PYTHON STARTER GUIDE

*on  
Linters, Pre-commit, and Project Generators*

***Revisions***

|  |  |  |  |
| --- | --- | --- | --- |
| ***Version*** | ***Modified on*** | ***Description*** | ***Modified by*** |
| 0.1 | 18/05/2022 | Document creation | Thanh-Long NGUYEN |
|  |  |  |  |
|  |  |  |  |

***Objectives***

The purpose of this document is to help B&D’s new Python developers create Python projects conform to organization’s standards and follow best practices by using available tools such as code linters, code formatters, pre-commit, and project generators.

The document is divided into two parts: the first one emphasizes on the importance of following Python coding standards, presents some of the most popular guidelines, and introduces some on-the-selves technologies and tools that automate those processes.

The second part explains the importance of having good and consistent templates for Python projects, then presents some of the most popular tools for creating and managing project templates.

TABLE OF CONTENTS

**PART I: Coding Standards and Best Practices**

1. Style Guide for Python Code

* 1. ***The importance of following coding standards***

Nowadays, coding is not only making programs that run correctly according to functional requirements but also creating code that are readable, consistent, and even error-coverable.

Because it’s worth knowing that 40 – 80% of the total cost of software is spent on its maintenance [1]. Moreover, software is hardly fully supported by its original authors. Thus, code that well follows recommended standards will improve software readability by allowing developers to understand new code faster and better.

Also, some code repositories like PyPi even require code to be well formatted and respect specific standards before being submitted.

Finally, like any other product, the software must be “well-packaged” and clean.

A very simple example of bad coding style and good/recommended coding style can be given below:

|  |  |
| --- | --- |
| **Bad** | **Good** |

* *Bad*: multi-statement per line, code is too dense
* *Good*: one line for each statement, a new line after *if/else* expression.
  1. ***Python Enhancement Proposals***

The authors of Python proposed an OSS (Open-Source Software) design process for Python called [PEPs](https://peps.python.org) (Python Enhancement Proposals). PEPs are the main mean for proposing new features, for collecting community input on an issue, and for documenting chosen design decisions [2].

Some PEPs describe new features for Python, while some others specify more general information about the process or organization of the Python community.

**PEP-8: Style guides for Python**

PEP8 is one of the most well-known PEPs, which is a document that provides guidelines and best practices on how to write Python code. It was written in 2001 by Guido Van Rossum, Barry Warsaw, and Nick Coghlan. The primary focus of PEP8 is to improve the readability and the consistency of Python code [3].

Some examples of conventions required by PEP8 [4] may include:

* Spaces are the preferred indentation method (rather than tabs)
* Use 4 spaces per indentation level
* Limit all lines to a maximum of 79 characters
* Separate top-level function and class definitions with two blank lines
* Method definitions inside a class are surrounded by a single blank line
* …

For more examples and concrete usages of PEP8, have a look at: <https://pep8.org> [22]

*🡪 In addition, a cheat sheet of following PEP8 rules can be found in* ***Appendix A*** *of this document.*

* 1. ***Other popular PEPs***

Until now, there have been a vast number of PEPs proposed, divided into several categories [21]:

* *Meta-PEPs*: about PEPs or processes. E.g.:
  + PEP1: PEP purposes and guidelines
  + PEP6: Bug fix releases
  + PEP8: Style guides
  + PEP387: Backwards compatibility policy
  + …
* *Informational PEPs*, e.g.:
  + PEP20: The Zen of Python (very well-known)
  + PEP257: Docstring conventions
  + PEP603: Annual release cycle for Python
  + PEP483: The theory of type hints
  + …
* *Accepted PEPs (validated but may not be implemented yet)*
* *Open PEPs (under consideration)*
* *Finished PEPs (done, with a stable interface)*
* *...*

Let’s have a look at some popular PEPs that provide style guide, beside PEP8.

**PEP-257: Docstrings Convention**

This PEP documents the semantics and conventions associated with Python docstrings. Some examples include:

* Always use triple quotes around docstrings
* Multi-line docstrings must contain a one-line summary followed by a blank line, then a more elaborate description.
* Insert a blank line after each docstring
* …

**PEP-20: The Zen of Python**

PEP20 is a collection of 19 guiding principles for writing computer programs that influence the design of Python, written by Tim Peters on the Python mailing list in 1999 [23].

