

Maintenir un code lisible

Sébastien Corbin





Préparez-vous

Attention danger, risque de :

- hémorragie rétinienne
- hystérie soudaine
- malaise vagal
- nausée

Les premiers rangs risquent d'être arrosés par les derniers

Imaginez... un monde sans code style

```
var lol, HAhaHA_I_M_notCONSisTANT =function(  
) {return lol;;;;;;;;;;  
                                ;;;;}; lol=  
"Coucou"; alert(HAHAHA_I_M_CONSTANT(  
)[ '0' ])
```

```
function zoubida(lavabo)  
  
{  
    if(confirm        (lavabo)) for (;;) return true; else {  
        return false  
    }  
}
```

Connaissez-vous l'indentation inversée ?

```
    for(int i = 0; i < 10; i++)
    {
myFunc();
    }
    if(something)
    {
// do A
    }
    else
    {
// do B
    }
```

Oui oui, ça existe <https://stackoverflow.com/q/218123>
<https://softwareengineering.stackexchange.com/q/1338>

Et avec python ?

- Syntaxe claire
- Pas d'accolades
- Indentation obligatoire
- Moins de parenthèses
- Pas de bloc anonymes
- Peu de mots-clés

Mais python peut aussi faire de mauvaises syntaxes

```
if (((bar == None))): bar = 2
```

```
bar = 3; print(bar)
```

Défi : créez/trouvez un obfuscateur en Python !

A green and yellow patterned snake, likely a Python, is coiled around the text. The snake's body is covered in a complex pattern of dark green and yellowish-green spots and bands. It is positioned in the center of the image, with its head visible on the left side. The background is a solid, light green color.

Une chance de coder sous Python

Tableau récapitulatif des *coding standards*

Langage	Statut	Nom	Commentaires
PHP	Officiel	PHP-FIG / PSR	Récent, d'autres standards sont apparus avant
Javascript	Communautaire	ESLint	C'est le bordel, mais eslint est le plus utilisé
Typescript	Communautaire	Google a pondu gts	
Ruby	Communautaire		
Java	Communautaire		
Go	Officiel	gofmt	Voir Effective Go
Rust	Officiel	rustfmt	
C#	Officiel		
Python	Officiel	PEP8	

Introduction à la PEP8

- indentation de 4 espaces
- ligne de 79 caractères
- imports sur plusieurs lignes
- utf-8 par défaut
- snake_case pour variables et fonctions/méthodes
- CamelCase pour classes

Ensuite chaque framework/bibliothèque/projet a son propre coding standard.

**Un outil de prédilection :
le *linter***

Les **outils de base** pour vérifier la compatibilité PEP8 :

- **pylint** : vérificateur d'erreurs de syntaxe
- **pyflakes** : vérificateur d'erreurs de syntaxe
- **autoflake** : formatteur automatique basé sur `pyflakes`
- **pycodestyle** : vérificateur de style (anciennement `pep8`)
- **autopep8** : formatteur automatique basé sur `pycodestyle`
- **yapf** : formatteur automatique
- **mypy** : vérificateur de typage statique
- **pychecker** : † projet mort †
- **pep8ify** : † projet mort †
- **pyntch** : † projet mort †

<http://meta.pycqa.org>

Aller plus loin



Nombre de caractères d'une ligne (en vrai)

Certains sont nazis avec 70 caractères, d'autres plus laxes 88 caractères car on est plus sur écran DOS.

Sur les docstring, c'est parfois 72, parfois plus...

Wrapping (retour à la ligne)

Des arguments/paramètres

```
my_super_long_method_name(super_long_argument1, super_long_argument2, super_long_a
```

```
my_super_long_method_name(super_long_argument1, super_long_argument2,  
                           super_long_argument3)
```

```
my_super_long_method_name(  
    super_long_argument1, super_long_argument2, super_long_argument3  
)
```

```
my_super_long_method_name(  
    super_long_argument1,  
    super_long_argument2,  
    super_long_argument3  
)
```

D'une callchain

```
result = MyLongClassName.my_long_method_name(arg1).my_chained_call(arg4, arg5).my_
```

```
result = MyLongClassName.my_long_method_name(arg1)\  
        .my_chained_call1(arg4, arg5)\  
        .my_chained_call2()
```

```
result = MyLongClassName.my_long_method_name(  
    arg1  
) .my_chained_call1(  
    arg4, arg5  
) .my_chained_call2()
```

```
result = (  
    MyLongClassName.my_long_method_name(arg1)  
    .my_chained_call1(arg4, arg5)  
    .my_chained_call2()  
)
```

Des imports

```
from mon_package import ma_methode_1, ma_methode_2, ma_methode_3, ma_methode_4, ma
```

```
from mon_package import ma_methode_1, ma_methode_2, ma_methode_3, ma_methode_4,\n    ma_methode_5
```

```
from mon_package import (ma_methode_1, ma_methode_2, ma_methode_3, ma_methode_4,\n    ma_methode_5)
```

```
from mon_package import (\n    ma_methode_1, ma_methode_2, ma_methode_3, ma_methode_4, ma_methode_5\n)
```

```
from mon_package import (\n    ma_methode_1,\n    ma_methode_2,\n    ma_methode_3,\n    ma_methode_4,\n    ma_methode_5,\n)
```

et ça continue avec les listes, les dictionnaires, etc...

