Workflow basics

Tips to improve productivity

Jason DeBacker

January 17, 2025

Roadmap

- Text Editors/IDEs
- Navigating the command line
- Coding Structure
- Code formatting
- Unit Testing
- Continuous Integration
- Virtual environments

Why do you want one of these?

- Syntax highlighting
- Autocorrect/completion of code
- Previews of data objects
- Previews of compiled code (e.g., Markdown, HTML, TeX)
- Integrated terminal to execute code
- Coding copilot

Text editor or IDE

- Text Editors: Lightweight, fast, extensible
- IDEs: Full-featured, integrated tools like debugging, typically software-specific

Which ones to look at for programming in Python?

- Text Editors:
  - → **Visual Studio Code**
    - Free, open-source, excellent Python extension, git integration, integrated terminal, large extension ecosystem, lightweight yet powerful
  - → Atom
    - Close to VS Code, but not quite as good
- IDEs:
  - → Pycharm
    - Advanced debugging, database tools, scientific computing support, but is resource intensive
  - → Spyder
    - Scientific computing focus, IPython integration variable explorer, built for data science

Extensions for Text Editor/IDE

- Code linters (for language-specific errors: Python, Stata, R, etc.)
- Git integration
- DataWrangler (or similar for data preview)
- Jupyter Notebook display
- Preview tools (for Markdown, HTML, LaTeX, etc.)
- TeX compiler

Roadmap

Text Editors/IDEs

Navigating the command line

Coding Structure

Code formatting

Unit Testing

Continuous Integration

Virtual environments

Why use the command line?

- More efficient than GUI
- Essential for development
- Required for many tools
- Enables automation

Essential commands

- Navigation
  - → `pwd` - Print working directory
  - → `ls` / `dir` - List contents
  - → `cd` - Change directory
  - → `cd ..` - Move up a directory level
  - → `cd ~` - Home directory
- File Operations
  - → `mkdir` - Create directory
  - → `touch` / `echo` - Create file
  - → `cp` - Copy
  - → `mv` - Move/rename
  - → `rm` / `rmdir` - Remove

Power user moves

- Tab completion
- Command history (↑/↓)
- `ctrl + r` - Search history
- Wildcards: `*, ?, []`
- `gzip` / `gunzip` - Compress/decompress
- Writing bash scripts
- Git workflow (see Git slides)

Python-specific commands

- `python --version` - Check Python version
- `pip list` / `conda list` - List installed packages
- `pip install package_name` / `conda install package_name` - Install package
- `conda activate venv_name` - Activate conda virtual environment called `venv_name` (More on this later)

Roadmap

Text Editors/IDEs

Navigating the command line

Coding Structure

Code formatting

Unit Testing

Continuous Integration

Virtual environments

Use the object oriented programming paradigm

- Write your code in functions of classes
- This makes the code more modular and flexible
- It's easier to test code written this way (see below)
- The code is usually easier to read
- **Tip**: you can import functions from a script as a module and use them in another script

The `os` package

- `os` is a Python package that provides a way to interact with the operating system
- Useful functions:
  - → `os.getcwd()` - Get current working directory (I use this in Notebooks)
  - → `os.path.dirname(os.path.realpath(__file__))` - Get the directory of the current script (I usually use this is scripts)
  - → `os.listdir()` - List files in a directory
  - → `os.path.join()` - Join paths
  - → `os.path.exists()` - Check if a path exists (e.g., see if a big data file you created exists)
  - → `os.makedirs()` - Make directories
  - → `os.remove()` - Remove a file (helpful to clean up temporary files)

Standalone scripts

- When you run a Python script, the code in the script is executed
- Sometimes you want to write a script that can be run as a standalone script or imported as a module
- The `if __name__ == '__main__':` block allows you to write code that will only be executed if the script is run as a standalone script
- This is not necessary in some systems, but it's a good habit to get into
- Example:

```
1 def function1():
2     print("Hello, world!")
3
4
5 if __name__ == "__main__":
6     function1()
```

The `argparse` package

- `argparse` is a Python package that provides a way to parse command line arguments
- Useful for writing scripts that can be run from the command line
- Example:
- `python my_script.py --input_file my_data.csv`
- `argparse` can be used to parse the input file name

Roadmap

Text Editors/IDEs

Navigating the command line

Coding Structure

Code formatting

Unit Testing

Continuous Integration

Virtual environments

Why format code?

- Improves readability/maintainability
- Ensures consistency
- Prevents conflicts
- Speeds up code reviews/debugging
- Reduces errors

Who says what is good formatting?

- **PEP 8** - Python Enhancement Proposal 8
- Official style guide for Python
- Some thought went into this, and many follow
- But who reads?

Thoughtless code formatting

- Two packages (pick one or the other) that can help you format your code without any work:
  - → `black`
    - "Uncompromising" formatter, zero configuration, PEP8 compliant
  - → `autopep8`
    - More configurable, PEP8 based
- I use `black`
- Both can integrate with VS Code or PyCharm (e.g., you can configure to format on save)
- Both can be run from the command line, e.g., `black my_file.py`

Roadmap

Text Editors/IDEs

Navigating the command line

Coding Structure

Code formatting

Unit Testing

Continuous Integration

Virtual environments

What is unit testing?