Those principles are listed below:

*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!*

Some code snippets are given below to demonstrate the ideas of 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. Using Linters and Formatters for Python Code

* 1. ***Difference between code linter vs code formatter?***

They are not the same!

[5] When working in a team that shares the same code base, we want that:

* The same code format is applied to every place (e.g. tabs, spaces, blank lines …)
* Best practices are well applied to the code (e.g. library importing, exception handling …)

*Code formatters solve the first problem*, it tries to “make up” the entire code in a consistent way while keeping the program’s behaviors unchanged.

*Code linters solve the second problem*, it helps to use the better syntaxes or new features of the programming languages and prevent possible errors (but some difficult problems like variable naming have not been carried out till the time of this writing). It’s worth noting that code linters can also solve the first problem, but it’s not as efficient as code formatters.

Another difference is that code linters often only detect issues then they let developers do their corrections while code formatters automatically fix issues they find.

In practices, it’s recommended to combine both code linters and code formatters for better programming quality.

* 1. ***Popular code linters for Python***

There are dozens of linters for Python that are available today, the below table describes and compares some of the most popular ones [6].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Linter*** | ***Description*** | ***Pros*** | ***Cons*** |
| [pylint](https://pylint.org) | One of the oldest linters (since 2006) and is still well-maintained today | - A mature linter, the contributors have fixed most major bugs and core features are well-developed | - Quite slow to run  - Need a lot of configurations  - Too verbose |
| [pyflakes](https://pypi.org/project/pyflakes/) | First released in 2005 and still supported today, Pyflakes makes a simple promise: never complain about style but try to never emit false positives | - Very fast to run  - Focus very well on checking logical code issues and prevent potential errors | - Hardly check coding style e.g. missing docstrings or naming styles …  - Assume that the code is already good by default |
| [pycodestyle](https://pypi.org/project/pycodestyle/) | First released in 2016, formerly called pep8, it is used to check Python code against some of style conventions in PEP8 | - Fast to run  - Well-look outputs, the lint is labeled by category so it’s easy to be filtered | - Naming conventions are not checked, and neither are docstrings |
| [pydocstyle](http://www.pydocstyle.org/en/stable/) | First released in 2014, it is a static analysis tool for checking compliance with Python docstring conventions.  Pydocstyle supports most of PEP257 out of the box (but it should not be considered as a reference implementation) | - Label and categorize various errors found.  - Can be combined with pydocstyle or others since all the errors are prefixed with a “D” (for docstring). | - Only check docstring conventions |
| [flake8](https://pypi.org/project/flake8/) | A wrapper around  - pyflakes  - pycodestyles  - Mccabe script (to check Mccabe complexity) | - Take advantages of several linters together  - One of the most used and supported linters today  - Check both logical and stylist lint, following PEP8 | - A little slow to run |
| [mypy](https://pypi.org/project/mypy/) | An optional static type checker for Python that checks programs having type annotations conforming to PEP484 | - Have a powerful type system with features such as type inference, gradual typing, generics and union types  - A mature product and well-maintained since 2009 | - Limited only to type checking |
| [pyright](https://pypi.org/project/pyright/) | A fast type checker mean for large Python code base | - Much faster than mypy and other type checkers  - Well developed and maintained (by Microsoft)  - Support many features following dozens of PEPs | - Limited only to type checking |

🡪 *A small tutorial of flake8 usages can be found in* ***Appendix B***.

* 1. ***Popular code formatters for Python***

Indeed, there are quite many code formatters for Python, the below table describes and compares some of the most popular ones [4][7][8][9][10].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Formatter*** | ***Description*** | ***Pros*** | ***Cons*** |
| [autopep8](https://pypi.org/project/autopep8/) | Automatically format Python code, conforming to PEP8. It uses pycodestyle utility. | - One of the earliest Python formatters and still well-maintained  - Not too strict | - Focus only on fixing PEP8 issues, which makes code formatting ununiform sometimes |
| [yapf](https://github.com/google/yapf) | *Yet Another Python Formatter*, maintained by Google.  Yapf takes the code and reformats it to the best formatting that conforms to PEP8, even if the code doesn’t violate the style guides. | - Highly configurable  - Make code uniform | - Very slow to run because it even reformats code that doesn’t violate the style guides.  - Too strict reformatting |
| [black](https://pypi.org/project/black/) | An uncompromising Python code formatter, first released in 2018 (quite recent) | - Very fast to run  - Make code uniform | - Not configurable (except line-length)  - Too strict reformatting |
| [isort](https://pypi.org/project/isort/) | Sort imports in Python files automatically following PEP8 | - Very fast to run | - Focus only on reformatting imports |

*Question: Do code formatters change program’s behaviors?  
Answer: No, because they try to reformat the code without changing the program’s execution i.e. Python bytecodes are kept unchanged.*

*🡪 A small tutorial of black usages can be found in* ***Appendix C****.*

* 1. ***Combine code linters and code formatters***

It’s always possible that we can combine several linters along with one or more formatters for the same code base [11].

