GUIDE STARTER PYTHON

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

Révisions

|  |  |  |  |
| --- | --- | --- | --- |
| ***Version*** | ***Modifié le*** | ***Description*** | ***Modifié par*** |
| 0.1 | 18/05/2022 | Créer le document en anglais | Thanh-Long NGUYEN |
| 0.2 | 20/05/2022 | Traduire en français et reformatter le doc  Ajouter la table des matières | Thanh-Long NGUYEN  Estelle ROSSET |
| 0.3 | 24/05/2022 | Intégrer flake8 et black dans VS Code  Relecture complète du document (corrections) + passage du texte en justifié | Thanh-Long NGUYEN  Estelle ROSSET |
| 0.4 | 27/05/2022 | Ajout des parties sur la POO  +  Tester son code | Estelle ROSSET |
|  |  |  |  |

Objectifs

Le présent document a pour but d’aider de 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**, le **pré-commit**, et des **générateurs** de projet.

Ce document est divisé en trois 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 outils connus pour **créer et gérer des modèles** de projet.

Enfin, la dernière partie revient sur les principes de bases de la **programmation orientée objet (POO)** ainsi que sur l’importance des **tests unitaires** pour vérifier le bon fonctionnement du code.

***Table des matières***

[PARTIE I: Normes de codage et bonnes pratiques 3](#_Toc104561249)

[1. Guide de style pour le code Python 3](#_Toc104561250)

[1.1. L’importance de respecter les normes de codage 3](#_Toc104561251)

[1.2. Python Enhancement Proposals 3](#_Toc104561252)

[1.3. Autres PEPs 4](#_Toc104561253)

[2. Utiliser des Linters et des Formateurs pour le code Python 7](#_Toc104561254)

[2.1 Quelles sont les différences entre code linters et code formateurs ? 7](#_Toc104561255)

[2.2 Les linters connus pour Python 7](#_Toc104561256)

[2.3 Les formateurs connus pour Python 9](#_Toc104561257)

[3. Pre-commit 10](#_Toc104561258)

[3.1 Qu’est-ce que c’est Git hooks ? 10](#_Toc104561259)

[3.2 Le package pre-commit de Python 10](#_Toc104561260)

[3.3 Quelques hooks connus pour le pre-commit 11](#_Toc104561261)

[PARTIE II: Structurer un Projet Python 12](#_Toc104561262)

[1. Pourquoi avons-nous besoin de modèles de projet ? 12](#_Toc104561263)

[2. Créer une structure avec un générateur de projet 13](#_Toc104561264)

[3. Structuration du Starter Kit 16](#_Toc104561265)

[PARTIE III: La programmation objet et les tests unitaires 17](#_Toc104561266)

[1. La Programmation Orientée Objet (POO) 17](#_Toc104561267)

[1.1 Les avantages de la POO 17](#_Toc104561268)

[1.2 Les concepts clefs de la POO 17](#_Toc104561269)

[1.3 Organisation du code dans le starter kit 19](#_Toc104561270)

[2. Comment tester son code 20](#_Toc104561271)

[2.1 Mettre en place des tests unitaires 20](#_Toc104561272)

[2.2 Lancer les tests avec pytest sur VS Code 20](#_Toc104561273)

[2.3 Contrôler l’exhaustivité de ses tests avec coverage 21](#_Toc104561274)

[ANNEXE A: Feuille de triche pour les règles PEP8 [24] 22](#_Toc104561275)

[ANNEXE B: Intégration de flake8 dans nos projets 25](#_Toc104561276)

[Utiliser flake8 avec l’outil de lignes de commande 25](#_Toc104561277)

[Utiliser flake8 avec Visual Studio Code [25] 26](#_Toc104561278)

[ANNEXE C: Intégration de black dans nos projets 29](#_Toc104561279)

[Utiliser black avec l’outil de lignes de commande 29](#_Toc104561280)

[Utiliser black avec Visual Studio Code [27] 30](#_Toc104561281)

[Utiliser l’extension « Black Formatter » de Microsoft : 31](#_Toc104561282)

[ANNEXE D: Intégration de pre-commit dans nos projets [14] 32](#_Toc104561283)

# 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 du code lisible**, **cohérent**, et même **sans erreur**.

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

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.

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

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

* *Pas OK*: multi-déclaration par ligne, le 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**

Le PEP8 est l’un des PEPs les plus connus, ce document fournit des **guidelines** et décrit les **bonnes** **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].

Voici quelques exemples de règles requises par le PEP8 [4] :

* 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]

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

### Autres PEPs

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

* *Meta-PEPs*: sur PEPs ou processus. E.g.:
  + PEP1: Objectifs de PEP et guidelines
  + PEP6: Correctifs de bogues
  + PEP8: Guide de style Python
  + PEP387: Politique de rétrocompatibilité
  + …
* *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 des indices 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 docstrings** Python (commentaires de code). 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 des programmes** 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 illustrer les idées du 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 **bonnes pratiques** soient bien appliquées au code (e.g. importations, exceptions …)

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

*Les linters résolvent le deuxième problème*, ils aident à utiliser au mieux les syntaxes ou les nouvelles fonctionnalités du langage de programmation et ainsi empêcher des erreurs potentielles (même si des problèmes compliqués comme les règles de nommage des variables ne sont pas encore gérés 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 aussi efficaces que des formateurs de code.

Une autre différence entre ces 2 types d’outils : les linters détectent des problèmes puis ils laissent le développeur les corriger lui-même, tandis que des formateurs de code font des modifications automatiquement pour régler les erreurs qu’ils viennent de trouver.

