Beyond assertion: setup and teardown

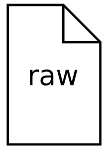
UNIT TESTING FOR DATA SCIENCE IN PYTHON

In this lesson, we are going to look at functions whose tests require more than assert statements.

Dibya Chakravorty
Test Automation Engineer

The preprocessing function

```python
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```




```
1,801    201,411
1,767565,112
2,002    333,209
1990    782,911
1,285    389129
```

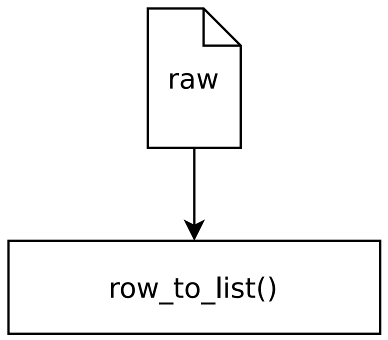
2. The preprocessing function
As an example, consider the function preprocess(), which accepts paths to a raw data file and a clean file as arguments. Let's say that the raw data file looks like this.

3. The preprocessing function
The function first applies row_to_list() on the rows. The second row has no tab separator,

The preprocessing function

```
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```

```
1,801    201,411
1,767565,112          # dirty row, no tab
2,002    333,209
1990    782,911
1,285    389129
```

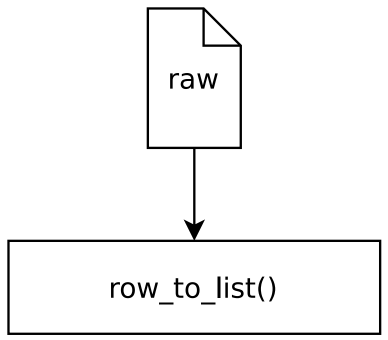


4. The preprocessing function
so row_to_list() filters it out.
5. The preprocessing function
convert_to_int() is applied next. The fourth and fifth rows are dirty
because the area and the price entry are missing commas respectively.

The preprocessing function

```
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```

```
1,801    201,411
2,002    333,209
1990    782,911
1,285    389129
```

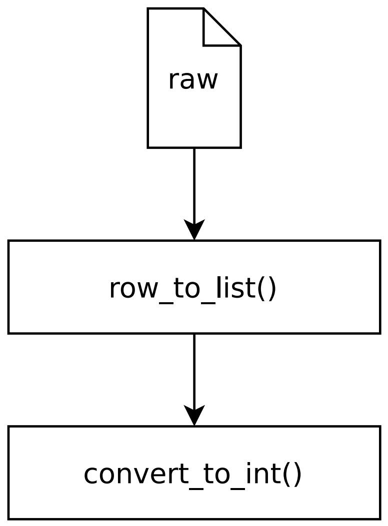


6. The preprocessing function
convert_to_int() filters them out.
7. The preprocessing function
For the two valid rows, convert_to_int() converts the comma separated strings into integers. The result is written to the clean file.

The preprocessing function

```python
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```

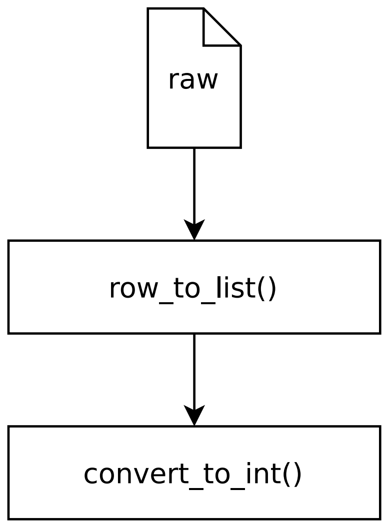
```
1,801    201,411
2,002    333,209
1990    782,911      # dirty row, no comma
1,285    389129      # dirty row, no comma
```



