Lab: Logistics

Goals

After successfully completing this assignment, you should be able to:

- accept GitHub Classroom invitations
- open an assignment repository in VSCode running in a Codespace (or on your computer)
- locate and work on assignment source files
- run included tests and and interpret their output
- commit and submit (push) your work

Overview

This assignment is primarily designed to help guide you through the steps you must follow to accept, work on, test, and submit assignments. You will follow the same steps for each subsequent assignment, so it's important that you understand them well --- be sure to ask for help if you're unclear on how to perform any step, or *why* you're doing it!

Accepting assignments

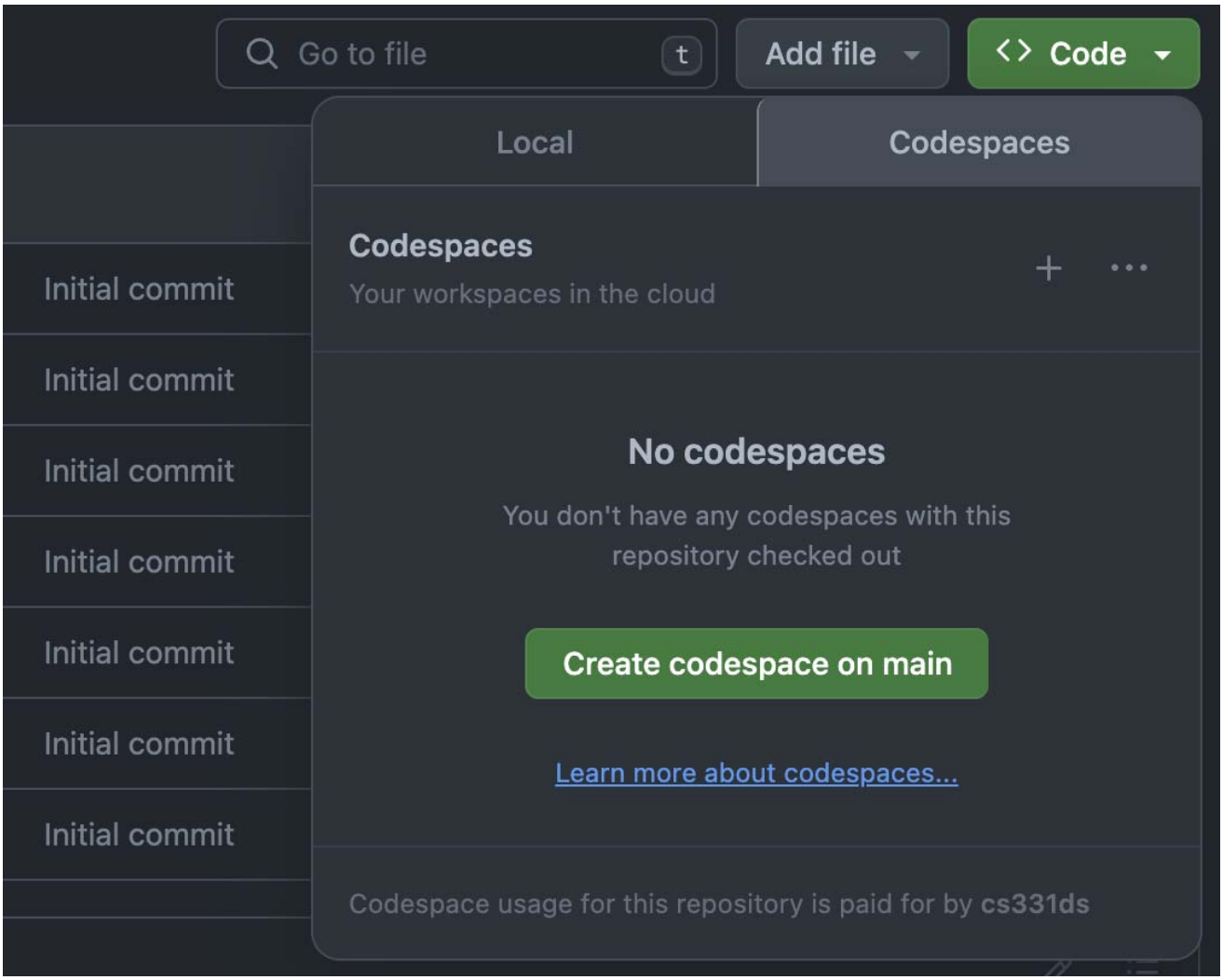
We will use a platform called GitHub Classroom to distribute and collect all assignments. The first step in accessing each assignment is to accept the corresponding GitHub Classroom invitation; you will find a separate invitation link for each assignment on the class website. If you're reading this writeup, you already accepted the first assignment via a new or pre-existing account on GitHub. You should also have picked your Hawk username from a list --- you will not need to do this again, as GitHub Classroom will have permanently associated your GitHub account with your Hawk username.