En pratique, **on peut combiner des linters et des formateurs** selon 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*** | ***Avantages*** | ***Inconvénients*** |
| [pylint](https://pylint.org) | L’un des plus anciens (depuis 2006) et encore bien-maintenu aujourd’hui | - Un linter mature, les contributeurs ont corrigé 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 et essaie de ne jamais émettre de faux positifs | - Très rapide  - Se focalise bien dans la vérification de la logique du code et empêche des erreurs potentielles | - Peine à 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  - Les sorties générées par le linter sont libellées par catégorie donc facilite l’analyse des problèmes | - 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 | - Classifie les erreurs trouvées  - Peut-être facilement combiné avec Pycodestyle ou autres car les erreurs sont préfixées avec un « D » (pour Docstrings) | - Vérifie seulement des docstrings |
| [flake8](https://pypi.org/project/flake8/) | Une combinaison de :  - pyflakes  - pycodestyles  - mccabe script (pour vérifier la complexité Mccabe) | - Réunit les 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 de type statique pour Python, qui vérifie des programmes ayant des annotations de type conformément au PEP484 | - Possède un système puissant de types avec des fonctionnalités tels que l’inférence de type, la 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 type statique | - 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  - Ne s’intègre que dans les versions récentes de python (> 3.0) |

à *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 en compare quelques-uns [4][7][8][9][10].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Formatter*** | ***Description*** | ***Pros*** | ***Cons*** |
| [autopep8](https://pypi.org/project/autopep8/) | Reformatte automatiquement le code Python conformément au PEP8. Il utilise le linter pycodestyle pour déterminer quelles parties de code sont à modifier. | - L’un des plus anciens et encore bien maintenu aujourd’hui  - Pas trop strict | - Se focalise seulement (et trop) sur le PEP8, cela rend parfois le format du code non-uniforme |
| [yapf](https://github.com/google/yapf) | Un formateur de code maintenu par Google. Yapf prend le code et le restitue au meilleur format même si le code n’enfreint pas les règles | - Très configurable  - Rend le code uniforme | - Lancement très lent (parce qu’il reformate le tout le code quelque ce soit)  - Trop strict |
| [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 strict |
| [isort](https://pypi.org/project/isort/) | Trie automatiquement les importations par ordre alphabétique et de les séparer automatiquement en sections et par type conformément au PEP8 | - Très rapide | - Se focalise seulement sur les 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.*

*à Un 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 de **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 les imports dans le code
* *flake8*: pour empêcher des erreurs potentielles et corriger les problèmes de style de code
* *black*: pour reformatter strictement 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 avec black.

## 3. Pre-commit

### Qu’est-ce que c’est Git hooks ?

**Les Git hooks sont des scripts exécutés par Git** avant ou après des instructions Git tels que : commit, push, et receive [13]. Les Git hooks ont des fonctionnalités intégrées et ils sont exécutés localement. Chaque dépôt Git contient un dossier *.git/hooks* dont un script pour chaque hook qu’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.

Les Git hooks permettent d’automatiser et donc fiabiliser une partie du travail. Cependant ils peuvent aussi être bloquants. Le code de sortie des scripts **bloquants** est analysé pour savoir si le traitement doit être interrompu ou non. Souvent, on utilise les [codes de sorties standards](http://www.tldp.org/LDP/abs/html/exitcodes.html) qu’on peut résumer à :

* **0** (zéro) : tout est OK, on ne bloque pas,
* **≥1** : erreur, on bloque le traitement.

Par exemple, si nous produisons une sortie exit 1 dans un script de *pre-commit* alors le commit ne sera pas effectué.

### 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.

Dans ce cas, un package Python appelé « pre-commit » devient utile. **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 le pre-commit gérera l’installation et l’exécution de ces hooks avant chaque instruction Git commit.

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

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

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

### Quelques hooks connus pour le pre-commit

En effet, avec pre-commit en Python, on peut utiliser et combiner autant de hooks qu’on veut, d’abord en 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 un 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

# PARTIE 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 n’y pensent peut-être pas, ou les dossiers et les fichiers sont nommés de manière différente entre les 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 et plus généralement entre les développeurs.

***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 le fichier 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 au démarrage des développements.

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

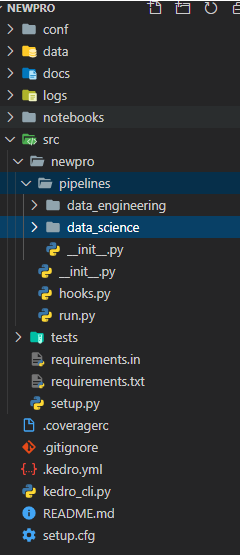
Il existe de nombreux modèles pour différents types de projet Python (data science, ETL, dataviz & reporting …), et ils sont disponibles sur les dépôts tel que Github. Beaucoup d’entre eux sont très bien faits sur la base des expériences précieuses de leur créateur à travers de vrais projets Python.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

**Kedro : génération automatique d’une structure 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 : création et utilisation de structures personnalisées**

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.



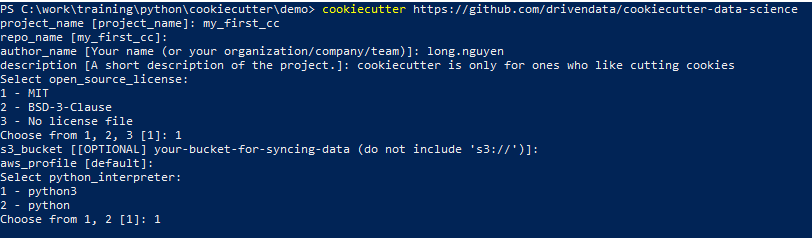
Si le créateur du projet laisse un paramètre vide, sa valeur sera remplacée par celle par défaut définie dans le fichier *cookiecutter.json*. Tous les paramètres du modèle doivent être définis dans ce fichier JSON, par exemple :

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

Cookiecutter permet non seulement de créer nos propres modèles, mais on peut également utiliser facilement des modèles cookiecutter prédéfinis sur des dépôts comme Github, s’ils 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 du modèle choisi.

**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 Copier ainsi que Yeoman [20].

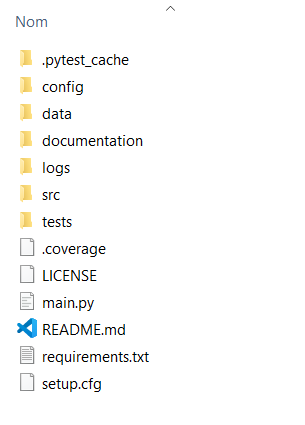
|  |  |  |  |
| --- | --- | --- | --- |
| ***Caractéristiques*** | ***Cookiecutter*** | ***Copier*** | ***Yeoman*** |
| *Peut paramétrer le nom des fichiers* | Oui | Oui | Oui |
| *Configuration* | Un seul JSON | Un seul YAML | Module JS |
| *Migrations* | Non | Oui | Non |
| *Programmé en* | Python | 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).