The preprocessing function

```python
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```

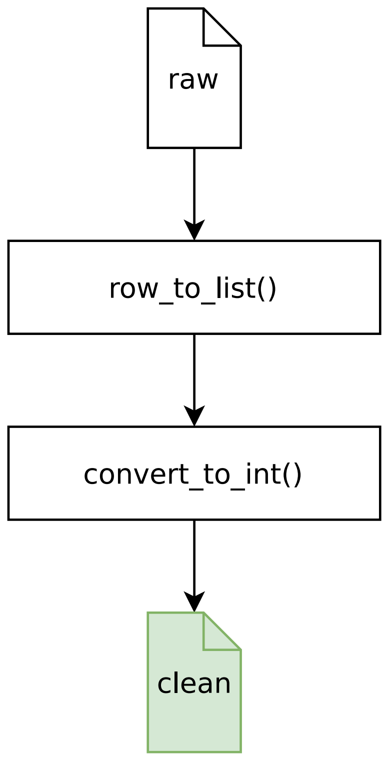
```
1,801     201,411
2,002     333,209
```



The preprocessing function

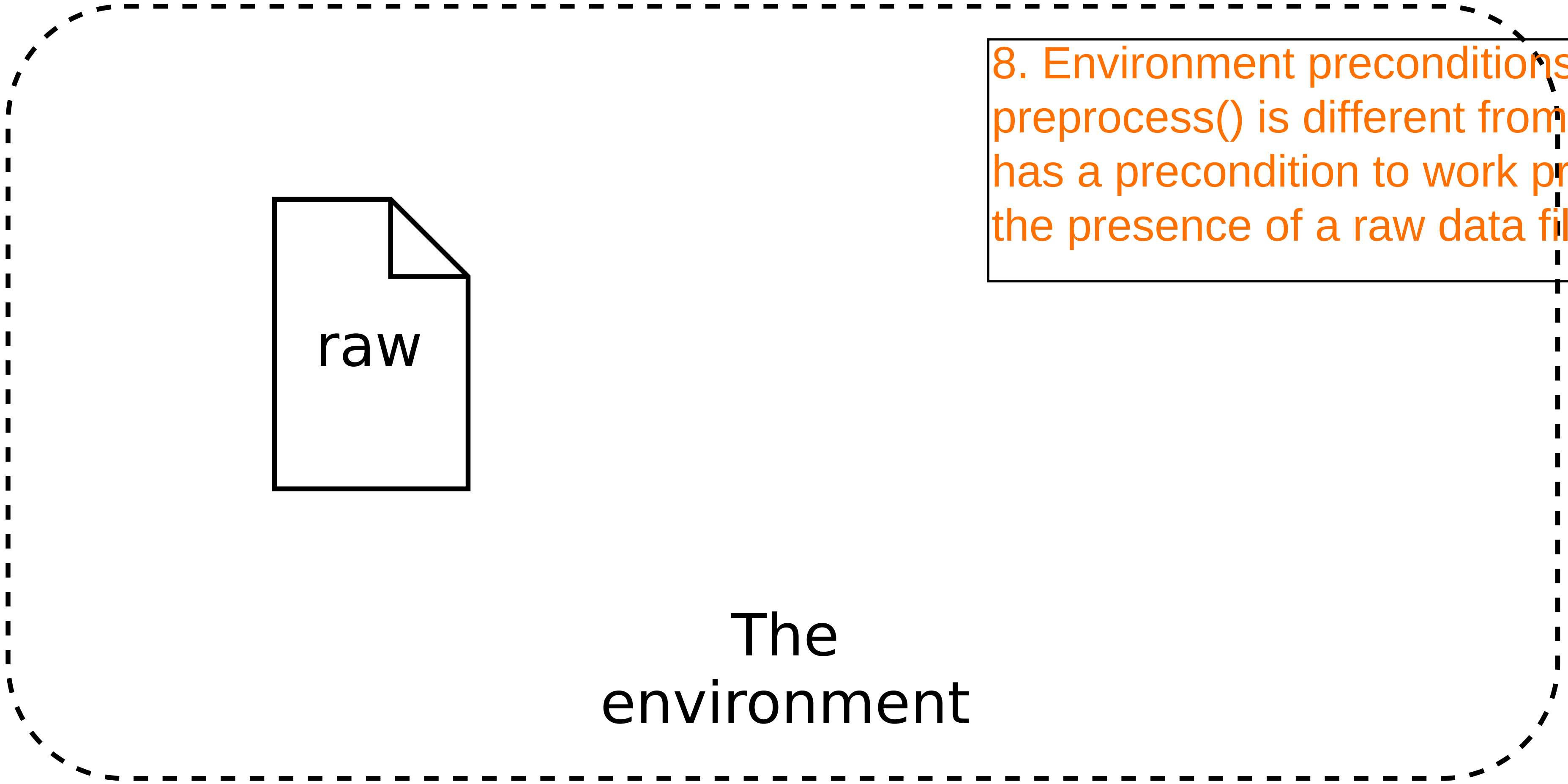
```
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```

```
1801    201411
2002    333209
```



