Modern Good Practices for Python Development

Python has a long history, and it has evolved over time. This article describes some agreed modern best practices.

Using Python

Install Python With Tools That Support Multiple Versions

Use a tool like mise or pyenv to install Python on your development systems, so that you can switch between different versions of Python for your projects. This enables you to upgrade each project to a new version of Python without interfering with other tools and projects that use Python.

Alternatively, consider using Development Containers, which enable you to define an isolated environment for a software project. This also allows you to use a separate version of Python for each project.

Ensure that the tool compiles Python, rather than downloading standalone builds. The standlone builds are modified versions of Python that are maintained by a third-party. Both the pyenv tool and the Visual Studio Code Dev Container feature automatically compile Python, but you must change the mise configuration to use compilation. PDM and Hatch always download standalone builds when you use them to set up versions of Python.

If your operating system includes a Python installation, avoid using it. This Python installation is for operating system tools. It is likely to use an older

version of Python, and may not include all of the standard features. An operating system copy of Python should be marked to prevent you from installing packages into it, but not all operating systems set the marker.

Use The Most Recent Version of Python That You Can

For new projects, choose the most recent stable version of Python 3. This ensures that you have the latest security fixes, as well as the fastest performance.

Upgrade your projects as new Python versions are released. The Python development team usually support each version for five years, but some Python libraries may only support each version of Python for a shorter period of time. If you use tools that support multiple versions of Python and automated testing, you can test your projects on new Python versions with little risk.

> ⚠ *Avoid using Python 2.* It is not supported by the Python development team or by the developers of most popular Python libraries.

Use pipx To Run Developer Applications

Use pipx to run Python applications on development systems, rather than installing the applications with *pip* or another method. This ensures that each application has the correct libraries, because *pipx* automatically puts the libraries for each application into a separate Python virtual environment.

Consider using the pipx run command, rather than *pipx install*. The *pipx run* command downloads and runs the application without installing it. Each

application is cached for several days after the first download, which means that *pipx run* may not be slower than running a manually installed application.

The Python Packaging Authority maintain *pipx*, but it is not included with Python. To install *pipx*, run this command:

```
python3 -m pip install --user pipx
```

This enables you to use any Python command-line application with *pipx*. For example, this command downloads and runs the latest version of **bpytop**, a system monitoring tool:

```
pipx run bpytop
```

> *"PEP 668 - Marking Python base environments as "externally managed" recommends that users install Python applications with pipx."*

Developing Python Projects

Avoid Using Poetry

Avoid using **Poetry** for new projects. Poetry predates many standards for Python tooling. This means that it uses non-standard implementations of key features, such as the dependency resolver and configuration formats in *pyproject.toml* files.

If you would like to use a similar tool to develop your applications, consider using PDM. Hatch is another alternative to Poetry, but it is most useful for developing Python libraries. Both of these tools follow modern standards, which avoids compatibility issues.

Use a pyproject.toml File

Create a *pyproject.toml* file in the root directory of each Python project. Use this file as the central place to store configuration information about the project and the tools that it uses. The pyOpenSci project documentation on pyproject.toml provides an introduction to the file format.

Modern Python tools use the *pyproject.toml* file to store configuration. Some tools support *pyproject.toml*, but do not use it by default. Python project management tools like PDM and Hatch automatically create and use a *pyproject.toml* file.

> *"The various features of pyproject.toml files are defined these PEPs: PEP 517, PEP 518, PEP 621 and PEP 660."*

Create a Directory Structure That Uses the src Layout

Python itself does not require a specific directory structure for your projects. The Python packaging documentation describes two popular directory structures: the src layout and the flat layout. The pyOpenSci project documentation on directory structures explains the practical differences between the two.

For modern Python projects, use the src layout. This requires you to use **editable installs** of the packages in your project, but tools like **PDM** and **Hatch** will handle this for you.

Use Virtual Environments for Development

The **virtual environments** feature enables you to define separate sets of packages for each Python project, so that the packages for a project do not conflict with any other Python packages on the system. Always use Python virtual environments for your projects.

Several tools automate creating and switching between virtual environments. The **mise** version manager includes **support for virtual environments**. The **pyenv** version manager supports virtual environments with the **virtualenv plugin**. If you use a tool like **PDM** or **Hatch** to develop your projects, these also manage Python virtual environments for you.

You can set up and use virtual environments with *venv*, which is part of the Python standard library, but this is a manual process.

Use requirements.txt Files to Install Packages Into Environments

Avoid using *pip* commands to install packages into virtual environments. If you use **PDM** or **Hatch**, they manage packages in development and test environments. For other cases, create a *requirements.txt* file that specifies all of the packages that are required in the environment.

