Bonnes pratiques et joli code

Laurent Signac – CC-BY-SA – 10-04-24 0954 3df6c1e664ffb6afd8e2

> *«Any fool can write code that a computer can understand. Good programmers write code that humans can understand.» — Martin Fowler*

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

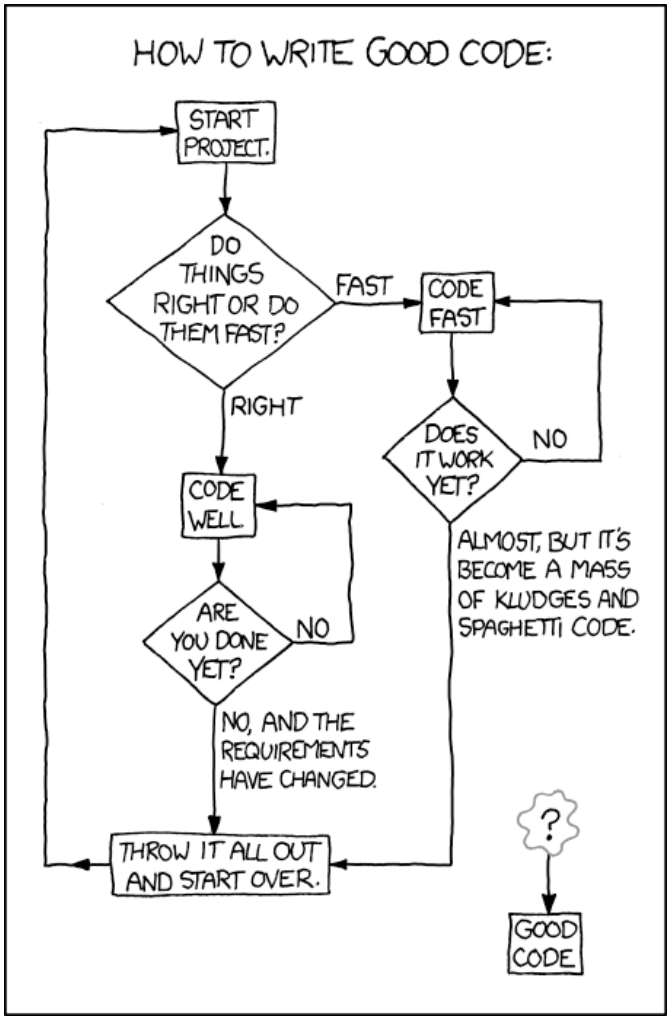
> *«Only ugly languages become popular. Python is the one exception.» — Donald Knuth*

La beauté est subjective, alors, donnons quelques précisions. Du joli code est du code :

- facile à vérifier
- concis
- intelligent et efficace

C'est avéré, un code est très souvent lu par un humain : celui qui l'a écrit, celui qui reprend un projet, celui qui veut vérifier le travail d'une autre personne… Du joli code est donc du code qui facilite la relecture. La facilité de vérification (et la non redondance, mais ça va finalement ensemble) est probablement le point le plus important.

> *« Good programming is good writing » — John Shore*



xkcd 882

1 Règles générales en programmation

Les objectifs sont d'obtenir des programmes ou des portions de code qui ont les propriété suivantes :

- le code doit **être facile à lire**, pas soi-même mais aussi par les autres.
- l'**organisation** du code doit être **logique et évidente**.
- le code doit être explicite et **montrer clairement les intentions du programmeur**.

Les règles qui suivent visent toutes à réaliser les objectifs précédents.

■ Les noms de variable doivent être descriptifs [Jolicode 1]

> *«Always use descriptive names, the longer the scope, the longer the name.» — E. Franchi (Idiomatic Python)*

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 [Jolicode 4] [Jolicode 5]

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 (et pas beaucoup moins efficace)… c'est celui-ci qu'il faut employer.

À ne pas faire :

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

Convenable :

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

■ Réserver `i`, `j`, `k` aux compteurs *numériques* de boucles [Jolicode 1]

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])
```

Itérer sur les valeurs plutôt que sur les indices présente plusieurs avantages :
- meilleure lisibilité ;
- plus rapide à l'exécution ;
- bytecode plus court ;
- fonctionne aussi sur des itérables qui ne supportent pas `len` (comme les générateurs).

■ Bien utiliser les commentaires [Jolicode 3]

Le nombre de citations témoigne de l'importance du problème…

> *"A common fallacy is to assume authors of incomprehensible code will be able to express themselves clearly in comments."* – *Kevlin Henney*
>
> *«Don't comment bad code — rewrite it.»* — *Brian Kernighan*
>
> *«When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.»* — *Martin Fowler*

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**.

Par ailleurs les témoignages qui visent à privilégier un code clair à un code commenté (sous prétexte qu'il n'est pas clair) sont multiples.

(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)
```

