- Pipetex -

Coding Styleguide

Author
Max Weise
Contact Email: `maxfencing@web.de`

August 25, 2022

Contents

1 Introduction

This document contains a very basic summary and guideline to what is considered „good practice“ or „good codingstyle“ for this project. This is done for two reasons:

- Future me working on the code surely has forgotten what present me consideres good code and this is a way of reminding future me.
- In the case of multiple contributers I want to help in keeping the code as uniform as possible.

Following, I will give a brief overview of what tools I use and how I define good quality code. The list will surely be incomplete and I'm not entirely sure if I will ever be able to complete it. But additions will be made if needed and requested.

2 Styleguide

2.1 General Guides

As I'm sure the list following in section **??** will be incomplete for some time to come, as some situations won't occur or are not foreseen at the time of writing. As it would be too much effort to address every edge case I present the following compromise:

Most situations handling codestyle problems are handled in section 2.3. If for some reason a situation is not handled there, please refer to the Google Python Style Guide. In the rare case that something is not handled *there*, please refer to the official PEP8 Style Guide written by Guido van Rossum. Please note, that neither the style guide provided by Google, not the PEP8 Guide overrule the decisions made in this style guide.

2.2 Tools

Following will list the tools used to help in keeping the code clean and uniform:

2.2.1 Linter

Currently, the **flake8** linter is used. The repository also provides a flake8rc which configures the linter according to the requirements. To run the linter, simply run it on the source directory using `$ flake8 src/` . I also recommend integrating it into your text editor or IDE to run it automatically on the edited file or on the press of a button.

2.2.2 Typechecking

As typehints are greatly encouraged in the project, I recomend to use **mypy** when modifying the code. In this case, no configuration is needed and you can simply run `$ mypy src/` on the source directory to check for any type incompatibilities. This can also be integrated into your text editor or IDE.

2.3 Style Decisions

Now, on to how the code actually should look.

2.3.1 Project Layout

The rogh layout of the project looks like this:

Ideally, only new modules have to be introduced to the codebase and no new packages.

```
/
├── docs
├── src
│   └── pipetex
│       ├── __init__.py
│       ├── main.py
│       ├── pipeline.py
│       └── various_utils.py ..................... This is just an example module
└── tests
```

Figure 1: Directory Structure

2.3.2 Modules

Modules should be named with descriptive, snake_case names. In the best case scenario, one word modules should be enough, but multiple words are possible as a module name.

Every module should start with a docstring which is used to generate documentation. The docstrings should include *what* belongs inside the module, why is it needed and (if possible) a common use case description. Please also include the name of the author and the date of creation. For more information on how to write docstrings (not only for modules) please refer to section 2.3.4. A module docstring may look something like this:

Listing 1: Example Module Docstring

```python
""" Operations and utility functions used in the pipeline class

The functions described in this module define a single operation which will be
executed by the pipeline object. All public functions must adhere to the same
signature so the pipeline can dynamically execute them one by one.

@author: Max Weise
created: 23.07.2022
"""

from pipetex import exceptions
from pipetex.enums import SeverityLevels, ConfigDictKeys
```

Modules may contain multiple classes, especially if they are small in lines of code and have some semantic or logical connection to each other (e.g: enumerations can be located in the same module). As classes are getting bigger or need to be more isolated for testing purposes they should be moved to a separate file.

2.3.3 Classes

Classes are a great feature for combining data and behavior, implementing design patterns and encapsulating similar things together. Python offers the typical behavior of a object oriented programming language and it is usually a good idea to harvest the benefits of this. But luckily, we are not using Java (no offense) and are not *forced* to rely on classes. So before implementing, take a moment to reflect on the planned implementation and determine if a class is the best option for the task or if it can be realized by for example a simple function.

I usually use 3 questions when determining if I need a class or not.

- Do I need to store data in the class?
- Do I need to reuse the class?
- Do I need the class for a type definition or inheritance hierarchy?

If you answer 'yes' to at least two of these questions then its probably best to use a class. Otherwise consider using functions, especially if you don't need to store data in an object.

Classes should be named in the PascalCase style, numbers and symbols like underscores or dashes are discouraged. Each class must contain a docstring below the class definition describing the class and listing any public attributes.

2.3.4 Docstrings