Environment preconditions

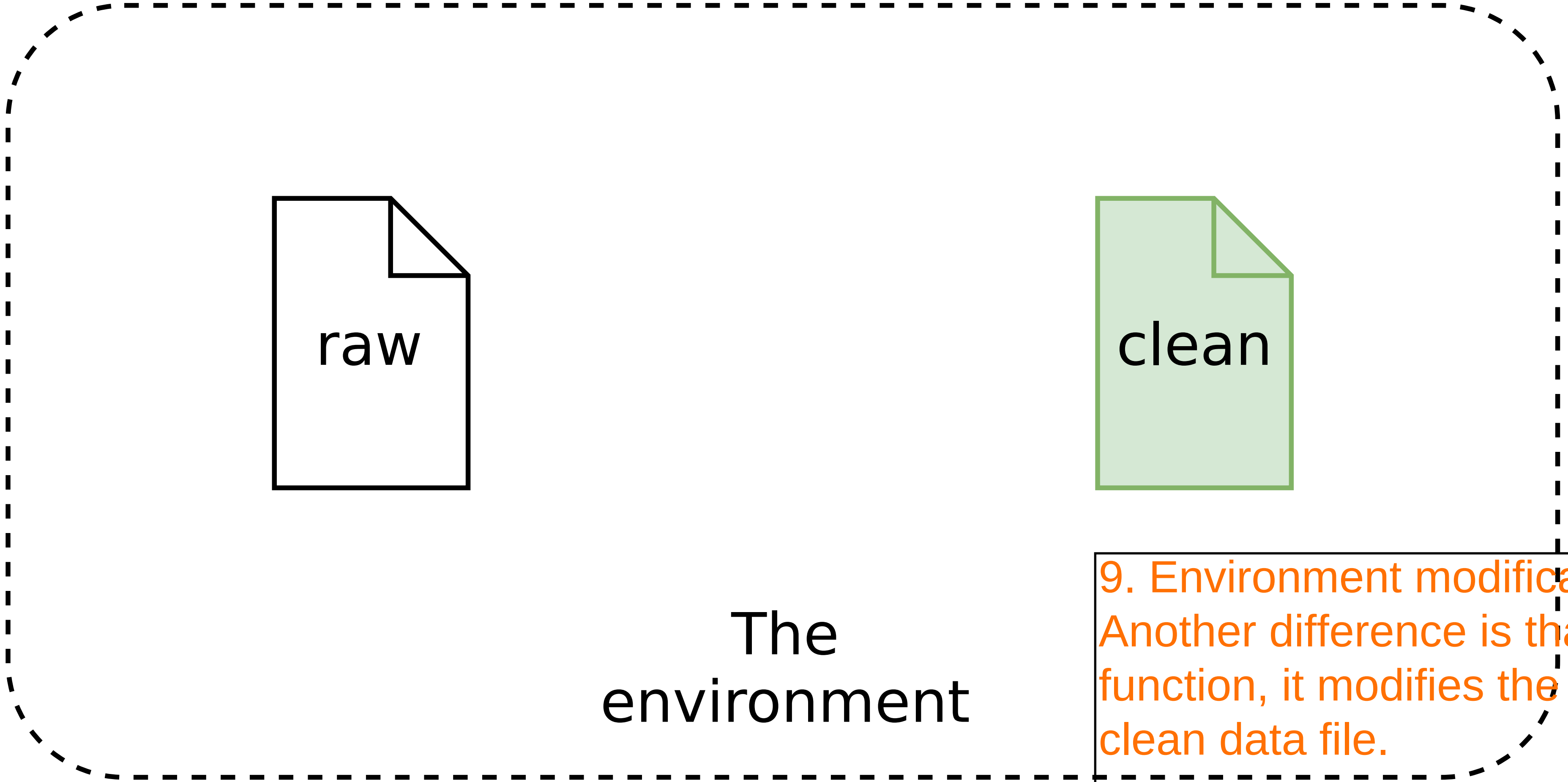
- `preprocess()` needs a raw data file in the environment to run.



8. Environment preconditions
preprocess() is different from other functions because it has a precondition to work properly. The precondition is the presence of a raw data file in the environment.

