Python Best Practice

**TABLE OF CONTENTS**

[Unit Testing 1](#_Toc2096604679)

[Unit Testing Framework 1](#_Toc1514162301)

[Coverage Tests 1](#_Toc1320020702)

[Styling 2](#_Toc506192930)

[Styling Tooling 2](#_Toc2081078370)

[Styling: Python Variable Naming 2](#_Toc724930925)

[Styling: Python Class Naming 2](#_Toc523773318)

[Styling: Python Variable Naming 2](#_Toc319044767)

[Styling: Python Function Naming 3](#_Toc1915860350)

[Styling: Python Naming Side Rules 3](#_Toc14786785)

[AWS SAM Code Conventions 4](#_Toc1879031802)

[Styling further reading 5](#_Toc1561535263)

[Code Documentation 5](#_Toc1638373774)

[Documentation further reading 6](#_Toc1948579096)

[Versioning 6](#_Toc8233389)

[Pip Dependency Versioning 6](#_Toc1309435036)

[Software Versioning 6](#_Toc1606871860)

[Semantic versioning Cliffnotes 6](#_Toc1596310480)

[Python Version 7](#_Toc1523701461)

[Multi Python Environment Help 7](#_Toc1407724862)

[Further Reading 7](#_Toc261877473)

# Unit Testing

## Unit Testing Framework

* You **SHOULD** use pytest to test python project
* You **SHOULD** prefix all test files test\_\*.py where \* is the file under test

<https://docs.pytest.org/en/stable/>

## Coverage Tests

* use pytest-cov (or favoured tool) to generate coverage
* you **SHOULD** aim for 80% coverage
* all new code **MUST** have a test\_x.py file and tests

# Styling

## Styling Tooling

* **SHOULD** use tooling to detect python styling rules (pylint or other tool say flake8)
* **SHOULD** use tooling to enforce python styling rules (black? tbd)
* <insert formatter of choice, maybe black ?>

## Styling: Python Variable Naming

Follow PEP-008 standard naming conventions with the follow additions

### Styling: Python Class Naming

|  |  |
| --- | --- |
| **Class** |  |
| Rules | \* classes **MUST** follow *CapitalizedWords* (also know as *CamelCase*)  \* classes **SHOULD** be Nouns  \* **SHOULD** try to keep your class names simple and descriptive.  \* **SHOULD** use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML) |
| Good Examples | InvalidVariant**Header**  **FixtureRequest** |
| Poor Examples | some\_class  SomethingToGet |

### Styling: Python Variable Naming

|  |  |
| --- | --- |
| **Variables** |  |
| Rules | \* variables **MUST** follow lower\_case\_with\_underscores (also known as *snake\_case*)  \* variables **SHOULD** start with a Noun  \* Variable names **SHOULD** be short yet meaningful.  \* The choice of a variable name **SHOULD**be mnemonic- that is, designed to indicate to the casual observer the intent of its use.  \* One-character variable names **SHOULD** be avoided except for temporary "throwaway" variables. |
| Good Examples | response\_header  commit\_status\_url |
| Poor Examples | run\_thing  header\_statusCode  getStatus |

### Styling: Python Function Naming

|  |  |
| --- | --- |
| **Functions** |  |
| Rules | \* variables **MUST** follow lower\_case\_with\_underscores (also known as *snake\_case*)  \* variables **SHOULD** start with Verbs |
| Good Examples | get\_response\_header  run\_job |
| Poor Examples | class\_header  runTheJob  Get\_Status |

