Bonne pratiques

Laurent Signac – CC-BY-SA – 03-10-22 0840 aea1b44c459feb254afe

Ce document regroupe un certain nombre de pratiques que vous devrez (ou pourrez) mettre en œuvre cette année.

■ Les noms de variable doivent être descriptifs

Extrait de *Idiomatic Python* de E. Franchi :

> Always use descriptive names, the longer the scope, the longer the name.

Si on peut se contenter d'un nom court pour une variable qui n'existe que dans un (tout) petit bloc de code, il faudra choisir un nom évocateur pour des variables qui traversent tout le code, qui sont en paramètres de fonctions, ou qui sont globales (si possible, éviter les variables globales).

■ Viser la simplicité

Le fait qu'un code *marche* ne signifie pas qu'il est bon. En particulier, s'il existe un moyen plus simple et plus clair d'arriver au résultat… c'est celui-ci qu'il faut employer.

À ne pas faire :

```
if ((j // 100) - 1) >= 0:
    ...
```

Convenable :

```
if j >= 100:
    ...
```

Extrait du Zen Python :

```
>>> 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.
...
```

■ Réserver `i`, `j`, `k` aux compteurs *numériques* de boucles

Une habitude tenace est d'utiliser `i`, `j` ou `k` comme compteur des boucles `for`. En Python, on utilise une boucle `for` pour parcourir les indices, mais aussi pour parcourir directement les objets. **N'utilisez pas** `i`, `j`, et `k` si vous parcourez les objets, cela va à l'encontre d'habitudes bien établies.

À ne pas faire :

```
noms = ["Bilbo", "Frodo", "Sam"]
for i in noms:
    print(i)
```

Convenables :

```
noms = ["Bilbo", "Frodo", "Sam"]
for nom in noms:
    print(nom)
```

ou

```
noms = ["Bilbo", "Frodo", "Sam"]
for i in range(len(noms)):
    print(noms[i])
```

■ Bien utiliser les commentaires

Hors contexte scolaire, les commentaires sont faits pour **aider le lecteur humain** à comprendre le fonctionnement du programme. **C'est une erreur de commenter ce qui est une évidence**.

(très) Mauvais

```
# Initialise s à {1}
s = {1}
# Pour i variant de 2 à n-1
for i in range(2, n):
    # si le reste de la division
    # de n par i est nul
    if n % i == 0:
        # ajouter i à s
        s.add(i)
```

Bon

```
# Met les diviseurs stricts de n
# dans l'ensemble s
s = {1}
for i in range(2,n):
    if n % i == 0:
        s.add(i)
```

■ Écrire des docstrings

Les docstrings sont des chaînes de caractères placées au début d'un objet à documenter (module, fonction…). La docstring peut ensuite être appelée avec la fonction `help`. Une docstring n'explique pas **comment fonctionne** le code (c'est le rôle des commentaires), mais **à quoi il sert** et **comment on s'en sert**.

Par exemple :

```
import random

def melange(s):
    """ Renvoie une chaîne de caractère obtenue après
        mélange de la chaîne passée en paramètre
    """
    lst = list(s)
    r = ""
    while len(lst) > 0: # TODO : utiliser while lst ?
        # À chaque tour, un élément est choisi dans lst, ajouté
        # à la chaîne r et supprimé de la liste lst
        i = random.randint(0, len(lst) - 1)
        r = r + lst[i]
        del lst[i]
    return r
```

Les commentaires sont à destination de la personne qui **relira** / tentera de comprendre le code. Les docstrings sont à destination de la personne qui **utilisera** le code (d'un module par exemple).

■ Importation des modules

On n'importe pas un module en écrivant : `from xxx import *`. Cette écriture pollue l'espace des noms et est source de bugs et de comportements difficiles à comprendre. Les seules écritures acceptables sont :

```
import math                 # écriture à privilégier
import random as rnd        # si le nom est trop long
from math import sqrt, cos # si vous n'avez besoin que d'une ou deux fonctions
```

Source (entre autres) : https://docs.python-guide.org/writing/structure/#modules

■ Vérifier son code

Une fois le programme écrit, à défaut de pouvoir prouver qu'il est correct, il faut vérifier sur des exemples qu'il donne le bon résultat. Pour cela, on le teste sur des valeurs particulières. Le choix de ces valeurs est important :

- tester avec des entrées pour lesquelles on connaît la sortie correcte
- tester de manière extensive, afin que chaque portion de code soit exécutée
- penser aux cas extrêmes qui posent parfois des problèmes

C'est d'autant plus vrai avec Python (pas de vérification statique des types)

■ Factoriser (au sens de la programmation) les calculs

Encore une fois, on garde à l'esprit qu'un programme est aussi écrit pour être lu par un humain. On cherchera donc à faciliter cette lecture.

Cette fonction donne un résultat correct :

```
def aire_triangle(a, b, c):
    """ Renvoie l'aitre d'un triangle
        connaissant les longueurs de 3 côtés
    """
    return math.sqrt( ((a + b + c) / 2) * (((a + b + c) / 2) - a) *  \
                      (((a + b + c) / 2) - b) *  (((a + b + c) / 2) - c))
```

Mais elle est tellement plus facile à lire écrite ainsi :

```
def aire_triangle(a, b, c):
    """ Renvoie l'aitre d'un triangle
        connaissant les longueurs de 3 côtés
    """
    m = (a + b + c) / 2
    s2 = m * (m - a) * (m -b) * (m - c)
    return math.sqrt(s2)
```