez vos *docstrings* au moment où vous écrivez vos modules, fonctions, classes ou méthodes. Passer une journée (voire plusieurs) à écrire les *docstrings* d'un gros projet est particulièrement pénible. Croyez-nous !

15.3 Outils de contrôle qualité du code

Pour évaluer la qualité d'un code Python, c'est-à-dire sa conformité avec les recommandations de la PEP 8 et de la PEP 257, on peut utiliser des sites internet ou des outils dédiés.

Le site pep8online, par exemple, est très simple d'utilisation. On copie / colle le code à évaluer puis on clique sur le bouton *Check code*.

Les outils `pycodestyle`, `pydocstyle` et `pylint` doivent par contre être installés sur votre machine. Avec la distribution Miniconda, cette étape d'installation se résume à une ligne de commande :

```
1  $ conda install -c conda-forge pycodestyle pydocstyle pylint
```

Définition

Les outils `pycodestyle`, `pydocstyle` et `pylint` sont des **linters**, c'est-à-dire des programmes qui vont chercher les sources potentielles d'erreurs dans un code informatique. Ces erreurs peuvent être des erreurs de style (PEP 8 et 257) ou des erreurs logiques (manipulation d'une variable, chargement de module).

Voici le contenu du script `script_quality_not_ok.py` que nous allons analyser par la suite :

```
1  """Un script de multiplication.
2  """
3
```

```
4  import os
5
6  def Multiplie_nombres(nombre1,nombre2 ):
7      """Multiplication de deux nombres entiers
8      Cette fonction ne sert pas à grand chose.
9
10     Parameters
11     ----------
12     nombre1 : int
13         Le premier nombre entier.
14     nombre2 : int
15         Le second nombre entier.
16
17         Avec une description plus longue.
18         Sur plusieurs lignes.
19
20     Returns
21     -------
22     int
23         Le produit des deux nombres.
24
25     """
26     return nombre1 *nombre2
27
28
29 if __name__ == "__main__":
30     print(f"2 x 3 = {Multiplie_nombres(2, 3)}")
31     print (f"4 x 5 = {Multiplie_nombres(4, 5)}")
```

Ce script est d'ailleurs parfaitement fonctionnel :

```
1  $ python script_quality_ok.py
2  2 x 3 = 6
3  4 x 5 = 20
```

On va tout d'abord vérifier la conformité avec la PEP 8 avec l'outil `pycodestyle` :

```
1  $ pycodestyle script_quality_not_ok.py
2  script_quality_not_ok.py:6:1: E302 expected 2 blank lines, found 1
3  script_quality_not_ok.py:6:30: E231 missing whitespace after ','
4  script_quality_not_ok.py:6:38: E202 whitespace before ')'
5  script_quality_not_ok.py:26:21: E225 missing whitespace around operator
6  script_quality_not_ok.py:31:10: E211 whitespace before '('
```

Ligne 2. Le bloc `script_quality_not_ok.py:6:1:` désigne le nom du script (`script_quality_not_ok.py`), le numéro de la ligne (6) et le numéro de la colonne (1) où se trouve la non-conformité avec la PEP 8. Ensuite, `pycodestyle` fournit un code et un message explicatif. Ici, il faut deux lignes vides avant la fonction `Multiplie_nombres()` .

Ligne 3. Il manque un espace après la virgule qui sépare les arguments `nombre1` et `nombre2` dans la définition de la fonction `Multiplie_nombres()` à la ligne 6 (colonne 30) du script.

Ligne 4. Il y un espace de trop après le second argument `nombre2` dans la définition de la fonction `Multiplie_nombres()` à la ligne 6 (colonne 38) du script.

Ligne 5. Il manque un espace après l'opérateur `*` à la ligne 26 (colonne 21) du script.

Ligne 6. Il y a un espace de trop entre `print` et `(` à la ligne 31 (colonne 10) du script.

Remarquez que curieusement, `pycodestyle` n'a pas détecté que le nom de la fonction `Multiplie_nombres()` ne respecte pas la convention de nommage.

Ensuite, l'outil `pydocstyle` va vérifier la conformité avec la PEP 257 et s'intéresser particulièrement aux *docstrings* :

```
1  $ pydocstyle script_quality_not_ok.py
2  script_quality_not_ok.py:1 at module level:
3          D200: One-line docstring should fit on one line with quotes (found
2)
4  script_quality_not_ok.py:7 in public function `Multiplie_nombres`:
5          D205: 1 blank line required between summary line and description
(found 0)
6  script_quality_not_ok.py:7 in public function `Multiplie_nombres`:
7          D400: First line should end with a period (not 's')
```

Lignes 2 et 3. `pydocstyle` indique que la *docstring* à la ligne 1 du script est sur deux lignes alors qu'elle devrait être sur une seule ligne.

Lignes 4 et 5. Dans la *docstring* de la fonction `Multiplie_nombres()` (ligne 7 du script), il manque une ligne vide entre la ligne résumé et la description plus complète.