For example, in the same project, we can apply:

* *isort*: to reformat import statements
* *flake8*: to prevent possible errors and check stylish issues
* *black*: to strictly reformat codes based on PEP8
* *pyright*: to provide type checking

Some pre-combined tools are also available such as [flake8-black](https://pypi.org/project/flake8-black/) which provides flake8 plugin for validating code style alongside with black.

3. Pre-commit

* 1. ***What are Git hooks?***

Git hooks are scripts that Git executes before or after events such as: commit, push, and receive [13]. Git hooks are a built-in feature and are run locally. Each Git repository has a *.git/hooks* folder with a script for each hook we can bind to. Some simple examples of using hook scripts include:

* *pre-commit*: check the commit message for spelling errors
* *pre-receive:* enforce project coding standards
* *post-commit*: email/SMS team members of a new commit
* *post-receive*: push the code to production

In order to implement Git hooks, just simply overwrite (or create) one of the scripts in the *.git/hooks* folder and make it executable.

* 1. ***Python’s pre-commit package***

Git hook scripts are useful for identifying simple issues before submission to code review [14][15]. However, as we create more libraries and projects, sharing pre-commit hooks across projects is sometimes painful. We copy and paste unwieldy bash scripts from project to project and must manually change the hooks to work for different project structures.

So a Python package named “*pre-commit*” comes in handy to solve those hook issues. It is a multi-language package manager for pre-commit hooks. We just specify a list of hooks we want then pre-commit will manage the installation and execution of the hooks (written in any language) before every commit.

To install pre-commit, simply run: *pip install pre-commit*

In a Python project, we have to add “pre-commit” to the requirements.txt file.

*🡪 For a simple tutorial of pre-commit, see* ***Appendix D****.*

* 1. ***Popular pre-commit hooks***

Indeed, with Python pre-commit we can use and combine as many pre-commit hooks as we want, by firstly declaring them in the *.pre-commit-config.yaml* file [16].

The below example shows how to integrate several popular hooks [17]. For each one, we specify the repository and the hook ids that we need e.g. black, isort, mypy, etc.

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: Structuring a Python Project**

1. Why do we need project templates?

***Keep development consistent among different teams***

If there is no template at all for coding projects, notably in Python, each one will be created differently. For example, someone may prefer adding a logs directory to the project’s structure while some others may not, or folders and files are named inconsistently by different teams although they may have the same objectives (e.g. “src” vs “source”, etc.). Therefore, having at least a template for each type of project will keep the development consistent among different teams.

***Save time when starting a new project***

Creating a new Python project is sometimes tired. There are numerous repeated steps such as creating setup.py, configuring pytest, constructing a dozen of files and folders … If we already have a template for the target project, that will save a lot of time.

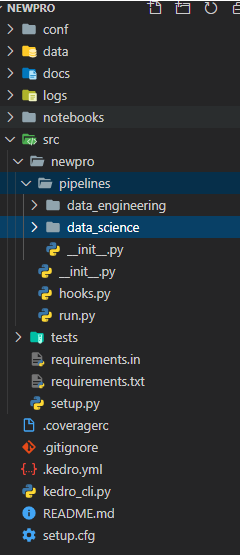
***Code better with well-defined templates***

There are a lot of templates for different types of Python projects (data science, ETL, dataviz & reporting …), and they are available on many code repositories such as Github. Many of them are very well-made based on valuable experiences of their creators through real Python projects. We can, of course, create our own templates, but before doing so, try not to reinvent the wheel if there are already good templates for the same problem.

1. Stick with a project generator

Automatically generating project templates will save a lot of time and bring some advantages, and there are many tools that are available for doing so.