### Styling: Python Naming Side Rules

|  |  |
| --- | --- |
| **Single character variable names** |  |
| Rules | \* One-character variable names **SHOULD** be avoided except for temporary "throwaway" variables.  \* **MUST** not use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.  In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead. |
| Good Examples | c:int = 0 # c for temp loop counter :) |
| Poor Examples | I:int = 0 # see how similar to 1 and l  l:int = 0 # see how similar to 1, L is better  O:int = 0 # see how similar to 0 and o |

|  |  |
| --- | --- |
| **Private / Protected Class Attributes** |  |
| Rules | Private variables / function should be prefixed "\_" |
| Good Examples | \_response\_header  \_get\_response\_header |
| Poor Examples | hidden\_variable  \_\_private\_function\_\_ |

|  |  |
| --- | --- |
| **list and dict naming** |  |
| Rules | list and dict variables **MAY** include a plural noun  this can help different sequence/maps stdTypes from singular stdTypes aka str/int etc |
| Good Examples | dogs: list = ["harry", "bonnie"] |
| Poor Examples | dog: list = ["harry", "bonnie"] |

|  |  |
| --- | --- |
| **boolean naming** |  |
| Rules | bool variables **MAY** include a possessive verb prefix aka "is\_" "has\_" (also "can" / "any" / "all" / "are") |
| Good Examples | is\_bool: bool = True  has\_variable\_been\_populated:bool = True |
| Poor Examples | get\_header\_already:bool = False  first:bool = True (this one debatably good) |

|  |  |
| --- | --- |
| **Exception / Error naming** |  |
| Rules | Exception **MUST** follow class rules  \* Exception name **SHOULD** end with Error |
| Good Examples | is\_bool: bool = True  has\_variable\_been\_populated:bool = True |
| Poor Examples | get\_header\_already:bool = False  first:bool = True (this one debatably good) |

### AWS SAM Code Conventions

Some excellent additional code conventions to consider following

<https://github.com/aws/serverless-application-model/blob/develop/DEVELOPMENT_GUIDE.md>

* Don't write any code in \_\_init\_\_.py file unless there is a really strong reason.
* ~~Module-level logger variable must be named as LOG\_~~ THIS ONE IS SAM SPECIFIC
* If your method wants to report a failure, it *must* raise a custom  
  exception. Built-in Python exceptions like TypeError, KeyError  
  are raised by Python interpreter and usually signify a bug in your  
  code. Your method must not explicitly raise these exceptions because  
  the caller has no way of knowing whether it came from a bug or not.  
  Custom exceptions convey are must better at conveying the intent and  
  can be handled appropriately by the caller. In HTTP lingo, custom  
  exceptions are equivalent to 4xx (user's fault) and built-in  
  exceptions are equivalent to 5xx (Service Fault)
* Don't use \*args or \*\*kwargs unless there is a really strong  
  reason to do so. You must explain the reason in great detail in  
  docstrings if you were to use them.
* Do not catch the broader Exception, unless you have a really  
  strong reason to do. You must explain the reason in great detail in  
  comments.

