How to organize a growing set of tests?

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

What you've done so far

16

2. What you've done so far
You wrote about 16 unit tests for the functions row_to_list(), convert_to_int(), get_data_as_numpy_array() and split_into_training_and_testing_sets() in Chapter 1 and 2. Well done!

- `row_to_list()`
- `convert_to_int()`
- `get_data_as_numpy_array()`
- `split_into_training_and_testing_sets()`

What you've done so far

32

- `row_to_list()`
- `convert_to_int()`
- `get_data_as_numpy_array()`
- `split_into_training_and_testing_sets()`
- `...`

What you've done so far

64

- `row_to_list()`
- `convert_to_int()`
- `get_data_as_numpy_array()`
- `split_into_training_and_testing_sets()`
- `...`

What you've done so far

128

- `row_to_list()`
- `convert_to_int()`
- `get_data_as_numpy_array()`
- `split_into_training_and_testing_sets()`
- `...`

Need a strategy to organize tests

128

Need a strategy to organize tests

Project structure

```
src/                                    # All application code lives here
```

Project structure

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
    |-- __init__.py
```

Project structure

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
    |-- __init__.py
    |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
```

Project structure

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
    |-- __init__.py
```

Project structure

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
    |-- __init__.py
    |-- as_numpy.py                     # Contains get_data_as_numpy_array()
```

Project structure

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
|   |-- __init__.py
|   |-- as_numpy.py                     # Contains get_data_as_numpy_array()
|-- models/                             # Package for training and testing linear regression model
    |-- __init__.py
```

Project structure

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
|   |-- __init__.py
|   |-- as_numpy.py                     # Contains get_data_as_numpy_array()
|-- models/                             # Package for training/testing linear regression model
    |-- __init__.py
    |-- train.py                        # Contains split_into_training_and_testing_sets()
```

8. Project structure
Assume that the four functions that you tested are present in the following project structure. There's a top level directory called src, which holds all application code. Inside, there's the data package. This package deals with functions that preprocess data. It has a Python module called preprocessing_helpers.py, containing the functions row_to_list() and convert_to_int(). Then there's the features package, which deals with extracting features from the preprocessed data. It has a module called as_numpy.py, containing the function get_data_as_numpy_array().

13. Project structure
Finally, there's the models package, which deals with training and testing the linear regression model.

The tests folder

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
|   |-- __init__.py
|   |-- as_numpy.py                     # Contains get_data_as_numpy_array()
|-- models/                             # Package for training/testing linear regression model
|   |-- __init__.py
|   |-- train.py                        # Contains split_into_training_and_testing_sets()
tests/                                  # Test suite: all tests live here
```

The tests folder mirrors the application folder

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
|   |-- __init__.py
|   |-- as_numpy.py                     # Contains get_data_as_numpy_array()
|-- models/                             # Package for training/testing linear regression model
|   |-- __init__.py
|   |-- train.py                        # Contains split_into_training_and_testing_sets()
tests/                                  # Test suite: all tests live here
|-- data/
|   |-- __init__.py
|-- features/
|   |-- __init__.py
|-- models/
    |-- __init__.py
```

16. The tests folder mirrors the application folder
Inside this folder, we simply mirror the inner structure of src and create empty packages called data, features and models respectively.

Python module and test module correspondence

- `my_module.py` $\Longleftrightarrow$ `test_my_module.py`.

```
src/                                    # All application code lives here
|-- data/                               # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py        # Contains row_to_list(), convert_to_int()
|-- features/                           # Package for feature generation from preprocessed data
|   |-- __init__.py
|   |-- as_numpy.py                     # Contains get_data_as_numpy_array()
|-- models/                             # Package for training/testing linear regression model
|   |-- __init__.py
|   |-- train.py                        # Contains split_into_training_and_testing_sets()
tests/                                  # Test suite: all tests live here
|-- data/
|   |-- __init__.py
|   |-- test_preprocessing_helpers.py   # Corresponds to module src/data/preprocessing_helpers.py
|-- features/
|   |-- __init__.py
|-- models/
    |-- __init__.py
```

17. Python module and test module correspondence

The general rule is that for each python module my_module.py, there should be a corresponding test module called test_my_module.py. For example, for the module preprocessing_helpers.py, we create a test module called test_preprocessing_helpers.py. Since preprocessing_helpers.py belongs to the data package, we put the corresponding test module in the mirrored package inside the tests directory. The mirroring in the directory structure and test module names ensure that if we know where to find application code, we can follow the same route inside the test directory to access corresponding tests. The test module test_preprocessing_helpers.py should contain tests for row_to_list() and convert_to_int().

Structuring tests inside test modules

- Test module: `test_preprocessing_helpers.py`

```
import pytest
from data.preprocessing_helpers import row_to_list, convert_to_int

def test_on_no_tab_no_missing_value():    # A test for row_to_list()
    ...

def test_on_two_tabs_no_missing_value():  # Another test for row_to_list()
    ...

...

def test_with_no_comma():                 # A test for convert_to_int()
    ...

def test_with_one_comma():                # Another test for convert_to_int()
    ...

...
```

18. Structuring tests inside test modules
We could just put the tests sequentially like this, but this is an organizational nightmare, because there's no way to tell where the tests for one function ends and another function begins.

Test class

19. Test class
pytest solves this problem using a construct called the test class.
20. Test class is a container for a single unit's tests
A test class is just a simple container for tests of a specific function.
21. Test class: theoretical structure
To declare a test class, we start with the class keyword

Test class is a container for a single unit's tests

Test class: theoretical structure

- Test module: `test_preprocessing_helpers.py`

```
import pytest
from data.preprocessing_helpers import row_to_list, convert_to_int

class
```

19. Test class
pytest solves this problem using a construct called the test class.

20. Test class is a container for a single unit's tests
A test class is just a simple container for tests of a specific function.

21. Test class: theoretical structure
To declare a test class, we start with the class keyword and follow it up with the name of the class. The name of the class should be in CamelCase, and should always start with “Test”. The best way to name a test class is to follow the “Test” with the name of the function, for example, TestRowToList.

Test class: theoretical structure

- Test module: `test_preprocessing_helpers.py`

```
import pytest
from data.preprocessing_helpers import row_to_list, convert_to_int

class TestRowToList():                                  # Use CamelCase
```

23. Test class: theoretical structure
A test class takes one argument, and this argument is always called object. To know more about this argument, check out the DataCamp course on object-oriented Python, but we don't really need to for testing purposes, as we will never use this argument anywhere else. Now put all tests for the function under the test class as follows.

24. Test class: theoretical structure
Note that, this time, all tests should receive a single argument called self. This also comes from object-oriented Python.

Test class: theoretical structure

- Test module: `test_preprocessing_helpers.py`

```
import pytest
from data.preprocessing_helpers import row_to_list, convert_to_int

class TestRowToList(object):                        # Always put the argument object
    def test_on_no_tab_no_missing_value():          # A test for row_to_list()
        ...

    def test_on_two_tabs_no_missing_value():        # Another test for row_to_list()
        ...
```

24. Test class: theoretical structure
Note that, this time, all tests should receive a single argument called self. This also comes from object-oriented Python.

Test class: theoretical structure

- Test module: `test_preprocessing_helpers.py`

```
import pytest
from data.preprocessing_helpers import row_to_list, convert_to_int

class TestRowToList(object):                        # Always put the argument object
    def test_on_no_tab_no_missing_value(self):    # Always put the argument self
        ...

    def test_on_two_tabs_no_missing_value(self):  # Always put the argument self
        ...
```

Clean separation

- Test module: `test_preprocessing_helpers.py`

```
import pytest
from data.preprocessing_helpers import row_to_list, convert_to_int

class TestRowToList(object):                        # Always put the argument object
    def test_on_no_tab_no_missing_value(self):      # Always put the argument self
        ...

    def test_on_two_tabs_no_missing_value(self):    # Always put the argument self
        ...

class TestConvertToInt(object):                     # Test class for convert_to_int()
    def test_with_no_comma(self):                   # A test for convert_to_int()
        ...

    def test_with_one_comma(self):                  # Another test for convert_to_int()
        ...
```

24. Test class: theoretical structure
Note that, this time, all tests should receive a single argument called self. This also comes from object-oriented Python.

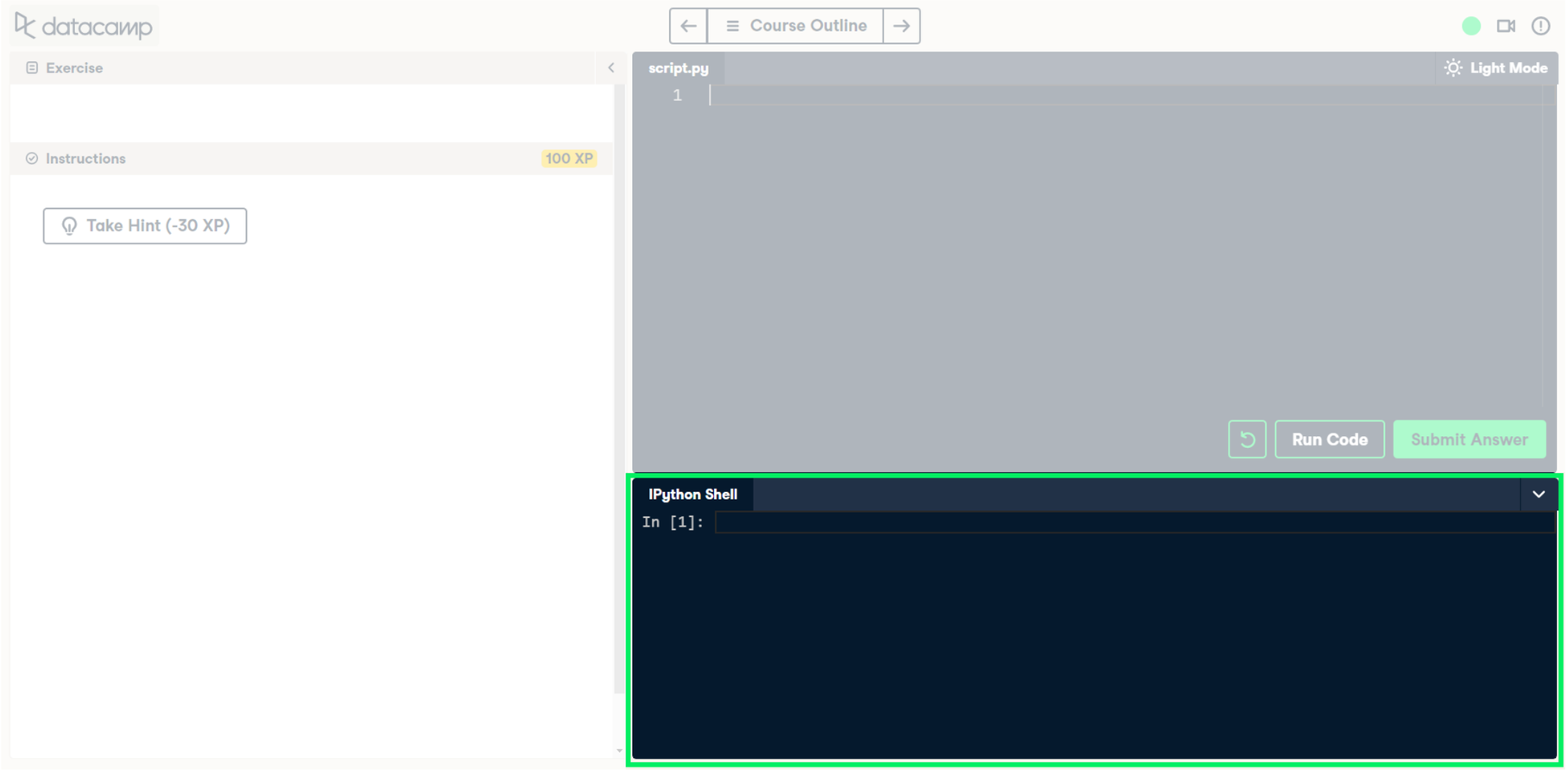
Final test directory structure

```
src/                                # All application code lives here
|-- data/                           # Package for data preprocessing
|   |-- __init__.py
|   |-- preprocessing_helpers.py    # Contains row_to_list(), convert_to_int()
|-- features/                       # Package for feature generation from preprocessed data
|   |-- __init__.py
|   |-- as_numpy.py                 # Contains get_data_as_numpy_array()
|-- models/                         # Package for training/testing linear regression model
|   |-- __init__.py
|   |-- train.py                    # Contains split_into_training_and_testing_sets()
tests/                              # Test suite: all tests live here
|-- data/
|   |-- __init__.py
|   |-- test_preprocessing_helpers.py   # Contains TestRowToList, TestConvertToInt
|-- features/
|   |-- __init__.py
|   |-- test_as_numpy.py            # Contains TestGetDataAsNumpyArray
|-- models/
    |-- __init__.py
    |-- test_train.py               # Contains TestSplitIntoTrainingAndTestingSets
```

26. Final test directory structure
This procedure is then repeated for test_as_numpy.py, which would hold the test class TestGetDataAsNumpyArray and test_train.py, which would hold the test class TestSplitIntoTrainingAndTestingSets.

Test directory is well organized!