■ Don't repeat yourself (DRY) : Factoriser (au sens de la programmation) les calculs [Jolicode 2]

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)
```

■ Don't repeat yourself (DRY) : Écrire une fonction [Jolicode 2]

Si vous avez des portions de code qui se ressemblent et qui apparaissent plusieurs fois, même si le code fonctionne correctement, récrivez le avec un fonction.

Par exemple, si on souhaite écrire du code qui sort la liste des diviseurs stricts communs à deux nombres, on pourrait écrire :

```
ens1 = set()
for k in range(1, n1):
    if n1 % k == 0:
        ens1.add(k)
for k in range(1, n2):
    if n2 % k == 0:
        ens2.add(k)
print(ens1 & ens2)
```

Mais il vaut mieux faire ainsi :

```
def ens_div(n):
    ens = set()
    for k in range(1, n):
        ens.add(n)
    return ens

print(ens_div(n1) & ens_div(n2))
```

Ainsi, le code pour la recherche des diviseurs n'apparaît qu'une fois. Si on trouve une solution plus efficace pour tous les trouver (il y en a…), on a une seule portion de code à modifier. En outre, la seconde version est plus facile à comprendre, et elle est moins sensible aux erreurs d'inattention de copier/coller.

2 Conseils plus spécifiques à Python

■ Préférer une boucle `for` à une boucle `while` [Jolicode 4]

Une boucle `while`, bien qu'elle facilite les éventuelles preuves et la manipulation d'invariants, est plus sujette à des erreurs d'inattention : évaluation incorrecte de la condition, oubli de l'initialisation, oubli de

l'incrémentation. Une boucle `for`, en Python, a un côté automatique, et cette alternative est préférable lorsqu'on parcourt une collection. **Lorsque vous pouvez, utilisez de préférence une boucle `for`.**

■ Écrire des docstrings [Jolicode 3]

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

■ Manipuler des séquences plutôt que de scalaires

Pour la manipulation des composantes d'une image, on lit souvent ceci :

```
pix = image.getpixel((x, y))
pix[0] = pix[0] // 2
pix[1] = pix[1] // 2
pix[2] = pix[2] // 2
image.putpixel((x, y), (pix[0], pix[1], pix[2]))
```

Ce qui correspond malheureusement au moins bon des deux mondes. On préférera donc une de ces deux écritures :

```
r, v, b = image.getpixel((x, y))
r, v, b = r // 2, v // 2, b // 2
image.putpixel((x, y), (r, v, b))
```

```
pix = image.getpixel((x, y))
pix[0] = pix[0] // 2
pix[1] = pix[1] // 2
pix[2] = pix[2] // 2
```

```
image.putpixel((x, y), pix)
```

■ Don't repeat yourself (DRY) : Utiliser une liste [Jolicode 5]

Ce n'est pas une règle absolue, mais si vous avez besoin d'une variable `a1`, d'une variable `a2`, `a3`, `a4` et `a5`… probablement que c'est d'une liste dont vous avez besoin.

Ainsi ce code :

```
import random

a1 = random.randint(1, 6)
a2 = random.randint(1, 6)
a3 = random.randint(1, 6)
a4 = random.randint(1, 6)
a5 = random.randint(1, 6)

s = (a1 + a2 + a3 + a4 + a5) / 5
print("En moyenne avec 5 lancers de dés : ", s)
```

pourrait avantageusement être remplacé par :

```
import random

lst = []
for i in range(5):
    lst.append(random.randint(1, 6))

s = sum(lst) / 5
print("En moyenne avec 5 lancers de dés : ", s)
```

Ici, on pourrait aussi écrire du code générique pour `n` nombres, ce serait encore mieux.

💡 Si vous avez quelques valeurs à traiter, posez-vous la question : « S'il y avait plus de valeurs, comment faudrait-il procéder ? ». Puis faites comme ça.

3 Tester et débuguer

■ Vérifier son code [Test 1] [Test 2]

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)

■ Analyser les messages d'erreur [Erreur 1]

Lorsque du code ne fonctionne pas, et qu'une exception est levée, le premier objectif n'est pas de corriger, mais de comprendre la nature de l'erreur. Ensuite on peut corriger. Une conséquence de ça est qu'il est **nécessaire** de lire le message d'erreur et de le comprendre *même si on avait la possibilité de corriger immédiatement.*