# Setting Up Python Development Environment for Use in a Small Classroom

Roman Yasinovskyy, Karina Hoff, and David Oniani

Department of Computer Science
Luther College
Decorah, IA 52101
{roman, hoffka04, oniada01} @ luther.edu

## Abstract

In this paper we describe a way to set up a classroom environment using the common tools used in software development with the focus on Python. We cover version control system *Git*, testing framework *pytest*, source formatter *black*, and linter *flake8*. Proper use of these tools helps students preserve history of code changes, provides feedback before a student submits their assignment, and eliminates subjective style preferences from the code review process.

The proposed setup works on Linux, macOS, and Windows without major modifications. We recommend publishing a standard *requirements.txt* for students to install the required packages and using Python's built-in module *venv* to isolate a class from the rest of the system. This approach helps eliminate (or at least greatly reduce) the common "works on my machine" problem by locking both Python version and installed packages.

## 1 Introduction

As students advance through the computer science sequence and become more proficient with Python they also discover more and more development tools. While it is possible to learn syntax using Python's IDLE or one of many online editors, at some point students realize the limitations of those tools and switch to a text editor or an IDE. Add a couple of operating systems to the mix and now instructors have to deal with different file naming strategies, formatting conventions, and Python versions used.

Regardless of the development environment, two major challenges students face in programming classes are making sure their code works as expected and getting feedback from instructors if it does not. On the other hand, instructors need to ensure consistent and predictable grading criteria, set up reproducible environment, and have a way to distribute and collect assignments efficiently.

The proposed setup allows instructors to enforce particular code style and project structure in their classes while providing immediate feedback to the students and not distracting from the course content. It was piloted in a typical CS2 (Algorithms and Data Structures) course and later successfully adopted in a CS3 (Advanced Algorithms and Data Structures) and other courses in our Computer Science curriculum.

We believe this setup works the best for small classes where instructors review each submission and provide feedback directly to students but it is reasonable to assume that the grading process can be automated or distributed among teaching assistants.

This approach can be seen as a variation of test-driven development, though tests are provided by instructors and not written by students. We are not going to argue the advantages of TDD as it has been done in multiple studies[2, 1, 5]. We focused on practical aspects of setting up a class and leave it for the individual instructor to be more intentional as they adopt TDD in their class.

## 2 Tools

### 2.1 Python

Python 2 reached its end-of-life on January 1st, 2020 and we would discourage its use in a new project or a class. Annual surveys by JetBrains[4, 3] show Python 3 adoption rate at 84% in 2018 (increased to 87% in 2019) and the majority (84%) of respondents were using the two latest released versions of the language available at the time (3.7 and 3.6).

We used Python 3.6 to set up the class environment on Ubuntu 18.04 and macOS 10.13; Python 3.7 was used on Windows 10 Pro (1909). The biggest challenge we have noticed in classes with this setup is that instructions for Windows must take into account method of installation (executable downloaded from Python.org or an app from the Microsoft Store) on a machine. Windows users may have to use `python` or `py -3` instead of `python3` and system-specific path separator.

## 2.2 Virtual Environment

A *virtual environment* is a directory that contains an installation of a particular version of Python and any additional packages required by an application. It can be isolated from the rest of the system to create a reproducible, locked environment. There are several ways to accomplish this task, including *pipenv* and *virtualenv*. We chose *venv* for it is included in the standard library since Python 3.3 and requires no changes to the usual development workflow once the environment is activated.

Listing 1 shows the sequence of commands necessary to create and activate a new virtual environment using the version of Python aliased as `python3` on Ubuntu. Once activated, the environment uses that version of Python as default, so there is no need to specify `python3` inside the environment.