The following image shows an example of a Python project that is generated by Kedro [18]. There are directories for config, data, docs, logs, scripts, and even specific directories for pipeline processing.



*However, if we want to define our custom templates for project structure, tools like* ***Cookiecutter*** *can be a good choice* [19]*.*

**Cookiecutter**

It is a command-line utility that creates Python projects from templates so-called cookiecutters, e.g. creating a Python package project from a Python package project template.

Cookiecutter is available on PyPi: *pip install cookiecutter*

In order to create a project template with Cookiecutter, just define firstly the desired structure with all the necessary files and folders for the project, then call the cookiecutter utility applied on that structure’s root directory to generate the project template.

One interesting thing when using cookiecutter is that we can parameterize as many items as we want for the template, just simply define the items to be parameterized as variables with the double “{“ mark, for example:

{{cookiecutter.directory\_name}}

{{cookiecutter.file\_name}}.py

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

Then cookiecutter will iterate all the items (files & folders) of the template to replace parameters inside the {{ }} by values inputted from project’s creator when they are prompted by cookiecutter.



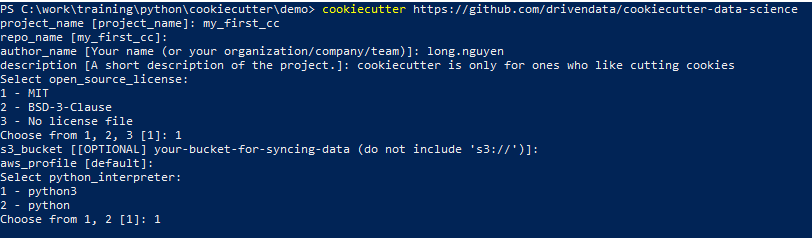
If the project’s creator leaves a parameter blank, its value will be replaced by the default one defined in the *cookiecutter.json* file. All template’s parameters must be defined in this JSON file, for instances:

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

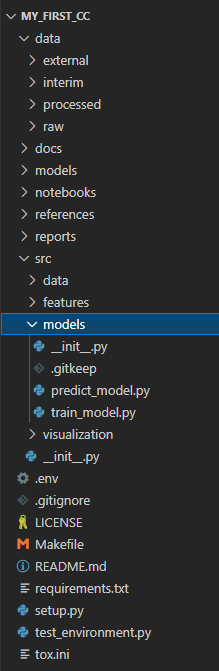
*Not only be able to create our own templates, but we can also easily find pre-made cookiecutter templates* on repositories like Github that fits our needs. For example, the following command pulls a cookiecutter template for a data science project:

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

Entering that command line will prompt a series of question to the project’s creator, with hint values which are already defined in the cookiecutter’s JSON file:



After answering the above questions, a new project will be generated into our local directory with the following structure:



**Compare Cookiecutter to other project generators?**

Cookiecutter is perhaps the most used tool to generate a Python project from templates today, but there are also some other tools for doing so. The below table shows a comparison between Cookiecutter and some others [20].

|  |  |  |  |
| --- | --- | --- | --- |
| ***Feature*** | ***Cookiecutter*** | ***Copier*** | ***Yeoman*** |
| *Can template file names* | Yes | Yes | Yes |
| *Configuration* | Single JSON file | Single YAML file | JS module |
| *Migrations* | No | Yes | No |
| *Programmed in* | Python | Python | Node JS |
| *Require handwriting JSON* | Yes | Yes | Yes |
| *Require installing templates separately* | No | No | Yes |
| *Require programming* | No | No | Yes (JS) |
| *Require templates to have a suffix* | No, not configurable | Yes, configurable | You choose |
| *Task hooks* | Yes | Yes | Yes |
| *Template in a subfolder* | Yes, required | You choose | Yes, required |
| *Template package format* | Git or Mercurial | Git | NPM |
| *Template updates* | No | Yes | No |
| *Template engine* | Jinja | Jinja | EJS |

Comparing the three project generators above, [Yeoman](https://yeoman.io/learning/) seems not to be the best choice for Python developers as it requires some knowledge in Node JS. [Copier](https://pypi.org/project/copier/) brings mainly two advantages over Cookiecutter: it supports template updates and uses YAML file for configuration (YAML is more recommended than JSON to define config parameters). However, Cookiecutter is more popular to the Python community as it was first released in 2013 and still very well-maintained today whereas Copier is quite recent (2019).

