



tianocore

Python Development Process and Coding Standards Specification

TABLE OF CONTENTS

EDK II Python Development Process and Coding Standards Specification

1 Introduction:

2 Python coding guidelines and tools:

3 Development steps and flowchart:

4 Environment setup and example

4.1 Install and setup environment

4.2 Create a project configuration file for flake8:

4.3 Write python code :

4.4 Run flake8:

4.5 Run mypy for type hints:

4.6 Write a unit test using pytest :

4.7 Generate documents using pydoc:



EDK II Python Development Process Specification

DRAFT FOR REVIEW

09/27/2020 03:15:06

Acknowledgements

Redistribution and use in source (original document form) and 'compiled' forms (converted to PDF, epub, HTML and other formats) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (original document form) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, epub, HTML and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS DOCUMENTATION IS PROVIDED BY TIANOCORE PROJECT "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL TIANOCORE PROJECT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 2020, Intel Corporation. All rights reserved.

Revision History

Revision	Revision History	Date
0.10	Initial release.	Jan 2017
1.0	#2626 Typo in Python development Process and coding standards draft spec	March 2017

1 INTRODUCTION

This specification defines a set of python coding standards, development flow, and tools to help to identify and fix deviations in written code. These standards, flow and tools to establish

- Uniformity of style
- Uniform conventions
- To maintain consistency
- To maintain extensibility
- To improve readability
- To improve maintainability, reusability

These rules apply to all code developed in Python for inclusion in the EDK II , and are intended as an enabling philosophy. All changes or additions from this point on shall conform to this specification. Pre-existing code does not need to be updated for the sole purpose of conforming to this specification. As conforming updates are made, the developer may update other content within the modified file to bring it within compliance with this specification. This specification addresses the chronic problem of providing accurate documentation of the code base by embedding the documentation within the code. A document generation system, using inbuilt python module, then produce formatted documentation by extracting information from specially formatted comment blocks and the syntactic elements of the code.

2 PYTHON CODING GUIDELINES AND TOOLS

This section covers python coding style guidelines followed.

PEP 8- Style guide for python code:

PEP8 covers python code style guide and helps to maintain consistency in code. A style guide is about consistency. Consistency within a project or module or function is most important. Complete specification available at <https://www.python.org/dev/peps/pep-0008/>

PEP 257- Docstring Conventions:

PEP257 covers semantics and conventions associated with python docstrings. The aim of PEP257 to standardize the high-level structure of docstrings. Complete specification available at <https://www.python.org/dev/peps/pep-0257/>

PEP 484- Type Hints:

PEP484 introduces a provisional module to provide the standard definitions and tools, along with some conventions for situations where annotations are not available. More details on PEP484 available at <https://www.python.org/dev/peps/pep-0484/>

Flake8:

Flake8 is a Python library wrapper around PyFlakes, pycodestyle and Ned Batchelder's McCabe script

PyFlakes:

- A simple program that checks Python source files for errors.
- It is available on PyPI <https://pypi.org/project/pyflakes/>

Pydocstyle:

- Pycodestyle used to called pep8 is a tool to check your Python code against some of the style conventions in PEP8.
- It is available on PyPI <https://pypi.org/project/pycodestyle/>

McCabe:

- Ned's script to check for the McCabe complexity for Python code.
- It is available on PyPI <https://pypi.org/project/mccabe/>

pytest:

It is important to validate the classes, methods, and functions we write. This will help to initiate the core software functionality of the modules. This is possible by writing a unit test which is the first level of software functionality validation. The pytest framework helps to write small test, yet scales to support complex functional testing for applications and libraries.

Complete features and documentation available at: <https://docs.pytest.org/en/latest/contents.html>

mypy:

The `mypy` tool aims to combine the benefits of dynamic and static typing. `mypy` combines a powerful type system and compile-time type checking. It is an optional static type checker for python. `Mypy` type checks standard Python programs; run them using any Python VM with basically no runtime overhead. More details available at <https://mypy.readthedocs.io/en/stable/>

3 DEVELOPMENT STEPS AND FLOWCHART

- Write code which follows PEP8, PEP257 and PEP484 recommendations
- Run `flake8` on source code.
- Run `mypy` for type hints checking of the code
- Write unit tests using `pytest`
- Run all unit tests.
- Send review
- Generate documents using `pydoc`
- Write/modify/generate specifications wherever applicable

Following flow chart explains complete development lifecycle.

The plugin you are using seems to generated a bad URL.
This URL does not look like DEFLATE data.
It looks like your plugin is using HUFFMAN encoding.

This means you have now to add an header ~1 to your data. For example, you have to change:
`http://www.plantuml.com/plantuml/png/UDfjaDqAWa0GXU_tAUPImUv3eOJeBY...`
to
`http://www.plantuml.com/plantuml/png/~1UDfjaDqAWa0GXU_tAUPImUv3eOJeBY...`

It will work this way

You may contact the PlantUML team at plantuml@gmail.com

But you should also probably contact the plugin authors you are currently using and send them this image

For the record, here is your data:

UDfjaDqAWa0GXU_tAUPImUv3eOJeBYYo7c3JgPRsHtPc4zy-5PL0vsOECvmv7oDoOkav62q68JpMAES6
C0k0j731bmYbgp2RLz2ybKQ8hi72KyiFRXCmRTtkOr04qT5xvoaU3cns5qcK2R5q1giO68bfzC5VJJSc
VpfC3nr6n_1Ls2JYRBMo29KhWq7Bwo4ZNa-Nz86tFYyHiAYJUwbIwWw40Yw85ZX5U1Z5dRZNbMI45HXY
LuiSRTL_-WUzZdBG

4 ENVIRONMENT SETUP AND EXAMPLE

4.1 Install and setup environment

- **Install flake8:**

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

- **Install pytest:**

```
python<version> -m pip install -u pytest
```

- **Install mypy:**

```
python<version> -m pip install mypy
```

4.2 Create a project configuration file for flake8

Create a file at root level of the project directory and name it as **".flake8"**.

