FLAP Styleguide

written by Gabor Cseh
last modified: 16th January 2019

The purpose of this document is to give an overview of the coding style requirements for developing the main components of the Fusion Library of Analysis Programs (FLAP) package. (It is not a requirement for developing your own user programs, however, it is still suggested to use this coding style, even when just using FLAP. Or always.).

The coding style in FLAP package follows the PEP 8 style guide (see examples later), so if you are already know it's rules, you are good to go. Otherwise here are some useful tips about installing and using a style- and code-checking extension (flake8) in Visual Studio Code and Spyder.

Extensions/packages for style checking

Before getting to the actual code style guide, here is the list of packages/extensions, one can use to automatically check the style of one's coding.

Python extension for Visual Studio Code

Visual Studio Code is an open source, cross-platform code editor, which is widely used in many environments, e.g. it is available from the Anaconda Python distribution.

To install flake8 code analyzer, you have to install first the Python extension for VSCode:

1. Go to the Extension sidebar (or press Ctrl+Shift+X).
2. Type Python into the "Search Extension in Marketplace" search field.
3. Click on the green Install button.
4. That's it!

After installing the Python extension, a text will appear on the bottom sidebar of the VSCode window, which says Python a version number and 32/64 bit in the form of "Python 3.7.2 64-bit". By clicking on this text, you can select the Python version, you want to use for code analysis, and running your Python programs. This list contains (in principle) all the available Python installations on your machine, not just the ones, which are in the PATH variable.

To have your code checked, you need to install a linting module to your Python installation. The recommended module for this is flake8 (as mentioned before), which not only checks your code, but also analyze your programs and notices you about uninitialized/unused variables etc.

To install flake8, you just have to set VSCode settings accordingly. This means, that go to File –> Preferences –> Settings (Ctrl+,). Then type "python." in the search bar, and click on the "edit in settings.json" option (it appears multiple times, it does not matter, which one you click on). Then you have to insert the following lines or - if they are present - check, that the settings are the same as below.

```
{
    "python.linting.pylintEnabled": false,
    "python.linting.flake8Enabled": true,
    "python.linting.enabled": true,
    "python.linting.flake8Args": [
        "--max-line-length=120"
    ],
}
```

If you have no flake8 installed with your Python distribution, an automatic message will come up, offering the possibility to install this module automatically. You can choose either this to install flake8 (in this case, an embedded command line will come up inside the VSCode window and install the extension), or you can install it manually by using the

```
> python -m pip install flake8
```

In this manual case, take care that you use the python version you mean, and not another installation.

After installing flake8, the style and code analysis messages will appear in a dedicated block, if you click the error/warning sign icons in the navigation bar at the bottom of the screen.

Python with Spyder

Since Spyder is optimized solely around the Python language, Spyder is coming pylint preinstalled, so to have a good linter, you just have to switch it on. You have to check on the:

```
Tools --> Preferences --> Editor --> Code Instrospection/Analysis -->
Real-time code style analysis
```

checkbox, and you are good to go. After checking this checkmark and applying the settings, the warning/error messages will appear next to the line numbering with a yellow warning sign. Hovering the mouse over these warning signs will give you the exact message.

Style guide - a short summary

Since we follow the guidelines articulated in Python Enchancment Proposal 8 (PEP 8), if something is not clear and/or not written in this document, this is a good web page to start. Otherwise I try to give a short, but comprehensive summary about the ideas described there.

1. Use 4 spaces for indentation - no tabs!! (It can be easily set in modern code editors, even VI(m) has this possibility. It is called either “indent using spaces” or “soft tab”.)

2. No trailing spaces at the end of the line.

3. Line continuation:

   ```python
   # Aligned with opening delimiter.
   foo = long_function_name(var_one, var_two,
                            var_three, var_four)

   # Add 4 spaces (an extra level of indentation) to distinguish
   # arguments from the rest.
   def long_function_name(
           var_one, var_two, var_three,
           var_four):
       print(var_one)

   # The closing brace/bracket/parenthesis on multiline constructs
   # line up under the first non-whitespace character of the last
   # line of list (or the first character of the line)
   my_list = [
       1, 2, 3,
       4, 5, 6,
       ]

   result = some_function_that_takes_arguments(
       'a', 'b', 'c',
       'd', 'e', 'f',
       )
   ```

4. Line length: **79 characters** for code, **72 characters** for docstrings/comments. The original argument for this decision is:

   > Limiting the required editor window width makes it possible to have several files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

   However, if this is not the case at your coding style (means you use one), it is allowed (as a local rule) to use **120 characters** for code and/or docstrings/comments.

