GUIDE STARTER PYTHON

*sur  
la Mise en Forme & la Structuration de Codage*

Révisions

|  |  |  |  |
| --- | --- | --- | --- |
| ***Version*** | ***Modifié le*** | ***Description*** | ***Modifié par*** |
| 0.1 | 18/05/2022 | Créer le document en anglais | Thanh-Long NGUYEN |
| 0.2 | 18/05/2022 | Traduire en français et reformatter le doc  Ajouter la table des matières | Thanh-Long NGUYEN  Estelle ROSSET |
|  |  |  |  |
|  |  |  |  |

Objectifs

Le présent document est pour aider des nouveaux développeurs Python de B&D à créer des projets Python **conformes aux normes** de l’organisation, et à suivre les **bonnes pratiques** en utilisant des outils tels que des **linters**, des **formateurs**, **pré-commit**, et des **générateurs** de projet.

Ce document est divisé en deux parties : **la première** insiste l’importance de respecter les normes de codage Python, puis présente certaines guidelines les plus connues, ainsi que les outils qui aident à **automatiser ces processus**.

**La deuxième partie** explique l’importance d’avoir de modèles cohérents pour des projets Python, puis présente certains des outils connus pour **créer et gérer des modèles** de projet.

***Table des matières***

[PARTIE I: Normes de Codage et Bonnes Pratiques 3](#_Toc104281321)

[1. Guide de Style pour le Code Python 3](#_Toc104281322)

[1.1. L’importance de respecter les normes de codage 3](#_Toc104281323)

[1.2. Python Enhancement Proposals 3](#_Toc104281324)

[1.3. Autres PEPs 4](#_Toc104281325)

[2. Utiliser des Linters et des Formateurs pour le code Python 6](#_Toc104281326)

[2.1 Quelles sont les différences entre code linters et code formateurs ? 6](#_Toc104281327)

[2.2 Les linters connus pour Python 6](#_Toc104281328)

[2.3 Les formateurs connus pour Python 7](#_Toc104281329)

[3. Pre-commit 9](#_Toc104281330)

[3.1 Qu’est-ce que c’est Git hooks ? 9](#_Toc104281331)

[3.2 Le package pre-commit de Python 9](#_Toc104281332)

[3.3 Quelques hooks connus pour pre-commit 9](#_Toc104281333)

[PART II: Structurer un Projet Python 11](#_Toc104281334)

[1. Pourquoi avons-nous besoin de modèles de projet ? 11](#_Toc104281335)

[2. Créer une structure avec un générateur de projet 12](#_Toc104281336)

[APPENDIX A: Feuille de triche pour les règles PEP8 [24] 16](#_Toc104281337)

[APPENDIX B: Intégration de flake8 dans nos projets 19](#_Toc104281338)

[APPENDIX C: Intégration de black dans nos projets 20](#_Toc104281339)

[APPENDIX D: Intégration de pre-commit dans nos projets Python [14] 22](#_Toc104281340)

# PARTIE I: Normes de Codage et Bonnes Pratiques

## 1. Guide de Style pour le Code Python

### L’importance de respecter les normes de codage

De nos jours, le codage consiste non seulement à créer des programmes qui fonctionnent correctement conformément aux exigences fonctionnelles, mais aussi à **créer de code lisible**, **cohérent**, et même **erreur-évitable**.

Parce qu’il faut savoir que **40-80% du coût total d’un logiciel est consacré à sa maintenance** [1]. De plus, des logiciels sont **rarement toujours supportés** **par leurs auteurs originaux**. Ainsi, un code qui respecte bien les normes recommandées amélioreront la lisibilité du logiciel en permettant aux développeurs de **comprendre plus rapidement** le nouveau code.

En outre, certains repositories de code comme PyPi exigent même que le code soit bien formaté et respecte quelques normes spécifiques avant d’être soumis.

Finalement, comme tout autre produit, le logiciel doit être « bien packagé » et propre.

Un exemple très simple d’un mauvais style et d’un bon style de code est montré ci-dessous.

|  |  |
| --- | --- |
| **Pas OK** | **OK** |

* *Pas OK*: multi-déclaration par ligne, code est trop dense
* *OK*: une ligne pour chaque déclaration, une nouvelle ligne après expression *if/else*.

### Python Enhancement Proposals

Les auteurs de Python ont proposé un **processus de conception** OSS (Open-Source Software) pour Python, appelé PEPs (**Python Enhancement Proposals**). Les PEPs sont le principal moyen de proposer de nouvelles fonctionnalités, de recueillir des commentaires de la communauté sur un problème et de documenter les décisions de conception choisies [2].

Certains PEPs décrivent de **nouvelles fonctionnalités** pour Python, tandis que d’autres spécifient des informations plus générales sur le **processus** ou **l’organisation** de la communauté Python.

**PEP-8: Guide de style pour Python**

PEP8 est l’un des PEPs les plus connus, il est un document qui fournit des **guidelines** et les **meilleurs** **pratiques** sur la façon d’écrire du code Python. Il a été rédigé en 2001 par Guido Van Rossum, Barry Warsaw et Nick Coghlan. L’objectif principal de PEP8 est **d’améliorer la lisibilité** et la cohérence du code Python [3].

