::page{title="Hands-On Lab: Running Test With Nose"}

Estimated time needed: 30 minutes

Welcome to the **Running Tests with Nose** lab. In this lab, you will learn how to use fundamental tools for running unit tests in Python.

Learning Objectives

After completing this lab, you will be able to:

- Install Nose, Pinocchio, and Coverage
- Run unit tests with unittest and Nose
- Produce color-coded test output
- Add coverage reports to your test output

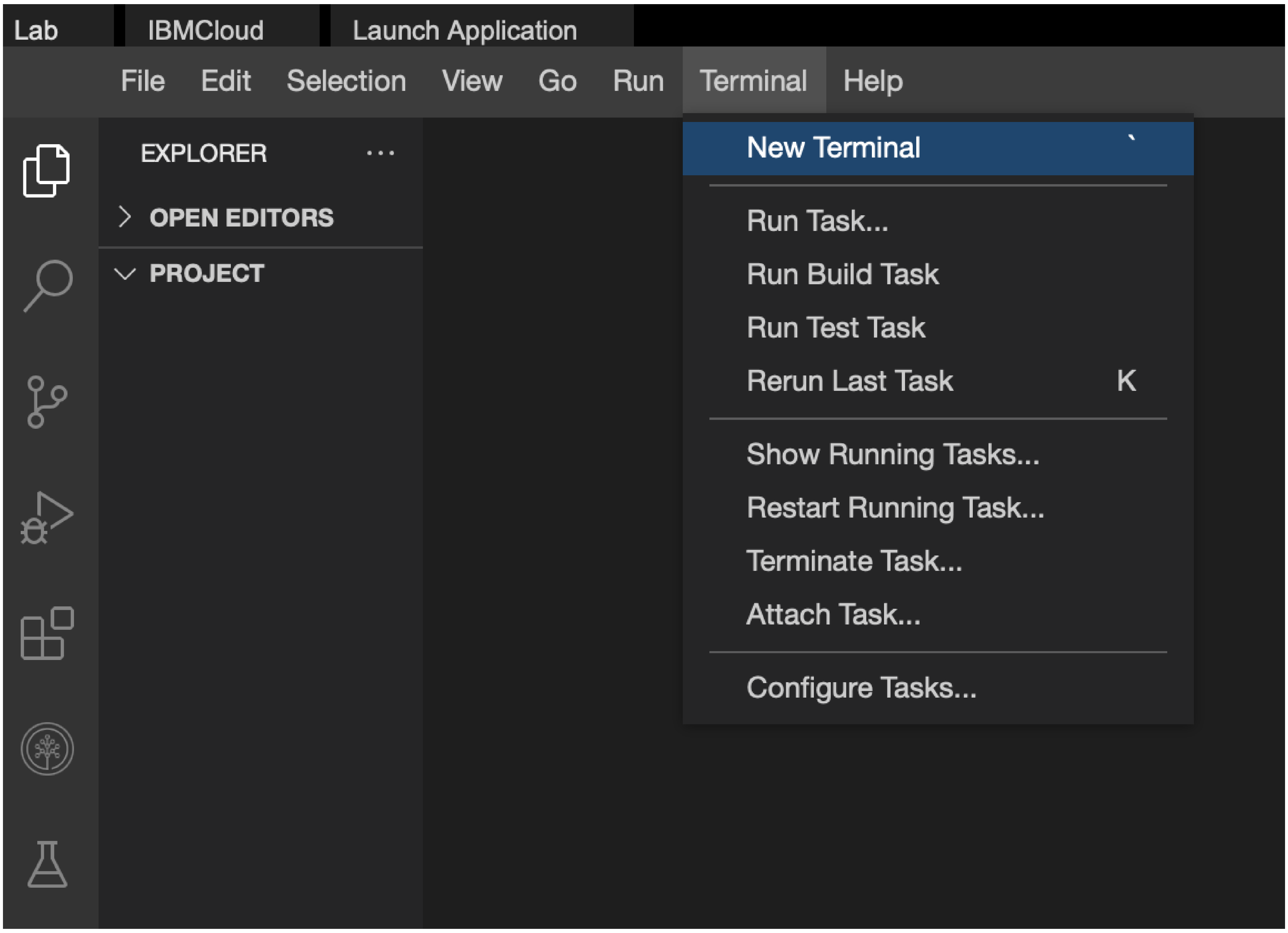
::page{title="About Theia"}

Theia is an open-source IDE (Integrated Development Environment), that can be run on desktop or on cloud. You will be using the Theia IDE to do this lab. When you log into the Theia environment, you are presented with a 'dedicated computer on the cloud' exclusively for you. This is available to you as long as you work on the labs. Once you log off, this 'dedicated computer on the cloud' is deleted along with any files you may have created. So, it is a good idea to finish your labs in a single session. If you finish part of the lab and return to the Theia lab later, you may have to start from the beginning. Plan to work out all your Theia labs when you have the time to finish the complete lab in a single session.

::page{title="Set Up the Lab Environment"}

You have a little preparation to do before you can start the lab.

Open a terminal window by using the menu in the editor: Terminal > New Terminal.



In the terminal, if you are not already in the `/home/projects` folder, change to your project folder now.