Lignes 6 et 7. Dans la *docstring* de la fonction `Multiplie_nombres()` (ligne 7 du script), il manque un point à la fin de la première ligne.

Les outils `pycodestyle` et `pydocstyle` vont simplement vérifier la conformité aux PEP 8 et 257. L'outil `pylint` va lui aussi vérifier une partie de ces règles mais il va également essayer de comprendre le contexte du code et proposer des éléments d'amélioration. Par exemple :

```
 1  $ pylint script_quality_not_ok.py
 2  ************* Module script_quality_not_ok
 3  script_quality_not_ok.py:6:29: C0326: Exactly one space required after
comma
 4  def Multiplie_nombres(nombre1,nombre2 ):
 5                               ^ (bad-whitespace)
 6  script_quality_not_ok.py:6:38: C0326: No space allowed before bracket
 7  def Multiplie_nombres(nombre1,nombre2 ):
 8                                       ^ (bad-whitespace)
 9  script_quality_not_ok.py:31:10: C0326: No space allowed before bracket
10      print ((f"4 x 5 = {Multiplie_nombres(4, 5)}")
11            ^ (bad-whitespace)
12  script_quality_not_ok.py:6:0: C0103: Function name "Multiplie_nombres"
13  doesn't conform to snake_case naming style (invalid-name)
14  script_quality_not_ok.py:4:0: W0611: Unused import os (unused-import)
```

```
15
16  ----------------------------------
17  Your code has been rated at 0.00/10
```

Lignes 3 à 5. `pylint` indique qu'il manque un espace entre les paramètres de la fonction `Multiplie_nombres()` (ligne 6 et colonne 29 du script). La ligne du script qui pose problème est affichée, ce qui est pratique.

Lignes 6 à 8. `pylint` identifie un espace de trop après le second paramètre de la fonction `Multiplie_nombres()`.

Ligne 9 à 11. Il y a un espace de trop entre `print` et `(`.

Lignes 12 et 13. Le nom de la fonction `Multiplie_nombres()` ne respecte pas la convention PEP 8. La fonction devrait s'appeler `multiplie_nombres()`.

Ligne 14. Le module `os` est chargé mais pas utilisé (ligne 4 du script).

Ligne 17. `pylint` produit également une note sur 10. Ne soyez pas surpris si cette note est très basse (voire négative) la première fois que vous analysez votre script avec `pylint`. Cet outil fournit de nombreuses suggestions d'amélioration et la note attribuée à votre script devrait rapidement augmenter. Pour autant, la note de 10 est parfois difficile à obtenir. Ne soyez pas trop exigeant.

Une version améliorée du script précédent est disponible en ligne.

15.4 Organisation du code

Il est fondamental de toujours structurer et organiser son code de la même manière. Ainsi, on sait tout de suite où trouver l'information et un autre programmeur pourra s'y retrouver. Voici un exemple de code avec les différents éléments dans le bon ordre :

```
1   """Docstring d'une ligne décrivant brièvement ce que fait le programme.
2
3   Usage:
4   ======
5       python nom_de_ce_super_script.py argument1 argument2
6
7       argument1: un entier signifiant un truc
8       argument2: une chaîne de caractères décrivant un bidule
9   """
10
11  __authors__ = ("Johny B Good", "Hubert de la Pâte Feuilletée")
12  __contact__ = ("johny@bgood.us", "hub@pate.feuilletee.fr")
13  __copyright__ = "MIT"
14  __date__ = "2030-01-01"
15  __version__= "1.2.3"
16
17  import module_interne
```

```python
import module_interne_2

import module_externe

UNE_CONSTANTE = valeur
UNE_AUTRE_CONSTANTE = une_autre_valeur


class UneSuperClasse():
    """Résumé de la docstring décrivant la classe.

    Description détaillée ligne 1
    Description détaillée ligne 2
    Description détaillée ligne 3
    """

    def __init__(self):
        """Résumé de la docstring décrivant le constructeur.

        Description détaillée ligne 1
        Description détaillée ligne 2
        Description détaillée ligne 3
        """
        [...]

    def une_méthode_simple(self):
        """Docstring d'une ligne décrivant la méthode."""
        [...]

    def une_méthode_complexe(self, arg1):
        """Résumé de la docstring décrivant la méthode.

        Description détaillée ligne 1
        Description détaillée ligne 2
        Description détaillée ligne 3
        """
        [...]
        return un_truc


def une_fonction_complexe(arg1, arg2, arg3):
    """Résumé de la docstring décrivant la fonction.

    Description détaillée ligne 1
    Description détaillée ligne 2
    Description détaillée ligne 3
    """
    [...]
    return une_chose


def une_fonction_simple(arg1, arg2):
    """Docstring d'une ligne décrivant la fonction."""
    [...]
    return autre_chose


```

```
75  if __name__ == "__main__":
76      # ici débute le programme principal
77      [...]
```

Lignes 1 à 9. Cette *docstring* décrit globalement le script. Cette *docstring* (ainsi que les autres) seront visibles si on importe le script en tant que module, puis en invoquant la commande `help()` (voir chapitre 14 *Création de modules*).