Quelques exemples de règles requises par PEP8 [4] peuvent inclure :

* Les espaces sont la méthode d’indentation préférée (plutôt que les tabulations)
* Utiliser 4 espaces par niveau d’indentation
* Limiter toutes les lignes à un maximum de 79 caractères
* Séparer les définitions de fonction et de classe avec deux lignes vides
* …

Pour plus d’exemples concrètes de l’utilisation de PEP8, regarder : <https://pep8.org> [22]

*🡪 En outre, une feuille de triche PEP8 avec code peut être consultée dans* ***l’annexe A*** *de ce document.*

### Autres PEPs

Jusqu’à présent, de très **nombreux PEP ont été proposés**, répartis en plusieurs catégories [21] :

* *Meta-PEPs*: sur PEPs ou processus. E.g.:
  + PEP1: Objectives de PEP et guidelines
  + PEP6: Bug fix releases
  + PEP8: Style guides
  + PEP387: Politique de compatibilité en arrière
  + …
* *PEPs informatifs*. E.g.:
  + PEP20: Le Zen de Python (très connu)
  + PEP257: Règles de docstring
  + PEP603: Cycle de publication annuelle de Python
  + PEP483: La théorie de l’indice de type
  + …
* *PEPs acceptés (validé peut-être pas encore implémenté)*
* *PEPs ouverts (sous considération)*
* *PEPs réalisés (terminé, avec une interface stable)*
* *...*

Jetons un coup d’œil à quelques PEPs connus qui fournissent un guide de style, autre que PEP8.

**PEP-257: Règles de docstring**

Ce PEP documente la **sémantique et les conventions associées aux docstring** Python. Quelques exemples peuvent inclure :

* Toujours utiliser des guillemets triples autour du docstring
* Des docstrings multiligne doivent contenir une ligne de résumé suivie par une ligne vide, puis une description plus élaborée
* Insérer une ligne vide après chaque docstring

**PEP-20: Le Zen de Python**

PEP20 est une collection de **19 principes pour coder de programmes** d’ordinateur qui influencent la conception de Python, écrits par Tim Peters sur le  « mailing list » en 1999 [23].

Ces principes sont listés ci-dessous (en anglais comme l’original) :

*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!*

Quelques morceaux de code sont donnés ci-dessous pour démontrer les idées de PEP20.

|  |  |  |
| --- | --- | --- |
|  | ***Bad*** | ***Good*** |
| *Explicit is better than implicit* |  |  |
| *Sparse is better than dense* |  |  |
| *In the face of ambiguity, refuse the temptation to guess* |  |  |
| *Now is better than never* |  |  |

## 2. Utiliser des Linters et des Formateurs pour le code Python

### Quelles sont les différences entre code linters et code formateurs ?

Ils ne sont pas pareils !

[5] Lorsqu’on travaille dans une équipe qui se partage la même base de code, on veut toujours que:

* Le **même format** de code est appliqué partout (e.g. tabulations, espaces, lignes vides …)
* Les **meilleurs pratiques** sont bien appliqués au code (e.g. importations, exceptions …)

*Les formateurs de code résolvent le premier problème*, ils essaient de « maquiller » l’ensemble de code de manière cohérente tout en gardant les comportements du programme inchangés.

*Les linters résolvent le deuxième problème*, ils aident à utiliser des mieux syntaxes ou des nouvelles fonctionnalités du langage de programmation, ainsi qu’empêcher des erreurs potentielles (mais les problèmes compliqués comme le nommage de variables n’ont pas été réalisés jusqu’au moment de la rédaction de ce document). Il faut savoir que des linters peuvent aussi résoudre le premier problème, mais ils ne seront pas efficaces comme des formateurs pour cette issue.

Une autre différence est que des linters normalement détectent des issues puis ils laissent le développeur les corriger lui-même, tandis que des formateurs automatiquement fixent des issues qu’ils viennent de trouver.

En pratique, **on peut combiner des linters et des formateurs** concernant les besoins du projet.

### Les linters connus pour Python

Il existe de dizaines de linters pour Python qui sont disponibles aujourd’hui, le tableau ci-dessous décrit et compare certains entre eux [6].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Linter*** | ***Description*** | ***Pros*** | ***Cons*** |
| [pylint](https://pylint.org) | L’un des plus anciens (depuis 2006) et encore bien-maintenu aujourd’hui | - Un linter mature, les contributeurs ont fixé la plupart des bugs.  - Les fonctionnalités de base sont bien développées | - Assez lent  - Besoin de beaucoup de configurations  - Trop verbeux |
| [pyflakes](https://pypi.org/project/pyflakes/) | Première sortie en 2005 et encore maintenu aujourd’hui, Pyflakes fait une simple promesse : il ne se plaint jamais du style mais essaie de ne jamais émettre de faux positifs | - Très rapide  - Focaliser bien dans la vérification de la logique du code et empêcher des erreurs potentielles | - A peine vérifier le style e.g. docstrings manquants ou style de nommage …  - Suppose que le code est déjà bon par défaut |
| [pycodestyle](https://pypi.org/project/pycodestyle/) | Première sortie en 2016, il s’appelait autrefois pep8.  Pycodestyle est utilisé pour vérifier le code conformément au PEP8 | - Rapide  - Belle outputs, le lint est libellé par catégorie donc c’est facile pour le filtrage. | - Les conventions de nommage ne sont pas vérifiées, et non plus les docstrings |
| [pydocstyle](http://www.pydocstyle.org/en/stable/) | Première sortie en 2014, il est un outil d’analyse statique pour vérifier les docstrings conformément au PEP257 (mais il ne doit pas être considéré comme une implémentation de réf. | - Libeller et classifier des erreurs trouvées.  - Peut-être facilement combiné avec Pycodestyle ou autres car les erreurs sont préfixées avec un « D » (pour Docstrings) | - Seulement vérifier des docstrings |
| [flake8](https://pypi.org/project/flake8/) | Un wrapper de :  - pyflakes  - pycodestyles  - mccabe script (pour vérifier la complexité Mccabe) | - Prendre des avantages de plusieurs linters ensemble  - L’un des linters les plus utilisés aujourd’hui  - Vérifier la logique et le style du code, conformément au PEP8 | - Assez lent |
| [mypy](https://pypi.org/project/mypy/) | Un vérificateur statique de types pour Python, qui vérifie des programmes ayant des annotations de type conformément au PEP484 | - Posséder un système puissant de types avec des fonctionnalités tels que l’inférence de type, dactylographie progressive, génériques, et types union  - Un linter mature, bien maintenu depuis 2009 | - Limité à la vérification de types uniquement |
| [pyright](https://pypi.org/project/pyright/) | Un vérificateur de types pour des grandes code bases Python | - Beaucoup plus rapide que mypy et d’autres vérificateurs de types.  - Bien développé et maintenu (par Microsoft)  - Supporter beaucoup de fonctionnalités | - Limité à la vérification de types uniquement |

