Python Environment and Repository Management

Nima Nooshiri

22 January 2025

`uv` for Python Environment Management

- uv is a Python package and project manager, written in Rust.
- Install `uv` using standalone installer:

(On Linux)

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

(On Windows)

```
$ powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

- Check out other installation methods on the official website.
- `uv` works with `pyproject.toml` file:
  - The `pyproject.toml` file is the Python standard for defining configuration for a project.
  - See official guide for `pyproject.toml` for more details.

Basic Usage of `uv`

- `$ uv init ...` creates a project:
  - The `--lib` flag is used to create a project for as a library.
  - A library provides functions for other projects to consume.

```
# Create a new project with Python 3.12
$ uv init aidan-benchmark --lib --python 3.12 --description "My Data Science Project"
```

- `$ uv add ...` installs tools and dependencies to the project:
  - Dependencies are added to the project's `pyproject.toml` file.

```
# Install NumPy and Plotly: latest versions
$ uv add numpy plotly

# Install Numpy and Plotly: specific versions
$ uv add numpy==2.2.2 plotly==5.24.1
```

- `$ uv remove ...` uninstalls tools and dependencies from the project:

```
# Uninstall Numpy
$ uv remove numpy
```

Working with `requirements.txt` Files

- Install all packages listed in a given `requirements.txt` file:

```
$ uv add --requirements my-requirements.txt
```

- Export the project's lockfile to a `requirements.txt` format:

```
$ uv export --format requirements-txt -o my-requirements.txt
```

`uv` and Virtual Environments

- `uv` requires using a virtual environment by default.
- As a default, `uv` create a virtual environment at `.venv`.
    - Specific name or path can also be specified before installing any tools: `$ uv venv my-env`.
- The virtual environment can be **activated** to make its packages available:

(on Linux)

```
$ source .venv/bin/activate
```

(on Windows)

```
$ .venv\Scripts\activate
```

- **Note:** `uv` searchs for a virtual environment in the following order:
    i. An activated virtual environment based on the `VIRTUAL_ENV` environment variable.
    ii. An activated Conda environment based on the `CONDA_PREFIX` environment variable.
    iii. A virtual environment at `.venv` in the current directory.

So, if you have an environment with a custom name rather than `venv`, you need to activate it before installing/removing tools. Otherwise, `uv` will create the default `venv` environment and install everything there.

Other Features of `uv`

- `$ uv lock` creates a lockfile for the project's dependencies called `uv.lock`.
- `$ uv sync` syncs the project's dependencies with the environment.
  - Syncing ensures that all project dependencies are installed and up-to-date with the lockfile `uv.lock`.

`mypy` for Static Type Checking

With mypy, we can add type hints to Python programs, and mypy raise **warning** when those types are used incorrectly. For example:

```python
# file: my-script.py

def my_func(x: int) -> str:
    return x * 2   # Problem: incompatible return value type (got "int", expected "str")
```

- To install mypy:

  ```
  $ uv add mypy
  ```

- To run it using mypy tool:

  ```
  $ mypy my-script.py
  ```

For the above example, it will return:

```
my-script.py:2: error: Incompatible return value type (got "int", expected "str")  [return-value]
Found 1 error in 1 file (checked 1 source file)
```

`mypy` Configuration

- `mypy` is configurable and it reads configuration settings from a file in this order:
  i. `./mypy.ini`
  ii. `./.mypy.ini`
  iii. `./pyproject.toml`
  iv. `./setup.cfg`
  v. `$XDG_CONFIG_HOME/mypy/config`
  vi. `~/.config/mypy/config`
  vii. `~/.mypy.ini`

Example `mypy.ini` file:

```
[mypy]
ignore_missing_imports = true
allow_redefinition = true
```

- Check out the official documentation for `mypy` configuration and available options.

Code Documentation

The most common docstring formats used are:

- Google docstrings
- NumPy/SciPy docstrings

Google docstring example

```python
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value

    """
    return True
```

NumPy/SciPy docstring example

```python
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

`black` for Python Auto Formatting

- Standard tool: part of the Python Software Foundation.
- Fully automated.
- Opinionated:
  - *Black* is a PEP 8 compliant opinionated formatter with its own style.
  - The coding style of the *Black* can be viewed as a strict subset of PEP 8.
- To install *Black*:

```
$ uv add black
```

- To format Jupyter Notebooks, install:

```
$ uv add "black[jupyter]"
```

- Basic usage:

```
$ black my-script.py
```

- To learn more about `black`, check out the following resources:
  - Python PEP 8
  - Google Python Style Guide
  - Getting started with `black`.

`pytest` for Testing Python Codes

- Standard, widely used Python testing framework.
- Very pythonic implementation.
- Powerful features:
  - Auto-discovery of test modules and functions.
  - Running parametrized variations of a test (the same test with multiple different values).
  - Shared resources across tests.
- To install `pytest`:

```
$ uv add pytest
```

- Run test(s):

```
# One test file
$ pytest ./tests/test_sample.py

# All test files in a directory
$ pytest ./tests
```

`pytest` Basic Example

```python
import pytest

@pytest.fixture
def mock_data():
    return [1, 2, 3, 4, 5]


def test_sum(mock_data):
    assert sum(mock_data) == 15


def test_max(mock_data):
    assert max(mock_data) == 5
```

- To learn more about `pytest`, check out the following resources:
  - Getting started with Pytest.
  - Anatomy of a test.

Next Steps

- Creating a GitLab CI pipeline for the project:
  - Automate the formatting, testing, and other processes.