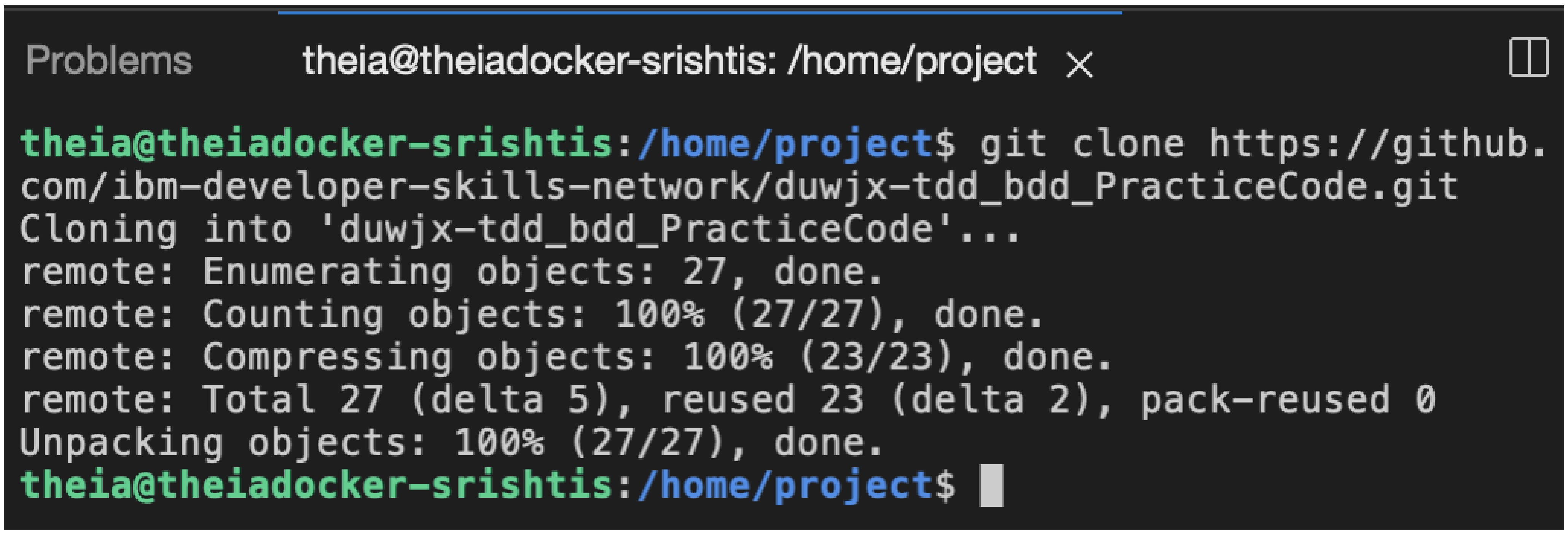
```
cd /home/project
```

Clone the Git Repository

Now let's get the code that you need to test. To do this, you will use the `git clone` command to clone the git repository:

```
git clone https://github.com/ibm-developer-skills-network/duwjx-tdd_bdd_PracticeCode.git
```

Your output should look similar to the image below:



```
theia@theiadocker-srishtis:/home/project$ git clone https://github.
com/ibm-developer-skills-network/duwjx-tdd_bdd_PracticeCode.git
Cloning into 'duwjx-tdd_bdd_PracticeCode'...
remote: Enumerating objects: 27, done.
remote: Counting objects: 100% (27/27), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 27 (delta 5), reused 23 (delta 2), pack-reused 0
Unpacking objects: 100% (27/27), done.
theia@theiadocker-srishtis:/home/project$ 
```

Change into the Lab Folder

Once you have cloned the repository, change to the directory named: `duwjx-tdd_bdd_PracticeCode`

```
cd duwjx-tdd_bdd_PracticeCode
```

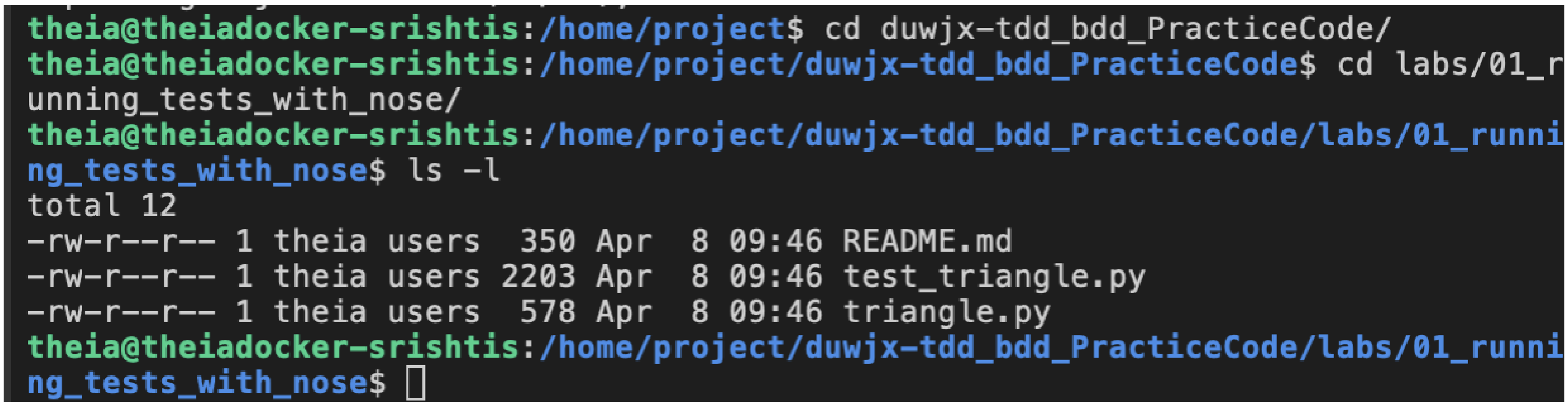
To go into the first set of labs, change into the `labs/01_running_tests_with_nose/` directory:

```
cd labs/01_running_tests_with_nose/
```

List the contents of this directory to see the artifacts for this lab.

```
ls -l
```

