- Pipetex -

Coding Styleguide

Author
Max Weise
Contact Email: maxfencing@web.de

August 31, 2022

Contents

Listings

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*[1] . 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.

[1] `https://google.github.io/styleguide/pyguide.html`

2.3 Style Decisions

Now, on to how the code actually should look.

2.3.1 Project Layout

The rough layout of the project looks like this:

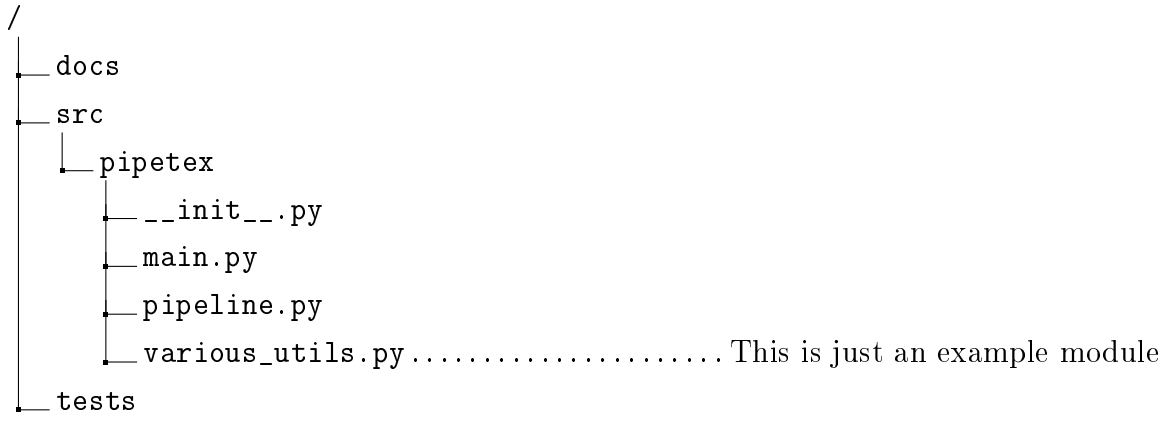

Figure 1: Directory Structure

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

2.3.2 Language and Style Rules

This section defines general rules to follow when contributing code to the project.

Imports Imports are only used for packages and modules. This is to avoid conflicting names inside the project and to keep track of the origin of functions and classes used in the code. Allways import the full package name and refrain from using relative imports.

Some additional aspects to consider:

- Use `from x import y` for classes and functions inside the package where `x` is the module and `y` is the class or function. Do not import methods from classes!
- Use `from x import y as z` if the imported name is inconveniently long or the. abbreviation (`z`) is commonly used (e.g.: `import numpy as np`).
- Members of the `typing` package do not need to used and inported with the full `package.module.class` syntax, as this would clutter the code more than it would help the readability.

Import statements are allways located at the top of the file after the module docstring. Imports get sorted after the following scheme:

1. Modules which are defined in the project.
2. Imports of the form `from x import y` which match the criteria listed above (mostly imports from typing or collections.abc)
3. Other modules and librarys used in the project. These get sorted alphabetically.

Line Length The maximum line length allowed is 80 characters. If needed, make use of implicit linejoining. The only exception for the 80 character limit is a long url which is placed inside a comment. If the linter flags this, add an igore comment.

Indentaion Use 4 spaces to indent your code. Never use tabs or mix tabs with spaces. Either align wrapped elements vertically or use a hanging indent of 4 spaces. In case of long method signatures use the vertical align method. When it would be necessary to align three or more lines vertically make each parameter of the function a seperate line and use a hanging indent.

Example:

Listing 1: Indentaion Example

```
# Good Example
def long_function_name(param1, param2,
                       param2, flag=False) -> bool:

def _long_method_name(
    self,
    param1,
    param2,
    param3
) -> str


# Bad Example
def long_function_name(param1, param2,
                       param2,
                       flag=False) -> bool:
```

Blank Lines and Whitespace Top level module members (Classes, functions) are separated by 2 blank lines. Methods inside classes are separated by 1 blank line. Keep the usage of blank line consistent with the rest of the code. The linters usually flag the incorrect usage of blank lines.

Follow standart typographic rules when using whitespace. When using arithmetic operations or comparisions, use whitespace: `if x == 5: print(x ** 2)`. Please refer to the *Google Python Style Guide*[2] for using whitespace, as this is a complete summary of the rules used in this project.

2.3.3 Naming

Naming conventions are described in this section. Following, functions and methods are referred to as functions.

[2] `https://google.github.io/styleguide/pyguide.html`

Modules *module_name.py*

Module names follow the snake_case convention. Uppercase letters, numbers and symbols other than the underscore are strongly discouraged.