Virtual environments have the following benefits:

- No need to have administrative privileges to install packages. This is especially useful if students are working in a lab rather than on a personal machine.
- Different versions of packages can be installed simultaneously (e.g. for different classes).
- They can be created and removed without changing the system installation of Python.

```
roman@ubuntu:~/mics2020$ python3.6 -m venv .venv
roman@ubuntu:~/mics2020$ source .venv/bin/activate
(.venv) roman@ubuntu:~/mics2020$ python --version
Python 3.6.9
(.venv) roman@ubuntu:~/mics2020$ python -m pip list
pip (9.0.1)
pkg-resources (0.0.0)
setuptools (39.0.1)
```

Listing 1: Activating Virtual Environment on Ubuntu

Once the environment has been activated, instructors can install all the required packages and generate the *requirements.txt* (see Listing 2) while students can recreate it with one command inside their own virtual environment (see Listing 3).

```
$ python -m pip install black flake8 pytest
$ python -m pip freeze > requirements.txt
```

Listing 2: Freezing the Installed Packages

```
$ python -m pip install -r requirements.txt
```

Listing 3: Installing the Required Packages

Windows users can invoke the activation script as `.\.venv\Scripts\Activate.ps1` when using PowerShell (see Listing 4) or `.venv\Scripts\activate.bat` from the command prompt.

```
PS D:\mics2020> python -m venv .venv
PS D:\mics2020> .\.venv\Scripts\Activate.ps1
(.venv) PS D:\mics2020> python --version
Python 3.7.4
(.venv) PS D:\mics2020> python -m pip list
Package    Version
---------- -------
pip        19.0.3
setuptools 40.8.0
```

Listing 4: Activating Virtual Environment on Windows

As long as the instructor and students use the same major version of Python the behavior of the code should be consistent and reproducible, potentially eliminating "works on my machine" explanation of code problems.

## 2.3 Source Version Control

Over the last ten years *Git* became the leading version control system used for open-source and proprietary projects, replacing Subversion as a de-facto VCS. While it is designed to handle large teams and distributed projects, individual students can benefit from the history of changes too. As *Git* grew in popularity, repository hosting services emerged too, notably GitHub, Bitbucket, and GitLab. We use GitHub to host public class repositories. Student repositories can be hosted on other platforms but usually it is GitHub too.

Code provided in Listing 5 assumes a *private* repository has been created using GitHub web interface and its local copy is stored in `mics2020` directory. Students should invite instructors as collaborators to their repositories, so only two of them would have access to the student's code. Student's local repository should now be connected to both the class public and their private repositories (see Figure 1).

```
$ cd mics2020
$ git init
$ git remote add origin \
> git@github.com:yasinovskyy/mics2020.git
$ git remote add upstream \
> git@github.com:yasinovskyy/mics2020-pub.git
$ git pull upstream master
$ git push -u origin master
$ git remote -v
origin    git@github.com:yasinovskyy/mics2020.git (fetch)
origin    git@github.com:yasinovskyy/mics2020.git (push)
upstream  git@github.com:yasinovskyy/mics2020-pub.git (fetch)
upstream  git@github.com:yasinovskyy/mics2020-pub.git (push)
```

Listing 5: Student's Repository Configuration

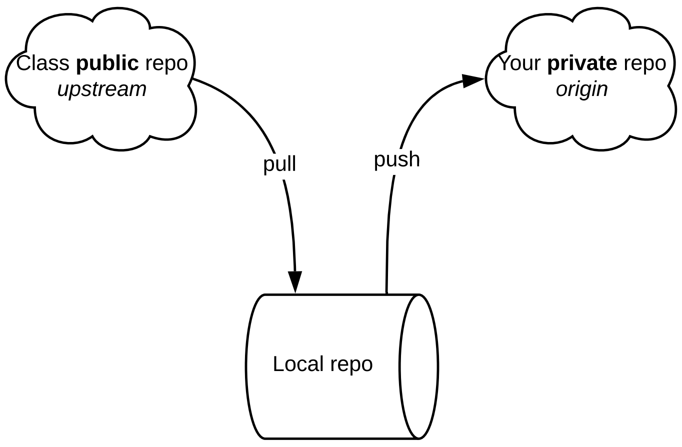

Figure 1: Student's repository

Once the repository is configured, students can use the usual `pull`, `add`, `commit`, and `push` commands to retrieve new assignments, lecture notes, and later submit completed projects for grading.

Instructors can use the public repository to publish assignments, class notes, and quickly post minor fixes and clarifications. Depending on their operating system instructors may use a script or a *Git* alias to collect assignments from the whole class rather than rely on their school's LMS. Once collected, code can be reviewed in the instructor's editor of choice.

Services like codePost, Gradescope, or GitHub Classroom offer code distribution, collection, and review functionality, but the advantage of our approach is its similarity to a "real" project setup. Our goal is not to hide VCS but to encourage its extensive and explicit use by the students. Our experience shows that a small non-graded project during the first week of a class is sufficient to make students comfortable using *Git* and resolve any issues.