5. Wrapping long lines:

   > The preferred way of wrapping long lines is by using Python’s implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation. Backslashes may still be appropriate at times. For example, long, multiple with-statements cannot use implicit continuation, so backslashes are acceptable:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

6. Line breaks with binary operators (short: operator should be *before* the operand):

   ```
   # Yes: easy to match operators with operands
   income = (gross_wages
             + taxable_interest
             + (dividends - qualified_dividends)
             - ira_deduction
             - student_loan_interest)
   ```

7. Blank lines.

   - Surround ***top-level functions*** and ***class definitions*** with ***two*** blank lines.
   - ***Method definitions*** inside a class are surrounded by a ***single*** blank line.
   - Use blank lines in functions, sparingly, to indicate logical sections.

8. Source file encoding: UTF-8. Set this every time. However, you should use English words and ASCII *characters* in those files. Except explicit test cases for non-ASCII characters and author names.

9. Naming conventions

   - Names to avoid: 'l' (lowercase letter L), 'o' (lowercase letter 'oh'), 'I' (uppercase letter 'eye') as single-letter variable names. Reason: confusing.
   - All the names should be ASCII compatibility (however, the character-encoding is UTF-8).
   - ***Module*** names should be ***short, lowercase*** letters (like flap).
   - ***Class names: CamelCase.***
   - ***Function names: lowercase***, words separated by ***underscores.***
   - Always use self for the first argument to instance methods.
   - Always use cls for the first argument to class methods.
   - At name collision (e.g. with a reserved keyword) use a trailing underscore. E.g. class_.

Logging

Instead of using print messages and verbose keywords etc. throughout the whole code, it it strongly advised to use Python's built-in logging system. It is capable of save the log messages in a stream, on the console or in a file (actually, the latter two are also kinds of streams) based on predefined criterions, e.g. severity. The logging system is easy-to-use and easy-to-config. Some examples are below.

- A logging.conf file for the logger setup.

  ```
  [loggers]
  keys=root,flapLogger

  [handlers]
  keys=consoleHandler,fileHandler

  [formatters]
  keys=fileFormatter,consoleFormatter

  [logger_root]
  level=DEBUG
  handlers=consoleHandler

  [logger_edvisLogger]
  level=DEBUG
  handlers=consoleHandler,fileHandler
  qualname=edvisLogger
  propagate=0

  [handler_fileHandler]
  class=logging.handlers.RotatingFileHandler
  ```

```
level=DEBUG
formatter=fileFormatter
args=('./flap.log', 'a', 5*1024*1024, 1)

[handler_consoleHandler]
class=StreamHandler
level=INFO
formatter=consoleFormatter
args=(sys.stdout,)

[formatter_fileFormatter]
format=%(asctime)s - %(name)s - %(filename)s - line: %(lineno)s -
%(levelname)s - %(message)s
datefmt=

[formatter_consoleFormatter]
format=%(asctime)s - %(levelname)s - %(message)s
datefmt=
```

- Usage of the logger after setting up a logging.conf file.

```python
import logging
import logging.config

# Loading the logger config file
logging.config.fileConfig('logging.conf')

# Create logger
logger = logging.getLogger('flapLogger')

logger.info("The FLAP logger facility has been initialized.")

# Doing some everyday work (only need info about that if we are in verbose mode)
print("Pam pam param...")
logger.debug("Here is some verbose, debug level logging message
              about 'Pam pam param...'")
```

Documentation strings

The Sphinx Python Documentation Generation system uses reStructuredText (RST) (for quick editing purposes, use this cheat sheet) to generate documentation for various formats include HTML, PDF, DOCX and various ebook formats.

It is capable of creating a whole documentation with separate documents (e.g. tutorials, detailed explanations, intentions etc.), but from the user point of view, the most important parts are the so-called docstrings. These are comment-like sections in the source code, where the user can on-the-fly describe the purpose and usage of the given part of the code. There are a few examples below.

A full code example (grabbed from the follwoing link) is below:

```python
"""
.. module:: useful_1
   :platform: Unix, Windows
   :synopsis: A useful module indeed.

.. moduleauthor:: Albert Example <albert@invalid.com>

"""


def public_fn_with_sphinxy_docstring(name, state=None):
    """This function does something.
    :param name: The name to use.
    :type name: str.
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError

    """
    return 0


class MyPublicClass(object):
    """We use this as a public class example class.

    You never call this class before calling :func:`public_fn_with_sphinxy_docstring`.

    .. note::

       An example of intersphinx is this: you **cannot** use :mod:`pickle` on this class.

    """

    def __init__(self, foo, bar='baz'):
        """A really simple class.

        :param foo: We all know what foo does.
        :type foo: str.
        :param bar: Really, same as foo.
        :type bar: str.

        """
        self._foo = foo
        self._bar = bar
```