After you accept the invitation, GitHub Classroom creates a copy (known as a *clone*) of the assignment files and places them in a private *Git repository* hosted on GitHub for you. Take some time to check out the homepage of the repository on GitHub --- its URL will look something like `https://github.com/cs331ds/labXX-USER` (where *XX* is the assignment number and *USER* is your GitHub username).

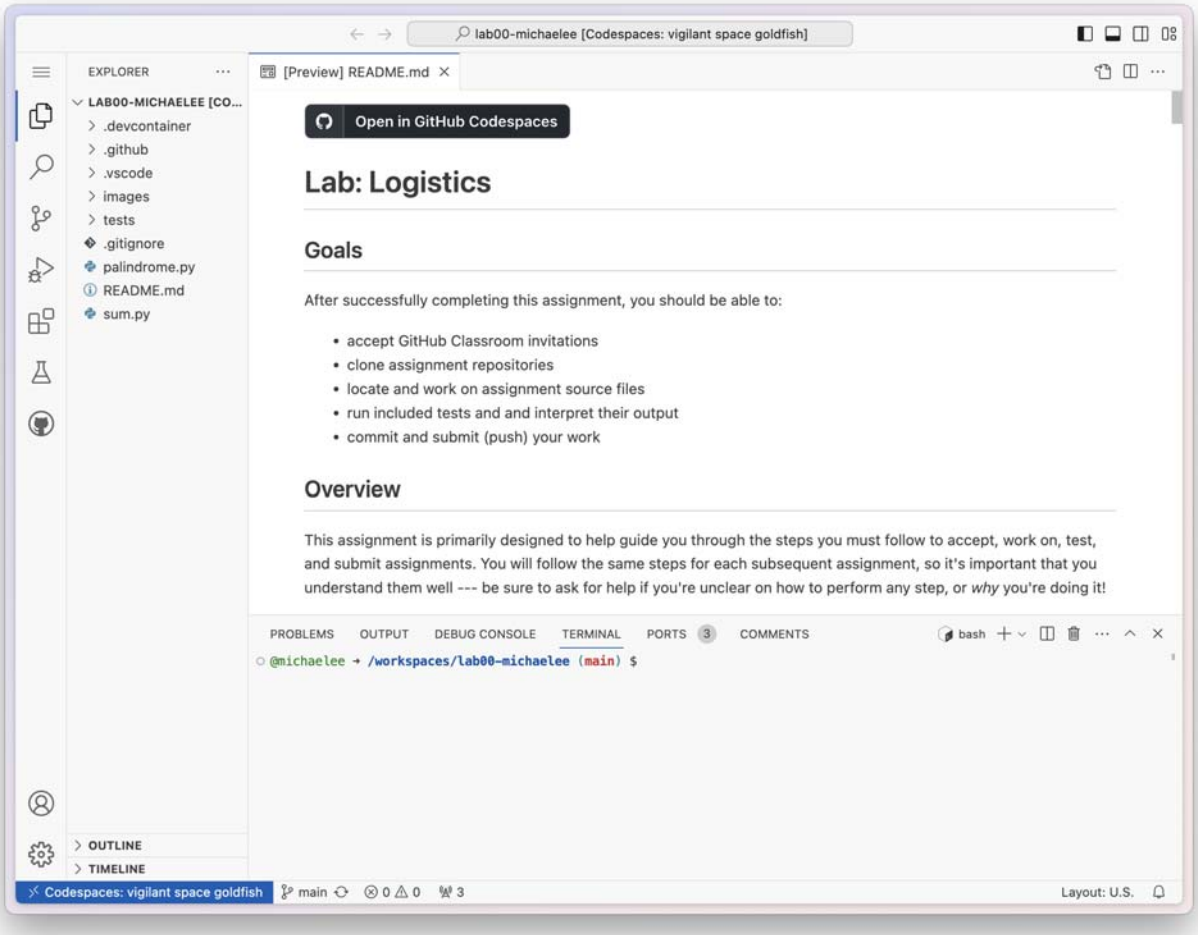
Up top is a list of all files found in the repository, and the contents of the `README.md` file --- which functions as the assignment writeup --- will be displayed under that list. The GitHub web interface allows you to edit files directly in the browser, but we *strongly recommend* that you do not do so. Instead, you should edit lab files in a properly configured development environment (e.g., VSCode). This will allow you to take advantage of features like syntax highlighting, code completion, and integrated testing. We'll walk you through how to do this in the next section.