IPython console's working directory is tests



IPython console's working directory is tests

```
src/
|-- data/
|   |-- __init__.py
|   |-- preprocessing_helpers.py
|-- features/
|   |-- __init__.py
|   |-- as_numpy.py
|-- models/
|   |-- __init__.py
|   |-- train.py
tests/                                # This is IPython console's working directory from now on
|-- data/
|   |-- __init__.py
|   |-- test_preprocessing_helpers.py
|-- features/
|   |-- __init__.py
|   |-- test_as_numpy.py
|-- models/
    |-- __init__.py
    |-- test_train.py
```

Let's practice structuring tests!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Mastering test execution

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

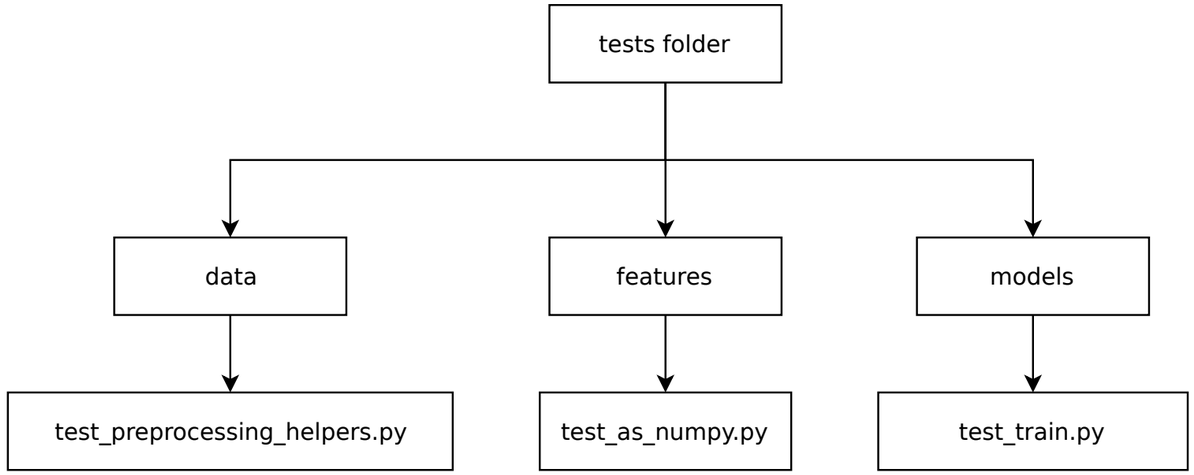
datacamp

Test organization

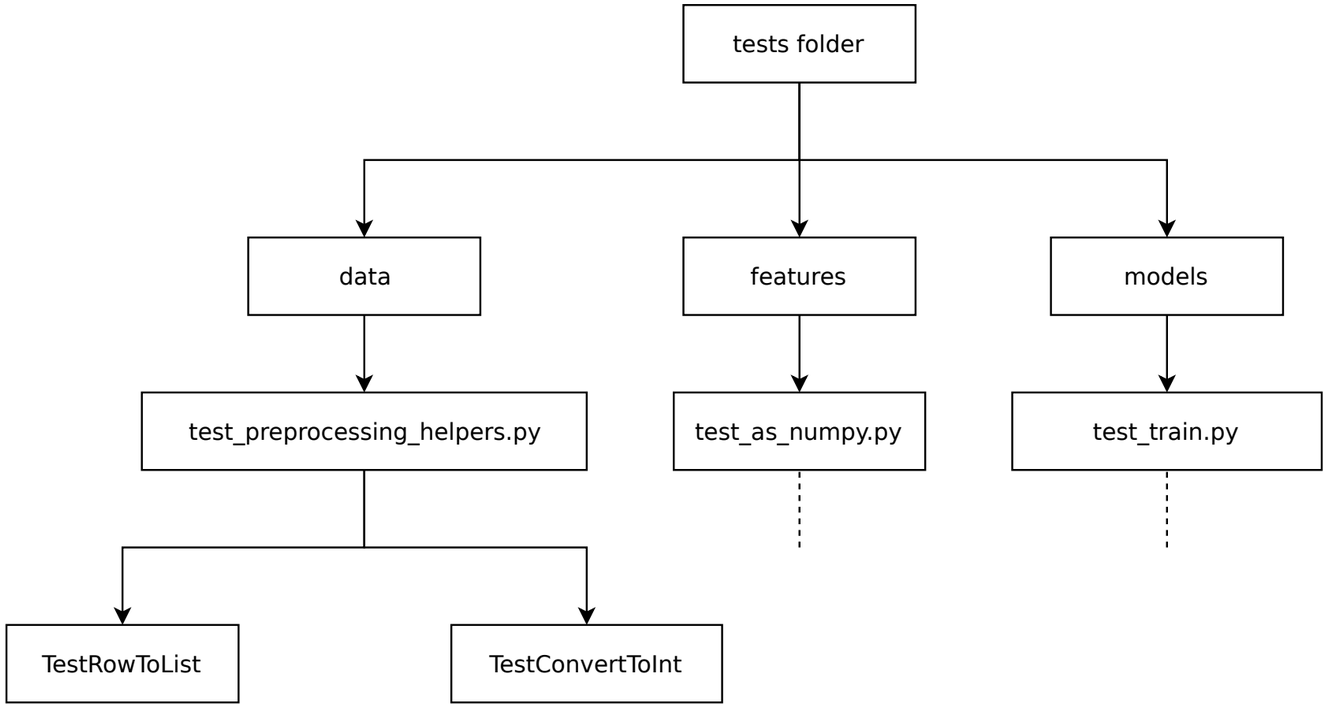
tests folder

Test organization

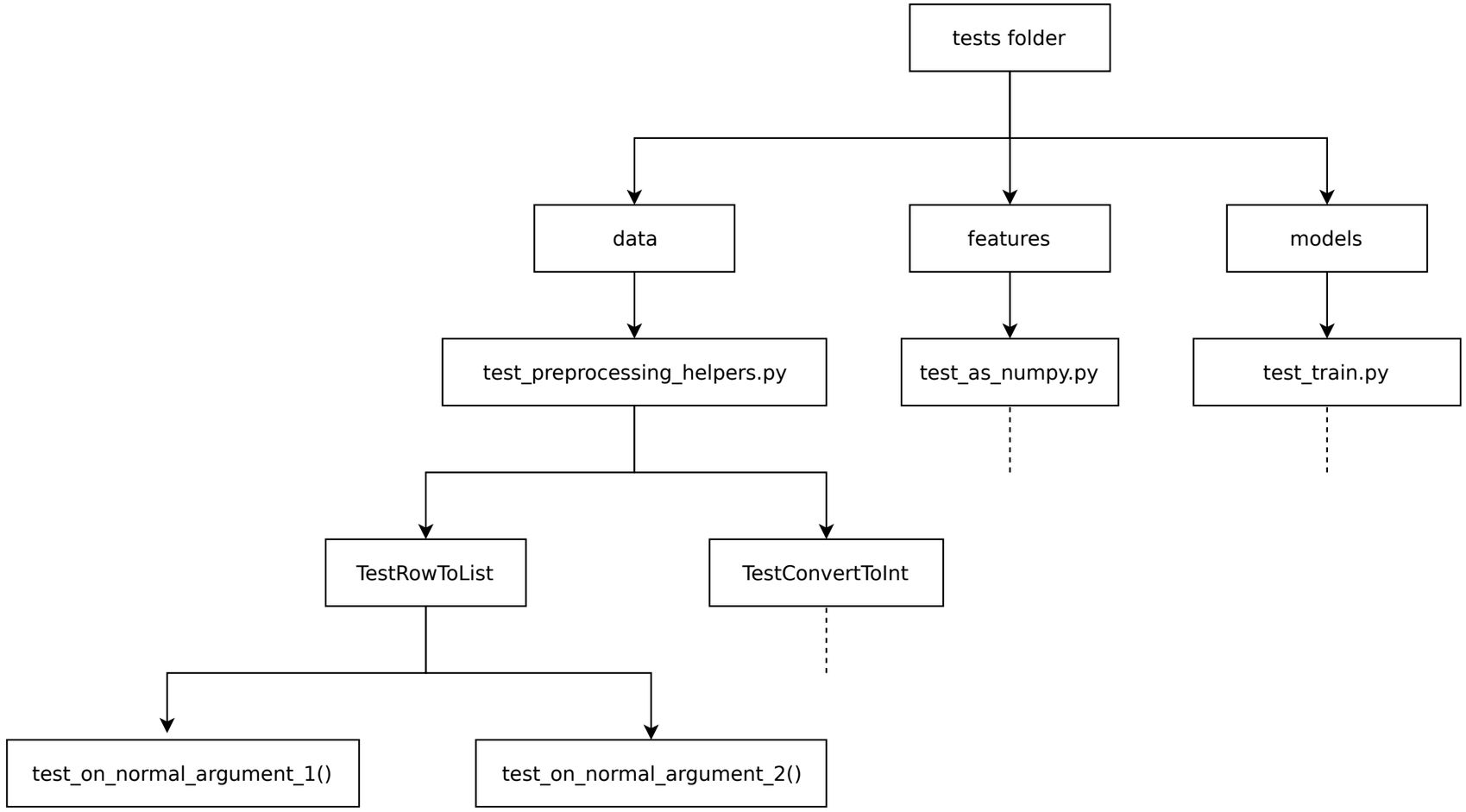
2. Test organization
The centerpiece was the tests folder, which holds all tests for the project.
3. Test organization
The folder contains mirror packages, each of which contain a test module.
4. Test organization
The test modules contain many test classes.
5. Test organization
A test class is just a container for unit tests for a particular function.



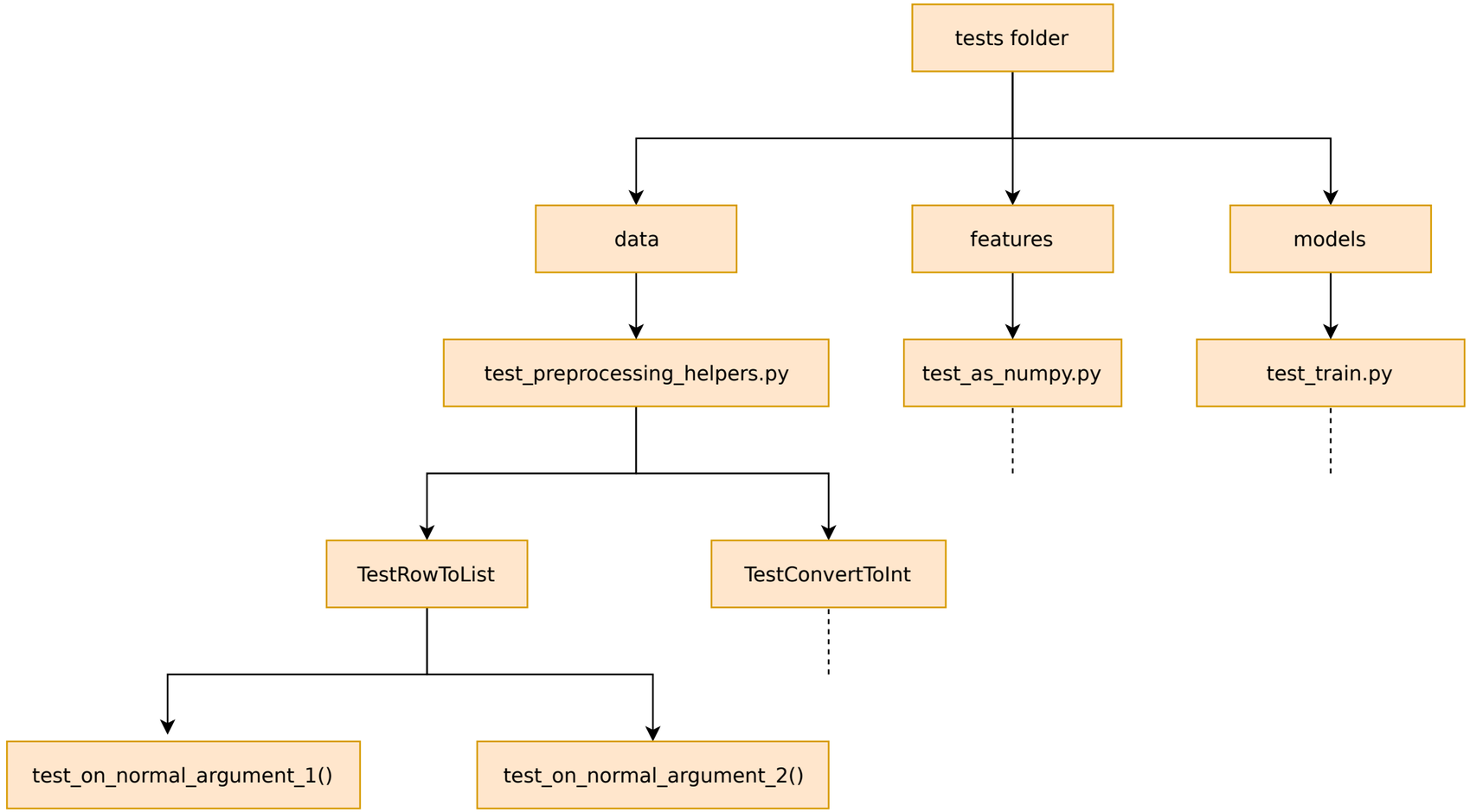
Test organization



Test organization



Running all tests



Running all tests

```
cd tests
pytest
```

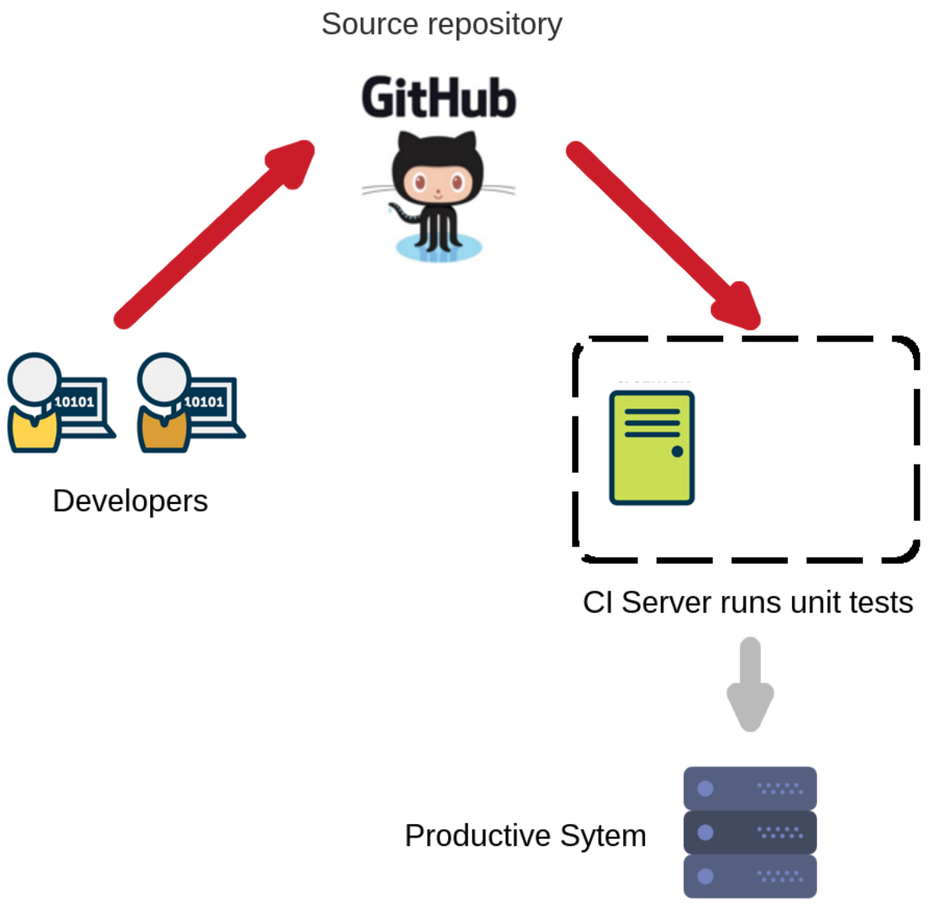
- Recurses into directory subtree of `tests/` .
  - Filenames starting with `test_` $\rightarrow$ test module.
    - Classnames starting with `Test` $\rightarrow$ test class.
      - Function names starting with `test_` $\rightarrow$ unit test.

