Python Best Practices

Managing projects

`uv`

Use `uv` to simplify project management. It will create layouts, manage dependencies, create virtual environments, and other management tasks.

Virtual Environments

Use virtual environments to isolate dependencies

Editable install

Install packages in editable mode during development. The installed package points back to the development folder, so changes to your codebase are immediately available in your environment (not just in the project). This is needed so that tests can import your packages.

Cookiecutter

Use `cookiecutter` to generate consistent layouts for projects.

Project layouts

Be consistent in project layouts. The following code layouts work well.

Script

```
myproject
├── README.md
├── main.py
└── pyproject.toml
```

Module

```
myproject
├── README.md
├── docs
├── pyproject.toml
├── src
│   └── myproject.py
└── tests
    └── test_myproject.py
```

Package

```
myproject
├──── README.md
├──── docs
├──── pyproject.toml
├──── src
│    ├──── myproject
│    │    ├──── __init__.py
│    │    ├──── __main__.py
│    │    └──── module1.py
└──── tests
     ├──── __init__.py
     └──── test_module1.py
```

NOTE These layouts are just suggestions and can be varied as needed.

Modularize your codebase

Put user interface code in main script; put everything else in modules.

Source code control

Use git to manage changes to your codebase

Source formatting

Pep 8

Follow Pep 8 to be consistent with formatting and naming conventions.

`pylint` will analyze your code and report inconsistancies.

Use type hints

Use type hints everywhere. They will prevent mistakes as well as providing documentation.

Use protocols for specifying "has-a" relationships. For example, you want a class instance that has a `read()` method. A protocol can specify this.

DRY

Avoid duplicate code (Don’t Repeat Yourself)

Docstrings

Add documentation strings for files, classes, and functions/methods.

Unit Tests

Use `pytest` to write unit tests for all your code.

Check coverage with `coverage`

Logging

Use the `logging` module to track events of all severity levels.

Globals

Avoid. If you must have globals, make them upper case and well-documented

Imports

Put at top of scripts. First builtins, then non-local, then local.

Import all

Don't use `from MODULE import *` more than once in the same script. Avoid using it at all if possible.

Source code suggestions

Dataclasses

Use the `dataclasses` module to save time when defining classes.

Requests

Use the `requests` module to make HTTP requests to web servers or APIs.

Plan ahead

Define the project before coding.

Handle errors

Use try-except-else-finally as needed

Classes

Classes are great, but don't use classes when they are not needed

Initializing containers

Use `[]` to initialize empty list, `{}` for dictionaries, `()` for tuples.

Concurrency

Use `concurrent.futures` or the `async` modules.

String formatting

Use f-strings. Don't use `str.format()` or the `%` operator.

argparse

Manage command line options and arguments

Zen

Follow the zen of Python (`import this`)