Virgules en fin de ligne

```
def ma_methode_au_nom_super_long(  
    argument1, argument2=None, argument3=None, argument4=None, argument5=None,  
    argument6=None, argument7=None, argument8=None, arg9=None, *args, **kwargs  
):
```

Doit-on la retirer pour éviter de dépasser les 79 caractères ?

Apostrophes

Double ou simple quote ? Forcé ou non ?

Formattage de chaînes

```
"Bonjour %s, bienvenue chez %s" % ("la vie", "les bisounours")
```

```
"Bonjour {}, bienvenue chez {}".format('Jean', 'nous')
```

```
"Bonjour {0}, bienvenue chez {1}".format('Kévin', 'McDonald')
```


Complexité cyclomatique

```
for i in ma_liste:
    if i < 10:
        try:
            while true:
                answer = ask("why ???")
                if answer = "idontknow"
                    raise DontKnowEception
            except DontKnowEception as e:
                for i in range(3):
                    print("ha")
```

Maintenabilité

Indice allant de 0 à 100 avec :

- 85+ : maintenabilité bonne
- 65-85 : maintenabilité moyenne
- < 65 : maintenabilité difficile

Nombre de ligne de code par module

Encore des outils :

- **radon/xenon** : métriques de Halstead
- **isort** : gère l'ordre, le regroupement et le style des imports de manière fine
- **McCabe** : analyse de complexité
- **pylama** : combinaison de `pycodestyle`, `pydocstyle`, `pyflakes`, `mccabe`, `pylint`, `radon`, `gjslint`
- **flake8** : combinaison de `PyFlakes`, `pycodestyle` et `mccabe`
- **black** : formatteur nazi

Tox

Sert à automatiser les tests, pratique car son fichier de config `tox.ini` est lu par *flake8* et *isort* notamment

```
[tox]
skipsdist = true
envlist =
    flake8
    isort

[testenv]
deps = -rdev-requirements.txt

[testenv:flake8]
deps = flake8
commands = flake8 .

[testenv:isort]
deps =
    -rdev-requirements.txt
    isort>=4.2.13
commands = isort --check-only --
```

Et le bon vieux grep !?

```
import pdb; pdb.set_trace()  
import ipdb; ipdb.set_trace()  
print("Je passe ici")
```

```
console.log("toto");  
alert("lol");
```

Qui c'est qu'à pas fini son merge !?







```
Si vous avez des questions :  
<<<<<< HEAD  
c'est pas maintenant.  
=====  
c'est à la fin du talk.  
>>>>>> branch-a
```

A quel moment vérifier ?



Pendant l'écriture

Configurer son IDE :

- Pycharm autoformatte où peut lancer un formatteur avec un raccourci
 -  +  +  sous Linux/Windows
 -  +  +  sous Max OS
- Lancer le linter régulièrement en ligne de commande

Avant de commiter

- Hook Git de pre-commit hook maison

```
# à mettre dans votre-projet/.git/hooks/pre-commit et à rendre executable
#!/bin/bash

source ~/.virtualenvs/votre-projet/bin/activate

echo "Checking for syntax"
if ! flake8 || ! isort -e --check-only
then
    echo ""
    echo "Correct the errors above"
    exit 1
fi
```

Un exemple en python <https://gist.github.com/streeter/1376856>

Avant de commiter

Fichier de configuration partagé avec **pre-commit**

- `pip install pre-commit` **ou** `brew install pre-commit`

```
# .pre-commit-config.yaml
-   repo: git://github.com/pre-commit/pre-commit-hooks
    sha: v0.9.1
    hooks:
      - id: trailing-whitespace
      - id: check-ast
      - id: check-docstring-first
      - id: check-merge-conflict
      - id: debug-statements
      - id: end-of-file-fixer
      - id: flake8
-   repo: git://github.com/RobinRamael/pre-commit-isort.git
    hooks:
      - id: isort
```

- `pre-commit install`

Possibilité d'ajouter ses propres hooks, tout ça, en python

Avec une Continuous Integration

- Travis CI

```
dist: trusty
language: python
python:
  - "3.6"
install:
  - pip install -r dev-requirements.txt
  - pip install flake8 isort
script:
  - flake8 .
  - isort -e --check-only
```

- GitlabCI

```
image: python
before_script:
  - pip install -r dev-requirements.txt
tests:
  script:
    - flake8 .
    - isort -e --check-only
```

Des questions ?

<https://makinacorporus.github.io/maintenir-code-lisible/>