6. Running all tests
pytest provides an easy way to run all tests contained in the tests folder.
7. Running all tests
We simply change to the tests directory and run the command pytest. This command automatically discovers tests by recursing into the subtree of the working directory. It identifies all files with names starting with “test_” as test modules. Within test modules, it identifies classes with names starting with “Test” as test classes. Within each test class, it identifies all functions with names starting with “test_” as unit tests. It collects these unit tests and runs them all.

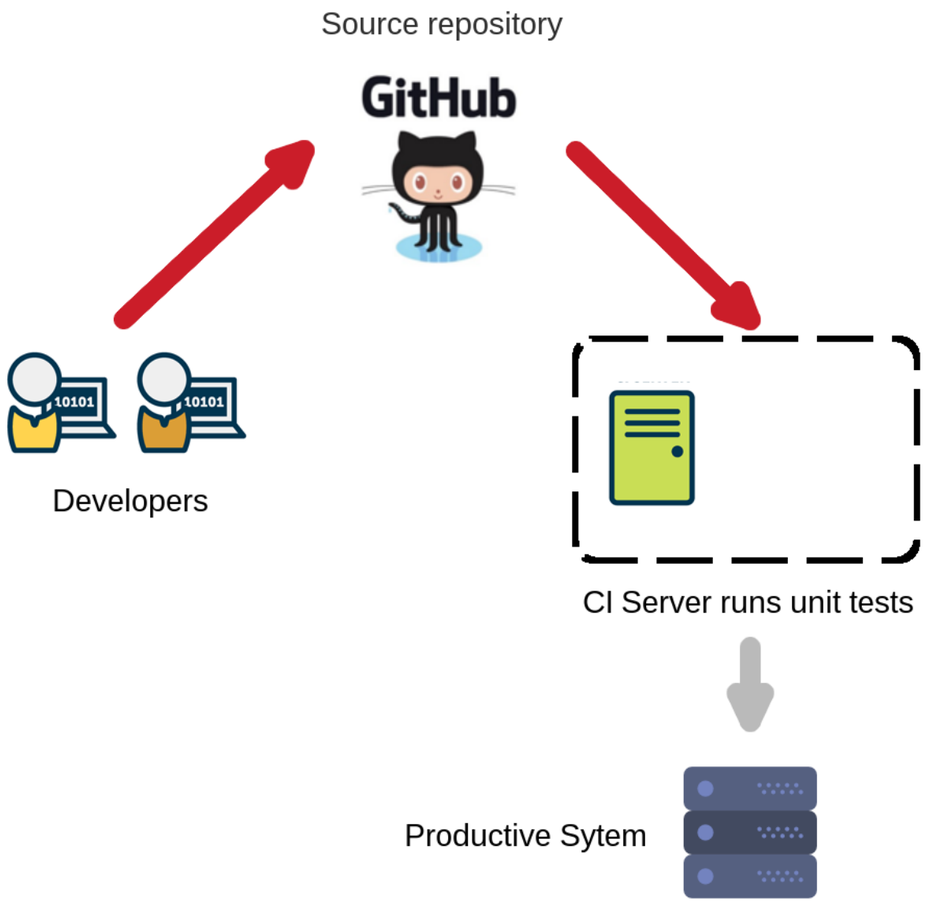
Running all tests

```
================================================================= test session starts ==========================================
data/test_preprocessing_helpers.py ........F....                                                                          [ 81%]
features/test_as_numpy.py .                                                                                               [ 87%]
models/test_train.py ..                                                                                                   [100%]

======================================================================= FAILURES ===============================================
_____________________________________________________ TestRowToList.test_on_one_tab_with_missing_value ___________________________

self = <tests.data.test_preprocessing_helpers.TestRowToList object at 0x7f6205475240>

    def test_on_one_tab_with_missing_value(self):    # (1, 1) boundary value
        actual = row_to_list("\t4,567\n")
>       assert actual is None, "Expected: None, Actual: {0}".format(actual)
E       AssertionError: Expected: None, Actual: ['', '4,567']
E       assert ['', '4,567'] is None

data/test_preprocessing_helpers.py:55: AssertionError
============================================================= 1 failed, 15 passed in 0.46 seconds ==================================
```

Typical scenario: CI server



Binary question: do all unit tests pass?



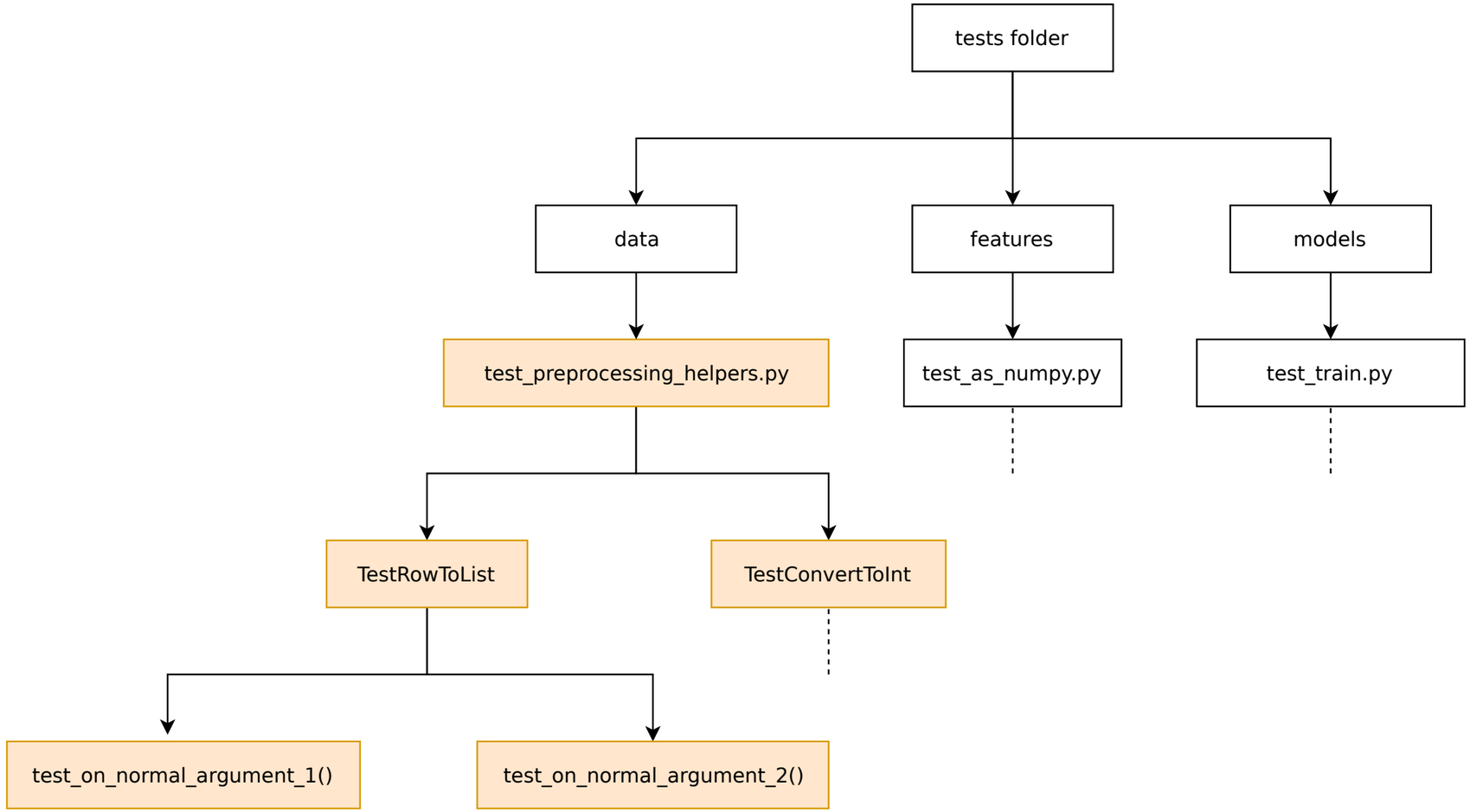
The -x flag: stop after first failure

```
pytest -x
```

```
========================================= test session starts ==========================================
data/test_preprocessing_helpers.py .......F

================================================ FAILURES ==============================================
_________________________ TestRowToList.test_on_one_tab_with_missing_value __________________________

self = <tests.data.test_preprocessing_helpers.TestRowToList object at 0x7f6309f17198>

    def test_on_one_tab_with_missing_value(self):    # (1, 1) boundary value
        actual = row_to_list("\t4,567\n")
>       assert actual is None, "Expected: None, Actual: {0}".format(actual)
E       AssertionError: Expected: None, Actual: ['', '4,567']
E       assert ['', '4,567'] is None

data/test_preprocessing_helpers.py:55: AssertionError
================================== 1 failed, 8 passed in 0.45 seconds ==================================
```

11. The -x flag: stop after first failure
In this case, adding the -x flag to the pytest command can save time and resources. This flag makes pytest stop after the first failing test, because a failing test already answers the binary question. In the report, we see that only 9 tests ran this time since execution stopped after the failing test test_on_one_tab_with_missing_value().

Running tests in a test module

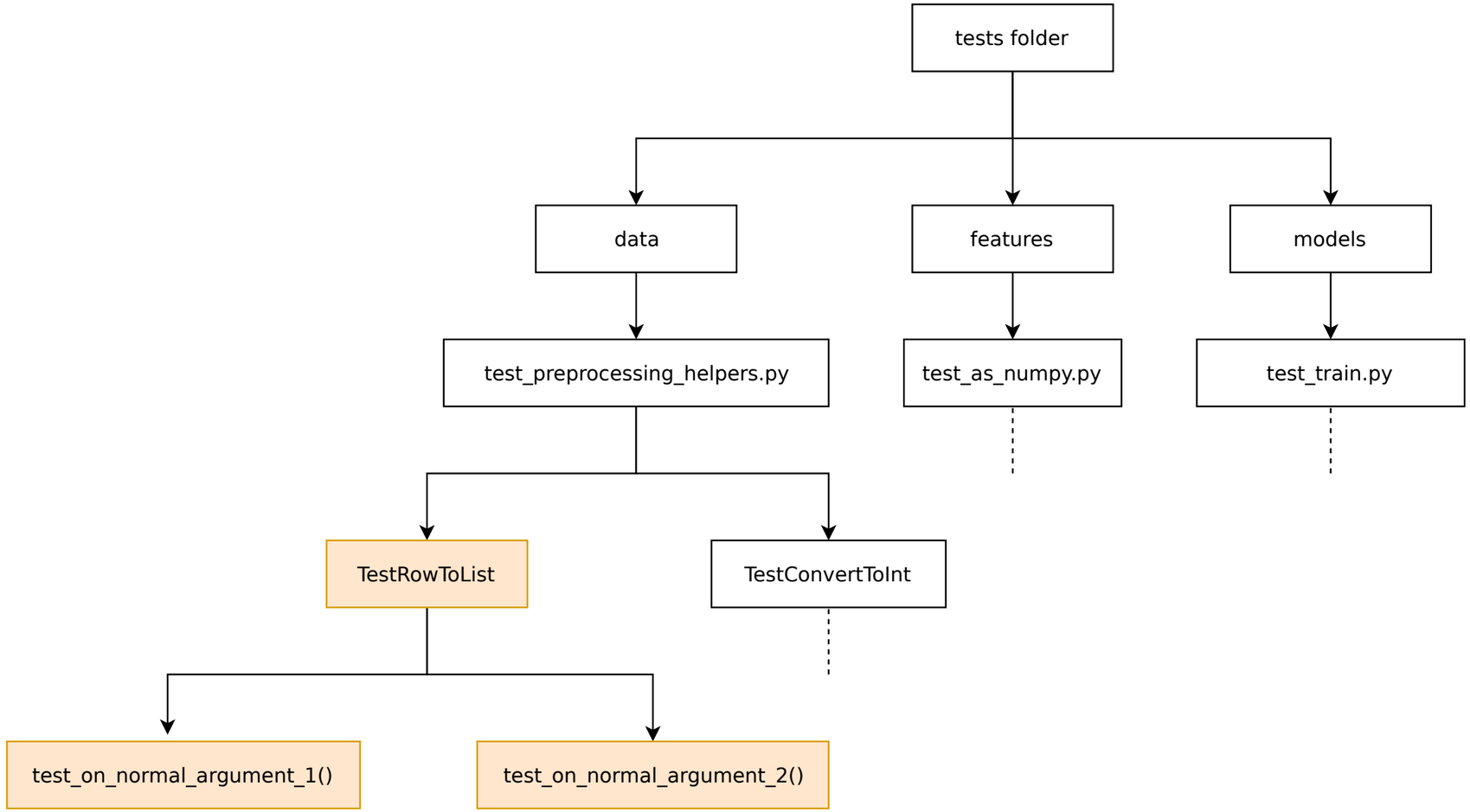


Running tests in a test module

```
pytest data/test_preprocessing_helpers.py
```

```
data/test_preprocessing_helpers.py ........F....                                          [100%]

================================================ FAILURES ================================================
__________________________ TestRowToList.test_on_one_tab_with_missing_value __________________________

self = <tests.data.test_preprocessing_helpers.TestRowToList object at 0x7f435947f198>

    def test_on_one_tab_with_missing_value(self):    # (1, 1) boundary value
        actual = row_to_list("\t4,567\n")
>       assert actual is None, "Expected: None, Actual: {0}".format(actual)
E       AssertionError: Expected: None, Actual: ['', '4,567']
E       assert ['', '4,567'] is None

data/test_preprocessing_helpers.py:55: AssertionError
================================== 1 failed, 12 passed in 0.07 seconds ==================================
```

Running only a particular test class



Node ID

- Node ID of a test class: `<path to test module>::<test class name>`
- Node ID of an unit test: `<path to test module>::<test class name>::<unit test name>`

The -k flag is really useful, because it helps you select tests and test classes by typing only a unique part of its name. This saves a lot of typing, and you must admit that TestSplitIntoTrainingAndTestingSets is a horrendously long name! In your projects, you will often run tests with the node IDs and the -k flag because you are often not interested in running all tests, but only a subset depending on the functions you are currently working on.