Creating a Codespace and Opening VSCode

The easiest way to work on assignments is to use VSCode running in a Codespace. A Codespace is a cloud-based development environment that is automatically configured with all the tools you need to work on assignments. To open a Codespace, click on the green "Code" button on your repository homepage, navigate to the "Codespaces" tab (shown below), then click on "Create codespace on main".



This will start the process of creating a Codespace in a new browser tab; it will take a few minutes to complete. If you click the "View logs" link (when it appears), you can monitor the detailed progress. Once the Codespace is ready, the loading screen will be replaced by a VSCode window (depicted below) with a file explorer on the left and a terminal window on the bottom. The main pane should display the `README.md` file (this writeup).



After creating a codespace for a given assignment repository, you can subsequently re-open the same Codespace by clicking on the name of the Codespace in the "Codespaces" tab.

Assignment files

Every assignment repository will consist, at minimum, of the following files / file types (all of which you should see in the file explorer pane of VSCode)):

- `README.md` : This is the assignment writeup (you're reading the first lab's writeup right now). This will typically include specifications for all the exercises that make up the assignment, along with additional information, hints, references, etc.
- Source files: At the top-level (i.e., not contained in sub-directories) of the repository, you will find one or more Python source files (ending with the `.py` extension). You will edit these files to add your own code. Typically, **these are the only files you need to modify in each assignment.**

- Test files: Under the `test` sub-directory, we provide tests that will help you evaluate the correctness of your implementation(s). We'll go over how to run and interpret the output of these tests later.
- Configuration files: In the "hidden" directories `.devcontainer`, `.github`, `.vscode`, and others, we include files used to configure the codespace and tests. Feel free to open them, but **do not alter their contents!**

Each assignment consists of one or more exercises, each of which requires you to add code to corresponding source files. Consider, for instance, the following two exercise specifications (please resist the urge to complete them right away):

Exercise 1 (5 points)

Edit `palindrome.py` so that the function `my_palindrome` returns a **palindrome** of at least 3 letters in length. A palindrome is a string that reads the same backwards as forwards.