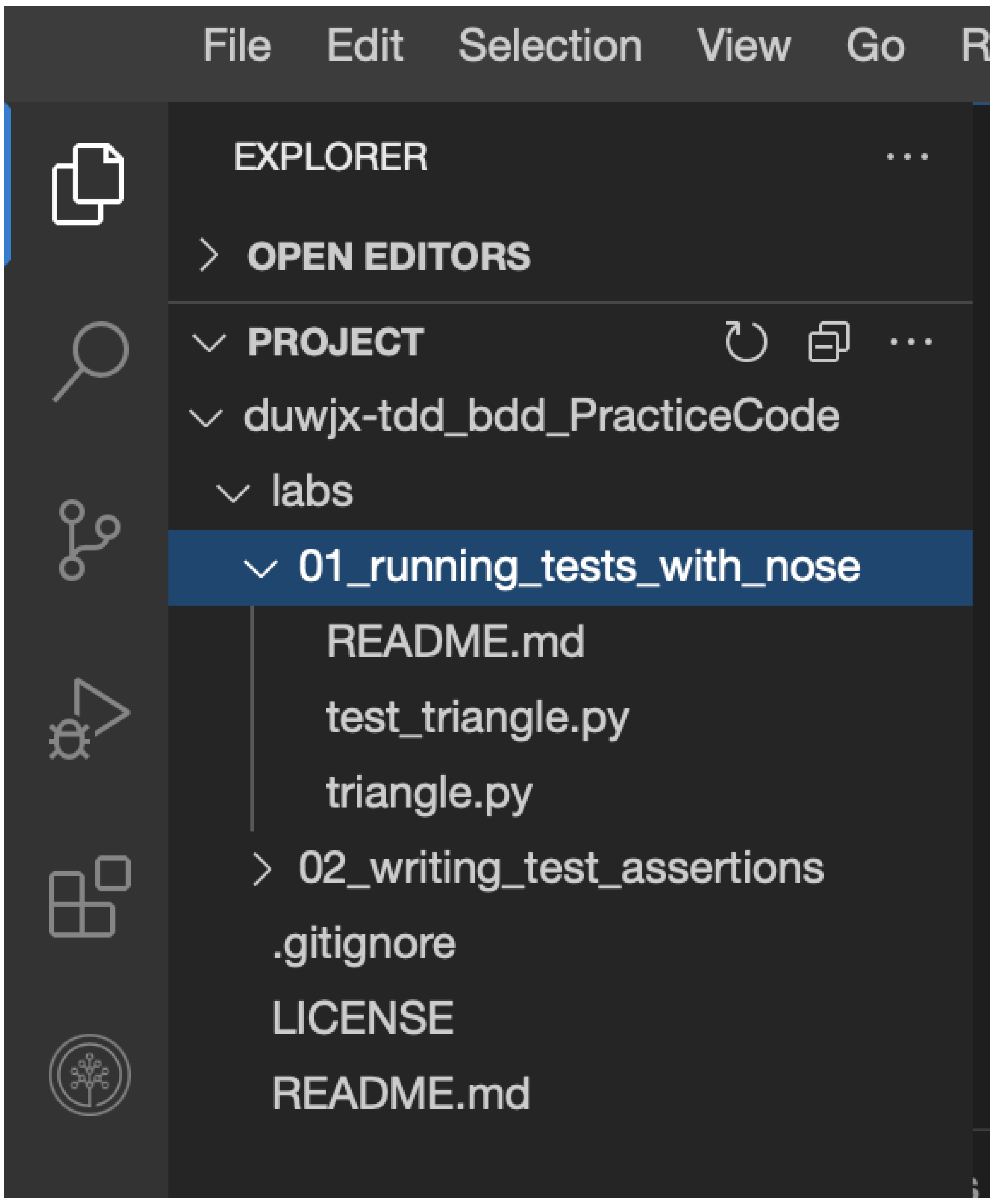
The directory should look like the listing below:



```
theia@theiadocker-srishtis:/home/project$ cd duwjx-tdd_bdd_PracticeCode/
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode$ cd labs/01_r
unning_tests_with_nose/
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_runni
ng_tests_with_nose$ ls -l
total 12
-rw-r--r-- 1 theia users  350 Apr  8 09:46 README.md
-rw-r--r-- 1 theia users 2203 Apr  8 09:46 test_triangle.py
-rw-r--r-- 1 theia users  578 Apr  8 09:46 triangle.py
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_runni
ng_tests_with_nose$ 
```

Note: *You must have a few exercise files that you will be running in the steps to follow.*

You can also view the files cloned in the file explorer.



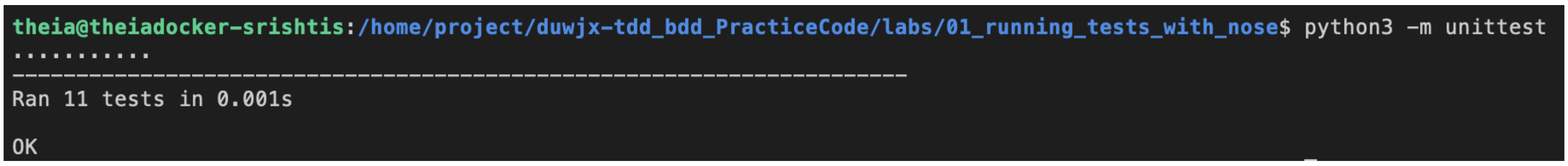
::page{title="Step 1: Working with unittest"}

You are ready to run your first tests. The `unittest` module is built into Python 3. You can invoke it through the Python interpreter by passing it the argument `-m` to run a module, and then `unittest` to give it the name of the module that you want to run.

In the terminal, run the following command to run the tests using the Python `unittest` package:

```
python3 -m unittest
```

In the output you will see that all 11 tests passed as indicated by the 11 dots. Had there been any errors, you would have seen the letter `E` in place of the dot, representing an error in that test.

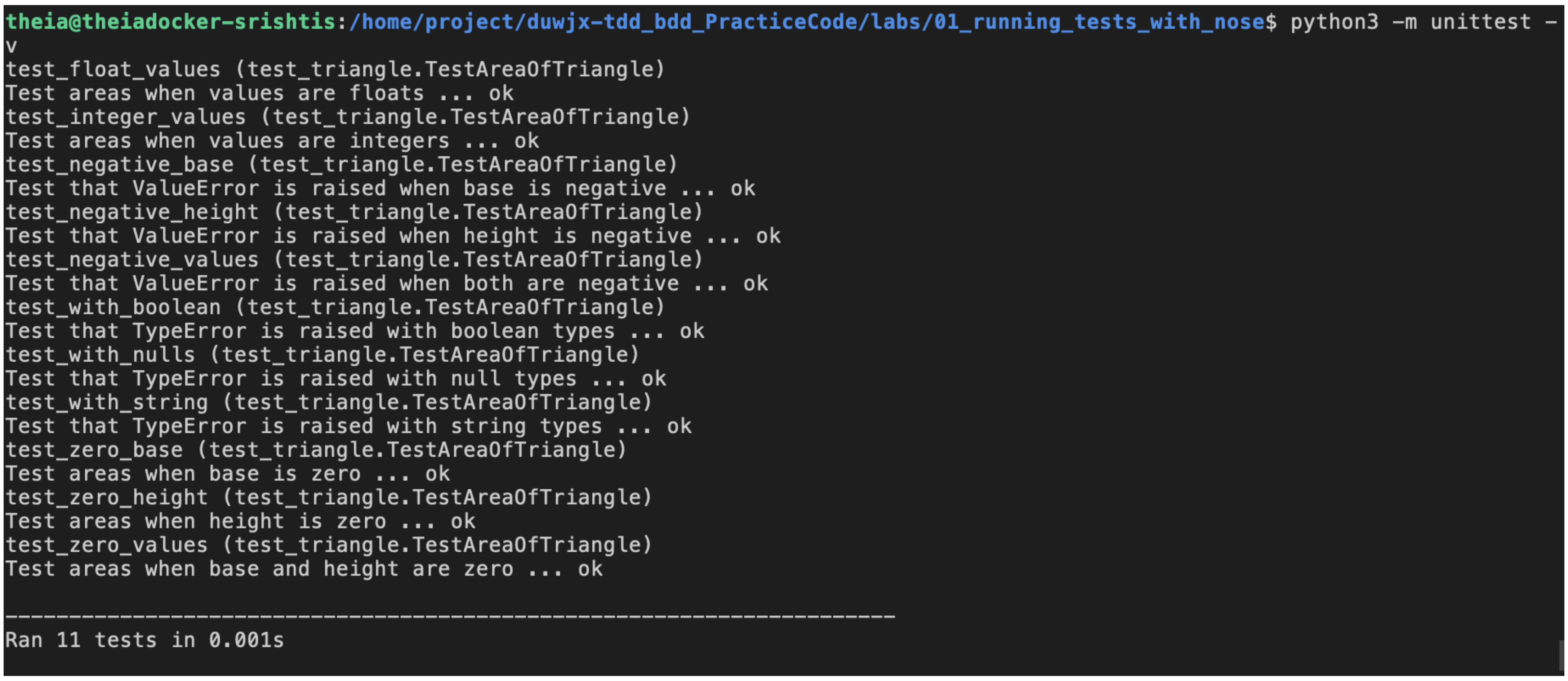


```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ python3 -m unittest
...........
----------------------------------------------------------------------
Ran 11 tests in 0.001s

OK
```