🡪 *Un petit tutoriel de flake8 peut-être trouvé dans* ***l’Annexe B*** *de ce document.*

### Les formateurs connus pour Python

En effet, il existe pas mal de formateurs de code pour Python aujourd’hui, le tableau ci-dessous décrit et compare quelques-uns [4][7][8][9][10].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Formatter*** | ***Description*** | ***Pros*** | ***Cons*** |
| [autopep8](https://pypi.org/project/autopep8/) | Automatiquement reformatter le code Python conformément au PEP8. Il utilise pycodestyle en derrière. | - L’un des plus anciens et encore bien maintenu aujourd’hui  - Pas trop stricte | - Focaliser seulement (et trop) à fixer des issues PEP8, cela parfois rend le format du code non-uniforme |
| [yapf](https://github.com/google/yapf) | *Yet Another Python Formatter*, maintenu par Google.  Yapf prend le code et le rend au meilleur format même si le code ne viole pas les règles | - Très configurable  - Rend le code uniforme | - Très lent pour lancer (parce qu’il reformate le tout le code quelque ce soit)  - Trop stricte |
| [black](https://pypi.org/project/black/) | Un formateur intransigeant, première sortie en 2018 | - Très rapide  - Rend le code uniforme | - Non configurable (sauf la longueur des lignes)  - Trop stricte |
| [isort](https://pypi.org/project/isort/) | Trier des imports en scripts Python automatiquement, conformément au PEP8 | - Très rapide | - Focaliser seulement dans des imports |

*Question: Est-ce que les formateurs vont modifier les comportements du programme ?  
Réponse: Non, parce qu’ils essaient de reformatter le code sans modifier l’exécution du programme i.e. les bytecodes Python restent inchangés.*

*🡪 A petit tutoriel de l’utilisation de black peut être trouvé dans* ***l’Annexe C*** *de ce document.*

* 1. ***Combiner des linters et des formateurs de code***

C’est toujours possible qu’on puisse **combiner plusieurs linters avec un ou plusieurs formateurs** pour la même base de code [11].

Par exemple, dans le même projet, on peut appliquer :

* *isort*: pour reformatter des imports dans le code
* *flake8*: pour empêcher des erreurs potentielles et vérifier des issues de style du code
* *black*: pour strictement reformatter le code
* *pyright*: pour vérifier les types dans le code

Quelques outils pré-combinés sont aussi disponibles tels que [flake8-black](https://pypi.org/project/flake8-black/) qui fournit un plugin flake8 pour valider le style du code à côté de black.

## 3. Pre-commit

### Qu’est-ce que c’est Git hooks ?

**Git hooks sont des scripts exécutés par Git** avant ou après des événements tels que : commit, push, et receive [13]. Git hooks est une fonctionnalité intégrée et ils sont exécutés localement. Chaque dépôt Git contient un dossier *.git/hooks* dont un script pour chaque hook auquel on peut appeler.

Voici quelques exemples simples de l’utilisation de hooks :

* *pre-commit*: vérifier des messages de commit pour des fautes d’orthographe
* *pre-receive:* renforcer les standards de codage du projet
* *post-commit*: informer par Email/SMS aux membres de l’équipe sur des nouveaux commits
* *post-receive*: pousser le code en production

Afin d’implémenter Git hooks, juste simplement écraser (ou créer) l’un des scripts dans le dossier *.git/hooks* puis le rendre exécutable.

### Le package pre-commit de Python

Les scripts Git hooks sont utiles pour identifier des problèmes simples avant de les soumettre à la révision du code [14][15]. Cependant, à mesure qu’on crée de plus en plus de librairies et de projets, le fait de **partager de hooks entre les projets est parfois fatigant** : on copie et colle les scripts d’un projet à l’autre et on doit modifier manuellement les hooks pour qu’ils fonctionnent avec de différents projets.

Donc, un package Python appelé « pre-commit » devient utile pour résoudre ces issues de hooks. **Pre-commit est un gestionnaire de packages pour des hooks pre-commit**. On a juste besoin de spécifier une liste de hooks qu’on veut (écrits dans n’importe quel langage) puis pre-commit gérera l’installation et l’exécution de ces hooks avant chaque commit Git.

Pour installer pre-commit, simplement exécuter : *pip install pre-commit*

Dans un projet Python, on doit ajouter « pre-commit » dans le fichier *requirements.txt*.

*🡪 Un petit tutoriel de l’utilisation de pre-commit peut être trouvé dans* ***l’Annexe D*** *de ce document.*

### Quelques hooks connus pour pre-commit

En effet, avec pre-commit en Python, on peut utiliser et combiner autant de hooks qu’on veut, en d’abord les déclarant dans le fichier *.pre-commit-config.yaml* [16].