### Styling further reading

<https://www.python.org/dev/peps/pep-0008/>

<https://docs.python-guide.org/writing/style/>

## Code Documentation

* Add classes and function **SHOULD** be self documented with Docstring
* *From PEP-0287* [*https://www.python.org/dev/peps/pep-0008/#documentation-strings*](https://www.python.org/dev/peps/pep-0008/#documentation-strings)
* Extended reStructuredText Docstring **MAY** used to help self document parameters and other function metadata
* *See PEP-008* [*https://www.python.org/dev/peps/pep-0287/*](https://www.python.org/dev/peps/pep-0287/)
* all functions **SHOULD** where possible use type hinting
* *to help static analysis find simple type coding errors*
* Comments that contradict the code are worse than no comments. **MUST** make a priority of keeping the comments up-to-date when the code changes!
* *From PEP-008* [*https://www.python.org/dev/peps/pep-0008/#comments*](https://www.python.org/dev/peps/pep-0008/#comments)
* Ensure that your comments are clear and easily understandable full sentences, try to avoid cryptic phrases, or partial sentence fragments that require outside context
* # good example""""Create a new attribute on a class.:param str key: attribute name:param str value: value to set attribute to"""# poor example""""The thing that you know what it does, same as previous function, etc""""
* *From PEP-008* [*https://www.python.org/dev/peps/pep-0008/#comments*](https://www.python.org/dev/peps/pep-0008/#comments)
* All comment **SHOULD** be in English
* *so that the whole team can read comments*
* *From PEP-008* [*https://www.python.org/dev/peps/pep-0008/#comments*](https://www.python.org/dev/peps/pep-0008/#comments)
* *exception: for light humour purposes in TODO's etc, aka Latin's "# WARNING: hic sunt dracones", French's "# TODO: remember to defenestrate this code once XXX migrated"*
* **SHOULD** use inline comments sparingly. Avoid inline comments that are unnecessary and in fact distracting if they state the obvious.
* useful inline example
* x = x + 1 # Compensate for border
* non useful inline example to avoid
* x = x + 1 # Increment x
* *From PEP-008* [*https://www.python.org/dev/peps/pep-0008/#inline-comments*](https://www.python.org/dev/peps/pep-0008/#inline-comments)
* **SHOULD** use TODO/FIXME/WARNING/WORKAROUND inline comments to highlight temporary solutions, a short term workarounds, and other pit falls to incoming developers
* # TODO: need to support oauth2 in near future # FIXME: Code doesn't seems right, this seems to loop infinitely# WORKAROUND: ADO#234 fix for persistent auth bug
* *debatable if useful see* [*https://www.python.org/dev/peps/pep-0350/#objections*](https://www.python.org/dev/peps/pep-0350/#objections) *for lots of counter points*

### Documentation further reading

* docstrings spec <https://www.python.org/dev/peps/pep-0257/>
* styling comments spec <https://www.python.org/dev/peps/pep-0008/#comments>
* param / author annotations in restructured docstrings <https://www.python.org/dev/peps/pep-0287/>

## Versioning

### Pip Dependency Versioning

* All dependencies **MUST** be managed via pip
* All dependencies **SHOULD** be versioned to a minor or patch level
* *exception: unless dependency is not following Semantic versioning*
* *see more here* [*https://www.python.org/dev/peps/pep-0440/*](https://www.python.org/dev/peps/pep-0440/)
* *25/03/2021 NOTE: other package managers than pip exist, as product grows worth investigating pipenv or poetry as easier alts to pip*

### Software Versioning

* All software versioning **MUST** follow python's "Version Identification and Dependency Specification"
* *see more here* [*https://www.python.org/dev/peps/pep-0440/*](https://www.python.org/dev/peps/pep-0440/)
* All deployed software versioning **MUST** following semantic versioning standard
* *ps: semantic versioning follows sub set of the PEP-0440*

### Semantic versioning Cliffnotes

Given a version number MAJOR.MINOR.PATCH, increment the:

* MAJOR version when you make incompatible API changes,
* MINOR version when you add functionality in a backwards compatible manner, and
* PATCH version when you make backwards compatible bug fixes.

<https://semver.org/>

## Python Version

As of writing use python 3.6+ (**tbd**)

**SHOULD** ensure only running python versions with software support

<https://endoflife.date/python>

Features to be careful with

* <https://docs.python.org/3/whatsnew/3.9.html>
* <https://docs.python.org/3/whatsnew/3.8.html>
* <https://docs.python.org/3/whatsnew/3.7.html>
* <https://docs.python.org/3/whatsnew/3.6.html>
* <https://docs.python.org/3/whatsnew/3.5.html>
* <https://docs.python.org/3/whatsnew/3.4.html>

### Multi Python Environment Help

If multiple python environments are requires venv is a excellent tool that can be used to have isolated environments

<https://docs.python.org/3/tutorial/venv.html>

# Further Reading

* Org Sun Java variable naming standards all other languages follow
* <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>
* PEP-008 rules
* <https://www.python.org/dev/peps/pep-0008/#function-and-variable-names>
* <https://www.python.org/dev/peps/pep-0008/#function-and-method-arguments>
* <https://www.python.org/dev/peps/pep-0008/#method-names-and-instance-variables>