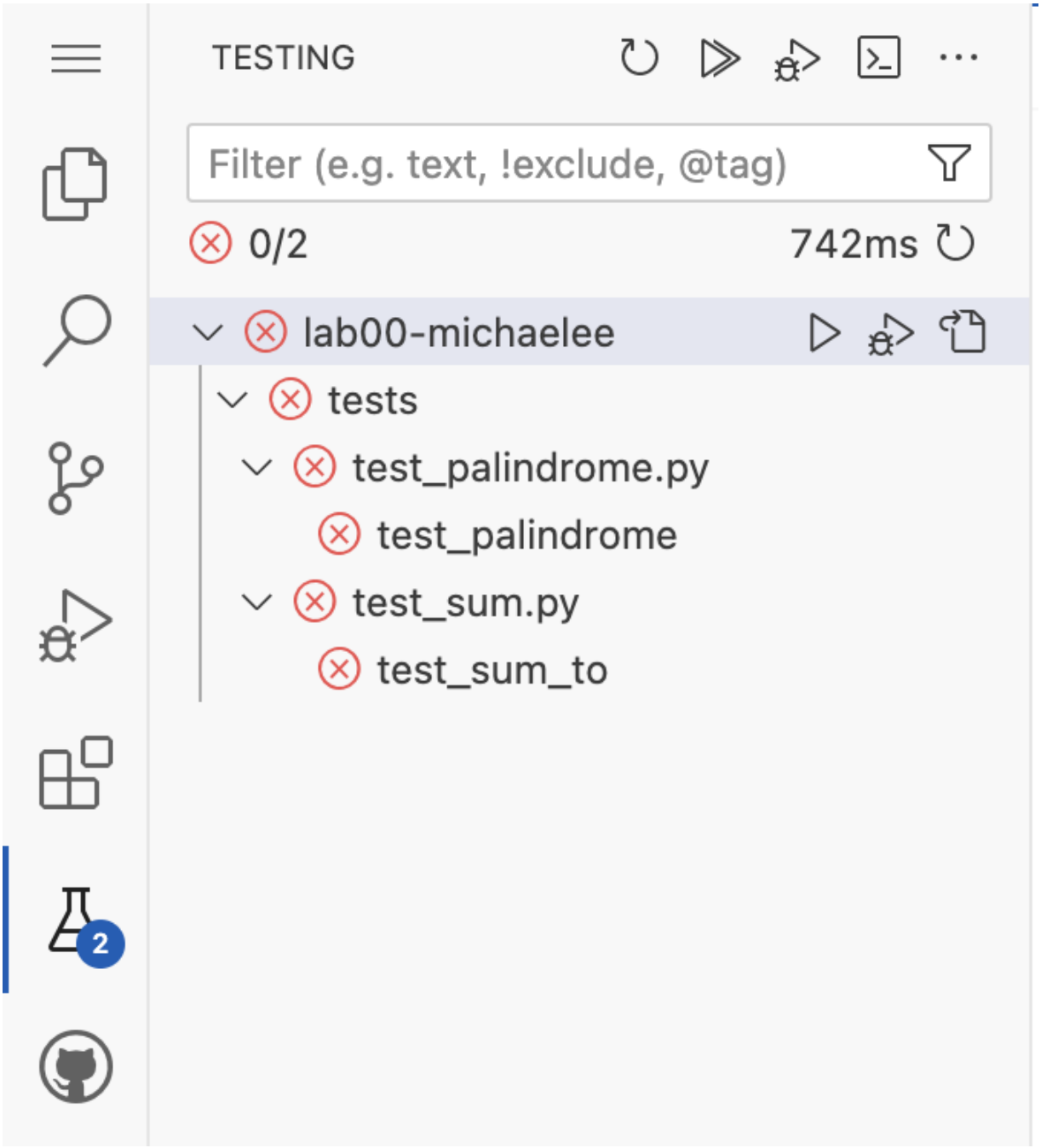
Exercise 2 (5 points)

Edit `sum.py` so that the function `sum_to`, which accepts an integer parameter k, returns the sum of the integers 0 up to k (inclusive). E.g., `sum_to(5)` should return the value $0 + 1 + 2 + 3 + 4 + 5 = 15$.

If you open the files `palindrome.py` and `sum.py`, you will find that the functions described in the exercises have already been defined for you, but with "stubs" for bodies (typically, just the keyword `pass`). The functions start with docstrings that specify their behavior, and often have type annotations. Instead of just duplicating specifications found in the source files, exercises will often just point you at which functions to implement and refer you to source file documentation.

Running tests

Before working on the exercises, let's see how to test them. In the "Testing" pane of the VSCode Side Bar (its icon looks like a test tube), you should find a list of all the tests that have been defined for you (you may need to expand the group). Click on the "Run Tests" icon (which looks like: ▶) at the top of the "Testing" pane to run all tests; you can also click on sub-groups to run just those tests. The tests should fail (since we haven't implemented anything yet) and be marked by ❌'s.