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

## Structuration du Starter Kit

Notre modèle créé dans le cadre du projet Start Kit pour vous aider au démarrage de vos projets est constitué des éléments suivants :



* Dossier **config** : on y met les fichiers de configuration
* Dossier **data** : répertorie les fichiers de données (entrée/sortie)
* Dossier **documentation** : permets d'enregistrer les documentations relatives à votre projet
* Dossier **logs** : contient les fichiers de logs générés lors de l'exécution des programmes
* Dossier **src** : les scripts Python nécessaires au fonctionnement du projet. Ils sont appelés dans le traitement principal main.py.
* Dossier **tests** : scripts permettant de faire les tests unitaires
* Fichier **COVERAGE** : il est automatiquement généré dès qu'on évalue la couverture (coverage) des tests de notre projet.
* Fichier **LICENSE** : spécifie par quelle license juridique est couvert notre projet.
* Fichier **main.py** : script principal du projet que l'on exécute pour que le traitement attendu soit réalisé. Il fait appel aux différentes classes du dossier src.
* Fichier **README.md** : il constitue la documentation principale du projet, c'est-à-dire celle qui doit être lue en premier par un utilisateur qui veut comprendre de quoi le projet traite et quelle en est la structure.
* Fichier **requirements.txt** : il contient les packages présents sur l'environnement de travail du développeur et qui sont donc nécessaires au bon fonctionnement du code.

# PARTIE III: La programmation objet et les tests unitaires

En plus des normes et de la structuration présentées dans les parties précédentes, on peut sécuriser davantage la qualité de son projet en ayant recours à la programmation orientée objet et à la réalisation de tests unitaires.

## La Programmation Orientée Objet (POO)

### Les avantages de la POO

La POO permets de nombreux avantages dont :

* Une meilleure structuration et organisation du code
* Le code est modulaire (découpé en modules) et donc évolutif
* Réduction des dépendances :
  + Les sections de code sont indépendantes
  + Le code est facilement réutilisable et maintenable par de nouvelles personnes

### Les concepts clefs de la POO

La POO repose sur des concepts, en voici les principaux pour vous aider à l’appréhender :

* **Objet :** quelque chose d’identifiable, réel ou non, du domaine du problème dont on parle en lui affectant des caractéristiques, ou qui doit être manipulé
  + Chaque entité peut avoir:
    - Des variables d’instance
    - Des variables de classe
* **Classes :** C'est la définition et l'implémentation de la structure d’un objet. C’est le “modèle” et on peut créer plusieurs objets à partir de celui-ci.
  + Particularité : par convention, un nom de classe est écrit en **CamelCase** (pratique qui consiste à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot).
* **Variable de classe :** La variable de classe s’applique une seule fois pour toutes les personnes. Python fait une copie de la variable de classe pour en faire une variable d’instance.
* **Instances :** À partir de la classe on peut créer un ou plusieurs objet (Logement 1, Logement 2, Logement 3…). On dit qu'on l'instancie. L'instance = l'objet.
* **Variables d’instance :** les attributs qui caractérisent l’objet (taille, position, état, valeur...)
  + Il faut toujours créer les variables d’instance à l’intérieur du constructeur.
  + Contrairement à la variable de classe, la variable d’instance est unique pour chaque objet (ex: chaque personne possède un nom différent)
* **Méthode d’instance** : une action appliquée à un objet. C'est exactement comme une fonction mais située l'intérieur de la classe.
  + Particularités :
    - Sont appelées par des objets
    - Doivent avoir self en premier paramètre
    - Il est possible d’accéder à des données ou autres méthodes de la classe instanciant en utilisant le “Self)
* **Constructeur :** il est aussi possible de "construire un ou plusieurs objets", c’est à dire instancier une classe. Il faudra définir un constructeur pour cela. Fonction à créer au moment de la construction de l'objet afin de l'initialiser.
  + Il s’écrit toujours \_\_init\_\_(self)
* **Héritage:** lorsque l'objet hérite d'une classe qui a des caractéristiques commune avec une classe déjà connue. Par exemple, si on souhaite créer un objet avec des caractéristiques spécifiques (ex : un œuf au chocolat). La classe obtenue sera la "classe enfant" (super classe), hérité de la "classe parent" (ou super classe). Facilite la réutilisation du code.
  + - Code:

Class Logement : => classe de base

Class Appartement(Logement) : => classe dérivée

L'objet enfant peut avoir ses propre caractéristiques et ses propres données.

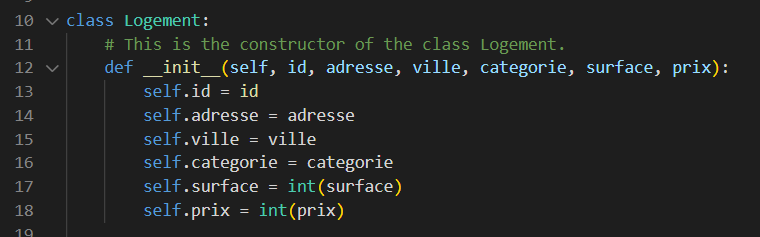
* **Self** ("soi-même") : il sert à spécifier de quelle variable ou méthode on parle. Quand on écrit self, on parle d'un objet créé lors de l'initialisation par le constructeur, dans ce self on stock les caractéristiques qui appartiennent à l'instance/objet.

Il permet par la suite de faire appel à toutes les caractéristiques que l'on souhaite, sans devoir préciser à chaque fois les paramètres associés aux caractéristiques d'un objet.

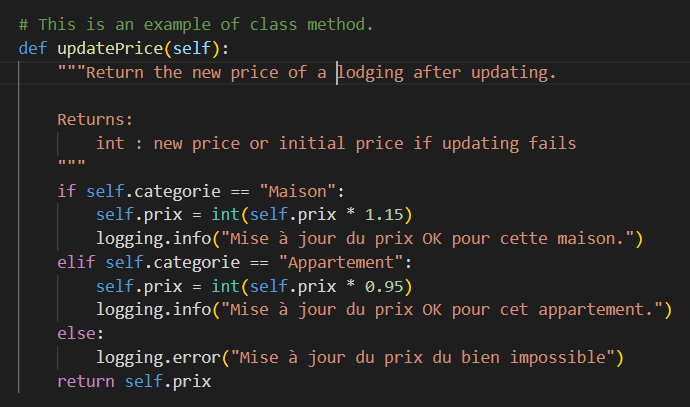
### Organisation du code dans le starter kit

Dans le starter kit, nous avons 3 classes (enregistrées dans le dossier src).

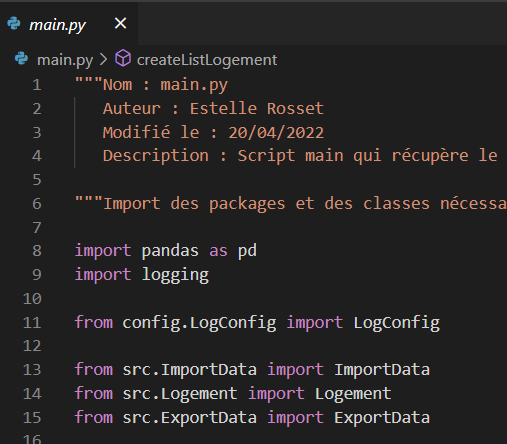
* + La classe **Logement** permet de définir les 6 **attributs** qui définissent un objet logement.