L’exemple ci-dessous démontre comment intégrer plusieurs hooks [17] dans notre projet. On peut spécifier plusieurs dépôts dont chacun peut posséder un ou plusieurs hooks (référencé par id).

Dans cet exemple, on fait appel aux dépôts de black, isort, mypy, … pour vérifier la conformité du code avant de les commiter.

repos:  
- repo: <https://github.com/pre-commit/pre-commit-hooks>  
rev: v3.2.0  
hooks:  
- id: check-ast  
- id: check-json  
- repo: <https://github.com/pre-commit/mirrors-mypy>  
rev: v0.782  
hooks:  
- id: mypy  
args: [--ignore-missing-imports]  
- repo: <https://github.com/asottile/seed-isort-config>  
rev: v2.2.0  
hooks:  
- id: seed-isort-config  
- repo: <https://github.com/pre-commit/mirrors-isort>  
rev: v5.4.2  
hooks:  
- id: isort  
- repo: <https://github.com/psf/black>  
rev: 20.8b1  
hooks:  
- id: black

# PART II: Structurer un Projet Python

## Pourquoi avons-nous besoin de modèles de projet ?

***Maintenir un développement cohérent entre les différentes équipes***

S’il y n’y a pas de modèle du tout pour des projets de codage, notamment en Python, chacun sera créé différemment. Par exemple, certains développeurs préfèrent ajouter un répertoire de « logs » à la structure du projet alors que d’autres ne veulent peut-être pas, ou les dossiers et les fichiers sont nommés de manière incohérente entre différentes équipes (e.g. « src » vs « source » vs « scripts »). Par conséquent, avoir au moins un modèle pour chaque type de projet assurera la cohérence du développement entre les différentes équipes.

***Gagner du temps lors du démarrage d’un nouveau projet***

Créer un nouveau projet Python est parfois fatigant : il y a de nombreuses étapes répétitives comme créer setup.py, configurer pytests, construire une dizaine de fichiers et dossiers… Si on a déjà un modèle pour le projet cible, ça fera gagner beaucoup de temps.

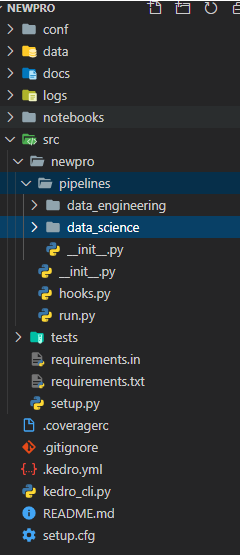
***Coder mieux avec des modèles bien définis***

Il existe de nombreux modèles pour différents types de projet Python (data science, ETL, dataviz & reporting …), et ils sont disponibles sur les dépôts tel que Github. Beaucoup d’entre eux sont très bien faits sur la base des expériences précieuses de leur créateur à travers de vrais projets Python. On peut, bien sûr, créer nos propres modèles, mais avant de le faire, essayez de ne pas réinventer la roue s’il existe déjà de bons modèles pour le même problème que le nôtre.

## Créer une structure avec un générateur de projet

La génération automatique de structure de projet permettra de **gagner beaucoup de temps** et apportera certains avantages, et de nombreux outils sont disponibles pour le faire.

L’image ci-dessous montre un exemple de projet Python généré par Kedro [18]. Il y a des répertoires pour la configuration, les données, la documentation, les logs, les scripts et même des dossiers spécifiques pour le traitement du pipeline.



*Cependant, si on veut définir nos modèles personnalisés pour la structure de projet, des outils comme* ***Cookiecutter*** *peut être une bonne solution [19].*

**Cookiecutter**

Il s’agit d’un outil de ligne de commande qui permet de créer des projets Python à partir de modèles appelés « cookiecutter », par ex. création d’un projet de package Python à partir d’un modèle de projet de package Python.

Cookiecutter est disponible sur Pypi : *pip install cookiecutter*

Afin de créer un modèle de projet avec Cookiecutter, définir d’abord la structure souhaitée avec tous les fichiers et dossiers nécessaires du projet, puis appeler l’outil cookiecutter appliqué sur le répertoire racine de cette structuration pour générer le modèle du projet.

Une chose intéressante lors de l'utilisation de cookiecutter est qu’on peut paramétrer autant d'éléments qu’on veut pour le modèle, il suffit simplement de définir les éléments à paramétrer en tant que variables avec la double marque "{", par exemple :

*{{cookiecutter.directory\_name}}*

*{{cookiecutter.file\_name}}.py*

*{{cookiecutter.other\_items\_eg\_file\_name\_or\_even \_file\_content}}*

Ensuite, cookiecutter itérera tous les éléments (fichiers et dossiers) du modèle pour remplacer les paramètres à l'intérieur de {{ }} par les valeurs saisies par le créateur du projet lorsqu'elles sont demandées par cookiecutter.



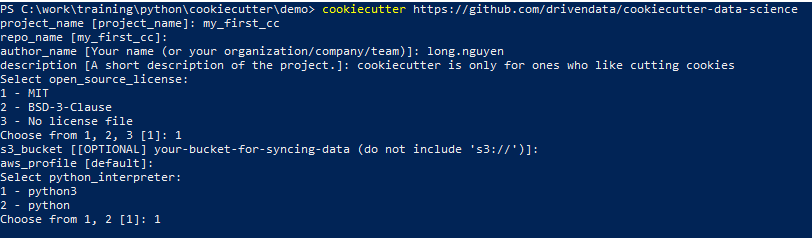
Si le créateur du projet laisse un paramètre vide, sa valeur sera remplacée par celle par défaut définie dans le fichier *cookiecutter.json*. Tous les paramètres du modèle doivent être définis dans ce fichier JSON, par exemple :