Environment modification

- `preprocess()` modifies the environment by creating a clean data file.



9. Environment modification
Another difference is that when we call the function, it modifies the environment by creating a clean data file.

10. Testing the preprocessing function
We create a test called test_on_raw_data() for this function.

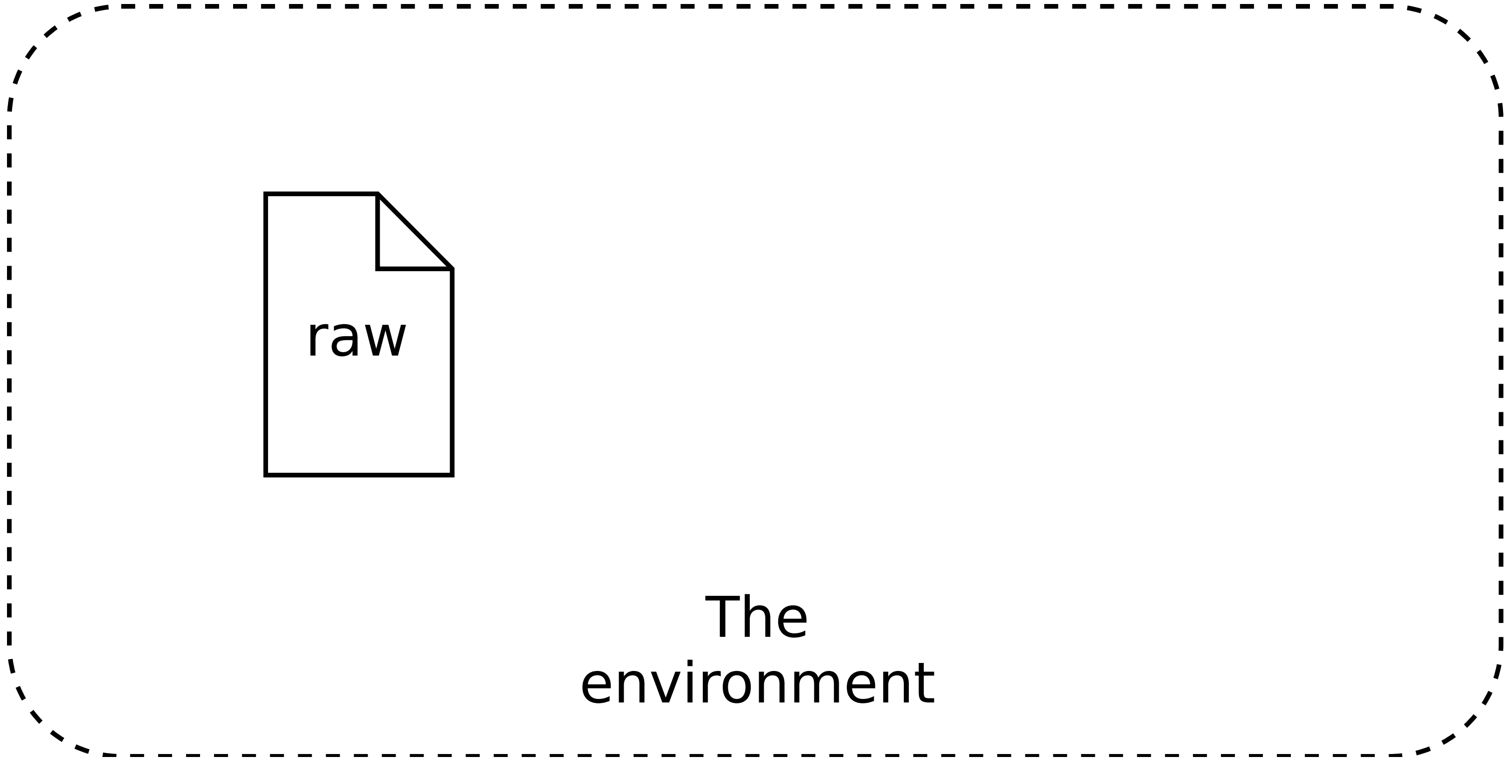
Testing the preprocessing function

```
def test_on_raw_data():
```

Step 1: Setup

```python
def test_on_raw_data():
    # Setup: create the raw data file
```