::page{title="More verbose output"}

You can make the output more useful by adding the `-v` flag to turn on verbose mode. Run the command below for more useful output:

```
python3 -m unittest -v
```



```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ python3 -m unittest -
v
test_float_values (test_triangle.TestAreaOfTriangle)
Test areas when values are floats ... ok
test_integer_values (test_triangle.TestAreaOfTriangle)
Test areas when values are integers ... ok
test_negative_base (test_triangle.TestAreaOfTriangle)
Test that ValueError is raised when base is negative ... ok
test_negative_height (test_triangle.TestAreaOfTriangle)
Test that ValueError is raised when height is negative ... ok
test_negative_values (test_triangle.TestAreaOfTriangle)
Test that ValueError is raised when both are negative ... ok
test_with_boolean (test_triangle.TestAreaOfTriangle)
Test that TypeError is raised with boolean types ... ok
test_with_nulls (test_triangle.TestAreaOfTriangle)
Test that TypeError is raised with null types ... ok
test_with_string (test_triangle.TestAreaOfTriangle)
Test that TypeError is raised with string types ... ok
test_zero_base (test_triangle.TestAreaOfTriangle)
Test areas when base is zero ... ok
test_zero_height (test_triangle.TestAreaOfTriangle)
Test areas when height is zero ... ok
test_zero_values (test_triangle.TestAreaOfTriangle)
Test areas when base and height are zero ... ok

----------------------------------------------------------------------
Ran 11 tests in 0.001s
```

Also notice that verbose mode gives you lots of duplicate information like the test function name followed by the docstring. Next, you will see how `nose` handles this.

::page{title="Step 2: Working with Nose"}

There is a test runner called **nose** that you can use to produce better test output. It is a Python package that you can install using the Python package manager `pip` utility.

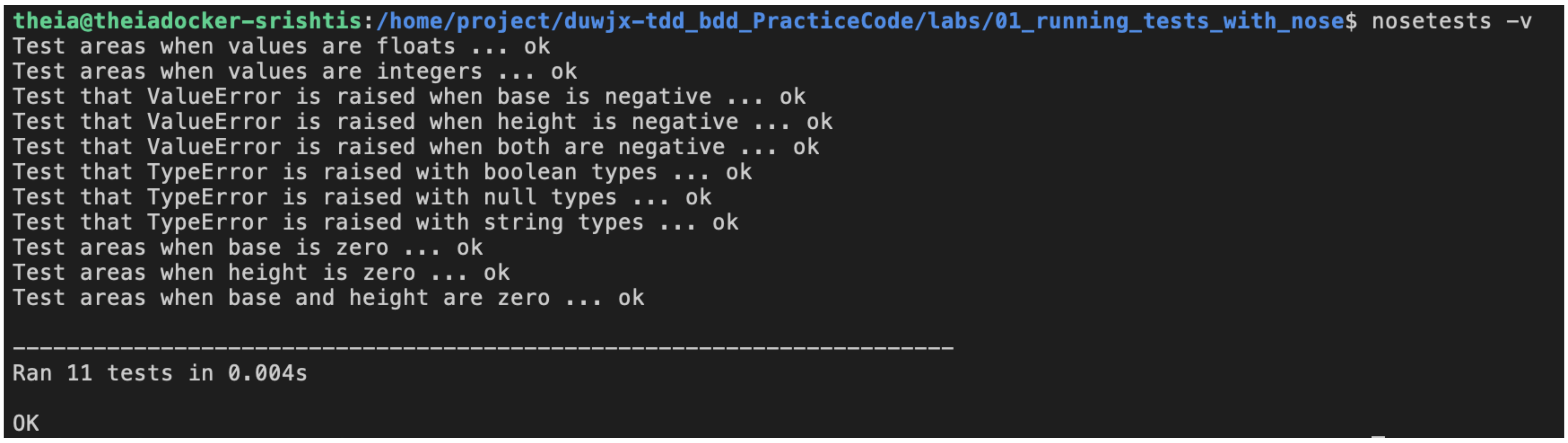
Install Nose using `pip`:

```
pip install nose
```

> ***Note:*** *To refresh your memory on Nose, review the "Running Tests with Nose" video.*

To see verbose output from `nose`, run `nosetests -v`. The verbose output from Nose will return nicer output than from unittest because it only returns the docstring comments:

```
nosetests -v
```



```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ nosetests -v
Test areas when values are floats ... ok
Test areas when values are integers ... ok
Test that ValueError is raised when base is negative ... ok
Test that ValueError is raised when height is negative ... ok
Test that ValueError is raised when both are negative ... ok
Test that TypeError is raised with boolean types ... ok
Test that TypeError is raised with null types ... ok
Test that TypeError is raised with string types ... ok
Test areas when base is zero ... ok
Test areas when height is zero ... ok
Test areas when base and height are zero ... ok

----------------------------------------------------------------------
Ran 11 tests in 0.004s

OK
```

::page{title="Step 3: Adding color with Pinocchio"}

Another way to make your output look better is with a plugin called `pinocchio`. With this plugin, you can get output as a specification similar to **Rspec** and also add color to the output. The color really gives you the Red/Green/Refactor workflow that TDD is famous for.