Alternatively, we can look at [Pyscaffold](https://pypi.org/project/PyScaffold/), another project generator for Python.

Specifically, if we are doing a data science project in Python, [Kedro](https://pypi.org/project/kedro/) is really a good choice too.

**APPENDIX A**: *Coding examples as a cheat sheet for PEP8 conventions [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

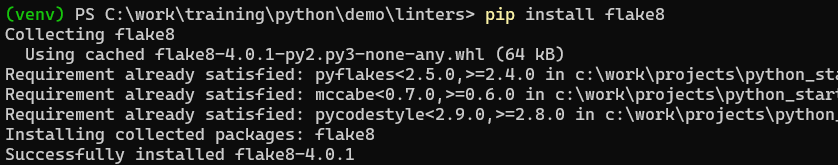
CapWords

"""

# Newline at end of file

**APPENDIX B**: *Integrating flake8 in your Python projects*

Flake8 is available in PyPi, we can simply use the pip command to get it into local (using virtual environment is recommended).



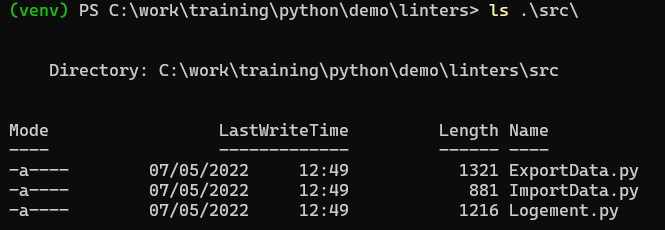
Then, using flake8 to verify code conform to PEP8 is so easy:

*flake8 /path/to/target/directory*

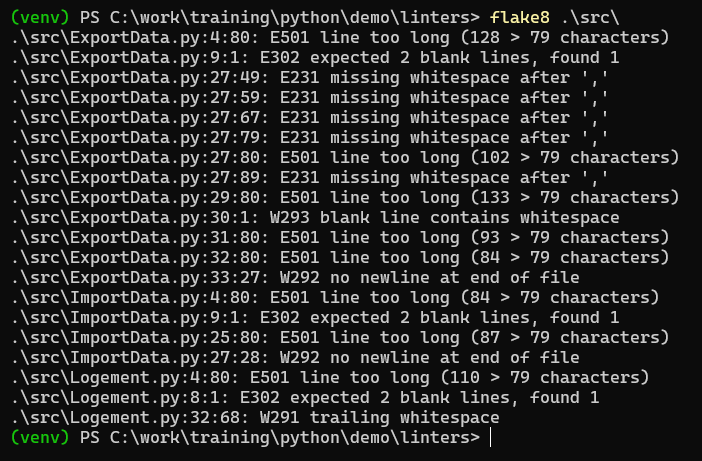
or

*flake8 /path/to/target/source/file.py*

In our example, the “*src*” folder contains three Python scripts, we will use flake8 to verify them.

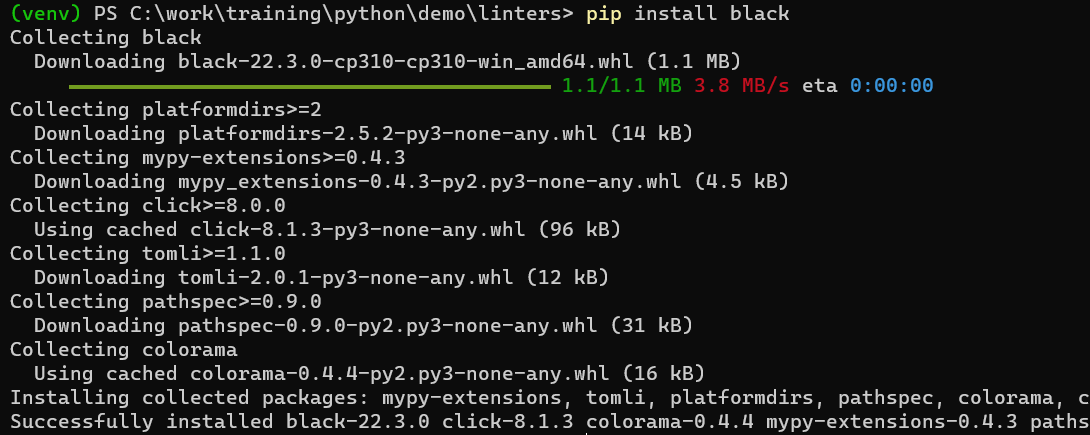


As shown below, flake8 prints out all PEP8 issues that it found with errors (E) and warnings (W):



**APPENDIX C**: *Integrating black in your Python projects*

Black is available in PyPi, we can simply use the pip command to get it into local (using virtual environment is recommended).