Flake8 configuration options needs to be in the flake8 section. The following options used for EDK II flake8 configuration.

```
[flake8]
# H903 Windows style line endings not allowed in code
# E266 too many leading '#' for block comment
# D203 : One blank line required before class docstring
# H306 : imports not in alphabetical order
ignore = H903, E266, D203, H306
exclude = .git,
max-complexity = 10
max_line_length = 120
```

4.3 Write python code

Create new file and name it as `sample.py` and start writing python code. Please note `sample.py` and `falke8` configuration files stored on same directory level.

Source code `sample.py` :

```
# Source code sample.py:

import os

class AddTen:
    """Class for add ten to a given number"""

    def __init__(self, user_input):
        self.user_input = user_input
        self.new_variable = 10
        d = {}

    def add_ten(self):
        """Init for calss."""
        try:
            return self.newvariable + self.user_input
        except:
            print("Unknown Error")
            return None
```

4.4 Run flake8

Run `flake8`. The output of `flake8` on `sample.py` shown below

```
C:\kpurma\PythonDevelopmentProcess>python -m flake8 sample.py
sample.py:1:1: D100 Missing docstring in public module
sample.py:30:1: F401 'os' imported but unused
sample.py:31:1: W293 blank line contains whitespace
sample.py:32:1: E302 expected 2 blank lines, found 1
sample.py:33:1: D400 First line should end with a period
sample.py:35:1: D107 Missing docstring in __init__
sample.py:36:13: E117 over-indented
sample.py:38:13: F841 local variable 'd' is assigned to but never used
sample.py:41:13: E117 over-indented
sample.py:43:21: E117 over-indented
sample.py:44:13: E722 do not use bare 'except'
sample.py:45:21: E117 over-indented
sample.py:48:1: E305 expected 2 blank lines after class or
                function definition, found 1
```

Fix `flake8` issues and run `flake8` again to check there is no errors reported.

Source code `sample_fixed.py` :

```
#Source code sample.fixed.py:

"""Sample file with flake8 errors fixed."""

class AddTen:
    """Class for add ten to a given number."""

    def __init__(self, user_input: int = 0):
        """Initialization."""
        self.user_input = user_input
        self.new_variable = 10

    def add_ten(self)->int:
        """Method to add ten to given number."""
        try:
            return self.new_variable + self.user_input
        except Exception as e:
            raise e
```

Run flake on fixed code.

```
C:\kpurma\PythonDevelopmentProcess>python -m flake8 sample_fixed.py

C:\kpurma\PythonDevelopmentProcess>
```

4.5 Run `mypy` for type hints

Run `mypy` on the source file to check to find type hints.

```
C:\kpurma\PythonDevelopmentProcess>mypy sample.py --strict
sample.py:35: error: Function is missing a type annotation
sample.py:40: error: Function is missing a return type annotation
sample.py:43: error: "AddTen" has no attribute "newvariable";
               maybe "new_variable"?
sample.py:48: error: Call to untyped function "AddTen" in typed context
Found 4 errors in 1 file (checked 1 source file)
```

`Mypy` output for fixed code:

```
C:\kpurma\PythonDevelopmentProcess>mypy sample_fixed.py --strict
Success: no issues found in 1 source file
```

4.6 Write a unit test using pytest

Use `pytest` library to write a unit test. Unit test for sample program is shown below for sample code.

```
from sample_fixed import AddTen

def test_answer():
    sum10 = AddTen(20)
    assert sum10.add_ten() == 30
```

Run the unit test and make sure all tests pass.

```
C:\kpurma\PythonDevelopmentProcess>python -m pytest test_sample.py
=====testsessionstarts=====
platform win32 -- Python 3.8.0, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: C:\kpurma\PythonDevelopmentProcess
collected 1 item

test_sample.py .                                     [100%]

=====1passedin 0.06s=====
```

4.7 Generate documents using pydoc

By using the `pydoc` module, documentation is generated in the desired format.

Following command generates html version of document at source level directory.

```
python -m pydoc -w sample_fixed
```

sample_fixed

sample_fixed.py.

Classes

builtins.object

AddTen

```
class AddTen(builtins.object)
```

```
AddTen(user_input: int = 0)
```

Class for add ten to a given number.

Methods defined here:

```
__init__(self, user_input: int = 0)
```

Initialization.

add_ten(self) -> int

```
Init for calss.
```

Data descriptors defined here:

dict

dictionary for instance variables (if defined)

weakref

```
list of weak references to the object (if defined)
```

Data

```
a = <sample_fixed.AddTen object>
```

c = 20