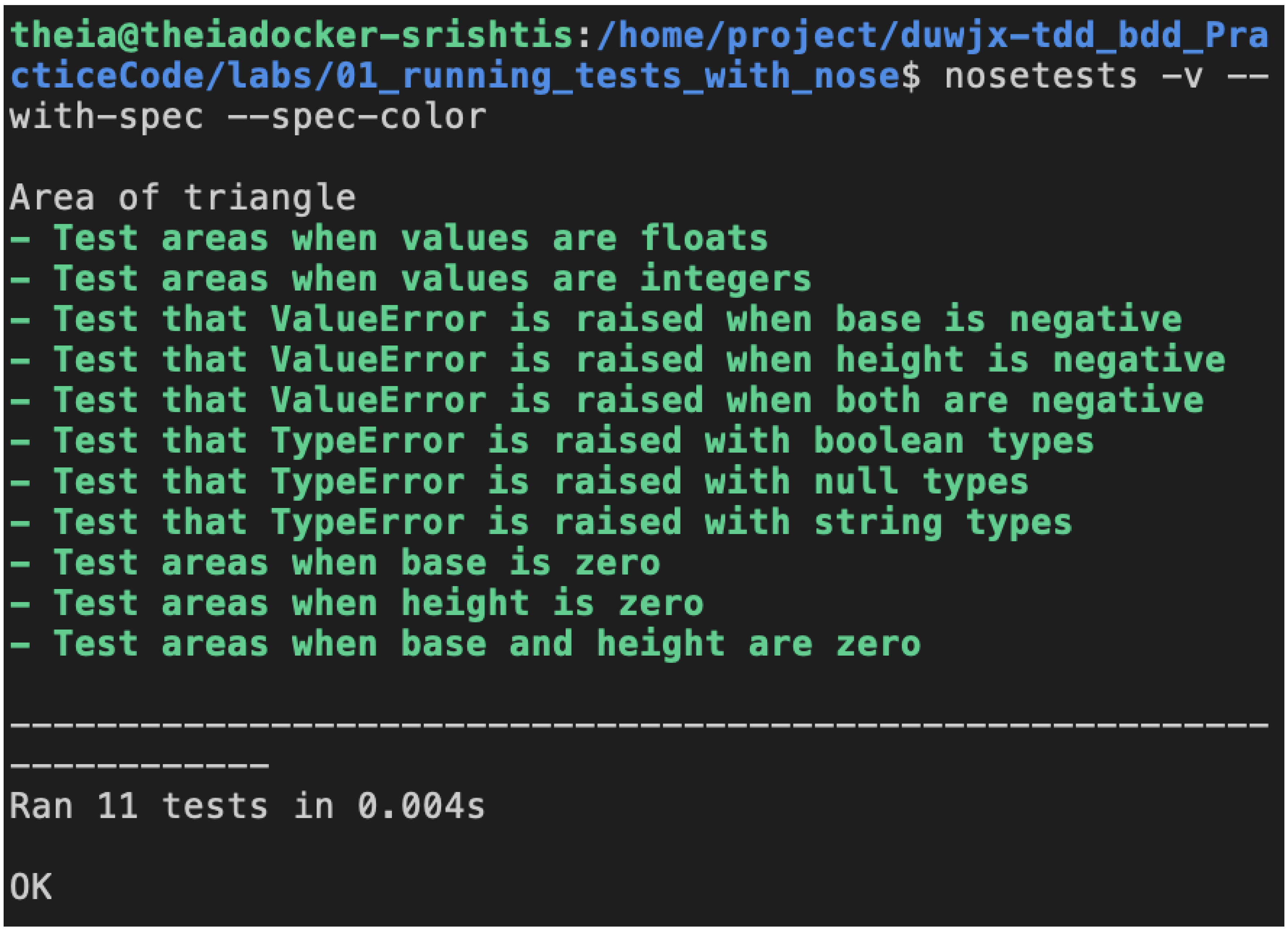
Install `pinocchio` using `pip`.

```
pip install pinocchio
```

To get nicer formatting and a colorful output, run `nose` again and add the `--with-spec --spec-color` parameters:

```
nosetests --with-spec --spec-color
```

> Note: In the output, green color indicates that all tests have passed. In case any test fails, the color for that test in the output will be red. Also note that you no longer need the `-v` because `--with-spec` already gives verbose output.



```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ nosetests -v --with-spec --spec-color

Area of triangle
- Test areas when values are floats
- Test areas when values are integers
- Test that ValueError is raised when base is negative
- Test that ValueError is raised when height is negative
- Test that ValueError is raised when both are negative
- Test that TypeError is raised with boolean types
- Test that TypeError is raised with null types
- Test that TypeError is raised with string types
- Test areas when base is zero
- Test areas when height is zero
- Test areas when base and height are zero

----------------------------------------------------------------------
Ran 11 tests in 0.004s

OK
```

::page{title="Step 4: Adding test coverage"}

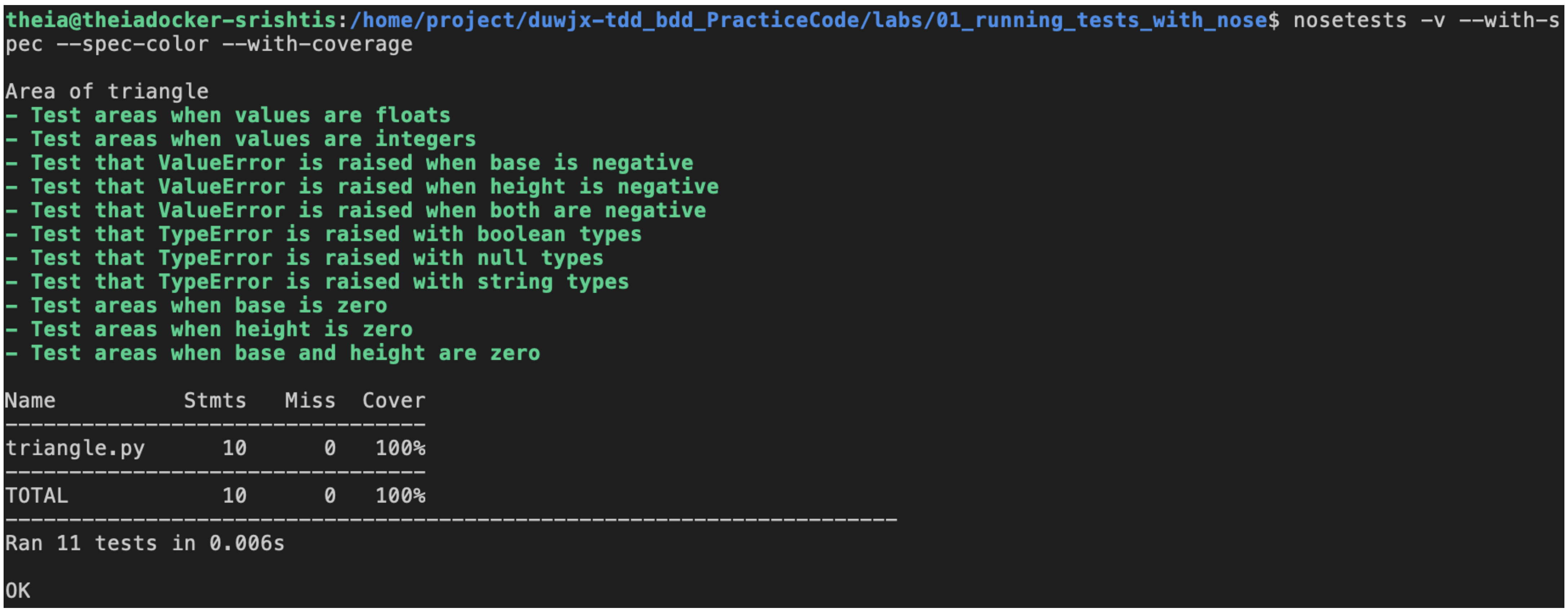
To know if you've written enough tests, you need to know how many lines of code your tests cover. The `coverage` tool will calculate the number of lines of code executed during your tests, against the total lines of code, and report that as a percentage of coverage.