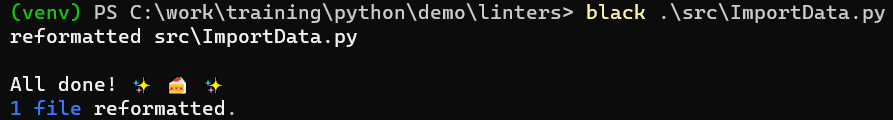
Then, using black to reformat the code is so easy:

*black /path/to/target/directory*

or

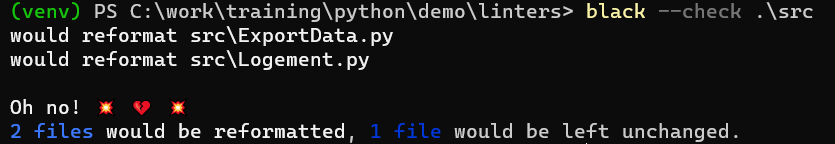
*black /path/to/target/source/file.py*

We take the same example as in Appendix B. Let’s use black to reformat the *ImportData.py* file:

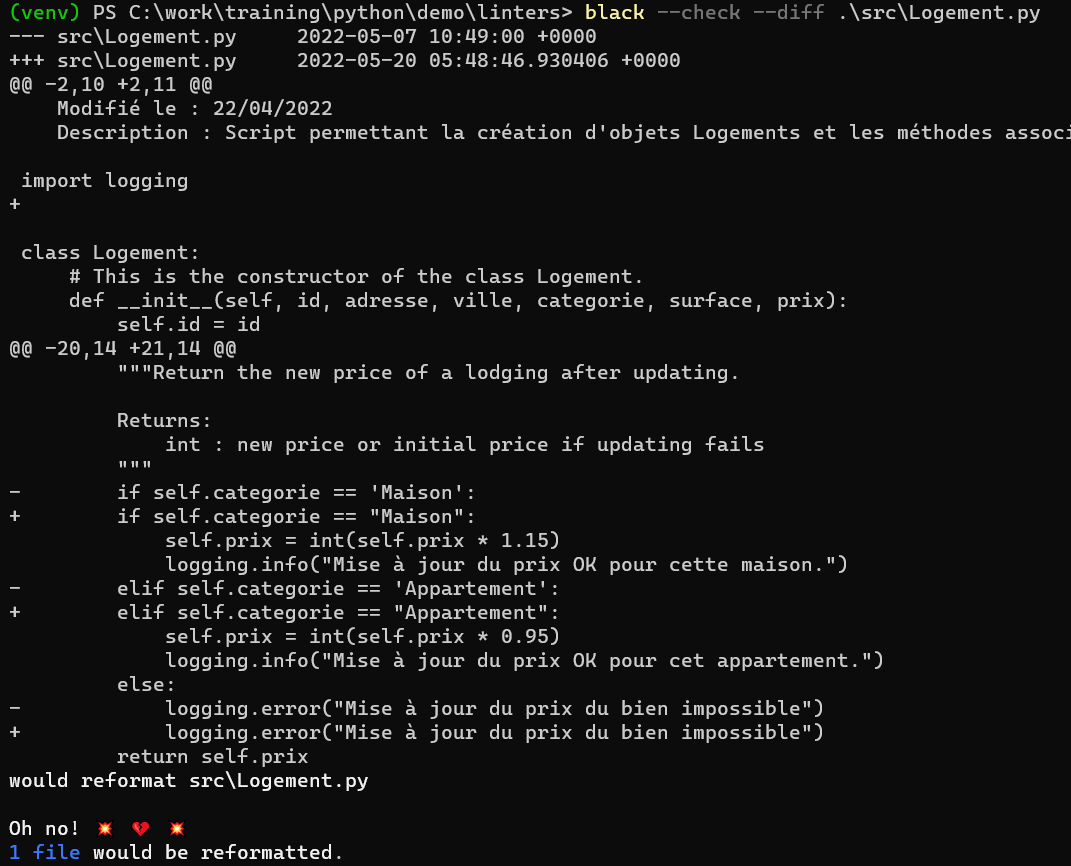


The target file was modified immediately (uncompromising) without any information to be displayed.