*{   
 “director\_name”: “Hello”,  
 “file\_name”: “NGUYEN”,  
 “other\_items\_eg\_file\_name\_or\_even \_file\_content”: “some\_value”,  
 …  
}*

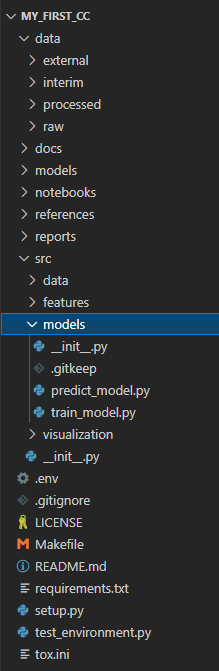
*Non seulement être en mesure de créer nos propres modèles, mais on peut également trouver facilement des modèles cookiecutter prédéfinis* sur des dépôts comme Github, qui correspondent à nos besoins. Par exemple, la commande suivante extrait un modèle cookiecutter pour un projet de science des données:

*cookiecutter https://github.com/drivendata/cookiecutter-data-science*

La saisie de cette ligne de commande entraînera une série de questions au créateur du projet, avec des valeurs indicatives déjà définies dans le fichier JSON de cookiecutter :



Après avoir répondu aux questions ci-dessus, un nouveau projet sera généré dans le répertoire local avec la structure suivante :



**Comparer Cookiecutter aux autres générateurs de projet ?**

Cookiecutter est peut-être l'outil le plus utilisé pour générer un projet Python à partir de modèles aujourd'hui, mais il existe également d'autres outils pour le faire. Le tableau ci-dessous montre une comparaison entre Cookiecutter et quelques autres [20].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Feature*** | ***Cookiecutter*** | ***Copier*** | ***Yeoman*** |
| *Peut paramétrer le nom des fichiers* | Oui | Oui | Oui |
| *Configuration* | Un seul JSON | Un seul YAML | Module JS |
| *Migrations* | Non | Oui | Non |
| *Programmé en* | Python | Python | Node JS |
| *Requiert JSON manuscrit* | Oui | Oui | Oui |
| *Requiert de modèles séparément* | Non | Non | Oui |
| *Requiert la programmation* | Non | Non | Oui (JS) |
| *Requiert suffixes pour modèles* | Non | Oui, configurable | Choix |
| *Taches hooks* | Oui | Oui | Oui |
| *Modèles dans un sous-répertoire* | Oui, requis | Choix | Oui, requis |
| *Format package de modèle* | Git ou Mercurial | Git | NPM |
| *Mise à jour de modèles* | Non | Oui | Non |
| *Engine de modèles* | Jinja | Jinja | EJS |

En comparant les trois générateurs de projet ci-dessus, [Yeoman](https://yeoman.io/learning/) ne semble pas être le meilleur choix pour les développeurs Python car il nécessite des connaissances en Node JS dans l’idéal. [Copier](https://pypi.org/project/copier/) apporte principalement deux avantages par rapport à Cookiecutter : il prend en charge les mises à jour de modèles et utilise le fichier YAML pour la configuration (YAML est plus recommandé que JSON pour définir les paramètres de configuration). Cependant, Cookiecutter est plus populaire auprès de la communauté Python car il a été publié pour la première fois en 2013 et il est toujours très bien entretenu aujourd'hui, tandis que Copier est assez récent (2019).

Alternativement, on peut regarder [Pyscaffold](https://pypi.org/project/PyScaffold/), un autre générateur de projet pour Python.

Spécifiquement, si nous réalisons un projet de science des données en Python, [Kedro](https://pypi.org/project/kedro/) est vraiment un bon choix aussi.

# APPENDIX A: Feuille de triche pour les règles PEP8 [24]

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""This module's docstring summary line.

This is a multi-line docstring. Paragraphs are separated with blank lines.

Lines conform to 79-column limit.

Module and packages names should be short, lower\_case\_with\_underscores.

Notice that this in not PEP8-cheatsheet.py

Seriously, use flake8. Atom.io with https://atom.io/packages/linter-flake8

is awesome!

See http://www.python.org/dev/peps/pep-0008/ for more PEP-8 details

"""

import os  # STD lib imports first

import sys  # alphabetical

import some\_thirvd\_party\_lib  # 3rd party stuff next

import some\_third\_party\_other\_lib  # alphabetical

import local\_stuff  # local stuff last

import more\_local\_stuff

import dont\_import\_two, modules\_in\_one\_line  # IMPORTANT!

from pyflakes\_cannot\_handle import \*  # and there are other reasons it should be avoided # noqa

# Using # noqa in the line above avoids flake8 warnings about line length!

\_a\_global\_var = 2  # so it won't get imported by 'from foo import \*'

\_b\_global\_var = 3

A\_CONSTANT = 'ugh.'

# 2 empty lines between top-level funcs + classes

def naming\_convention():

    """Write docstrings for ALL public classes, funcs and methods.

    Functions use snake\_case.

    """

    if x == 4:  # x is blue <== USEFUL 1-liner comment (2 spaces before #)

        x, y = y, x  # inverse x and y <== USELESS COMMENT (1 space after #)

    c = (a + b) \* (a - b)  # operator spacing should improve readability.

    dict['key'] = dict[0] = {'x': 2, 'cat': 'not a dog'}

class NamingConvention(object):

    """First line of a docstring is short and next to the quotes.

    Class and exception names are CapWords.