Classes *ClassName.py*

Class names follow the PascalCase convention. Only upper- and lowercase letters are allowed in the class name. Numbers technically work but are discouraged.

functions *function_name*

Functions follow the snake_case convention. To signify a function is internal (private or protected) add a single underscore (_) at the beginning of the function name. Double underscores (so called „dunder“, __) are a way of making it private to the class, however it impacts readability and testability without making the function „true private“. Prefer single underscores and use a linter, which will flag private member access[3].

Unittests should follow the function naming convention. Each test should start with „test_“[4] followed by the name of the function that is tested. If an error condition is tested, a short string in camelCase describing the error should follow the function name. See an example below:

Listing 2: Example Test Function

```
def test_compile_latex_file(simple_testfile, config_dict, mocker):
    """Tests the compilation of latex files."""
```

Listing 3: Example Test Function Testing Error

```
def test_compile_latex_file_notVerbose(simple_testfile, config_dict,
    mocker):
    """Tests the compilation of latex files."""
```

Variables *variable_name*

Variables names follow the snake_case conventions. To signify a class member is internal add a single underscore to the beginning of the name. In case of a constant, the name should be written in ALL_CAPS.

[3] Some code completion engines also support the single underscore notation, as members with a leading underscore won't be shown in the code completion preview

[4] I remember some frameworks force this convention, some do not. As it is technically possible to use different frameworks in the same project I decided to enforce this naming style.

Names to avoid Please avoid single character names like $a = 4$ or $b = function()$. Inside for loops it is common practice to use counting variables, for example i, j, k etc. This is accepted as it is commonly known. Dashes (-) should not be used in module-, class-, function- or variablenames. Please also avoid using slurs, political statements / abbreviations or any other offensive language.

2.3.4 Docstrings

Docstrings should be added at the top of modules, after the class name and after a function declaration. In general, all docstrngs follow the same scheme. Use three double quotes (" $\mapsto$ """) to start and end a docstring. The first line of every docstring should contain a brief summary of the module / class / function it documents followed by a blank line. If the author wants to write more (which is encouraged) this can be done after the blank line. The docstring should not exceed the 80 character line.

This is the basic way to write a docstring. What follows next depends on what the docstring documents.

Module Docstrings Module docstrings contain the author and date of creation of the module. The contents of the docstring should describe what belongs into the module and how to use it.

Example docstring:

Listing 4: Example Module Docstring

```
""" 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
"""
```

Class Docstrings Class docstrings should contain a list of public attributes after the summary. If you think its necessary, include a small example on how the class would commonly be used. After this the reader should find a list of public attributes. The docstring should be indented one level deeper than the classname.

Example docstring

Listing 5: Example Class Docstring

```
class Pipeline:
    """Representation of a pipeline object. Runs different tasks on the file.

    The pipeline keeps a reference of which operations should be run in which
    order. Also the pipeline will comunicate errors to the user by using
    logging stements.

    Common Usage:
        p = Pipeline(file_name, True, False, False, False)
        p.execute(p.file_name)

    Attributes:
        file_name: Name of the file which should be processed.
        config_dict: Contains metadata which should be shared
            with the operations.
        oder_of_operations: List of operations which will be run on the file.
    """
```

Function Docstrings [5] Function docstrings should describe all side effects that can occur during the execution (logging is not considered a side effect). After the summary, the docstring should include:

Args: A short list describing each argument passed to the function. What is it, how is it used, why is it there. If the argument is a default argument, describe what it default value is. `self` and `cls` variables do not need to be documented. If the function receives no arguments, this field is omitted.

Returns: A short description of the return value of the function. In most cases, this is the same for all functions, as monadic error handling is used in the project. If the return value is a variable, use the variable name to list it. Else, use its type. If the function does not return anything, this field is omitted.

[5] This also applies to methods. Function and method are used interchangeably.

Raises: A short description of which type of error the function raises and when. If the error is a `InternalException`, document which `severity_level` is / are used. If no exceptions are raised, this field is omitted.

Example docstring:

Listing 6: Example Function Docstring

```
def copy_latex_file(file_name: str, config_dict: dict[str, Any]) ->
    Monad:
    """Create a working copy of the specified latex file.

    To avoid losing data or corrupting the latex file a copy is created. The
    copy can be identified by a prefix added in this function. The new file
    name is written in the config dict. It is the responsibility of the
    dict owner to update the name of the working file accordingly.

    Args:
        file_name: The name of the file to be compiled. Does not contain any
            file extension.
        config_dict: Dictionary containing further settings to run the engine.

    Returns:
        Monad: Tuple which contains a boolean to indicate success of the
            function. If its true, the second value will be None. If its
            false, the second value will contain an InternalException object
            containing further information.

    Raises:
        InternalException: Indicates an internal error and is used to
            comunicate exceptions and how to handle them back to the calling
            interface.
            [Please see class definition]
        Raised Levels: CRITICAL
    """
```