Running tests using node ID

- Run the test class `TestRowToList`.

```
pytest data/test_preprocessing_helpers.py::TestRowToList
```

```
data/test_preprocessing_helpers.py ..F....                                          [100%]

=============================================== FAILURES ================================================
__________________________ TestRowToList.test_on_one_tab_with_missing_value ___________________________

self = <tests.data.test_preprocessing_helpers.TestRowToList object at 0x7ffb3bac4da0>

    def test_on_one_tab_with_missing_value(self):    # (1, 1) boundary value
        actual = row_to_list("\t4,567\n")
>       assert actual is None, "Expected: None, Actual: {0}".format(actual)
E       AssertionError: Expected: None, Actual: ['', '4,567']
E       assert ['', '4,567'] is None

data/test_preprocessing_helpers.py:55: AssertionError
================================= 1 failed, 6 passed in 0.06 seconds ==================================
```

12. Running tests in a test module
Very often, we would only want to run a subset of tests. For example, we might want to just run tests contained in a particular test module, say, test_preprocessing_helpers.py. You already know how to do that since you did this several times in the exercises.

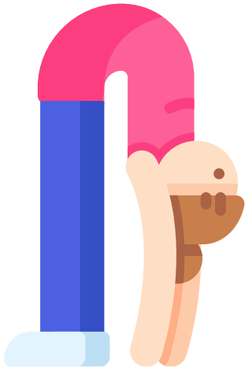
Running tests using node ID

- Run the unit test `test_on_one_tab_with_missing_value()`.

```
pytest data/test_preprocessing_helpers.py::TestRowToList::test_on_one_tab_with_missing_value
```

```
data/test_preprocessing_helpers.py F                                                    [100%]

=================================================== FAILURES ===================================================
___________________________ TestRowToList.test_on_one_tab_with_missing_value ___________________________

self = <tests.data.test_preprocessing_helpers.TestRowToList object at 0x7f4eece33b00>

    def test_on_one_tab_with_missing_value(self):    # (1, 1) boundary value
        actual = row_to_list("\t4,567\n")
>       assert actual is None, "Expected: None, Actual: {0}".format(actual)
E       AssertionError: Expected: None, Actual: ['', '4,567']
E       assert ['', '4,567'] is None

data/test_preprocessing_helpers.py:55: AssertionError
======================================== 1 failed in 0.06 seconds ========================================
```

Running tests using keyword expressions



The -k option

```
pytest -k "pattern"
```

- Runs all tests whose node ID matches the pattern.

19. The -k option
To run tests using keyword expressions, use the -k option. This option takes a quoted string containing a pattern as the value.
20. The -k option
For example, we can specify a test class such as TestSplitIntoTrainingAndTestingSets as the pattern, and this will run only the 2 tests within that test class. We can also enter only part of the test class name, as long as that is unique. This saves a lot of typing and has the same outcome.

The -k option

- Run the test class `TestSplitIntoTrainingAndTestingSets`.

```
pytest -k "TestSplitIntoTrainingAndTestingSets"
```

```
models/test_train.py ..                                                  [100%]

============================== 2 passed, 14 deselected in 0.36 seconds ==============================
```

```
pytest -k "TestSplit"
```

```
models/test_train.py ..                                                  [100%]

============================== 2 passed, 14 deselected in 0.36 seconds ==============================
```

Supports Python logical operators

```
pytest -k "TestSplit and not test_on_one_row"
```

```
models/test_train.py .                                                  [100%]

==================== 1 passed, 15 deselected in 0.36 seconds ====================
```

Let's run some tests!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

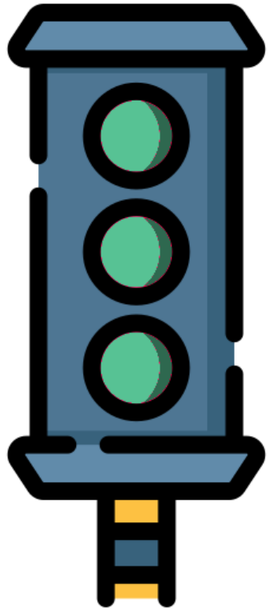
Expected failures and conditional skipping

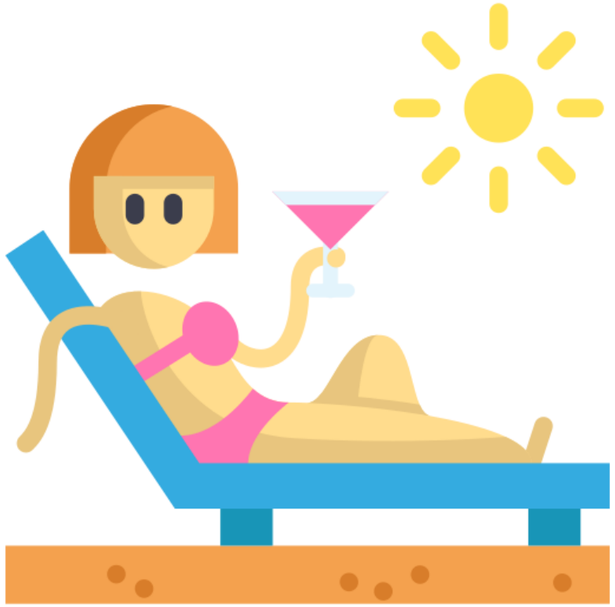
UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

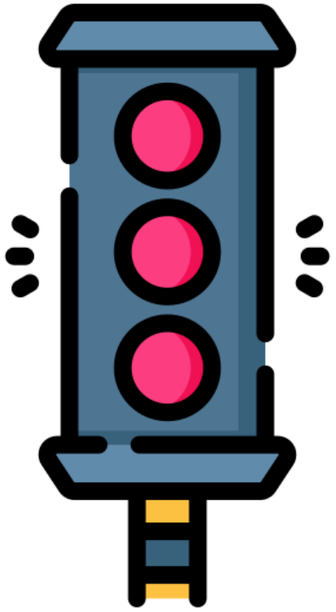
Test suite is green when all tests pass





Test suite is red when any test fails

3. Test suite is red when any test fails
If any test fails, then our test suite is red. This means we better work and fix it, otherwise our users will be very angry. This is good in theory, but, sometimes, the red light can be a false alarm that will ruin our beach vacations! An example will make things clear.



Implementing a function using TDD

- `train_model()` : Returns best fit line given training data.

```
import pytest


class TestTrainModel(object):
    def test_on_linear_data(self):
        ...
```

4. Implementing a function using TDD
Let's say we are implementing this new function train_model(), which returns the best fit line on the training data. Since we are gonna use TDD, the first step is to write tests, so we create a test class TestTrainModel and add a test to it.

5. The test fails, of course!
If we run pytest, this test will fail because the function train_model() is not yet implemented. And this is just a result of using TDD, it does not indicate a problem with the code base.

The test fails, of course!

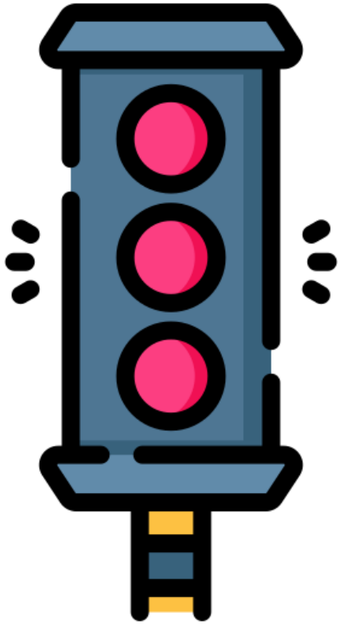
6. False alarm
But the CI server does not know this and will set off a false alarm when that test fails. It would be nice to have a way to tell pytest that we expect this test to fail.

```
pytest
```

```
================================================ test session starts ================================================
data/test_preprocessing_helpers.py .............                                                                  [ 76%]
features/test_as_numpy.py .                                                                                       [ 82%]
models/test_train.py ..F                                                                                          [100%]

======================================================= FAILURES ======================================================
_____________________________________________ TestTrainModel.test_on_linear_data _____________________________________________

self = <tests.models.test_train.TestTrainModel object at 0x7f5fc0f31978>

    def test_on_linear_data(self):
        test_input = np.array([[1.0, 3.0], [2.0, 5.0], [3.0, 7.0]])
        expected_slope = 2.0
        expected_intercept = 1.0
>       actual_slope, actual_intercept = train_model(test_input)
E       NameError: name 'train_model' is not defined

models/test_train.py:39: NameError
========================================== 1 failed, 16 passed in 0.22 seconds ==========================================
```

False alarm

7. xfail: marking tests as "expected to fail"
We do that by using the xfail decorator. The decorator goes on top of a test, and it starts with the character @.



xfail: marking tests as "expected to fail"

```
import pytest

class TestTrainModel(object):
    @
    def test_on_linear_data(self):
        ...
```

8. xfail: marking tests as "expected to fail"
This is followed by the name of the decorator pytest.mark.xfail.
After adding the decorator, if we run pytest again, we see that one test is xfailed. But there are no reported errors,

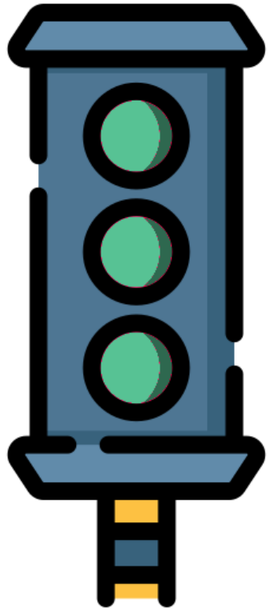
xfail: marking tests as "expected to fail"

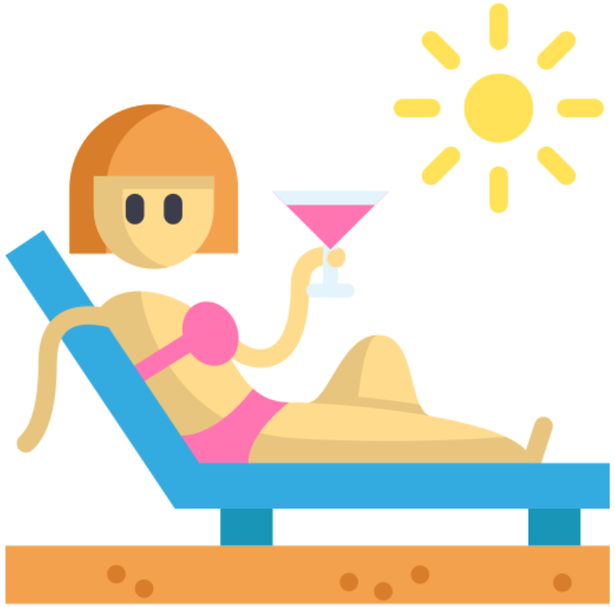
```python
import pytest

class TestTrainModel(object):
    @pytest.mark.xfail
    def test_on_linear_data(self):
        ...
```

```
pytest
```

```
========================================= test session starts =========================================
data/test_preprocessing_helpers.py .............                                              [ 76%]
features/test_as_numpy.py .                                                                   [ 82%]
models/test_train.py ..x                                                                      [100%]

================================ 16 passed, 1 xfailed in 0.21 seconds ================================
```

Test suite stays green





Expected failures, but conditionally

Tests that are expected to fail

- on certain Python versions.
- on certain platforms like Windows.

10. Expected failures, but conditionally
At other times, we might know that the test fails only under certain conditions, and we don't want to be warned about them. Common situations are when some function won't work under a particular Python version or a particular platform. As an example, we have deliberately added the unicode() function in the failure message for the test test_with_no_comma() that we wrote earlier. This only works on Python 2.7 or lower.

```
class TestConvertToInt(object):
    def test_with_no_comma(self):
        """Only runs on Python 2.7 or lower"""
        test_argument = "756"
        expected = 756
        actual = convert_to_int(test_argument)
        message = unicode("Expected: 2081, Actual: {0}".format(actual))  # Requires Python 2.7 or lower
        assert actual == expected, message
```

Test suite goes red on Python 3

```
pytest
```

```
========================================= test session starts ==========================================
platform linux -- Python 3.6.8, pytest-4.3.1, py-1.8.0, pluggy-0.9.0

============================================= FAILURES =================================================
_____________________________ TestConvertToInt.test_with_no_comma ______________________________

self = <tests.data.test_preprocessing_helpers.TestConvertToInt object at 0x7f2c479a76a0>

    def test_with_no_comma(self):
        test_argument = "756"
        expected = 756
        actual = convert_to_int(test_argument)
>       message = unicode("Expected: 2081, Actual: {0}".format(actual))
E       NameError: name 'unicode' is not defined

data/test_preprocessing_helpers.py:12: NameError
========================== 1 failed, 15 passed, 1 xfailed in 0.23 seconds ==========================
```

skipif: skip tests conditionally

```
class TestConvertToInt(object):
    @pytest.mark.skipif
    def test_with_no_comma(self):
        """Only runs on Python 2.7 or lower"""
        test_argument = "756"
        expected = 756
        actual = convert_to_int(test_argument)
        message = unicode("Expected: 2081, Actual: {0}".format(actual))
        assert actual == expected, message
```

11. Test suite goes red on Python 3
If we run pytest using Python 3, the test suite will go red.

12. skipif: skip tests conditionally
To tell pytest to skip running this test on Python versions higher than 2.7, we need the skipif decorator. The syntax is similar to xfail. The name of the decorator is pytest.mark.skipif.