If we want to check the target file/folder before doing any modification, just add *--check* option:



Finally, if we want to have more detail about the modification before reformatting, use *--check --diff*:

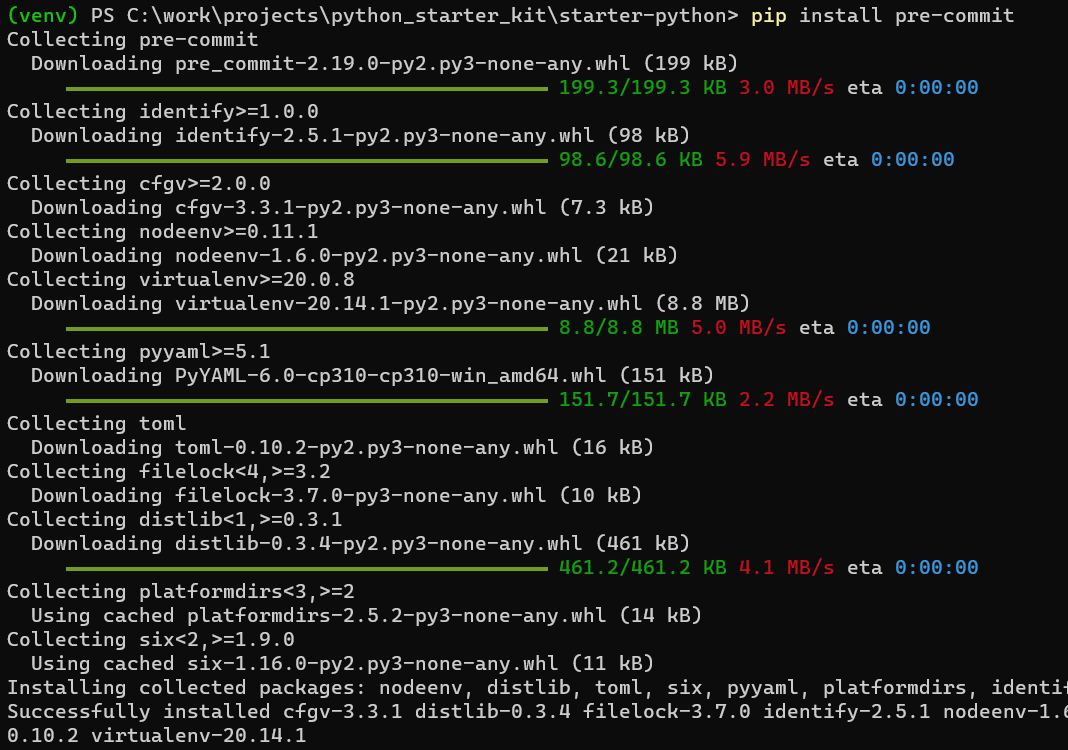


\* It’s worth noting that black limits the maximum line length to 88 (instead of 79 with flake8) characters, but we can change this value by using *-l* or *--line-length* option:

*black -l 79 src\_file.py*

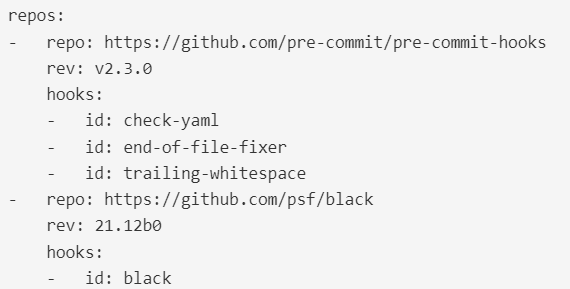
**APPENDIX D**: *Integrating pre-commit in your Python projects [14]*

Pre-commit is available in PyPi, we can simply use the pip command to get it into local.



Next, navigate to the local git repository of our project to add a pre-commit configuration by creating a file named *.pre-commit-config.yaml*:

We can generate a very basic configuration with *pre-commit sample-config*, but let’s try with the following example options. In this case, we utilize *pre-commit-hooks* and *black*:



Then use *pre-commit install* to setup the git hook scripts:



* Now pre-commit will run automatically on every *git commit* command!

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*