In the bottom pane, click on the "Test Results" tab to see the output of the tests. It should look something like this:

```
============================= test session starts ==============================
platform linux -- Python 3.12.1, pytest-7.4.4, pluggy-1.3.0
rootdir: /workspaces/lab00-michaelee
plugins: anyio-4.2.0
collected 2 items

tests/test_palindrome.py F                                               [ 50%]
tests/test_sum.py F                                                      [100%]

=================================== FAILURES ===================================
_____________________________ test_palindrome _____________________________
```

```
    @pytest.mark.points(5)
    def test_palindrome():
        pal = palindrome.my_palindrome()
>       assert isinstance(pal, str), 'Must return a string'
E       AssertionError: Must return a string
E       assert False
E        +  where False = isinstance(None, str)

tests/test_palindrome.py:7: AssertionError
_____________________________________ test_sum_to _____________________________________

    @pytest.mark.points(5)
    def test_sum_to():
>       assert sum.sum_to(5) == 15
E       assert None == 15
E        +  where None = <function sum_to at 0x7f01b2a55ee0>(5)
E        +    where <function sum_to at 0x7f01b2a55ee0> = sum.sum_to

tests/test_sum.py:6: AssertionError
============================== Grading Summary ==============================
Total Points: 0
=========================== short test summary info ============================
FAILED tests/test_palindrome.py::test_palindrome - AssertionError: Must retur...
FAILED tests/test_sum.py::test_sum_to - assert None == 15
============================== 2 failed in 0.04s ===============================
Finished running tests!
```

That's a lot of output! With a bit of practice, however, you should be able to hone in on the important information. In this case, they are:

1. Two tests failed (one for each exercise)
2. The `test_palindrome` test failed with the message `"Must return a string"`
3. The `test_sum_to` test failed the assertion `sum.sum_to(5) == 15`

Note that the report also shows you the relevant lines in the test functions that fail. Reading the test functions should give you a good idea of what the tests are checking for, and how your implementation should behave. For instance, the `test_palindrome` function, which is defined in `tests/test_palindrome.py`, looks like this:

```
def test_palindrome(self):
    pal = palindrome.my_palindrome()
    assert isinstance(pal, str), 'Must return a string'
    assert pal.strip() != '', 'String should not be empty'
    assert len(pal) >= 3, 'String should be at least 3 letters long'
    assert pal == pal[::-1], 'String is not a palindrome'
```

The first assertion checks that the return value of `my_palindrome` is a string. The second assertion checks that the string is not empty. The third assertion checks that the string is at least 3 letters long. The fourth assertion checks that the string is a palindrome. If any of these assertions fail, the test fails.

You should **not** alter any of the tests! Doing so may mislead you into believing that you correctly solved an exercise, when in fact you just changed the test to accept your incorrect implementation. (It also won't work to sneakily "pass" tests by altering them, as we'll be using the original tests -- and possibly others -- on our end to grade your submissions.)

Committing your work

Ok, now go ahead and implement your solutions to the exercises! (If you're still not quite comfortable with Python syntax, you can check out the included sample solutions). As you work on your solutions, you can periodically run the tests to help you check your work. Once all the ❌'s turn into ✅'s, as shown below, you're done!