    Closing quotes are on their own line

    """

    a = 2;

    b = 4

    \_internal\_variable = 3

    class\_ = 'foo'  # trailing underscore to avoid conflict with builtin

    # this will trigger name mangling to further discourage use from outside

    # this is also very useful if you intend your class to be subclassed, and

    # the children might also use the same var name for something else; e.g.

    # for simple variables like 'a' above. Name mangling will ensure that

    # \*your\* a and the children's a will not collide.

    \_\_internal\_var = 4

    # NEVER use double leading and trailing underscores for your own names

    \_\_nooooooodontdoit\_\_ = 0

    # don't call anything (because some fonts are hard to distiguish):

    l = 1

    O = 2

    I = 3

    # some examples of how to wrap code to conform to 79-columns limit:

    def \_\_init\_\_(self, width, height,

                 color='black', emphasis=None, highlight=0):

        if width == 0 and height == 0 and \

           color == 'red' and emphasis == 'strong' or \

           highlight > 100:

            raise ValueError('sorry, you lose')

        if width == 0 and height == 0 and (color == 'red' or

                                           emphasis is None):

            raise ValueError("I don't think so -- values are %s, %s" %

                             (width, height))

        Blob.\_\_init\_\_(self, width, height,

                      color, emphasis, highlight)

    # empty lines within method to enhance readability; no set rule

    short\_foo\_dict = {'loooooooooooooooooooong\_element\_name': 'cat',

                      'other\_element': 'dog'}

    long\_foo\_dict\_with\_many\_elements = {

        'foo': 'cat',

        'bar': 'dog'

    }

    # 1 empty line between in-class def'ns

    def foo\_method(self, x, y=None):

        """Method and function names are lower\_case\_with\_underscores.

        Always use self as first arg.

        """

        pass

    @classmethod

    def bar(cls):

        """Use cls!"""

        pass

# a 79-char ruler:

# 34567891123456789212345678931234567894123456789512345678961234567897123456789

"""

Common naming convention names:

snake\_case

MACRO\_CASE

camelCase

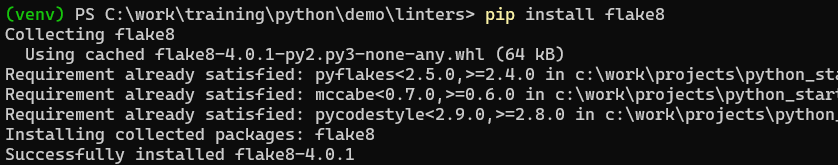
CapWords

"""

# Newline at end of file

# APPENDIX B: Intégration de flake8 dans nos projets

Flake8 est disponible sur PyPi, on peut simplement utiliser la commande pip pour le télécharger en local.



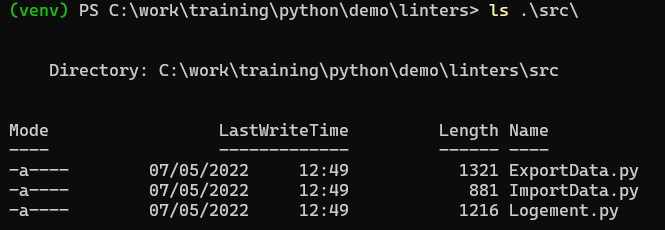
Après, utiliser flake8 pour vérifier le code conformément au PEP8 est tellement facile :

*flake8 /chemin/au/répertoire/cible*

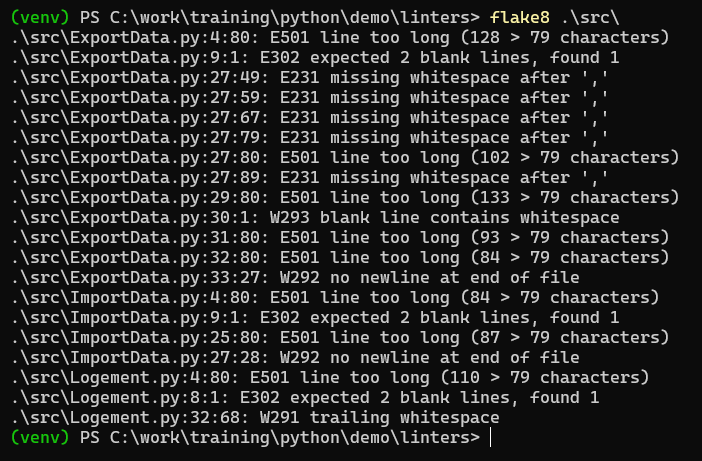
or

*flake8 /chemin/au/script/cible/.py*

Dans notre exemple, le répertoire « src » contient trois scripts Python, on va utiliser flake8 pour les vérifier.

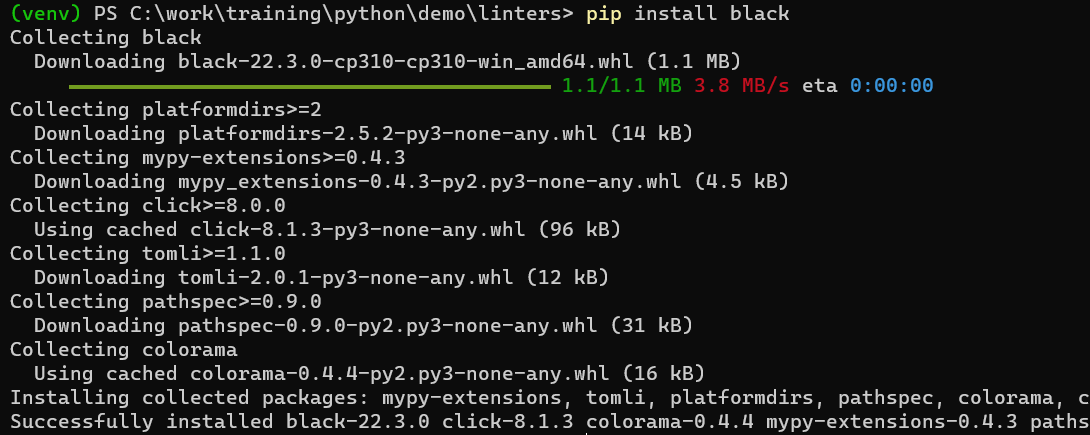


flake8 imprime toutes les issues PEP8 trouvés avec les préfixes E (pour erreur) et W (pour warning) :



# APPENDIX C: Intégration de black dans nos projets

Black est disponible sur Pypi, on peut simplement utiliser la commande pip pour le télécharger en local.