13. skipif: skip tests conditionally
It takes a single boolean expression as an argument. If the boolean expression is True, then the test will be skipped.

skipif: skip tests conditionally

```
class TestConvertToInt(object):
    @pytest.mark.skipif(boolean_expression)
    def test_with_no_comma(self):
        """Only runs on Python 2.7 or lower"""
        test_argument = "756"
        expected = 756
        actual = convert_to_int(test_argument)
        message = unicode("Expected: 2081, Actual: {0}".format(actual))
        assert actual == expected, message
```

12. skipif: skip tests conditionally
To tell pytest to skip running this test on Python versions higher than 2.7, we need the skipif decorator. The syntax is similar to xfail. The name of the decorator is pytest.mark.skipif.
13. skipif: skip tests conditionally
It takes a single boolean expression as an argument. If the boolean expression is True, then the test will be skipped.

- If `boolean_expression` is `True`, then test is skipped.

skipif when Python version is higher than 2.7

```
import sys


class TestConvertToInt(object):
    @pytest.mark.skipif(sys.version_info > (2, 7))
    def test_with_no_comma(self):
        """Only runs on Python 2.7 or lower"""
        test_argument = "756"
        expected = 756
        actual = convert_to_int(test_argument)
        message = unicode("Expected: 2081, Actual: {0}".format(actual))
        assert actual == expected, message
```

14. skipif when Python version is higher than 2.7
o construct the boolean expression, import the built in module sys and use the attribute sys.version_info. This attribute can be compared against a tuple containing the major and minor Python version, in this case, 2 and 7.

The reason argument

15. The reason argument
We must also add the required reason argument, which states why the test is skipped.

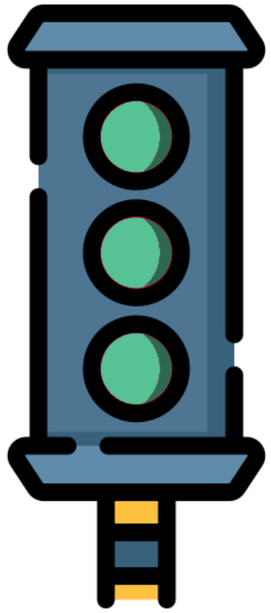
```
import sys

class TestConvertToInt(object):
    @pytest.mark.skipif(sys.version_info > (2, 7), reason="requires Python 2.7")
    def test_with_no_comma(self):
        """Only runs on Python 2.7 or lower"""
        test_argument = "756"
        expected = 756
        actual = convert_to_int(test_argument)
        message = unicode("Expected: 2081, Actual: {0}".format(actual))
        assert actual == expected, message
```

1 skipped, 1 xfailed

16. 1 skipped, 1 xfailed
Running pytest again confirms that one test was xfailed and another one was skipped.

```
pytest
```

```
========================================= test session starts =========================================
platform linux -- Python 3.6.8, pytest-4.3.1, py-1.8.0, pluggy-0.9.0
collected 17 items

data/test_preprocessing_helpers.py s...........                                          [ 76%]
features/test_as_numpy.py .                                                              [ 82%]
models/test_train.py ..x                                                                 [100%]

======================== 15 passed, 1 skipped, 1 xfailed in 0.21 seconds ========================
```



17. Test suite stays green
The test suite remains green. Perfect again!

Showing reason in the test result report

```
pytest -r
```

18. Showing reason in the test result report
We can make the reason for skipping show in the report. For that, we can use the -r option.
19. The -r option
The -r option can be followed by any number of characters.

The -r option

```
pytest -r[set_of_characters]
```

Showing reason for skipping

```
pytest -rs
```

20. Showing reason for skipping
If we add the character s, it will show us tests that were skipped in the short test summary section near the end.

```
==================================== test session starts ====================================
platform linux -- Python 3.6.8, pytest-4.3.1, py-1.8.0, pluggy-0.9.0
collected 17 items

data/test_preprocessing_helpers.py s...........                                        [ 76%]
features/test_as_numpy.py .                                                            [ 82%]
models/test_train.py ..x                                                               [100%]
================================== short test summary info ==================================
SKIPPED [1] tests/data/test_preprocessing_helpers.py:8: Requires Python 2.7 or lower

===================== 15 passed, 1 skipped, 1 xfailed in 0.21 seconds =====================
```

Optional reason argument to xfail

```
import pytest


class TestTrainModel(object):
    @pytest.mark.xfail
    def test_on_linear_data(self):
        ...
```

21. Optional reason argument to xfail
The xfail decorator also takes an optional reason argument.

22. Optional reason argument to xfail
For the test that we marked with xfail, we will add the reason "Using TDD, train_model() is not implemented".

Optional reason argument to xfail

```
import pytest


class TestTrainModel(object):
    @pytest.mark.xfail(reason="“Using TDD, train_model() is not implemented")
    def test_on_linear_data(self):
        ...
```

23. Showing reason for xfail
If we add the character x to the -r option, it will only show us tests that are xfailed along with the reason in the test summary info.

Showing reason for xfail

23. Showing reason for xfail
If we add the character x to the -r option, it will only show us tests that are xfailed along with the reason in the test summary info.

```
pytest -rx
```

```
==================================== test session starts ====================================
platform linux -- Python 3.6.8, pytest-4.3.1, py-1.8.0, pluggy-0.9.0
collected 17 items

data/test_preprocessing_helpers.py s...........                                        [ 76%]
features/test_as_numpy.py .                                                            [ 82%]
models/test_train.py ..x                                                               [100%]
================================== short test summary info ==================================
XFAIL models/test_train.py::TestTrainModel::test_on_linear_data
  Using TDD, train_model() is not implemented

===================== 15 passed, 1 skipped, 1 xfailed in 0.28 seconds =====================
```

Showing reason for both skipped and xfail

```
pytest -rsx
```

24. Showing reason for both skipped and xfail
We can show reasons for both by using the combination sx.

```
==================================== test session starts ====================================
platform linux -- Python 3.6.8, pytest-4.3.1, py-1.8.0, pluggy-0.9.0
rootdir: /home/dibya/startup-code/datacamp/univariate_linear_regression, inifile:
collected 17 items

data/test_preprocessing_helpers.py s...........                                        [ 76%]
features/test_as_numpy.py .                                                            [ 82%]
models/test_train.py ..x                                                               [100%]
================================== short test summary info ==================================
SKIPPED [1] tests/data/test_preprocessing_helpers.py:8: Requires Python 2.7 or lower
XFAIL models/test_train.py::TestTrainModel::test_on_linear_data
  Using TDD, train_model() is not implemented

===================== 15 passed, 1 skipped, 1 xfailed in 0.22 seconds =====================
```

Skipping/xfailing entire test classes

```
@pytest.mark.xfail(reason="“Using TDD, train_model() is not implemented")
class TestTrainModel(object):
    ...
```

```
@pytest.mark.skipif(sys.version_info > (2, 7), reason="requires Python 2.7")
class TestConvertToInt(object):
    ...
```

25. Skipping/xfailing entire test classes
If we are skipping and xfailing multiple tests, note that these decorators can be applied to entire test classes as well.

Let's practice xfailing and skipping!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Continuous integration and code coverage

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

Code coverage and build status badges

NumPy

Travis CI passing | AppVeyor passing | Azure Pipelines succeeded | codecov 85%

NumPy is the fundamental package needed for scientific computing with Python.

- **Website (including documentation):** https://www.numpy.org
- **Mailing list:** https://mail.python.org/mailman/listinfo/numpy-discussion
- **Source:** https://github.com/numpy/numpy

Code coverage and build status badges

NumPy

Travis CI passing | AppVeyor passing | Azure Pipelines succeeded | codecov 85%

NumPy is the fundamental package needed for scientific computing with Python.

- **Website (including documentation):** https://www.numpy.org
- **Mailing list:** https://mail.python.org/mailman/listinfo/numpy-discussion
- **Source:** https://github.com/numpy/numpy

Code coverage and build status badges

2. Code coverage and build status badges
We saw how NumPy increases user trust
3. Code coverage and build status badges
by adding code coverage
4. Code coverage and build status badges
and build status badges. In this lesson, we will learn to implement these badges for our own GitHub projects.

NumPy

Travis CI passing | AppVeyor passing | Azure Pipelines succeeded | codecov 85%

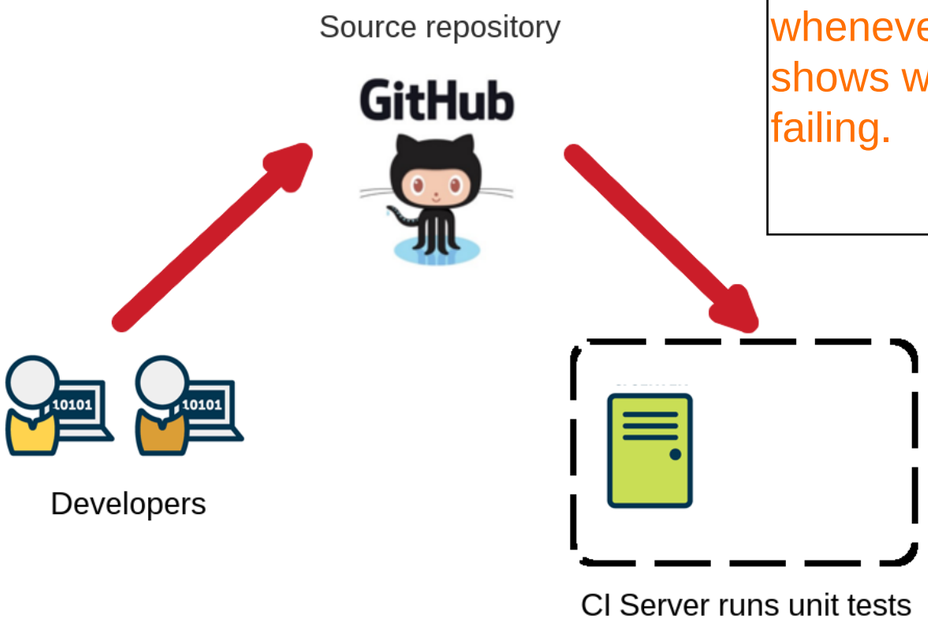
NumPy is the fundamental package needed for scientific computing with Python.

- **Website (including documentation):** https://www.numpy.org
- **Mailing list:** https://mail.python.org/mailman/listinfo/numpy-discussion
- **Source:** https://github.com/numpy/numpy

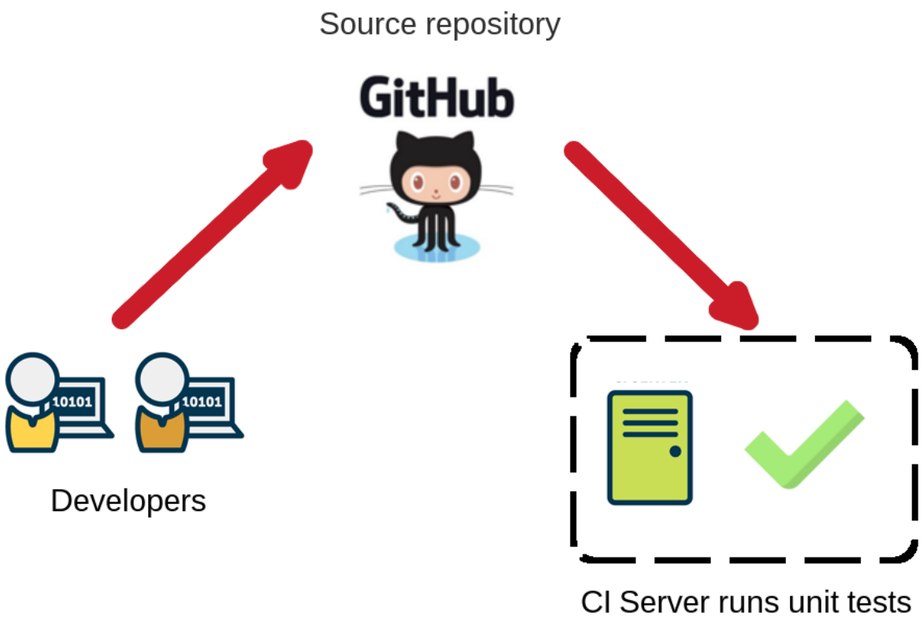
The build status badge

The build status badge

6. The build status badge
This badge uses a Continuous Integration server, which runs all tests automatically whenever we push a commit to GitHub. It shows whether tests are currently passing or failing.



Build passing = Stable project



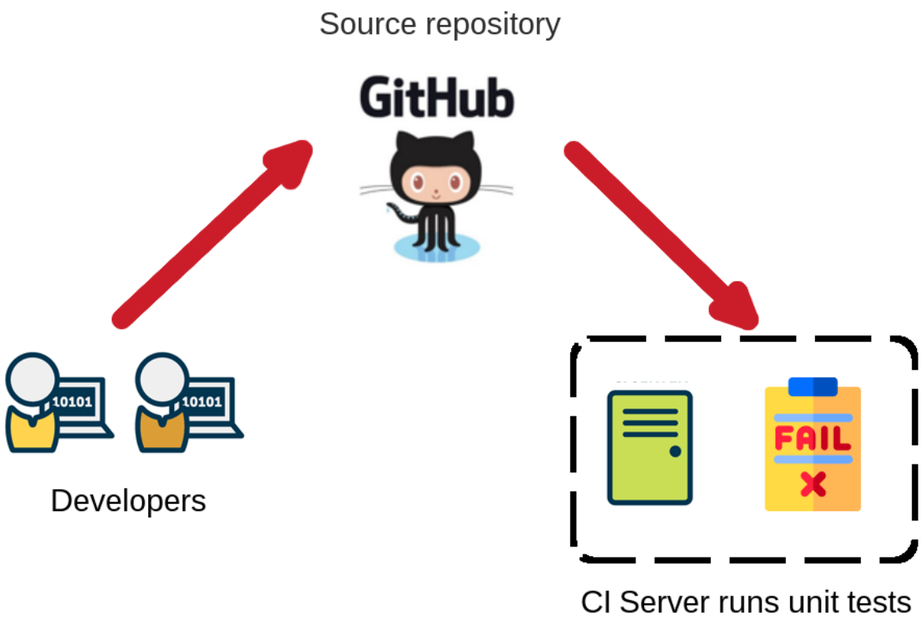
build passing

6. The build status badge
This badge uses a Continuous Integration server, which runs all tests automatically whenever we push a commit to GitHub. It shows whether tests are currently passing or failing.