- Testing individual units of code
- Typically functions or classes
- But can also adapt for data analysis (e.g., create data and see if estimator can recover parameters used to create data)

Why test?

- Catch bugs early
- Document behavior
- Enable refactoring
- Improve design

Testing in Python with `pytest`

- `pytest` is the industry standard
- Simple syntax
- Powerful fixtures (e.g., create temporary directories, parameterize tests)
- Rich plugin ecosystem

Testing example

```
import pytest
import numpy as np


# example_test.py
def test_add_numbers():
    result = np.add(2, 3)
    assert result == 5


def test_empty_list():
    my_list = []
    assert len(my_list) == 0
    assert not my_list

```

Running unit tests

- Run all tests in a directory: `pytest`
  - → will run all files that start with `test_` or end with `_test.py`
  - → will run all functions/classes that start with `test_` in those files
  - → can specify options (e.g., markers to skip certain tests)
- Run a specific test module in a directory:

  ```
  pytest test_example.py::test_add
  ```
- Run a specific test: `pytest test_example.py::test_add`

Test Best Practices

- Test one thing per test
- Use descriptive names
- Arrange-Act-Assert pattern
- Keep tests independent
- Don’t test implementation
- Use fixtures for setup
- Mock external dependencies

Test Coverage

- Coverage.py tool – or Codecov
- Aim for meaningful coverage
- Common targets: 80-90%
- Focus on critical paths

Roadmap

Text Editors/IDEs

Navigating the command line

Coding Structure

Code formatting

Unit Testing

Continuous Integration

Virtual environments

Continuous Integration (CI) CI Fundamentals

- Automated build & test
- Early issue detection
- Quality assurance
- Faster development

GH Actions

- Most popular CI tool for open-source projects
- Free for public repos
- YAML configuration
- Runs on push, pull request, schedule – many triggers you can customize

Example GitHub Actions YAML file

```yaml
name: Python CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.x'
      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest pytest-cov
      - name: Run tests
        run: |
          pytest --cov=./ --cov-report=xml
      - name: Check formatting
        run: |
          black --check .
          isort --check-only .
```

Example GitHub Actions

Working examples on GitHub: Cost-of-Capital-Calculator workflows

CI Best Practices

- Fast builds
- Cache dependencies
- Matrix testing
- Clear error reporting
- Automated formatting
- Security scanning

CI in GitLab

- Similar to GitHub Actions
- More built-in features
- More complex configuration
- More control over runners
- Still YAML-based, but syntax differs
  - → `.gitlab-ci.yml` used by default
  - → See this comparison of syntax: GitLab CI vs GitHub Actions

Complete Development Workflow

Daily Workflow:

- Pull latest changes
- Create feature branch
- Write tests
- Implement features
- Run local tests
- Format code
- Commit and push
- CI pipeline runs
- Code review
- Merge

Roadmap

Text Editors/IDEs

Navigating the command line

Coding Structure

Code formatting

Unit Testing

Continuous Integration

Virtual environments

What is an environment?

- **A Python environment**: a self-contained directory that contains a Python installation for a particular version of Python, plus a number of additional packages.
- **A virtual environment**: a Python environment that is isolated from other Python environments on the same machine.

Why use a virtual environment?

- Open-source projects often have many dependencies, for which new updates may be released at any time
- Virtual environments solve several issues related to this:
  - → Isolation
  - → Dependency management
  - → Reproducibility
  - → Avoid conflicts

How to create virtual environment

One way: with conda an a `enivronment.yml` file, e.g.,

```yaml
name: usitc-env
channels:
- conda-forge
dependencies:
- python>=3.9, <3.13
- numpy
- ipython
- scipy>=1.7.1
- pandas>=1.2.5
- numba
- jax
- matplotlib
- scikit-learn
- dask>=2.30.0
- dask-core>=2.30.0
- distributed>=2.30.1
- paramtools>=0.15.0
- sphinx>=3.5.4
- sphinx-argparse
- sphinxcontrib-bibtex>=2.0.0
- sphinx-math-dollar
- pydata-sphinx-theme
- jupyter-book>=0.11.3
```

How to create and use virtual environment

- `conda create --name myenv --file environment.yml`
- `conda activate myenv`
- To use in a Jupyter Notebook, install `ipykernel` and run `python -m ipykernel install --user --name=myenv`
- To deactivate, run `conda deactivate`

Special cases:

- Create a completely empty environment (named `myenv`):
  → `conda create --name myenv --no-default-packages`
- Dump all installed packages in an environment to a file:
  → `conda env export > environment.yml`

Tools Summary

- Editor/IDE: VS Code or PyCharm
- Formatting: `black` or `autopep8`
- Testing: `pytest`
- CI: GitHub Actions
- VCS: Git
- Virtual Environments: Conda