Nous avons également la méthode de classe updatePrice() qui permets de mettre à jour le prix d’un logement.



* + La classe **ImportData** permet de charger le fichier CSV source pour récupérer les informations des logements.
  + La classe **ExportData** permet de convertir les nouvelles informations de nos logements et de les exporter dans un nouveau fichier CSV.

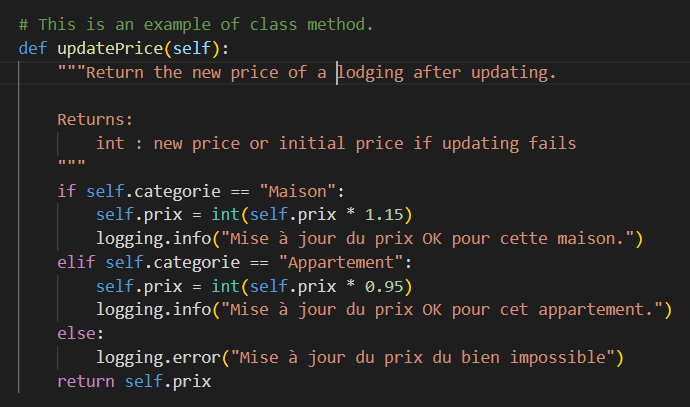
Ainsi, chacune de nos classes répond à un besoin fonctionnel précis, ce qui permet au développeur de savoir exactement où il devra modifier son code. Les classes sont ensuite importées/utilisées dans le fichier main.py. Nous aurions pu mettre l’ensemble du code dans le main.py mais cela en aurait compliqué son utilisation et sa maintenabilité.

## Comment tester son code

### Mettre en place des tests unitaires

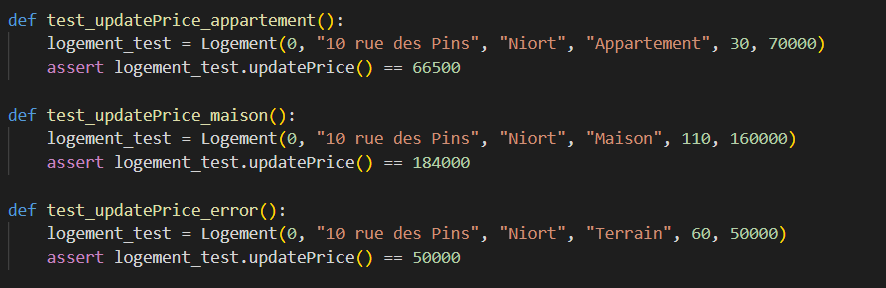
Les tests permettent de vérifier que le fonctionnement du code est bien celui attendu. Pour cela, on créé des scripts Python dédié au tests au sein de la structure du projet, sans le sous-dossier **tests** prévu à cet effet.

Si on reprend la fonction updatePrice() de la classe Logement :



On voit qu’un seul test de suffira pas car il y a plusieurs situations possibles et le mieux est de tester chacun d’elles.

Dans le dossier tests on créé un script **test\_Logement.py** : en effet, il est préférable d’organiser les tests selon la même logique que les modules de code du projet. Cela facilite la lisibilité des tests également.



On fait 3 tests pour tester les 3 modalités possibles de la fonction. De plus, pour tester la fonction updatePrice, on doit instancier des logements et donc on teste par la même occasion le constructeur de la classe Logement.

Dans les tests, l’instruction **assert** nous permets de comparer ce que le code fait (ex : logement\_test.updatePrice() avec le résultat qui est attendu (66500 dans l’exemple).

### Lancer les tests avec pytest sur VS Code

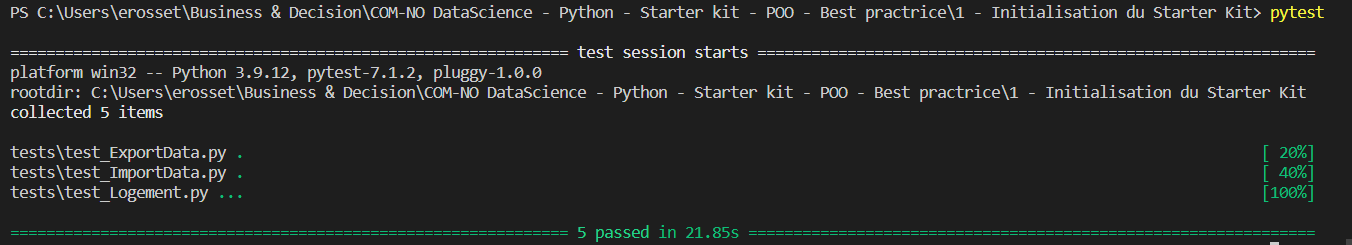
La rédaction des tests est la première étape. Il faut ensuite les exécuter pour s’assurer que notre code est opérationnel.

Pour cela, on utilise le package pytest de Python, qui est largement utilisé par les développeurs.

Il suffit de l’installer avec pip : *pip install pytest*

Puis, toujours dans le terminal, on exécute la commande : *pytest*

Celle-ci exécute tous les scripts du répertoire dont le nom commence par « test ». Dans le starter kit, il y a 3 scripts et 5 tests en tout. Après l’exécution, nous pouvons voir que les 5 tests sont « passed » donc OK.