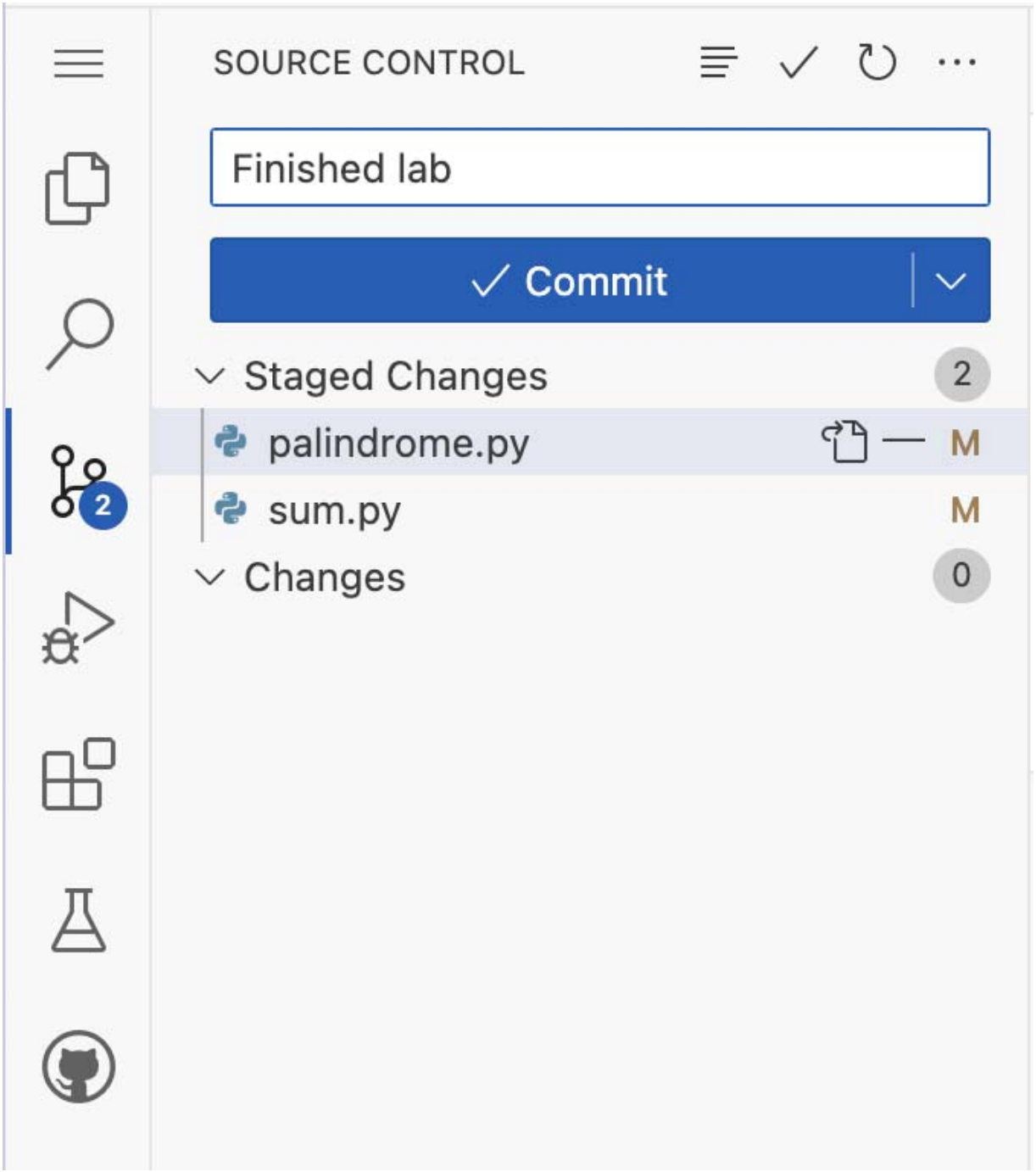
The test output should look like this:

```
============================= test session starts ==============================
platform linux -- Python 3.12.1, pytest-7.4.4, pluggy-1.3.0
rootdir: /workspaces/lab00-michaelee
plugins: anyio-4.2.0
collected 2 items

tests/test_palindrome.py .                                               [ 50%]
tests/test_sum.py .                                                      [100%]

============================== Grading Summary ===============================
Total Points: 10
============================== 2 passed in 0.01s ===============================
Finished running tests!
```

When you're satisfied with your work, you should *commit* and *push* your changes. What this does is save a snapshot of the updates you made to the code, and sync that snapshot to the GitHub repository. To do this, go to the "Source Control" pane in the VSCode Side Bar. You should see a list of all the files that have changed since the last commit. Hover over the `palindromes.py` and `sum.py` files and click the "+" icon to *stage* them for commit. Next, enter a commit message (something like "Finished lab" will do) and click on the blue "Commit" button.



After a successful commit, the blue "Commit" button should turn into a blue "Sync Changes" button. Click on it to push your recent commit(s) to GitHub.

As you work on future assignments, you should commit and push your changes frequently.

Summary

Here's a recap of the steps you need to follow to complete an assignment:

1. Accept the assignment invitation link found on the class website
2. Open the assignment repository in VSCode running in a Codespace
3. Refer to the writeup (`README.md`) for exercise specifications

4. Edit source files to solve the exercises
5. Run included tests to check your work
6. Commit and Sync your changes to GitHub (repeat steps 4-6 as needed)

Solutions

Here are sample solutions to the exercises. We include them for this assignment because the focus is on logistics, and you may still be busy learning Python syntax. No sample solutions in future assignments!

palindrome.py

```
def my_palindrome() -> str:
    return 'racecar'
```

sum.py

```
def sum_to(k: int) -> int:
    res = 0
    for i in range(1, k+1): # note: ranges are exclusive of the end
        res += i
    return res
```