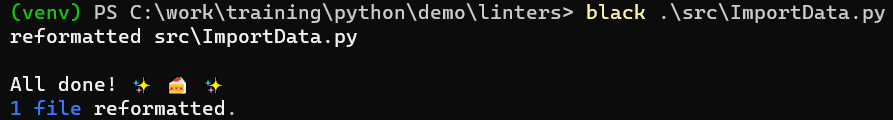
Puis, utiliser black pour reformatter le code est tellement facile :

*black /chemin/au/répertoire/cible*

or

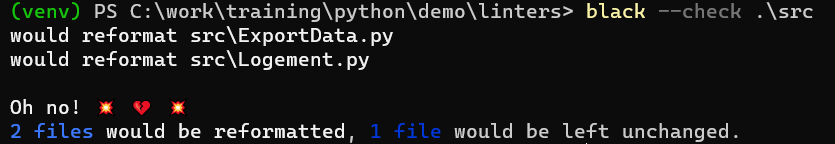
*black /chemin/au/script/cible/.py*

On prend le même exemple comme dans l’Annexe B. Utiliser black pour reformatter *ImportData.py* :

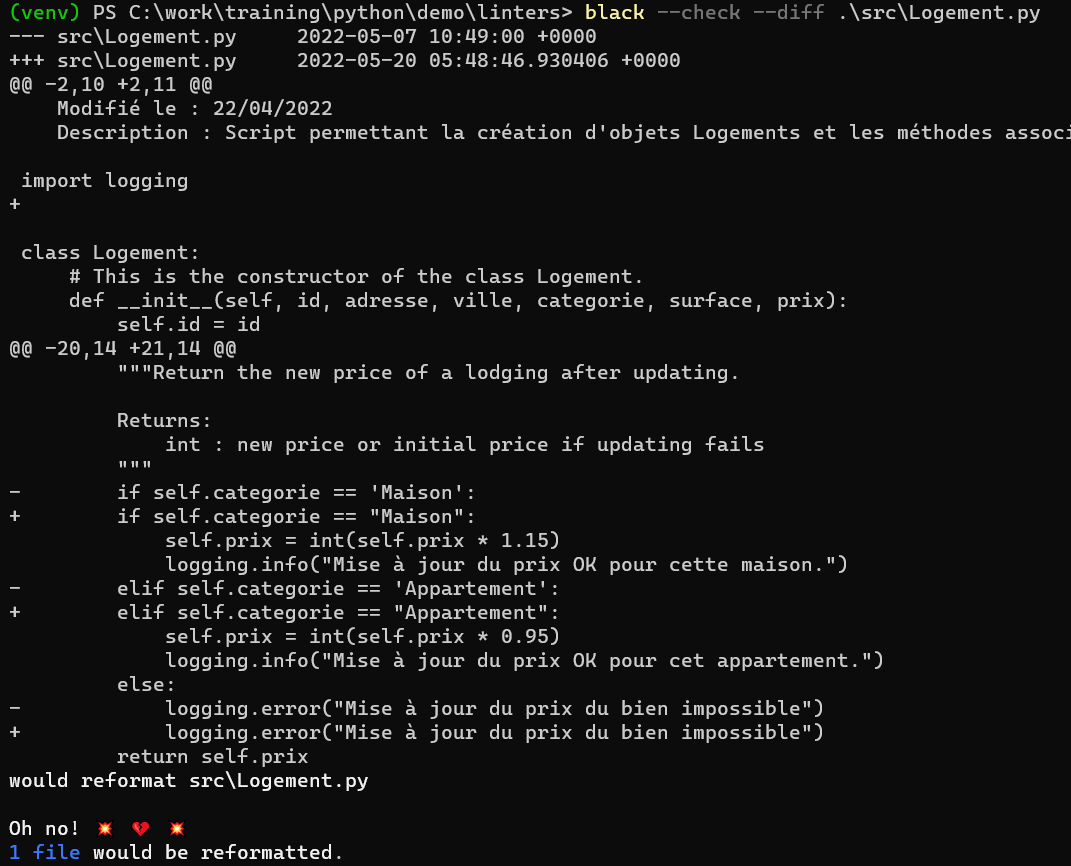


Le fichier cible a été modifié immédiatement (intransigéant) sans aucune information affichée.