7. Build passing = Stable project
To an end user, passing indicates a stable code base
8. Build failing = Unstable project
while failing indicates instability.

Build failing = Unstable project

8. Build failing = Unstable project while failing indicates instability.



build failing

CI server

9. CI server
We will use Travis CI as our CI server.

Travis CI

Step 1: Create a configuration file

```
repository root
|-- src
|-- tests
|--.travis.yml
```

10. Step 1: Create a configuration file
To integrate with Travis CI, we have to create a settings file called .travis.yml at the root of our repository.

Step 1: Create a configuration file

- Contents of `.travis.yml`.

```
language: python
python:
  - "3.6"
install:
  - pip install -e .
script:
  - pytest tests
```

11. Step 1: Create a configuration file
The file is arranged into sections. First, there's a language setting, and we set it to python. The python setting determines which Python version will be used to run the tests. We choose Python 3.6. The install setting is a list of commands to install our project and dependencies in the CI server. If we organized our tests in the recommended way, then we can use a local pip install using pip install -e dot. The script section lists the commands necessary to run the tests once everything is installed. We use pytest tests to run the test suite.

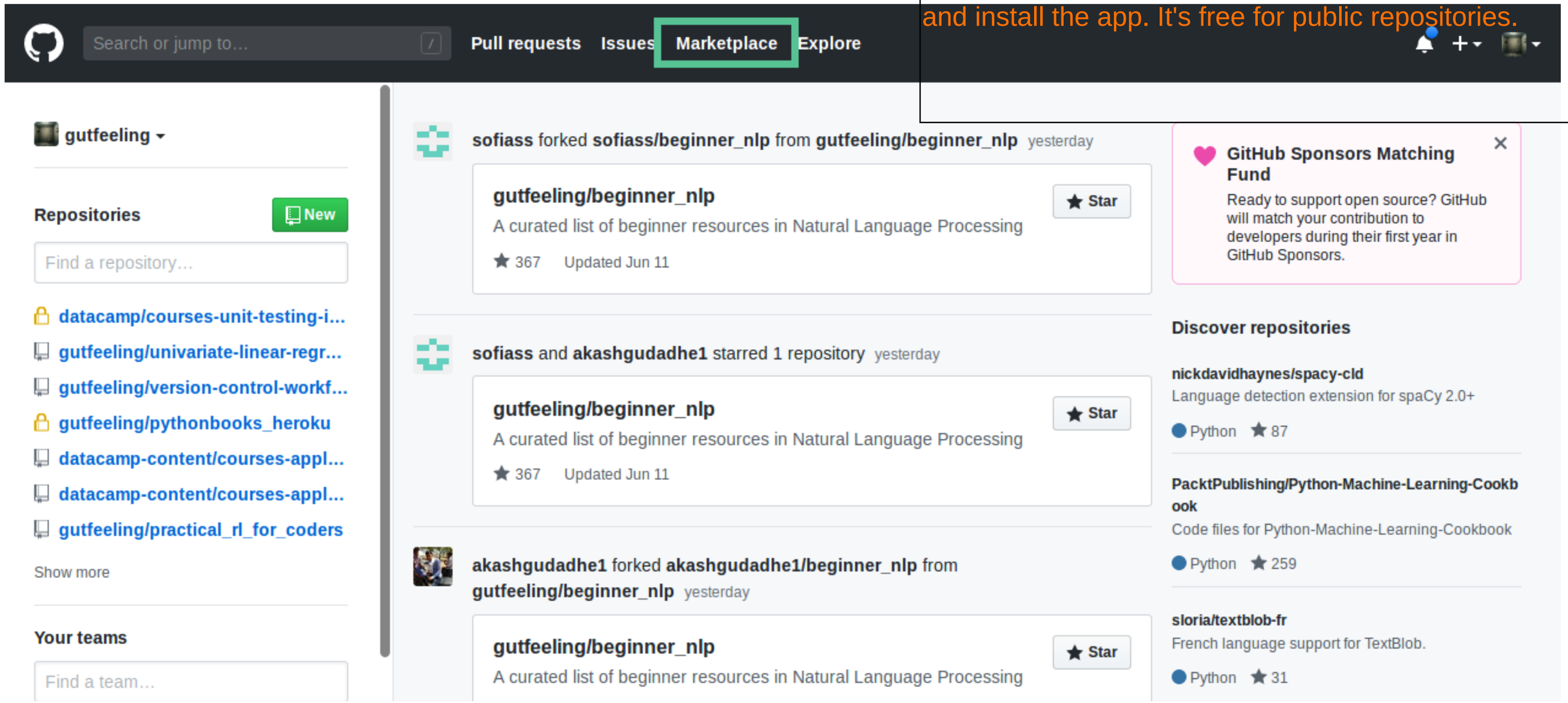
Step 2: Push the file to GitHub

```
git add .travis.yml
git push origin master
```

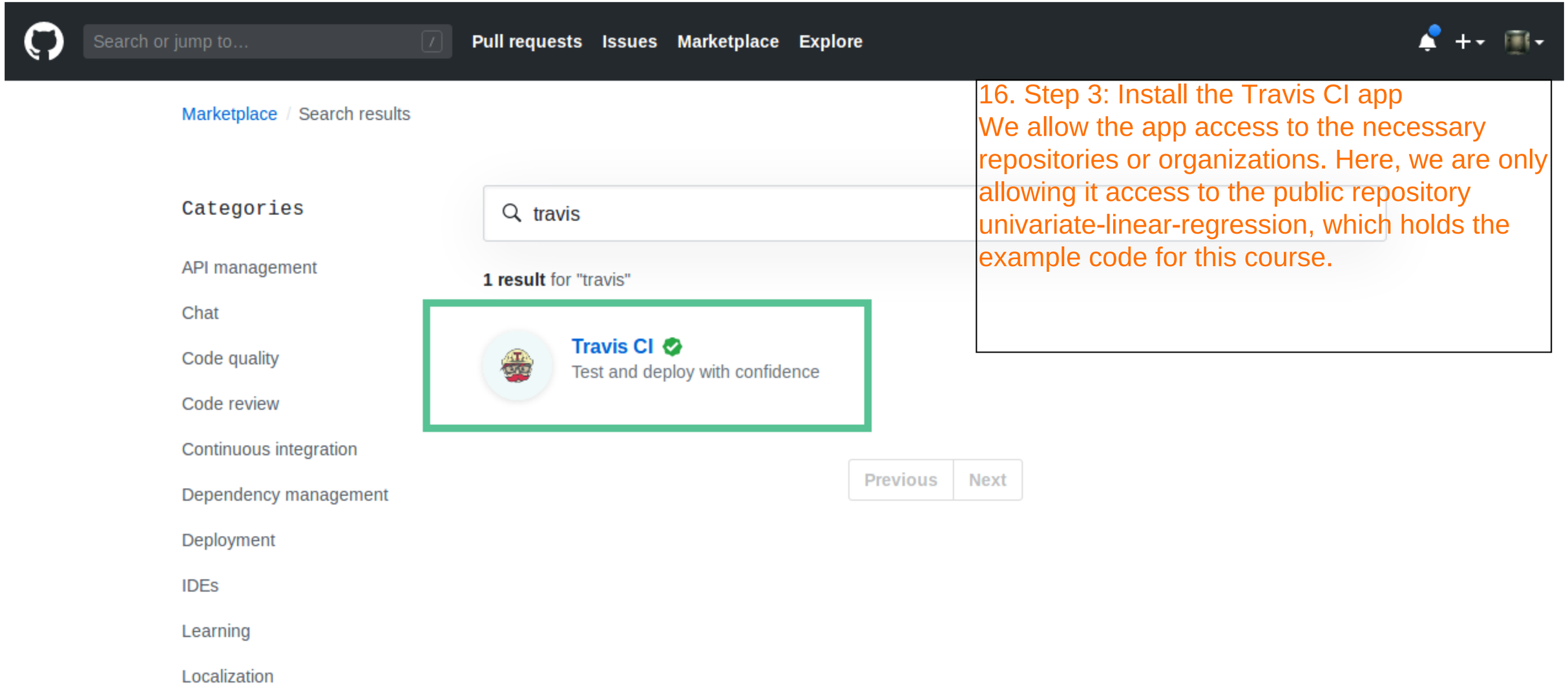
12. Step 2: Push the file to GitHub
We push this settings file to GitHub.
13. Step 3: Install the Travis CI app
Now we go to the GitHub profile page and click on MarketPlace.

Step 3: Install the Travis CI app

14. Step 3: Install the Travis CI app
We search for Travis CI, click on it
15. Step 3: Install the Travis CI app
and install the app. It's free for public repositories.



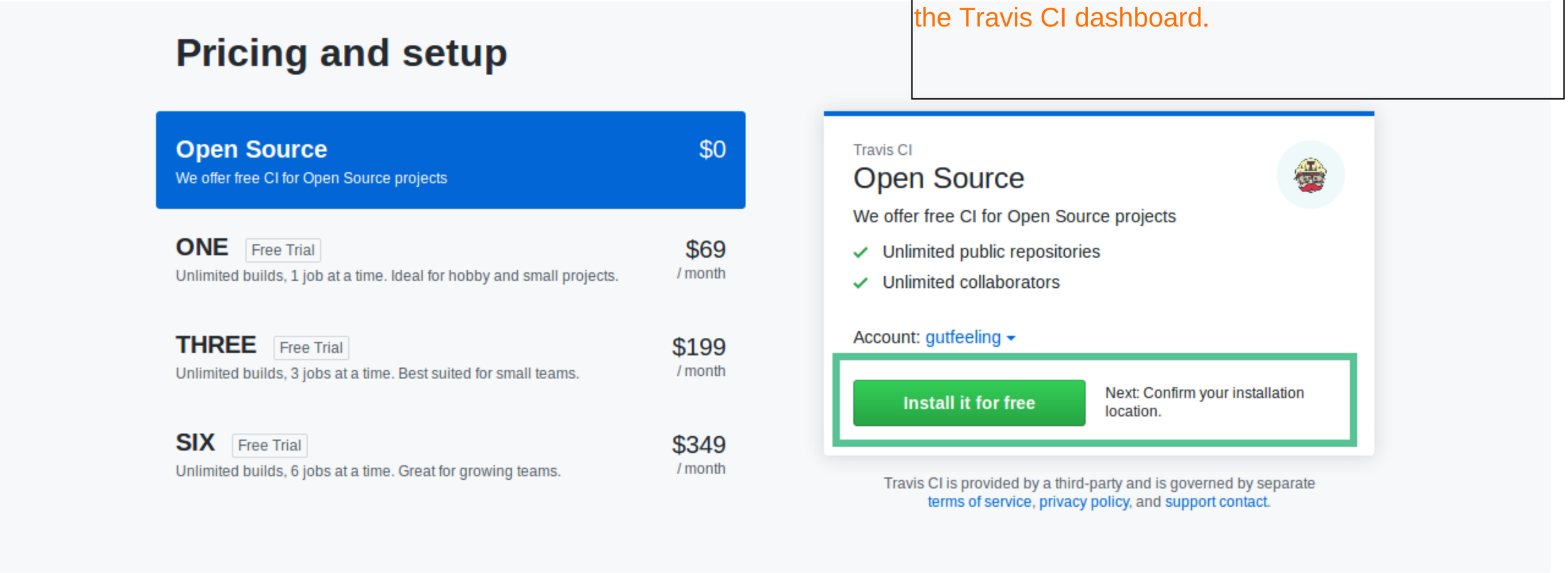
Step 3: Install the Travis CI app



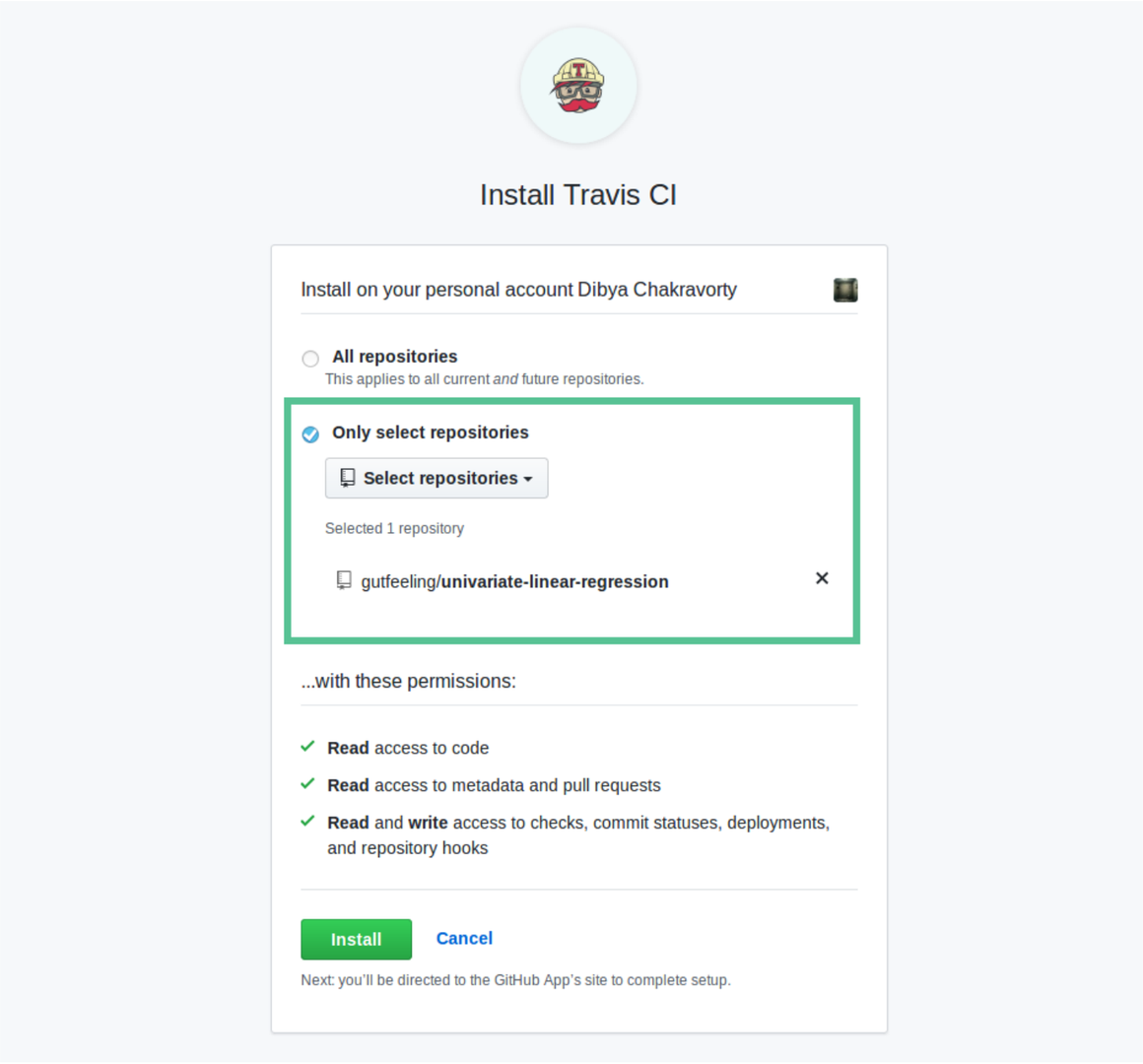
16. Step 3: Install the Travis CI app
We allow the app access to the necessary repositories or organizations. Here, we are only allowing it access to the public repository univariate-linear-regression, which holds the example code for this course.