You can create *requirements.txt* files with whichever tool is appropriate. For example, PDM includes **an export feature** that creates *requirements.txt* files. If you do not already have a tool to create *requirements.txt* files, use the *pip-compile* utility that is provided by **pip-tools**.

You can then use the *pip-sync* utility in **pip-tools** to add the packages that are specified in the *requirements.txt* file into a target virtual environment. The *pip-sync* utility ensures that the packages in a virtual environment match the list in the *requirements.txt* file.

If you need to install packages without using *pip-sync*, run *pip install* with a *requirements.txt* file. For example, these commands install the packages that are specified by the file *requirements-dev.txt* into the virtual environment *.venv*:

```
source ./.venv/bin/activate
python3 -m pip install -r requirements-dev.txt
```

> ***"Ensure that your requirements.txt files include hashes for the packages. The pip-compile utility generates requirements.txt files with hashes if you specify the generate-hashes option."***

Format Your Code

Use a formatting tool with a plugin to your editor, so that your code is automatically formatted to a consistent style.

Black is currently the most popular code formatting tool for Python, but consider using **Ruff**. Ruff provides both code formatting and quality checks for Python code.

Run the formatting tool with your CI system, so that it rejects any code that does not match the format for your project.

Use a Code Linter

Use a code linting tool with a plugin to your editor, so that your code is automatically checked for issues.

flake8 is currently the most popular linter for Python, but consider using Ruff. Ruff includes the features of both flake8 itself and the most popular plugins for flake8.

Run the linting tool with your CI system, so that it rejects any code that does not meet the standards for your project.

Test with pytest

Use pytest for testing. It has superseded *nose* as the most popular testing system for Python. Use the *unittest* module in the standard library for situations where you cannot add *pytest* to the project.

Package Your Applications

Use *wheel* packages for libraries, or for tools that are intended to be used with an existing installation of Python. If you publish your Python application as a *wheel*, other developers can use it with *pipx* and *pip-sync*. These packages cannot be used without a Python installation.

In most cases, you should package an application in a format that enables you to include your code, the dependencies and a copy of the required version of

Python. This ensures that your code runs with the expected version of Python, and has the correct version of each dependency.

Use container images to package applications that provide a network service, such as a Web application. Use **PyInstaller** to publish desktop and command-line applications as a single executable file. Each container image and PyInstaller file includes a copy of Python, along with your code and the required dependencies.

Language Syntax

Use Type Hinting

Current versions of Python support type hinting. Consider using type hints in any critical application. If you develop a shared library, use type hints.

Once you add type hints, the **mypy** tool can check your code as you develop it. Code editors can also read type hints to display information about the code that you are working with.

If you use **Pydantic** in your application, it can work with type hints. Use the **mypy plugin for Pydantic** to improve the integration between mypy and Pydantic.

> *"PEP 484 - Type Hints and PEP 526 – Syntax for Variable Annotations define the notation for type hinting."*

Format Strings with f-strings

The new **f-string** syntax is both more readable and has better performance than older methods. Use f-strings instead of *%* formatting, *str.format()* or

str.Template().

The older features for formatting strings will not be removed, to avoid breaking backward compatibility.

The f-strings feature was added in version 3.6 of Python. Alternate implementations of Python may include this specific feature, even when they do not support version 3.6 syntax.

> *"PEP 498 explains f-strings in detail."*

Use Datetime Objects with Time Zones

Always use *datetime* objects that are **aware** of time zones. By default, Python creates *datetime* objects that do not include a time zone. The documentation refers to *datetime* objects without a time zone as **naive**.

Avoid using *date* objects, except where the time of day is completely irrelevant. The *date* objects are always **naive**, and do not include a time zone.

Use aware *datetime* objects with the UTC time zone for timestamps, logs and other internal features.

To get the current time and date in UTC as an aware *datetime* object, specify the UTC time zone with *now()*. For example:

```
from datetime import datetime, timezone

dt = datetime.now(timezone.utc)
```

Python 3.9 and above include the **zoneinfo** module. This provides access to the standard IANA database of time zones. Previous versions of Python require a third-party library for time zones.

> *"PEP 615 describes support for the IANA time zone database with **zoneinfo**."*

Use enum or Named Tuples for Immutable Sets of Key-Value Pairs

Use the *enum* type in Python 3.4 or above for immutable collections of key-value pairs. Enums can use class inheritance.

Python 3 also has *collections.namedtuple()* for immutable key-value pairs. Named tuples do not use classes.

Create Data Classes for Custom Data Objects

The data classes feature enables you to reduce the amount of code that you need to define classes for objects that exist to store values. The new syntax for data classes does not affect the behavior of the classes that you define with it. Each data class is a standard Python class.