Install the `coverage` tool so that you can check your test coverage:

```
pip install coverage
```

Next, call `coverage` through `nose` by adding the `--with-coverage` parameter.

```
nosetests --with-spec --spec-color --with-coverage
```



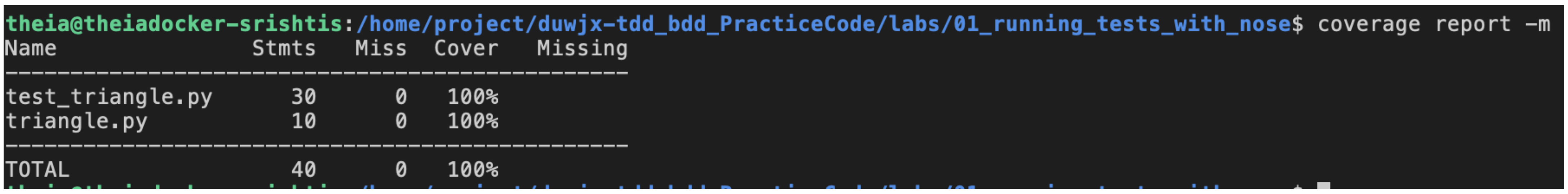
```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ nosetests -v --with-spec --spec-color --with-coverage

Area of triangle
- Test areas when values are floats
- Test areas when values are integers
- Test that ValueError is raised when base is negative
- Test that ValueError is raised when height is negative
- Test that ValueError is raised when both are negative
- Test that TypeError is raised with boolean types
- Test that TypeError is raised with null types
- Test that TypeError is raised with string types
- Test areas when base is zero
- Test areas when height is zero
- Test areas when base and height are zero

Name           Stmts   Miss  Cover
-----------------------------------
triangle.py       10      0   100%
-----------------------------------
TOTAL             10      0   100%
----------------------------------------------------------------------
Ran 11 tests in 0.006s

OK
```

::page{title="Step 5: Create missing coverage report"}

One useful feature of the coverage tool is that it can report which lines of code are missing coverage. With that information, you know the lines for which you need to add more test cases so that your testing executes those missing lines of code.

To get the missing coverage report, run the below command in the terminal:

```
coverage report -m
```



```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ coverage report -m
Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
test_triangle.py        30      0   100%
triangle.py             10      0   100%
--------------------------------------------------
TOTAL                   40      0   100%
```

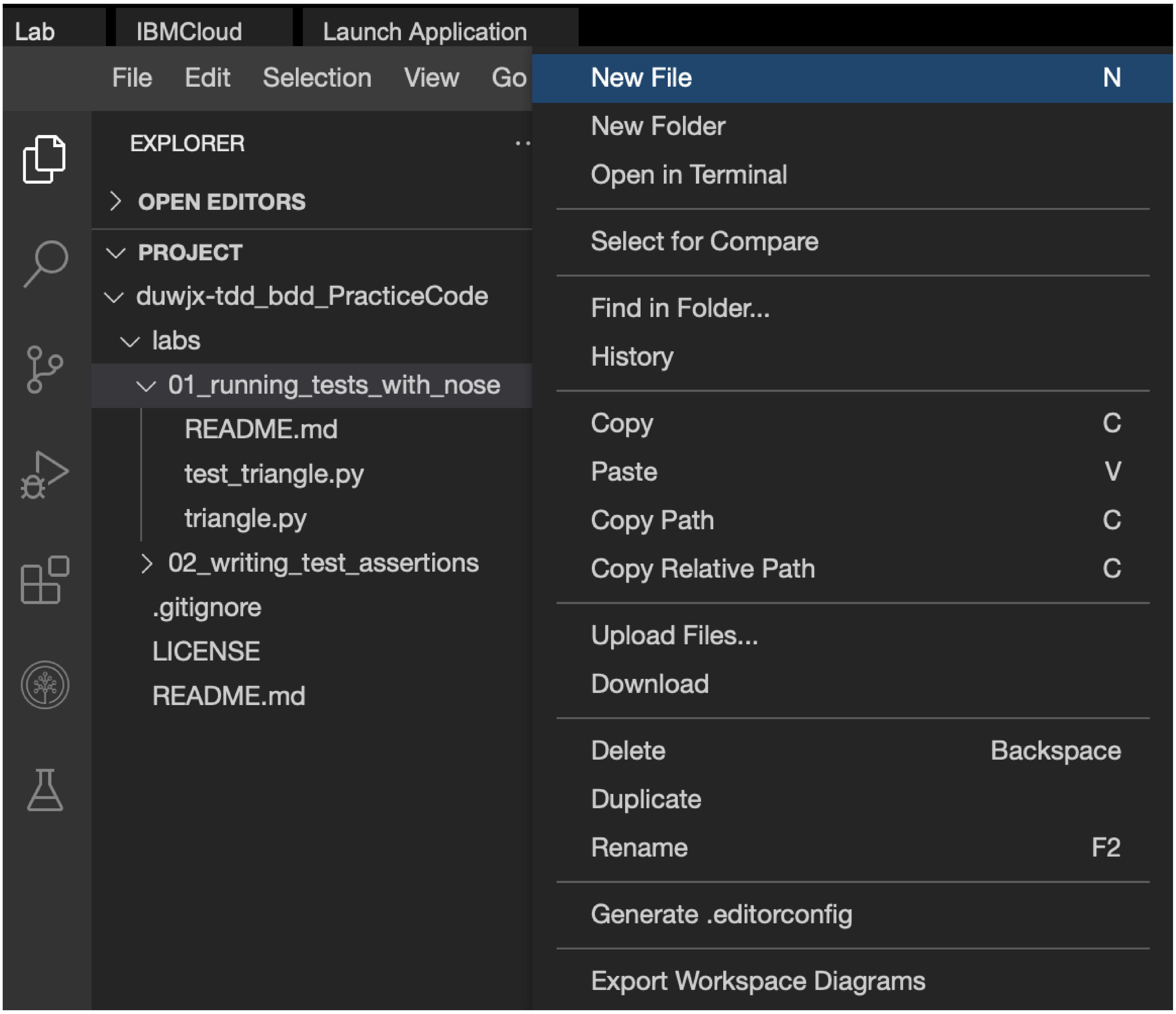
These test cases produce 100% coverage but notice that a new column has been added with the **Missing** heading. This is where any line numbers would show up for lines without test coverage.