Step 3: Install the Travis CI app

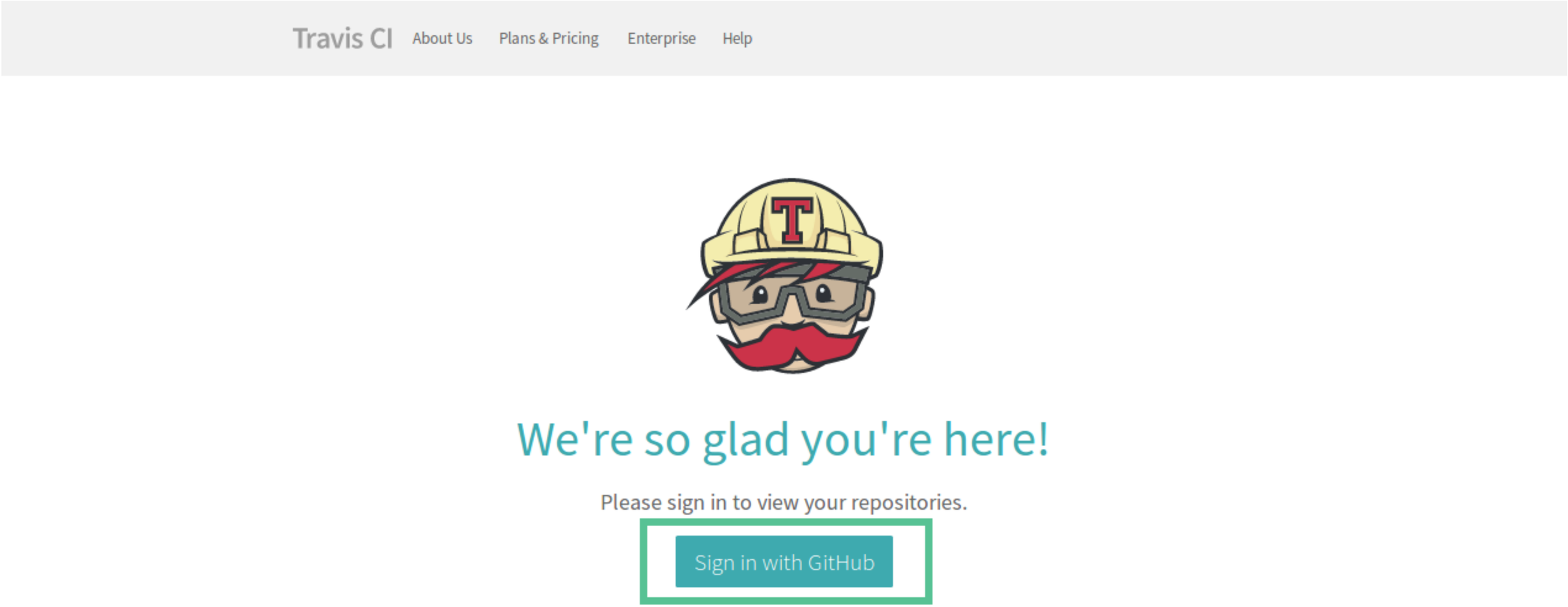
17. Step 3: Install the Travis CI app
We will be redirected to Travis CI, where we should login using our GitHub account. This will bring us to the Travis CI dashboard.