Using any cloud service raises a question of its compliance with FERPA and potential academic integrity infringements. We recommend using private repositories for individual students, thus protecting their code from unauthorized access. Private repositories do not appear on a user's GitHub main page so an outsider will not be able to determine that a student is enrolled in a specific class.

## 2.4 Testing Framework

*pytest* is a Python testing framework that uses `assert` statements to test conditions and can be used to run unit tests created for other frameworks, including standard Python `unittest`. Its core functionality is sufficient for the purposes of our example and can be extended using plugins. For example, `pytest-timeout` allows setting time limit on individual tests and marking inefficient solutions as failed.

Listing 6 shows a function (defined in a file *greetings.py*) that returns "Hello, " concatenated with the parameter and Listing 7 shows a test that seems to verify this function.

```
def hello(audience: str) -> str:
    """Greet the audience"""
    return "Hello, " + audience
```

Listing 6: Hello World

Parametrized tests are one of the features of the *pytest* framework. Instructors can specify edge cases and students can tweak their implementations trying to pass all test cases. Parameters for the test in Listing 8 combine input and expected output while Listing 9 tests if the function raises an error with the expected message.

```
import pytest
from greetings import hello

def test_hello():
    """Testing the output"""
    assert hello("class") == "Hello, class"
    assert hello("world") == "Hello, world"
```

Listing 7: Test for `hello`

```
AUDIENCE = [
    ("World", "Hello, World"),
    ("MICS 2020", "Hello, MICS 2020"),
]

@pytest.mark.parametrize("data, expected", AUDIENCE)
def test_hello(data, expected):
    """Testing the output"""
    assert hello(data) == expected
```

Listing 8: Improved Test for `hello`

```
AUDIENCE_ERR = [42, None, [1, 2]]

@pytest.mark.parametrize("data", AUDIENCE_ERR)
def test_hello_err(data):
    """Testing the exception"""
    with pytest.raises(TypeError) as exc:
        hello(data)
    assert str(exc.value) == "Unable to greet " + str(data)
```

Listing 9: Error Handling for `hello`

With function `test_hello_err` added to the test file, our naive implementation from Listing 6 fails those tests. At this point students should read the error message provided by *pytest* (see Listing 10), update the function to check the type of the parameter, and raise a `TypeError` if it is not `str`.

```
=============== short test summary info ================
FAILED test_hello.py::test_hello_err[42] - AssertionError:
 assert 'must be str, not int' == 'Unable to greet 42'
FAILED test_hello.py::test_hello_err[None] - AssertionError:
 assert 'must be str, not NoneType' == 'Unable to greet None'
FAILED test_hello.py::test_hello_err[data2] - AssertionError:
 assert 'must be str, not list' == 'Unable to greet [1, 2]'
```

Listing 10: `pytest` Output

Instructors may assign points to each passed test, thus decreasing uncertainty and bias from the grading process. It is also possible to order test by the increasing complexity of the tested functions, making sure students implement simpler units before attempting more complex tasks.

## 2.5 Automated Code Formatter

While Python's enforced use of indentation makes most code readable, it is still possible to have poorly formatted multi-line expressions, long lists, and misaligned dictionaries in one's source. We should encourage students to *write* PEP-8 compliant code but a formatter has the advantage of helping students focus on the solution substance rather than style.

One possible concern is that predefined function names and formatting rules make it more difficult to tell two submissions apart and detect cases of plagiarism. While we did not address this in our study, a tool like *Algae* could be integrated in the toolchain on the instructor's end to detect infringements.

*black* is not the only one code formatter available for Python and the recommendations for its use should apply to *yapf* and *autopep8*. It can be used as a standalone command line tool, configured to be invoked by an IDE, or as a pre-commit hook.

## 2.6 Linter

If writing proper code is one of the learning goals of a course, then automated code formatters may be considered "cheating". In that case students may be directed to use a linter like *pylint* or *flake8* and read their hints as if they are error messages, debugging and eliminating them just like source code bugs.

## 3 Student Feedback

In general, after some initial confusion students embraced the use of this environment configuration and unit tests helped them be more successful in class. We did not address this question in the course evaluation survey and only one student mentioned tests at that time (see the first comment in the list below).