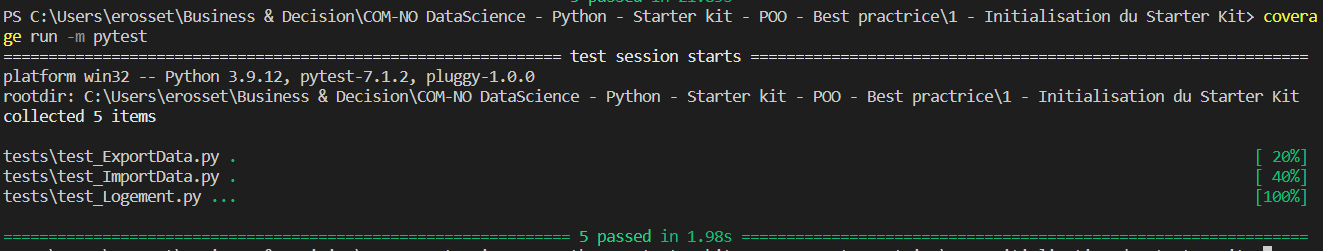


### Contrôler l’exhaustivité de ses tests avec coverage

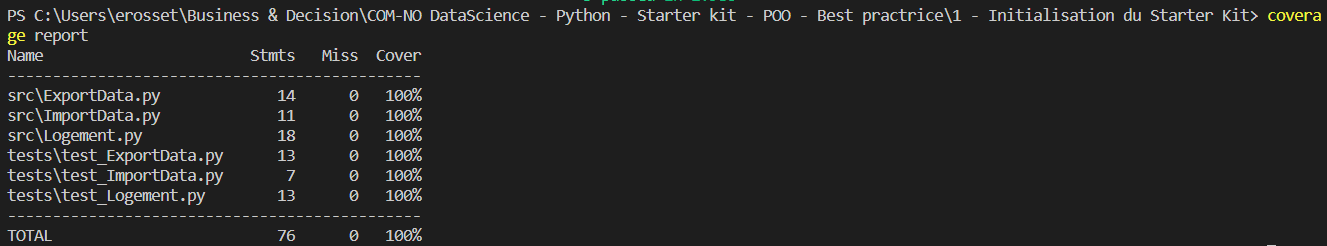
En complément de pytest, on utilise le package **coverage** pour mesurer quelle proportion de notre code source nos tests permettent de vérifier.

Il suffit de l’installer avec pip : *pip install coverage*

Puis dans notre terminal VS Code, on exécute : *coverage run –m pytest*



Enfin, la commande *coverage report*nous permets d’afficher le résultat. On peut ensuite réajuster nos tests unitaires si ceux-ci ne sont pas assez complets (coverage < 80%).



# PARTIE IV: Travail collaboratif avec Git

## Partager votre projet sur Git

## Commit des modifications

# ANNEXE 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

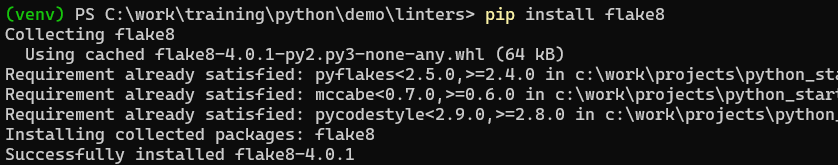
"""

# Newline at end of file

# ANNEXE B: Intégration de flake8 dans nos projets

## Utiliser flake8 avec l’outil de lignes de commande

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



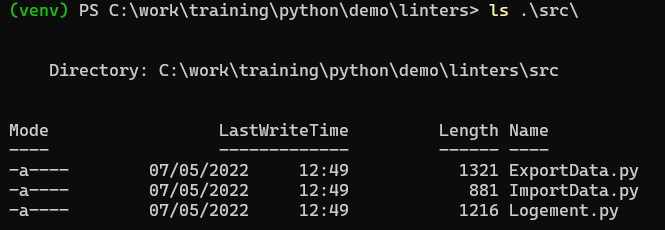
Après, utiliser flake8 pour vérifier le code conformément au PEP8 est tellement facile :

*flake8 /chemin/au/répertoire/cible*

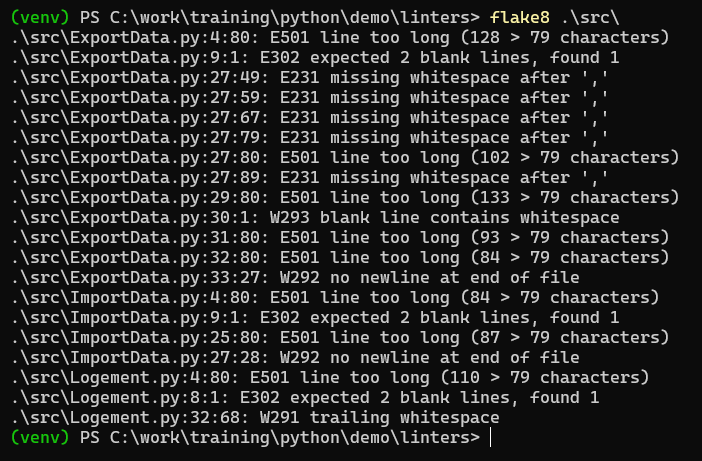
ou

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

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



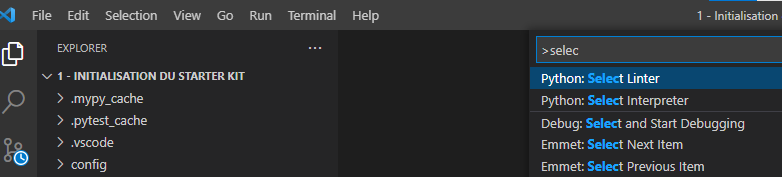
flake8 imprime toutes les issues PEP8 trouvés avec les préfixes E (pour erreurs) et W (pour warnings) :



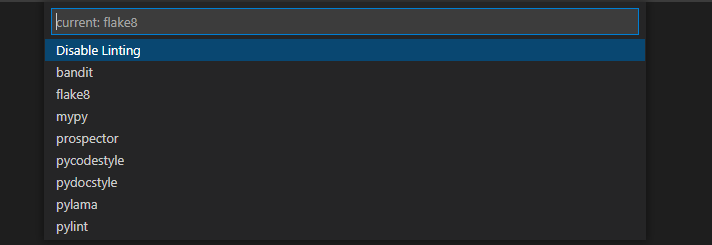
## Utiliser flake8 avec Visual Studio Code [25]

En effet, l’intégration de flake8 dans VS Code nous permet de vérifier la conformité de code de manière plus pratique : des erreurs liées à la PEP8 peuvent être affichés en immédiat pendant le coding. Voici comment intégrer flake8 dans VS Code.

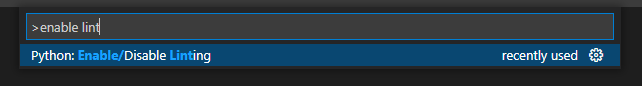
Ouvrir le projet Python souhaité avec VS Code, puis chercher l’option « Python : Select Linter » dans la palette de commandes (Ctrl + Maj + P).