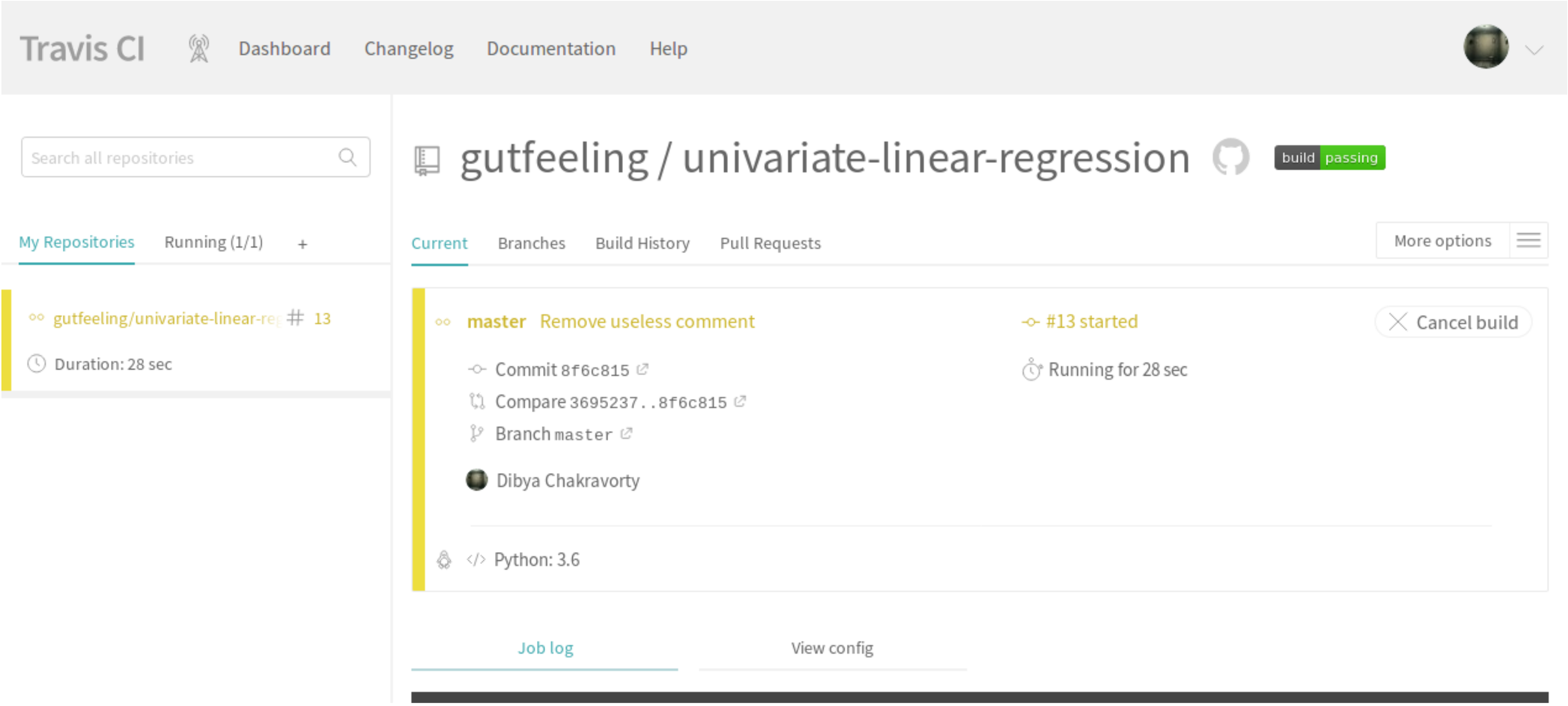
Step 3: Install the Travis CI app



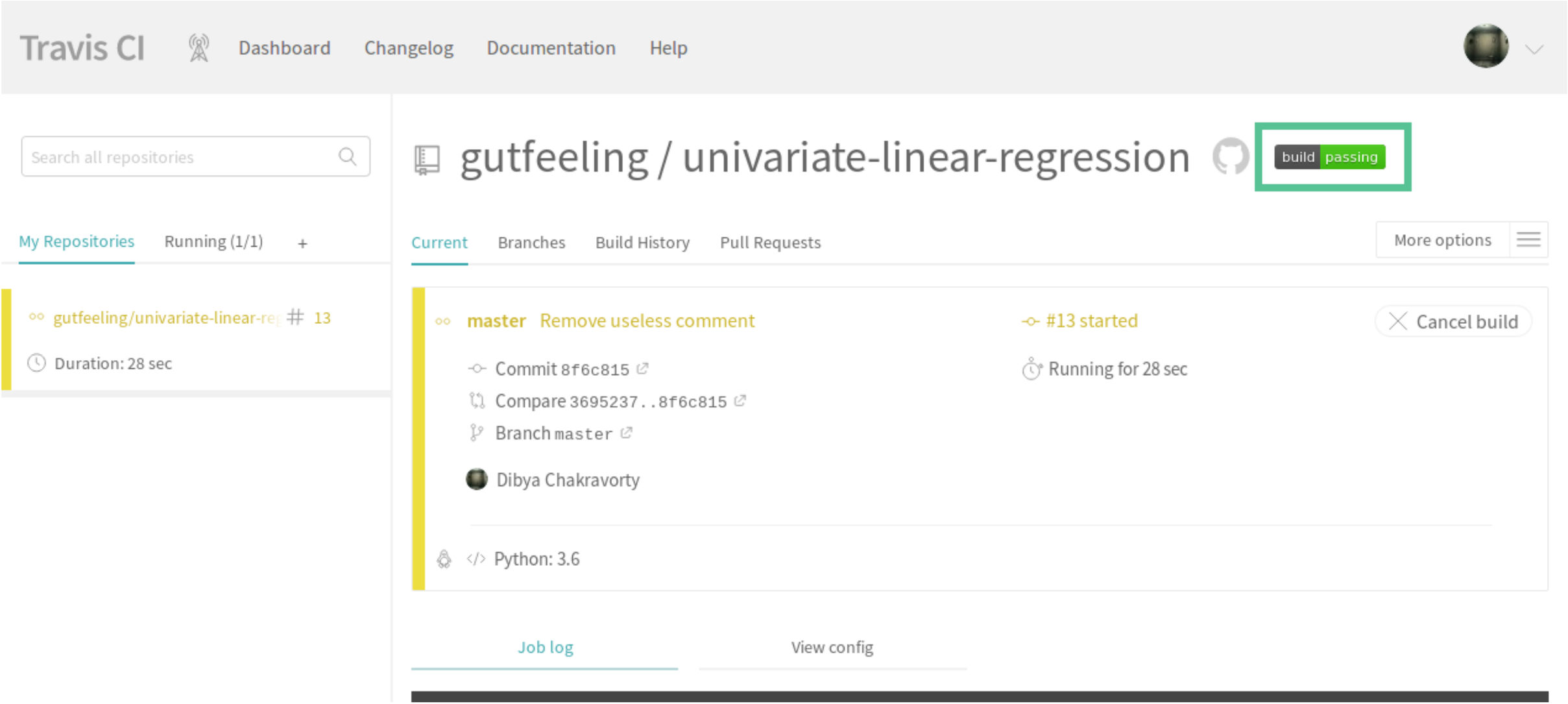
Step 3: Install the Travis CI app



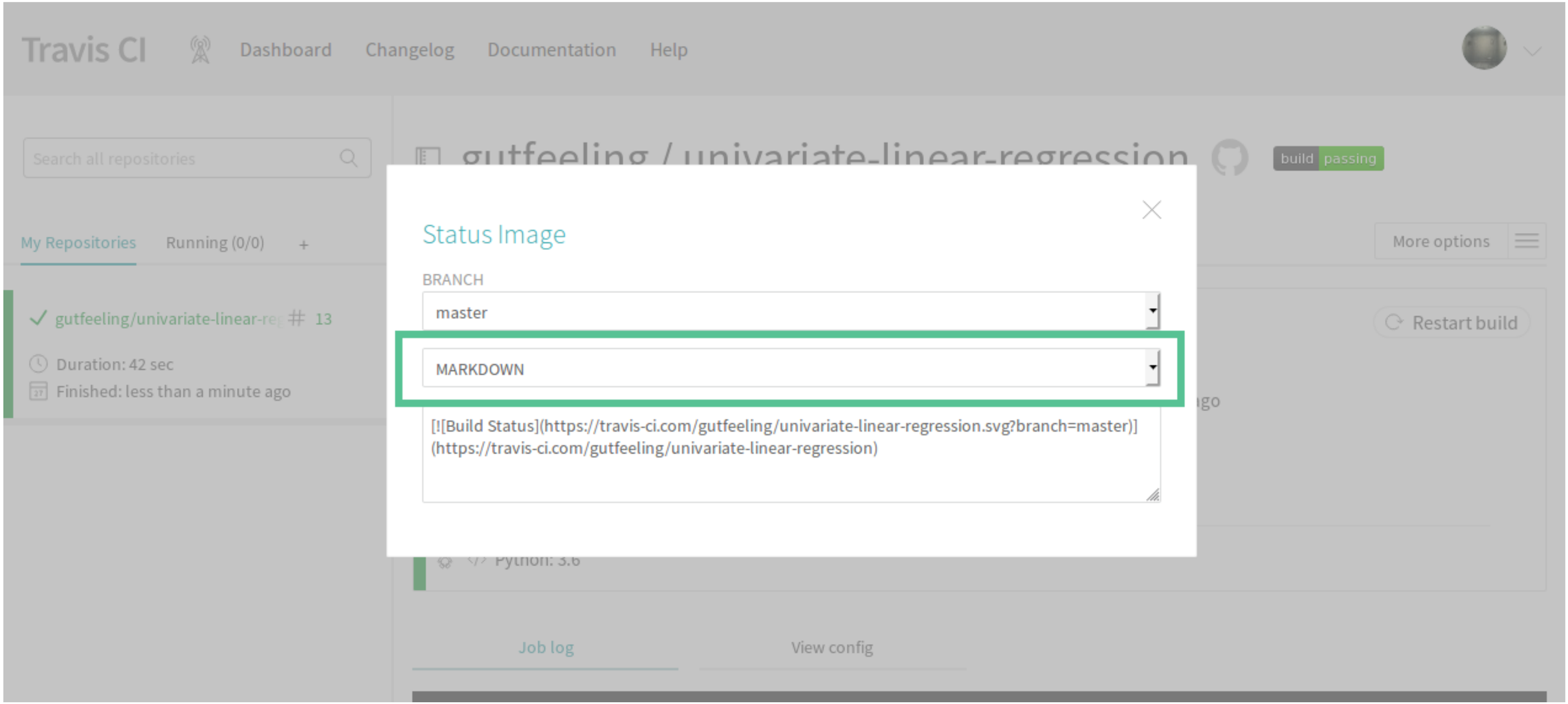
Every commit leads to a build



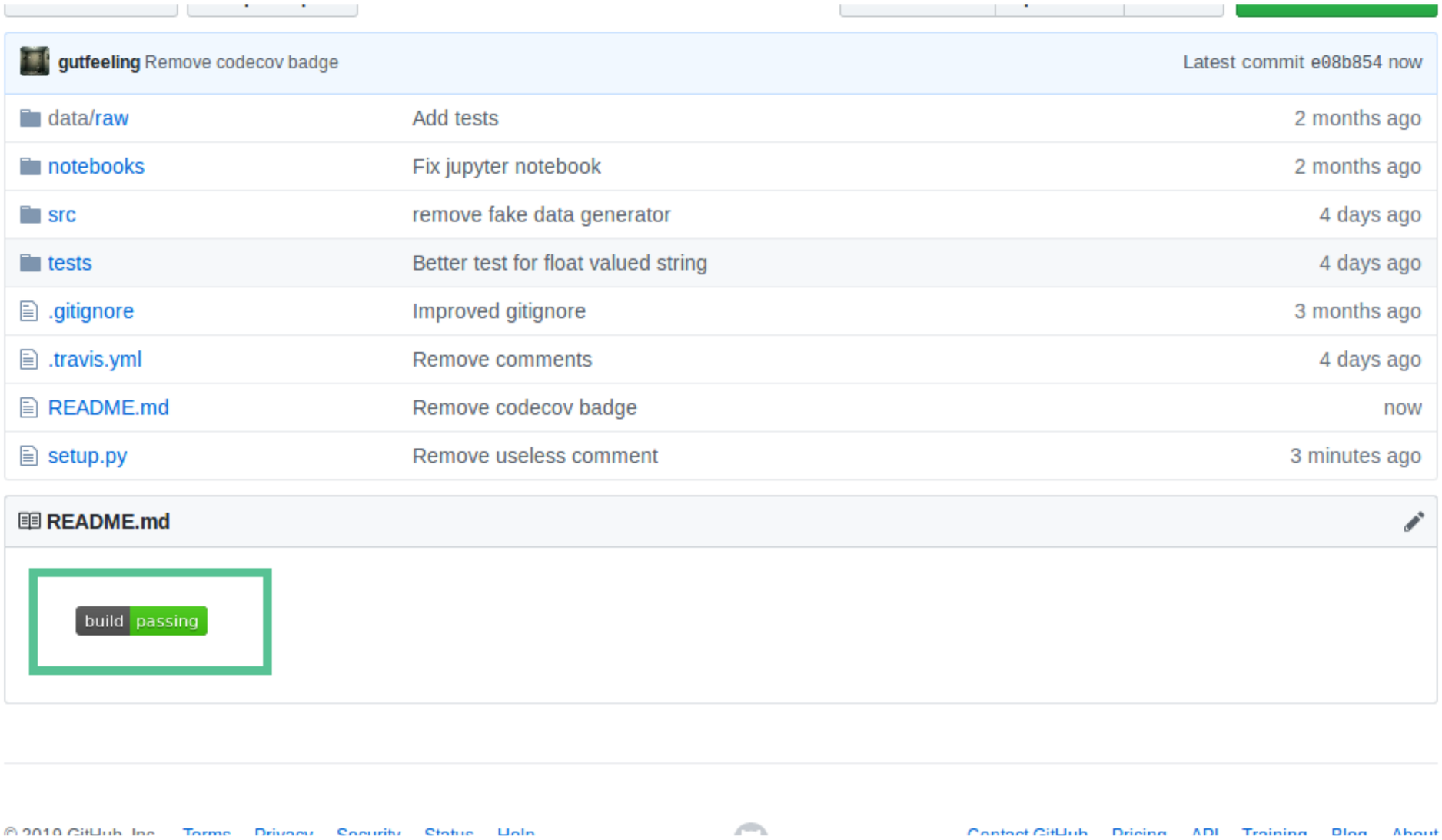
Step 4: Showing the build status badge



Step 4: Showing the build status badge



Step 4: Showing the build status badge



Code coverage

- $\text{code coverage} = \frac{\text{num lines of application code that ran during testing}}{\text{total num lines of application code}} \times 100$
- Higher percentages (75% and above) indicate well tested code.

Codecov

Step 1: Modify the Travis CI configuration file

- File: `.travis.yml`

```
language: python
python:
  - "3.6"
install:
  - pip install -e .

script:
  - pytest tests
```

27. Step 1: Modify the Travis CI configuration file
Finally, add a setting called after_success and add the command codecov. This makes Travis CI push the code coverage results to Codecov after every build.

28. Step 2: Install Codecov
To enable Codecov for our repository, we install the Codecov app in the GitHub marketplace in the same way we installed Travis CI.

Step 1: Modify the Travis CI configuration file

- File: `.travis.yml`

```
language: python
python:
  - "3.6"
install:
  - pip install -e .
  - pip install pytest-cov codecov    # Install packages for code coverage report
script:
  - pytest tests
```

Step 1: Modify the Travis CI configuration file

- File: `.travis.yml`

29. Commits lead to coverage report at codecov.io
From now, when we push a new commit, the code coverage report should show up in Codecov, accessible at codecov.io, after Travis CI completes the build.
30. Step 3: Showing the badge in GitHub
Go to the badge section in settings and paste the Markdown code to the GitHub README file.

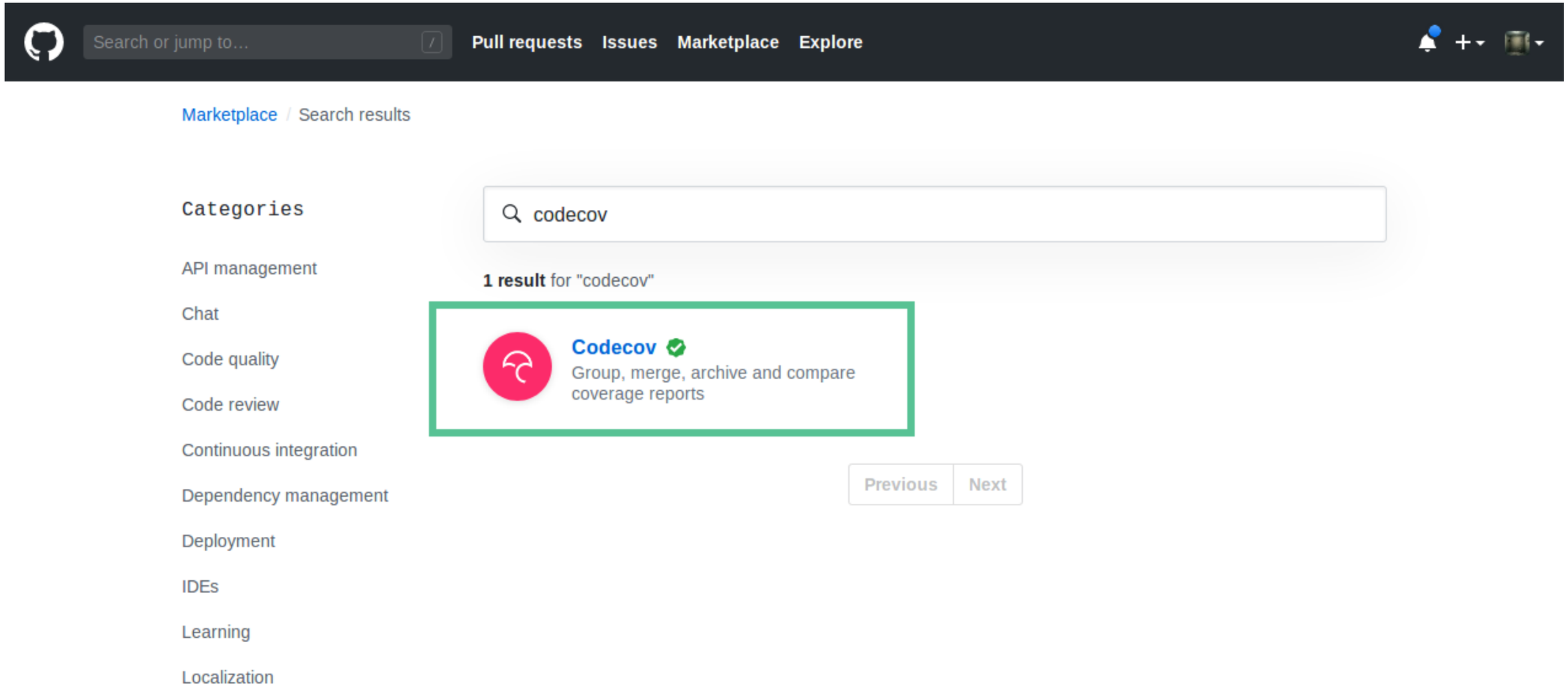
```
language: python
python:
  - "3.6"
install:
  - pip install -e .
  - pip install pytest-cov codecov    # Install packages for code coverage report
script:
  - pytest --cov=src tests            # Point to the source directory
```

Step 1: Modify the Travis CI configuration file

- File: `.travis.yml`

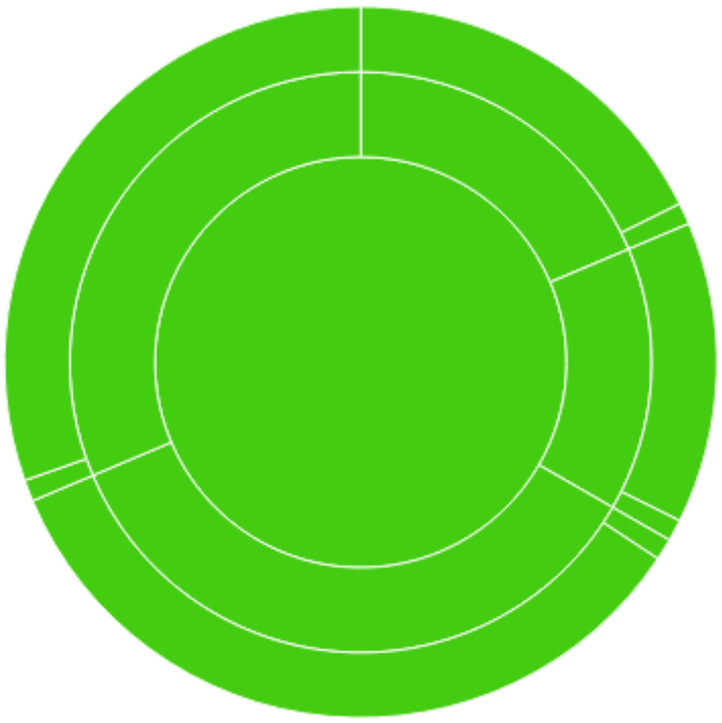
```
language: python
python:
  - "3.6"
install:
  - pip install -e .
  - pip install pytest-cov codecov    # Install packages for code coverage report
script:
  - pytest --cov=src tests            # Point to the source directory
after_success:
  - codecov                           # uploads report to codecov.io
```

Step 2: Install Codecov



Commits lead to coverage report at codecov.io

COVERAGE SUNBURST



/

ALL RECENT COMMITS

Fix side effects and call order in mocks — Browse Report
gutfeeling 7 days ago master 298091d CI Passed

Add pytest-mock — Browse Report
gutfeeling 7 days ago master 02e714a CI Passed

Add test for preprocessing function — Browse Report
gutfeeling 7 days ago master b8dfc76 CI Passed

Add baseline plots and add pytest-mpl requirement — Browse Report
gutfeeling 7 days ago master 752e0e0 CI Passed

Merge remote-tracking branch 'origin/master' — Browse Report
gutfeeling 7 days ago master 7ef5ef2 CI Passed

Add codecov badge — Browse Report
gutfeeling 10 days ago master 960fa3f CI Passed

View all recent commits

Step 3: Showing the badge in GitHub

gh gutfeeling univariate-linear-regression Docs Support

Overview Commits Branches Pulls Compare Settings

General

Yaml

Badge

Markdown Copy

```
[![codecov](https://codecov.io/gh/gutfeeling/univariate-linear-regression/branch/master/grapl
```

HTML Copy

```
<a href="https://codecov.io/gh/gutfeeling/univariate-linear-regression">
  <img src="https://codecov.io/gh/gutfeeling/univariate-linear-regression/branch/master/grapl
</a>
```

RST Copy

```
.. image:: https://codecov.io/gh/gutfeeling/univariate-linear-regression/branch/master/graph.
  :target: https://codecov.io/gh/gutfeeling/univariate-linear-regression
```

Step 3: Showing the badge in GitHub

gutfeeling Update README.md — Latest commit `0d34ed3` 4 minutes ago

data	Match the course code	12 minutes ago
notebooks	Match the course code	12 minutes ago
src	Match the course code	12 minutes ago
tests	Match the course code	12 minutes ago
.gitignore	Improved gitignore	4 months ago
.travis.yml	Remove comments	2 months ago
README.md	Update README.md	4 minutes ago
setup.py	Add pytest-mock	2 months ago

README.md

build passing | codecov 90%

This repository holds the code for the DataCamp course Unit Testing for Data Science in Python by Dibya Chakravorty.

Let's practice CI and code coverage!

UNIT TESTING FOR DATA SCIENCE IN PYTHON