Python Best Practices (TA1)

Michael P. Gerlek

mgerlek@inferlink.com

v1.1 / 3 Feb 2024

This document provides guidance on coding conventions and software development best practices.

Guidelines like these have three (admittedly overlapping) goals: (1) to make it easier for others to understand your code; (2) to promote better design/architecture, improve long-term maintainability, and reduce bugs; and (3) to promote build/test automation.

We acknowledge that not all these guidelines will make sense for every context of every project. For example, it may be better to continue to follow the existing style in a legacy code base than to suddenly impose a new one. Use your best judgement and consult with your teammates before doing anything radical.

*This document aims for breadth of topic coverage, not depth: there are lots of good articles and tutorials out there on all these topics. If you need help with a topic and are unsure where to look, feel free to reach out to us.*

# Python Style Conventions

There are four important best practices for writing Python code:

* **Style:** Everyone has opinions on what individual coding style decisions are “good” or “bad” and experience has shown that trying to reach consensus and document all the individual decisions is fruitless. Instead, we strongly recommend following Google’s Python style guide. It is well thought out, very comprehensive, and provides rationales for its decisions.
* **Formatting:** Similarly, you should use an auto-formatting tool to maintain consistent code layout. Your IDE (e.g. VS Code or PyCharm) may provide this for you. We also like the black tool for this.
* **“Linting”:** You should also rely on your IDE’s error checking to reduce or eliminate code errors and warnings, highlight potential code duplication, and improve portability: aim for a “no squiggles” policy, and take the time to keep your code clean. We also like the flake8 and pep8 tools for this.
* **Type Hints:** Using type hints (also known as function signatures) in your code is the single most important thing you can do to reduce run-time errors in your code. Try to use type hints in all your functions.

## Suggested Reading

* Google’s in-house python style guide: <https://google.github.io/styleguide/pyguide.html>
* A helpful article on Flake8 and Black: <https://medium.com/@huzaifazahoor654/improving-code-quality-with-flake8-and-black-a-guide-for-python-developers-c374168d5884>
* InferLink’s coding conventions: <https://docs.google.com/document/d/11BmhRhn-azl8ItoALvb_S7Xz2Q_vpef2wd0XpSe8rzk>

# Python Programming Conventions

Other best practices for Python development include:

* **Schema Support:** If you read or write JSON files, consider using “schema” classes to provide type checking and data validation. The pydantic and marshmallow packages can be used for this.
* **Exceptions I:** Handle system exceptions using the try-except-finally pattern where appropriate. For example, it is good practice to catch an IOError when reading a file to handle conditions like an incorrect file path, a non-existent file, lack of permission, etc. Catching such errors explicitly provides more control over the code and lets one handle various error situations more gracefully. Try not to let exceptions leak out to the user.
* **Exceptions II:** You may choose to raise exceptions your own code, including user-defined exception types. For example, if a user inputted file does follow an expected format, it can be helpful to raise a user-defined IncorrectFileSyntax exception.
* **Logging:** Use a logging system to record information about your application as it runs. Logging data can be critical when debugging long-running, non-interactive system. Loggers usually have levels like Debug, Info, Warn, Error, etc., to help distinguish the severity of the diagnostic. Python’s built-in logging module should be sufficient for most cases.
* **Object-oriented programming:** Write object-oriented code for better organization, reusability, and maintenance of the code. Define classes and instantiate objects in order to benefit from encapsulation and inheritance. Sometimes the nature of the project can be such that a whole lot of object-oriented programming might be an overkill – but even in such cases, at least consider using dataclasses for defining complicated objects.

## Suggested Reading

* InferLink’s own coding conventions document:
  + <https://docs.google.com/document/d/1tIYxdeUZVNoKGpAZdJyRbIjNGuSHZl61WAdOqz7ldhQ>
* Two tutorials on dataclasses:
  + [https://www.geeksforgeeks.org/understanding-python-dataclasses/](•%09https:/www.geeksforgeeks.org/understanding-python-dataclasses)
  + <https://www.dataquest.io/blog/how-to-use-python-data-classes/>
* Two articles on python errors and exception handling:
  + <https://docs.python.org/3/tutorial/errors.html>
  + <https://realpython.com/python-exceptions/>
* Two articles on logging:
  + <https://docs.python.org/3/library/logging.html>
  + <https://realpython.com/python-logging/>

# Repository Layout

Your git repository (repo) should follow these conventions:

* At the top of the repo, add these things:
  + A README file (in markdown or plain text). It should contain instructions for building, testing, and running your application.
  + A LICENSE.txt file with the appropriate legal boilerplate for your project.
* Also at the top of the repo, you should have these directories:
  + src: for your source code (or, better, use the name of your python package)
  + tests: for your unit and system tests.
  + docs: for your project documentation. This may consist of markdown files, PDFs, PowerPoint slides, etc.
  + config: for any configuration or settings files your project or application may require, e.g. for DB connections, server construction, etc.
  + build: for any files related to building, packaging, cleaning, etc. (Standard files like requirements.txt and setup.py should stay in their stand locations, at the top of the repo.)
  + etc: for any other miscellaneous files
* Organize your source code under a single dir at the top of the repo. Group files into further subdirectories as needed to organize your classes.
  + In your tests directory, try to mirror the layout under your source directory. For example, the file src/graph/node.py should have its unit tests in tests/graph/test\_node.py.
* Use one source code file per (nontrivial) class.
* Include a copyright line at the top of each source code file.

## Suggested Reading

* One reasonable way to lay out your repo: <https://docs.python-guide.org/writing/structure/>

# Source Control

Use git to provide version control.

* Use .gitignore, but be conservative: only add files to it as you need to. (Do not rely on tools that know how to “automatically” populate this file for you.) Be sure to include \_\_pycache\_\_ in your .gitignore.
* When writing code, do your work on a branch of the main tree.
* Use consistent and helpful branch names, e.g. mpg/532-topsort-bug indicates the branch belongs to user mpg and it is addressing ticket #532 which has something to do with a bug in the topsort algorithm.
* Commit to your branch early and often.
* Branches should not be “long-lived”. Generally, use a new branch for each feature or bug fix.
* When merging your branch, strongly consider “squashing” your merges.

# Environments and Builds

* It's advisable to use a virtual environment for each Python project, so as to keep careful control over your external dependencies. There are various tools to select from: venv, conda, poetry, etc. (Docker can also be used for this.)
* Provide a simple way to build your system, e.g. build.sh or Makefile or setup.py. This is critical to making an automated build/test system for your project.
* Use Python version 3.8 or higher. It’s preferable to use the latest stable version if there are no limitations or restrictions based on other dependencies and/or environments. (If can be helpful to check for the minimum required version in the code, especially when the code might be deployed on different servers and by different people. The value of sys.version\_info can be used to do this.)
* Use a requirements.txt file to specify all your project’s dependencies. “Pin” the versions of third-party libraries by using == with a specific version number whenever possible. Avoid using >= or wildcards.

# Testing and QA

Your project should provide a “hands-free” (non-interactive) means of testing itself to verify that all is working as expected.

There are many kinds of tests, forming a hierarchy: tests for individual classes; tests for multiple, interacting classes; and tests for the whole system. A minimum, be aware of the “layers” of your code and test accordingly.

* For testing Python at the class-level, use pytest.
* Try to write at least one test per (nontrivial) class. Stubbing in even one trivial test now will reduce the effort/friction when you need to add more tests later.
* For larger granularity tests, you may want to write helper scripts (in Python or Bash) to perform functions like populating a database or standing up a local server.
* Focus on the testing that will provide value to you. For example, it might be better to have a test that says "yes, a non-zero length file was produced" than to spend days trying to future-proof things by carefully checking the file's contents (e.g. dealing with drifting model outputs, numerical precision, etc.).

# Release Management

Your project should have a visible version string.

* Each project will have its own notion of what this string should be: examples include 1.2, 1.2.3, 1.2.3-beta, 20231214, etc. Whatever you choose, make sure that the version strings, when compared lexically, are always increasing.
* When your code is ready to be delivered to another team, e.g. for initial integration or when you’ve fixed a bug, increment the version string. Mark your repo with the version string via a git tag.
* Your applications should print this value via a --version switch.
* It is often easiest to have a file VERSION.txt at the top of your repo; your version string lives only in that one place, so it is easy to update. Your apps can read from this file to print the version.

# Command Line Apps

These guidelines may help your command line applications:

* Implement a top-level class that hides the “real” code. This class should handle all the command-line parsing, logging setup, and so forth.
* Command line apps should explicitly call sys.exit(n) with a status value of zero (if there were no errors) or nonzero (if something went wrong).
  + Do not call exit from within your “real” code: instead, let the error propagate up and out to the top-level wrapper class.
* Provide a --log-file switch to specify where the log file should be written. (This is very important for automation scenarios.)
* Provide a --version switch that prints out the version string and then exits (with a status of zero).

1. Further Reading

These sites have high signal / low noise content about writing good Python code:

* The Hitchhiker's Guide to Python: <https://docs.python-guide.org/>
* Python Tutorials – Real Python: <https://realpython.com/>