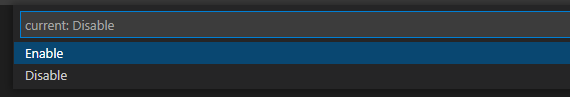
Une liste de linters disponibles sera affichée comme ci-dessous, choisir donc flake8 (ou autres linters)



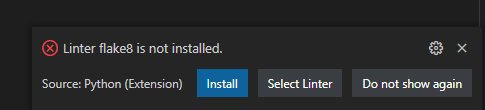
Ensuite, chercher l’option « Python : Enable/Disable Linting » dans la palette de commandes.



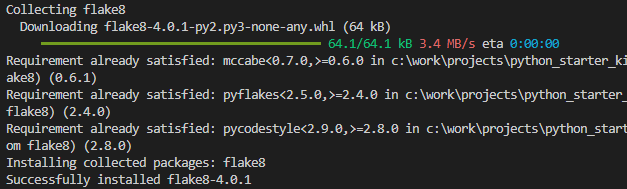
Puis choisir « Enable » pour activer l’utilisation de linters pour notre projet.



En ouvrant un fichier Python du projet, si flake8 n’a pas été installé, un message d’erreur sera affiché en bas à droit de l’IDE.

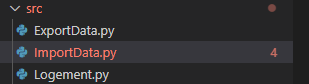


Cliquer sur « Install », flake8 sera téléchargé et installé dans (l’environnement virtuel) local.

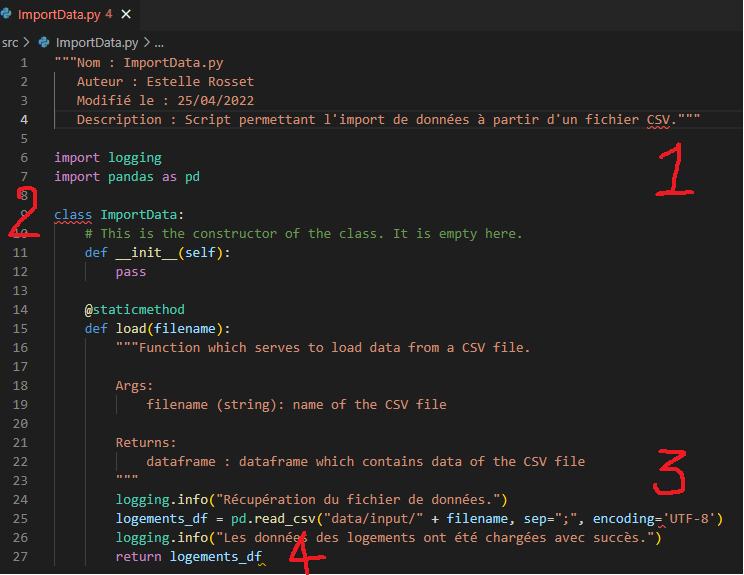


Maintenant, flake8 est bien intégré dans VS Code, et on peut l’utiliser pour vérifier le code conformément au PEP8. Un exemple est donné ci-dessous.

Le fichier *ImportData.py* comporte 4 erreurs PEP8, donc il est marqué *rouge :*

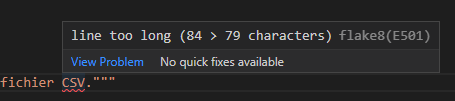


Dans ce fichier-là, on trouve facilement les erreurs rapportées (marquées par 1, 2, 3, et 4).

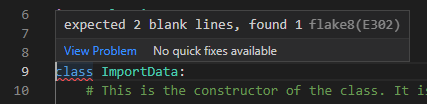


En pointant le curseur sur l’une des issues, on peut voir le message en plus de détail.

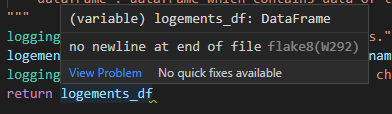
Par exemple, l’issue 1 indique que la longueur maximale de la ligne est dépassée (PEP8 E501) :



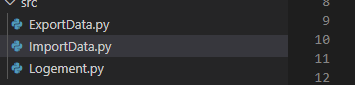
Dans l’issue 2, une ligne vide supplémentaire est requise avant la définition de la classe (PEP8 E302):



L’issue 4 : une ligne vide est demandée à la fin du fichier (PEP8 W292).

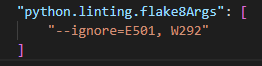


Une fois toutes les issues sont fixées, la couleur du fichier *ImportData.py* revient normale :



Afin de configurer l’utilisation de flake8 dans le projet, ouvrir le fichier *settings.json* avec la palette de commandes (Ctrl + Maj + P), puis ajouter la clé *python.linting.flake8Args* et ses valeurs.

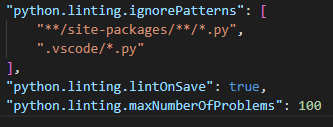
L’exemple ci-dessous montre comment on ignore la vérification pour les l’issue E501 et l’issue W292.



Par défaut, flake8 ignore E121, E123, E126, E226, E24, et E704.

Pour plus d’information sur la configuration de flake8, voir [26].

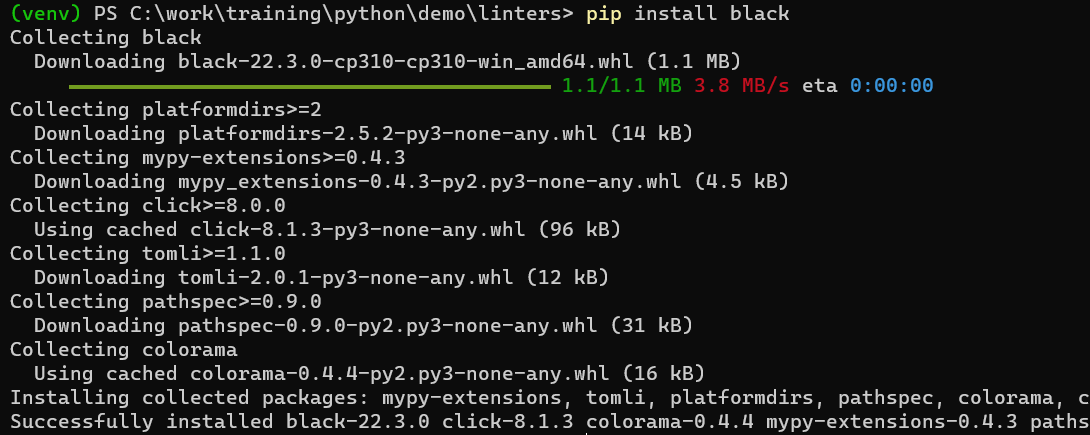
On peut aussi configurer d’autres paramètres de linting dans le fichier *settings.json*, par exemple :