Lignes 11 à 15. On définit ici un certain nombres de variables avec des doubles *underscores* donnant quelques informations sur la version du script, les auteurs, etc. Il s'agit de métadonnées que la commande `help()` pourra afficher. Bien sûr, ces métadonnées ne sont pas obligatoires, mais elles sont utiles lorsque le code est distribué à la communauté.

Lignes 17 à 20. Importation des modules. D'abord les modules internes à Python (fournis en standard), puis les modules externes (ceux qu'il faut installer en plus), un module par ligne.

Lignes 22 et 23. Définition des constantes. Le nom des constantes est en majuscule.

Ligne 26. Définition d'une classe. On a laissé deux lignes vides avant.

Lignes 27 à 32. *Docstring* décrivant la classe.

Lignes 33, 42 et 46. Avant chaque méthode de la classe, on laisse une ligne vide.

Lignes 58 à 72. Après les classes, on met les fonctions « classiques ». Avant chaque fonction, on laisse deux lignes vides.

Lignes 75 à 77. On écrit le programme principal. Le test ligne 76 n'est vrai que si le script est utilisé en tant que programme. Les lignes suivantes ne sont donc pas exécutées si le script est chargé comme un module.

15.5 Conseils sur la conception d'un script

Voici quelques conseils pour vous aider à concevoir un script Python.

- Réfléchissez avec un papier, un crayon... et un cerveau (voire même plusieurs) ! Reformulez avec des mots en français (ou en anglais) les consignes qui vous ont été données ou le cahier des charges qui vous a été communiqué. Dessinez ou construisez des schémas si cela vous aide.
- Découpez en fonctions chaque élément de votre programme. Vous pourrez ainsi tester chaque élément indépendamment du reste. Pensez à écrire les *docstrings* en même temps que vous écrivez vos fonctions.
- Quand l'algorithme est complexe, commentez votre code pour expliquer votre raisonnement. Utiliser des fonctions (ou méthodes) encore plus petites peut aussi être une solution.

- Documentez-vous. L'algorithme dont vous avez besoin existe-t-il déjà dans un autre module ? Existe-t-il sous la forme de pseudo-code ? De quels outils mathématiques avez-vous besoin dans votre algorithme ?
- Si vous créez ou manipulez une entité cohérente avec des propriétés propres, essayez de construire une classe. Jetez, pour cela, un œil au chapitre 19 *Avoir la classe avec les objets*.
- Utilisez des noms de variables explicites, qui signifient quelque chose. En lisant votre code, on doit comprendre ce que vous faites. Choisir des noms de variables pertinents permet aussi de réduire les commentaires.
- Quand vous construisez une structure de données complexe (par exemple une liste de dictionnaires contenant d'autres objets), documentez et illustrez l'organisation de cette structure de données sur un exemple simple.
- Testez toujours votre code sur un jeu de données **simple** pour pouvoir comprendre rapidement ce qui se passe. Par exemple, une séquence de 1000 bases est plus facile à gérer que le génome humain ! Cela vous permettra également de retrouver plus facilement une erreur lorsque votre programme ne fait pas ce que vous souhaitez.
- Lorsque votre programme « plante », **lisez** le message d'erreur. Python tente de vous expliquer ce qui ne va pas. Le numéro de la ligne qui pose problème est aussi indiqué.
- Discutez avec des gens. Faites tester votre programme par d'autres. Les instructions d'utilisation sont-elles claires ?
- Si vous distribuez votre code :
  - Rédigez une documentation claire.
  - Testez votre programme (jetez un œil aux tests unitaires).
  - Précisez une licence d'utilisation. Voir par exemple le site *Choose an open source license*.

15.6 Pour terminer : la PEP 20

La PEP 20 est une sorte de réflexion philosophique avec des phrases simples qui devraient guider tout programmeur. Comme les développeurs de Python ne manque pas d'humour, celle-ci est accessible sous la forme d'un « œuf de Pâques » (*easter egg* en anglais) ou encore « fonctionnalité cachée d'un programme » en important un module nommé `this` :

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
```

```
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Et si l'aventure et les *easter eggs* vous plaisent, testez également la commande

```
>>> import antigravity
```

Il vous faudra un navigateur et une connexion internet.

🚀 Pour aller plus loin

- L'article *Python Code Quality: Tools & Best Practices* du site *Real Python* est une ressource intéressante pour explorer plus en détail la notion de qualité pour un code Python. De nombreux *linters* y sont présentés.
- Les articles *Assimilez les bonnes pratiques de la PEP 8* du site *OpenClassrooms* et *Structuring Python Programs* du site *Real Python* rappellent les règles d'écriture et les bonnes pratiques vues dans ce chapitre.