You can set a *frozen* option to make frozen instances of a data class.

Data classes were introduced in version 3.7 of Python.

> *"PEP 557 describes data classes."*

Use collections.abc for Custom Collection Types

The abstract base classes in *collections.abc* provide the components for building your own custom collection types.

Use these classes, because they are fast and well-tested. The implementations in Python 3.7 and above are written in C, to provide better performance than Python code.

Use breakpoint() for Debugging

This function drops you into the debugger at the point where it is called. Both the **built-in debugger** and external debuggers can use these breakpoints.

The **breakpoint()** feature was added in version 3.7 of Python.

> *"PEP 553 describes the breakpoint() function."*

Application Design

Use Logging for Diagnostic Messages, Rather Than print()

The built-in *print()* statement is convenient for adding debugging information, but you should include logging in your scripts and applications. Use the **logging** module in the standard library, or a third-party logging module.

Use The TOML Format for Configuration

Use **TOML** for data files that must be written or edited by human beings. Use the JSON format for data that is transferred between computer programs. Avoid using the INI or YAML formats.

Python 3.11 and above include *tomllib* to read the TOML format. Use **tomli** to add support for reading TOML to applications that run on older versions of Python.

If your Python software needs to generate TOML, add **Tomli-W**.

> ***"PEP 680 - tomllib: Support for Parsing TOML in the Standard Library explains why TOML is now included with Python."***

Only Use async Where It Makes Sense

The **asynchronous features of Python** enable a single process to avoid blocking on I/O operations. To achieve concurrency with Python, you must run multiple Python processes. Each of these processes may or may not use asynchronous I/O.

To run multiple application processes, either use a container system, with one container per process, or an application server like **Gunicorn**. If you need to build a custom application that manages muliple processes, use the **multiprocessing** package in the Python standard library.

Code that uses asynchronous I/O must not call *any* function that uses synchronous I/O, such as *open()*, or the *logging* module in the standard library. Instead, you need to use either the equivalent functions from *asyncio* in the standard library or a third-party library that is designed to support asynchronous code.

The **FastAPI** Web framework supports **using both synchronous and asynchronous functions in the same application**. You must still ensure that asynchronous functions never call any synchronous function.

If you would like to work with *asyncio*, use Python 3.7 or above. Version 3.7 of Python introduced **context variables**, which enable you to have data that is local to a specific *task*, as well as the *asyncio.run()* function.

> *"PEP 0567 describes context variables."*

Libraries

Handle Command-line Input with argparse

The **argparse** module is now the recommended way to process command-line input. Use *argparse*, rather than the older *optparse* and *getopt*.

The *optparse* module is officially deprecated, so update code that uses *optparse* to use *argparse* instead.

Refer to **the argparse tutorial** in the official documentation for more details.

Use pathlib for File and Directory Paths

Use **pathlib** objects instead of strings whenever you need to work with file and directory pathnames.

Consider using the **the pathlib equivalents for os functions**.

The existing methods in the standard library have been updated to support Path objects.

To list all of the the files in a directory, use either the *.iterdir()* function of a Path object, or the *os.scandir()* function.

This **RealPython article** provides a full explanation of the different Python functions for working with files and directories.

The *pathlib* module was added to the standard library in Python 3.4, and other standard library functions were updated to support Path objects in version 3.5 of Python.

Use os.scandir() Instead of os.listdir()

The *os.scandir()* function is significantly faster and more efficient than *os.listdir()*. Use *os.scandir()* wherever you previously used the *os.listdir()* function.

This function provides an iterator, and works with a context manager:

```
import os

with os.scandir('some_directory/') as entries:
    for entry in entries:
        print(entry.name)
```

The context manager frees resources as soon as the function completes. Use this option if you are concerned about performance or concurrency.

The *os.walk()* function now calls *os.scandir()*, so it automatically has the same improved performance as this function.

The *os.scandir()* function was added in version 3.5 of Python.

> *"PEP 471 explains os.scandir()."*

Run External Commands with subprocess

The **subprocess** module provides a safe way to run external commands. Use *subprocess* rather than shell backquoting or the functions in *os*, such as *spawn*, *popen2* and *popen3*. The *subprocess.run()* function in current versions of Python is sufficient for most cases.

> *"PEP 324 explains the technical details of subprocess in detail."*

Use httpx for Web Clients

Use **httpx** for Web client applications. It **supports HTTP/2**, and **async**. The httpx package supersedes **requests**, which only supports HTTP 1.1.

Avoid using *urllib.request* from the Python standard library. It was designed as a low-level library, and lacks the features of httpx.