# ANNEXE C: Intégration de black dans nos projets

## Utiliser black avec l’outil de lignes de commande

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



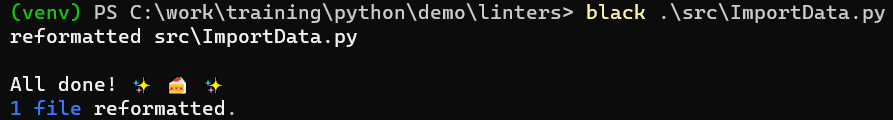
Puis, utiliser black pour reformatter le code est tellement facile :

*black /chemin/au/répertoire/cible*

ou

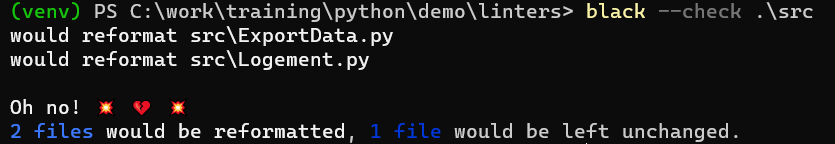
*black /chemin/au/script/cible/.py*

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

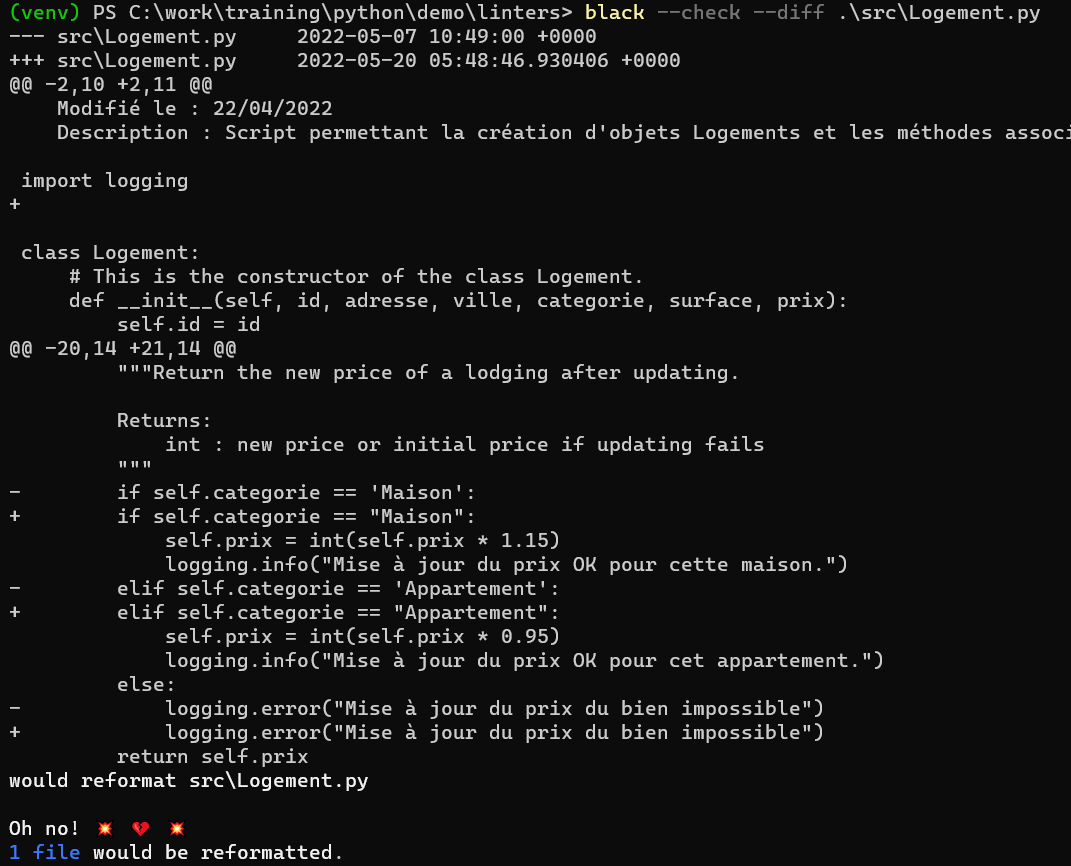


Le fichier cible a été modifié immédiatement (intransigeant) sans aucune information affichée.

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



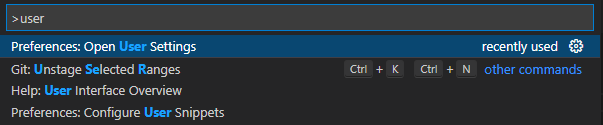
Finalement, si on veut avoir plus de détails sur les modifications à appliquer, utiliser *--check --diff*:



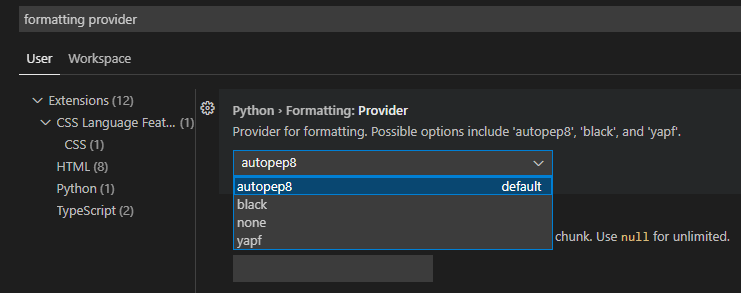
\* Il faut savoir que black limite la longueur maximale de lignes à 88 caractères (au lieu de 79 caractères en flake8), mais on peut changer cette valeur en utilisant en faisant : *black –line-length 79 src\_file.py*

## Utiliser black avec Visual Studio Code [27]

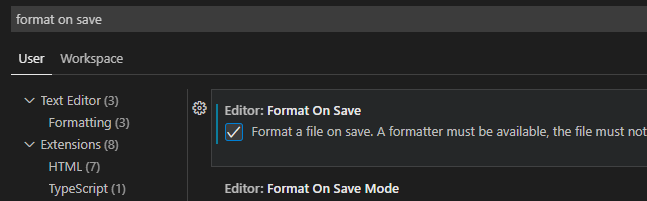
En supposant qu’on a déjà installé black dans l’environnement local (par ex. avec pip), ouvrir notre projet souhaité avec VS Code. Puis utiliser la palette de commandes (Ctrl + Maj + P) pour chercher l’option « Open User Settings » :

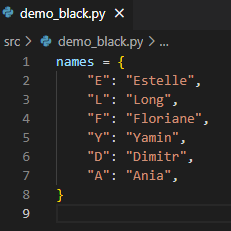


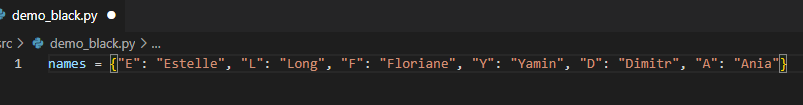
Ensuite, chercher l’option « Python Formatting : Provider », et choisir black comme le formateur.