::page{title="Step 6: Automating these parameters"}

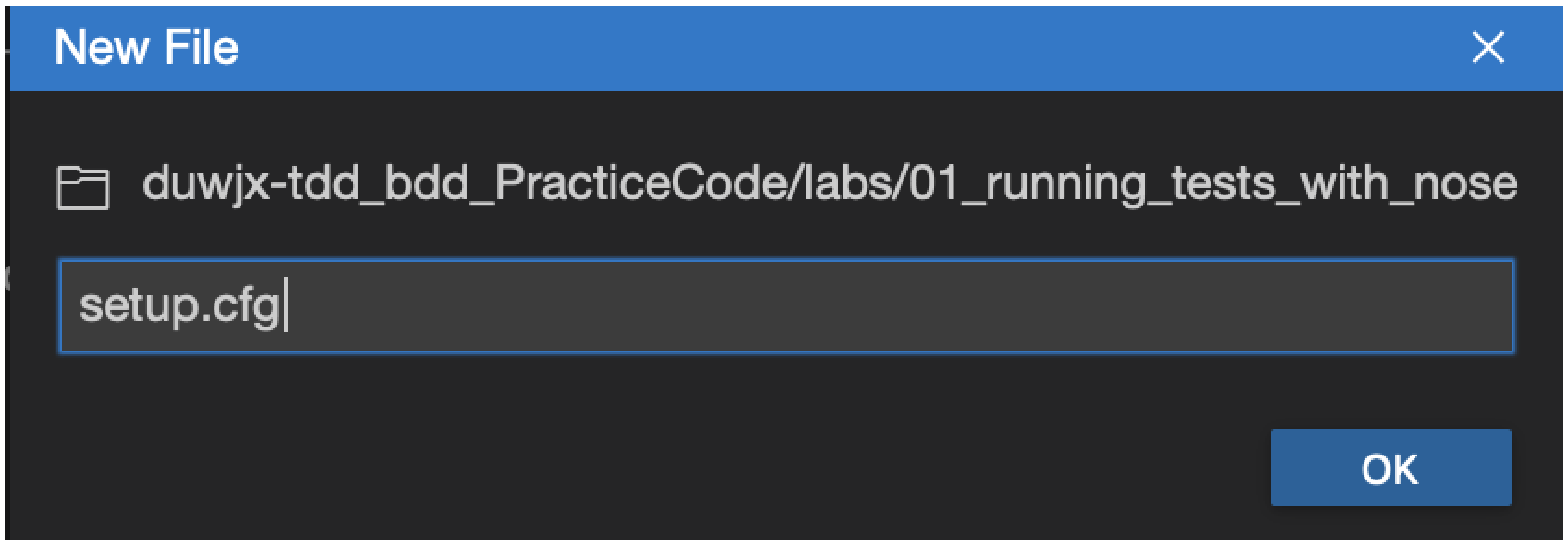
Up until now you have typed out a lot of command parameters when running tests with nose. Alternatively, you can save all the parameters in a configuration file so that you don't have to type them in every time.

Create a new file named `setup.cfg` under `duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose` directory. Here are the steps:

1. On the window to the right, click the File menu and select the New File option, as shown in the image below:



2. A pop-up appears with title New File, as shown in the image below. Enter `setup.cfg` as the file name and then click OK.



```
setup.cfg
```

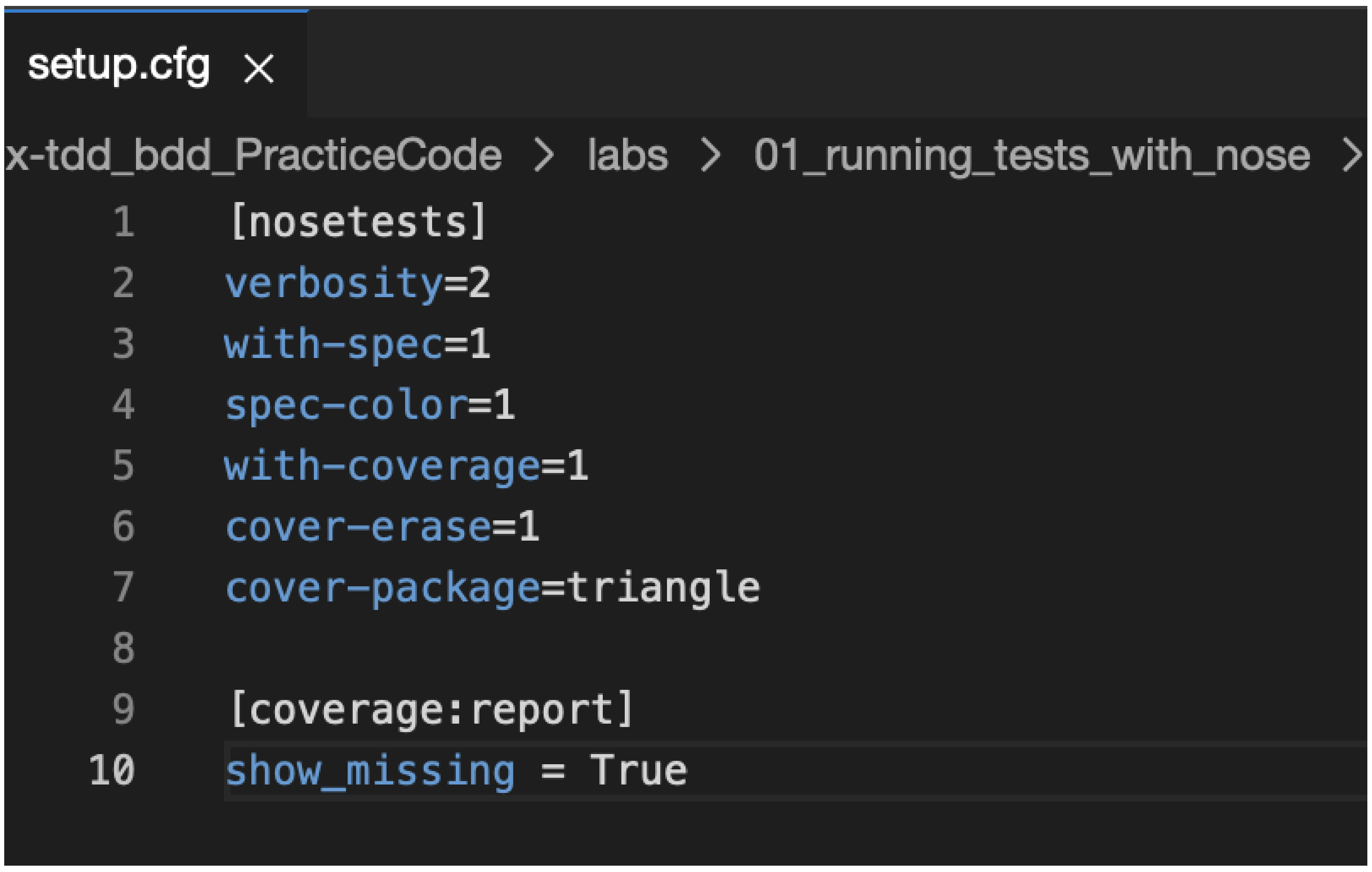
3. A file `setup.cfg` will be created for you.