Si on veut vérifier la répertoire/fichier avant de faire de modification, juste ajouter l’option *--check*:



Finalement, si on veut avoir plus de détail sur les modifications à appliquer, utiliser *--check --diff*:

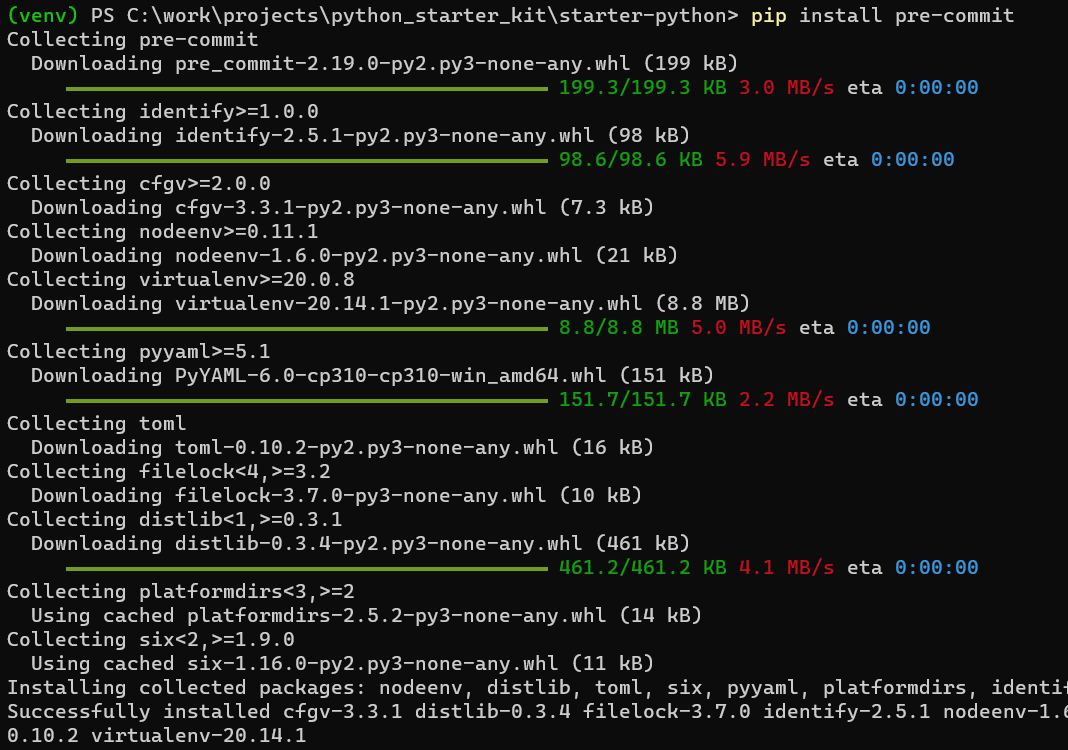


\* Il faut savoir que black limite la longueur maximale de lignes à 88 caractères (au lieu de 79 caractères en flake8), mais on peut changer cette valeur en utilisant l’option *-l* ou *--line-length*:

*black -l 79 src\_file.py*

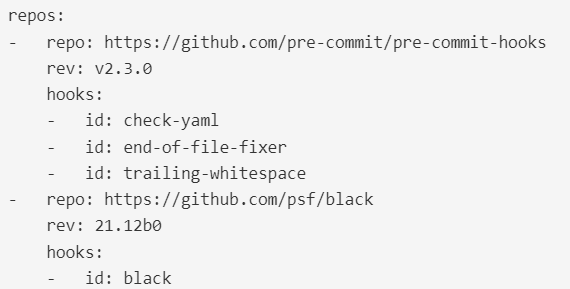
# APPENDIX D: Intégration de pre-commit dans nos projets Python [14]

Pre-commit est disponible sur PyPi, on peut donc simplement utiliser la commande pip pour le télécharger en local.



Ensuite, se naviguer au dépôt git local de notre projet afin d’ajouter une configuration de pre-commit en créant le fichier *.pre-commit-config.yaml*:

On peut générer une configuration très basique avec *pre-commit sample-config*, mais allons essayer les options d’exemple ci-dessous. Dans ce cas, on utilise *pre-commit-hooks* and *black*:



Puis utiliser *pre-commit install* pour mettre en œuvre les scripts git hook :



* Dès maintenant pre-commit sera lancé automatiquement sur chaque commande *git commit*!

REFERENCES

*[1] svitla.com/blog/why-where-and-when-to-use-coding-conventions*

*[2] Flore Barcellini et al., A study of online discussion in an OSS community, 2005*

*[3] https://realpython.com/python-pep8/*

*[4] https://books.agiliq.com/projects/essential-python-tools/en/latest/linters.html*

*[5]* [*https://medium.com/@awesomecode/format-code-vs-and-lint-code-95613798dcb3*](https://medium.com/@awesomecode/format-code-vs-and-lint-code-95613798dcb3)

*[6] https://realpython.com/python-code-quality/*

*[7] https://www.kevinpeters.net/auto-formatters-for-python*

*[8] https://deepsource.io/blog/python-code-formatters/*

*[9] https://blog.frank-mich.com/python-code-formatters-comparison-black-autopep8-and-yapf*

*[10]* <https://medium.com/@boxed/a-quick-performance-comparison-of-python-code-formatters-3a89478da8b8>

*[11] https://sbarnea.com/lint/black/*

*[12]* [*https://pypi.org/project/flake8-black/*](https://pypi.org/project/flake8-black/)

*[13] https://githooks.com*

*[14] https://pre-commit.com*

*[15] https://lorenzwalthert.github.io/precommit/articles/why-use-hooks.html*

*[16] https://www.architecture-performance.fr/ap\_blog/some-pre-commit-git-hooks-for-python/*

*[17] https://towardsdatascience.com/pre-commit-hooks-you-must-know-ff247f5feb7e*

*[18] https://www.lftechnology.com/blog/ai-pipeline-kedro/*

*[19] https://pypi.org/project/cookiecutter*

*[20] https://copier.readthedocs.io/en/stable/comparisons*

*[21]* [*https://peps.python.org*](https://peps.python.org)

*[22] https://pep8.org*

*[23] https://en.wikipedia.org/wiki/Zen\_of\_Python*

*[24] https://gist.github.com/RichardBronosky/454964087739a449da04*