Après, chercher l’option « Editor : Format on Save » puis cocher « format a file on save … », cela permet black de reformatter le fichier chaque fois qu’on le sauvegarde (par ex. avec Ctrl + S).



Maintenant, on peut utiliser black pour reformatter le code, comme l’exemple ci-dessous :



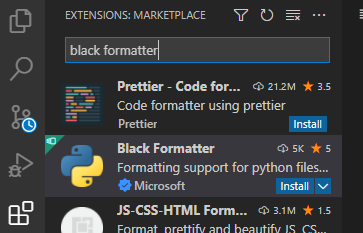
Après avoir sauvegardé le fichier (ex. Ctrl + S), le dictionnaire a été reformaté et une ligne vide est ajoutée à la fin du fichier Python automatiquement (intransigeant) !

Si on veut désactiver black, simplement choisir « none » dans la liste de « Python Formatting : Provider » mentionnée ci-dessus.

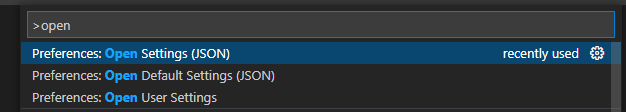
## Utiliser l’extension « Black Formatter » de Microsoft :

Au moment de la rédaction de ce document, Microsoft vient de sortir la « pré-release » de l’extension Black Formatter, mais elle fonctionne aussi bien.

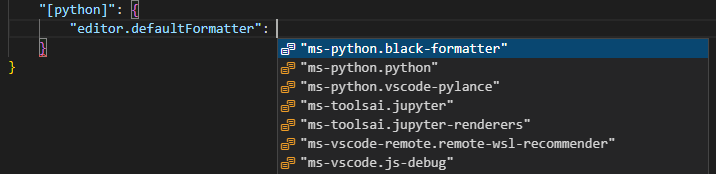
Simplement chercher sur la Marketplace pour l’extension « Black Formatter » puis l’installer.



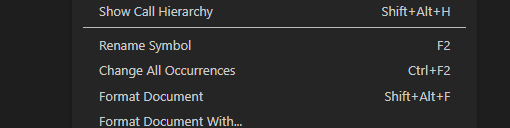
Ensuite, ouvrir le paramétrage « Open Settings (JSON) » avec la palette de commandes.



Ajouter l’option defaultFormater et choisir black comme le formateur par défaut.

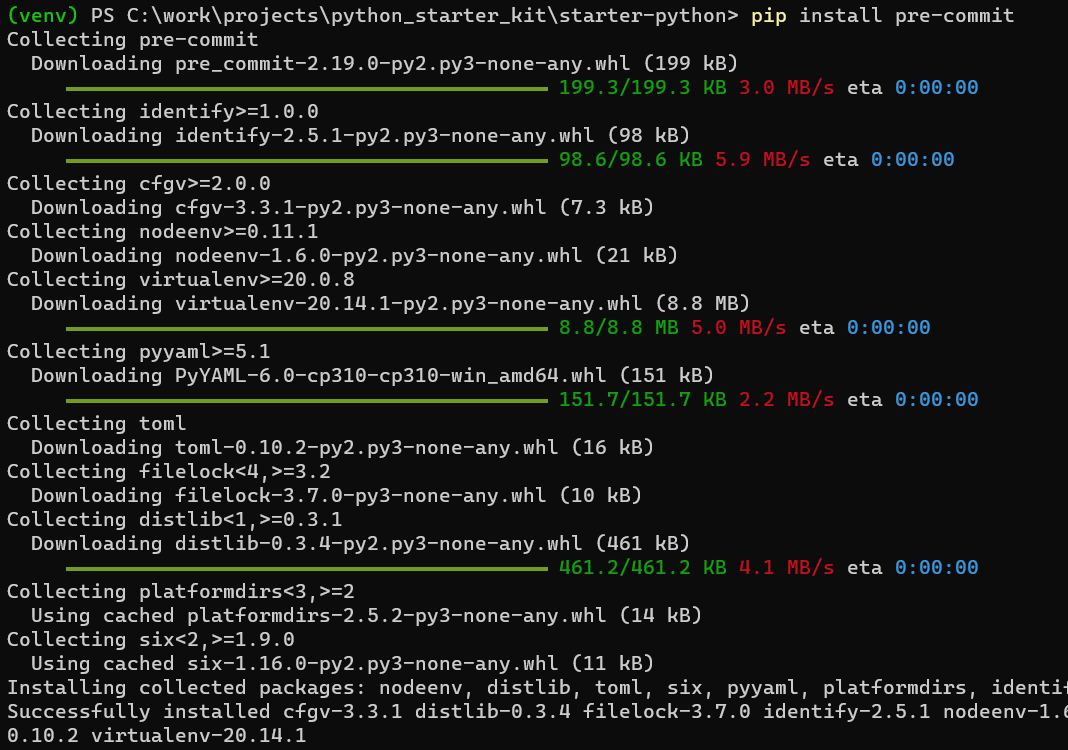


Maintenant, si on veut reformatter un fichier Python, simplement clic droit puis choisir « Format Document », le code sera reformaté immédiatement.



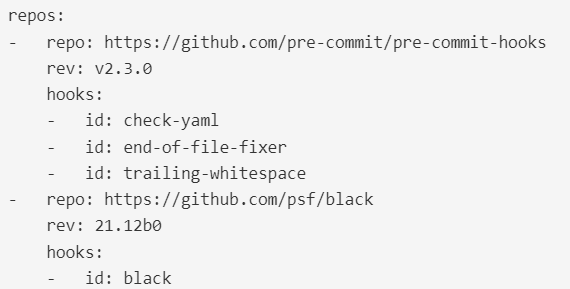
# ANNEXE D: Intégration de pre-commit dans nos projets [14]

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



Ensuite, se positionner au niveau du 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*

*[25] https://code.visualstudio.com/docs/python/linting*

*[26] https://flake8.pycqa.org/en/latest/user/configuration.html*

*[27] https://dev.to/adamlombard/how-to-use-the-black-python-code-formatter-in-vscode-3lo0*