Later we asked 15 students of the class the following question: "Do you think using unit tests helped you be more successful in class?" Seven of them responded (comments 2–8 in the list below). While mostly positive, it should be noted that the answers were not anonymous.

1. I think that the homework assignments and projects could have had more tests because sometimes the tests felt like they weren't testing the main focus of the assignment, but more superfluous things.

2. Unit testing helped me think through problems and consider edge cases, ultimately avoiding silly programming errors and saving a lot of time. Besides, having test cases in mind, it was a lot easier to refactor the code or change the programming logic. More importantly, practicing unit testing in the classroom got me familiar with how testing frameworks work and taught me the practicality, flexibility, and maintainability of well-tested code. Unit testing was a great success for the class!

3. I love unit tests always. I think it helped me a lot in understanding the problems. It definitely contributed a lot to my success in the class.

4. The pytest module was pretty helpful for me, because it gave concrete examples for you to test your programs against. Talking about "algorithms" as an abstract topic often makes them seem more daunting than they actually are, so being able to test code I wrote against edge cases and more involved cases helped a lot. Additionally, it saved me time because I didn't have to waste time double checking answers that I had come up with for tests were actually right; instead, I could easily get a handle on the algorithm in question based on the expected pytest answers.

5. I think that including testing was both helpful and important. My only point for improvement, is that I wish we had spent a little more time building tests ourselves or learning about how that process worked for Test Driven Development experience.

6. Yes, the test helps writing the code and to find the error. Maybe, a little more guidance for students may be needed to help understand the test. For example, how the test work, what the test assert and check etc.

7. At first I found the approach a little strange. Before, I had to be the one who was manually doing the testing. However, after I got used to the change, I found it a lot easier to test out my code. I no longer had to think of obscure examples that may

break my program. Most cases were taken care of with the tests that we should have passed.

8. In my opinion, unit testing allowed us to work on more challenging problems and dive deeper into understanding the course material. It allowed us to not only understand the algorithms and data structures broadly, but also allowed us to understand quirks about them because the tests pointed out the quirks to us. Additionally, I felt more confident as a student having something verifiable to let me know that I was on the right track or that I had gotten it. It was great and I wish something like it were in all of my classes.

## 4 Conclusion

The proposed approach helps bridge the gap between academia and industry, promotes consistency of students' code, and decreases grading time. Using testing framework like *pytest* allows students to test their program multiple times before submitting it and reduces the amount of uncertainty in the grading process.

One of the advantages of this setup is that individual components can be introduced into the course sequence separately. For example, students in CS1 may not be ready to dive into version control to submit their code but they could execute `git clone` and `git pull` commands to download class notes and code templates. If the full-featured unit testing framework is beyond the scope of a course, Python's `assert` statements could be used to make a hypothesis and check if code is correct.

## 5 Acknowledgments

This work originated as a student research project supported by a grant from the Dean of Luther College.

We thank students of the *Advanced Algorithms and Data Structures* class for their feedback.

## References

[1] Joel Adams. "Test-Driven Data Structures: Revitalizing CS2". In: *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*. SIGCSE '09. Chattanooga, TN, USA: Association for Computing Machinery, 2009, pp. 143–147. ISBN: 9781605581835. DOI: 10.1145/1508865.1508920. URL: https://doi.org/10.1145/1508865.1508920.

[2] Stephen H. Edwards. “Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action”. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’04. Norfolk, Virginia, USA: Association for Computing Machinery, 2004, pp. 26–30. ISBN: 1581137982. DOI: `10.1145/971300.971312`. URL: `https://doi.org/10.1145/971300.971312`.

[3] *Python 2019 - The state of Developer Ecosystem in 2019 Infographic*. `https://www.jetbrains.com/lp/devecosystem-2019/python/`. (Accessed on 03/19/2020).

[4] *Python Developers Survey 2018 Results*. `https://www.jetbrains.com/research/python-developers-survey-2018/`. (Accessed on 03/19/2020).

[5] Dee A. B. Weikle, Michael O. Lam, and Michael S. Kirkpatrick. “Automating Systems Course Unit and Integration Testing: Experience Report”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 565–570. ISBN: 9781450358903. DOI: `10.1145/3287324.3287502`. URL: `https://doi.org/10.1145/3287324.3287502`.