::openFile{path="/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose/setup.cfg"}

4. Copy and paste the below code into `setup.cfg`:

```
[nosetests]
verbosity=2
with-spec=1
spec-color=1
with-coverage=1
cover-erase=1
cover-package=triangle

[coverage:report]
show_missing = True
```

> Do not forget to save the setup.cfg file.


setup.cfg
x-tdd_bdd_PracticeCode > labs > 01_running_tests_with_nose >

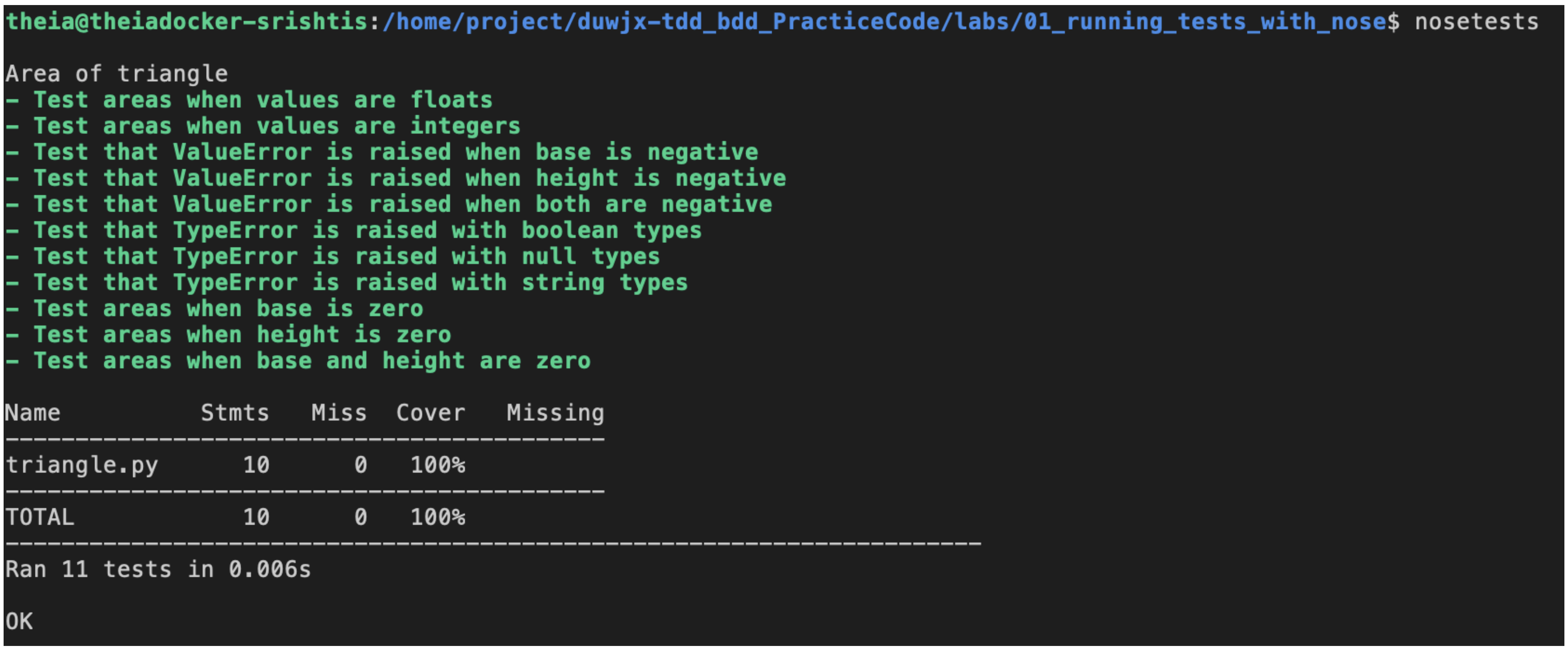
```
[nosetests]
verbosity=2
with-spec=1
spec-color=1
with-coverage=1
cover-erase=1
cover-package=triangle

[coverage:report]
show_missing = True
```

::page{title="Step 7: Run nosetests with config"}

Now that you have established your `setup.cfg` file, go to your terminal and run `nosetests` without any parameters:

```
nosetests
```

> Now you've applied all your parameters and gotten colorful output, by simply running **nosetests**.


```
theia@theiadocker-srishtis:/home/project/duwjx-tdd_bdd_PracticeCode/labs/01_running_tests_with_nose$ nosetests

Area of triangle
- Test areas when values are floats
- Test areas when values are integers
- Test that ValueError is raised when base is negative
- Test that ValueError is raised when height is negative
- Test that ValueError is raised when both are negative
- Test that TypeError is raised with boolean types
- Test that TypeError is raised with null types
- Test that TypeError is raised with string types
- Test areas when base is zero
- Test areas when height is zero
- Test areas when base and height are zero

Name           Stmts   Miss  Cover   Missing
--------------------------------------------
triangle.py       10      0   100%
--------------------------------------------
TOTAL             10      0   100%
----------------------------------------------------------------------
Ran 11 tests in 0.006s

OK
```

::page{title="Conclusion"}

Congratulations on Completing the Lab on Running Tests with Nose

You now know how to run basic and verbose unit tests with Nose. You also know how to use the Pinocchio and Coverage plugins to produce nicer test output that really brings the Red/Green/Refactor workflow to life. Feel free to play around with these commands some more or move on to the next lesson.

Now you should use these tools in your own projects to produce actionable test cases reports.

Author(s)

John Rofrano

Contributor(s)

Srishti Srivastava

Changelog

Date	Version	Changed by	Change Description
2022-04-08	1.0	Srishti	Create new lab
2022-04-17	1.1	Rofrano	Fixed image links
2022-04-17	1.2	Zach Rash	Proofreading and edits