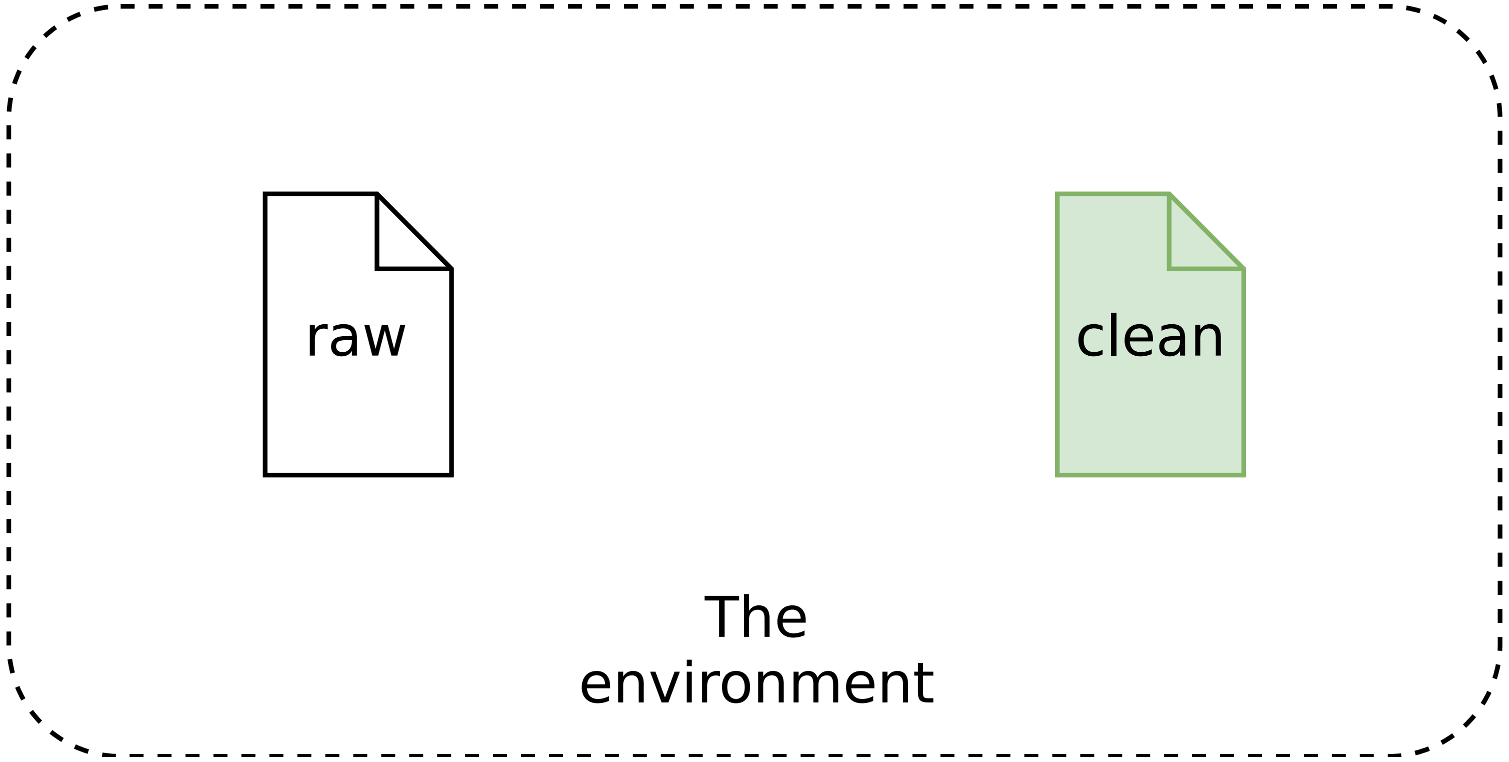
- Setup brings the environment to a state where testing can begin.



11. Step 1: Setup
We create the raw data file first. This step is called setup, and it is used to bring the environment to a state where testing can begin.

Step 2: Assert

```python
def test_on_raw_data():
    # Setup: create the raw data file
    preprocess(raw_data_file_path,
               clean_data_file_path
               )
    with open(clean_data_file_path) as f:
        lines = f.readlines()
    first_line = lines[0]
    assert first_line == "1801\t201411\n"
    second_line = lines[1]
    assert second_line == "2002\t333209\n"
```



12. Step 2: Assert
Then we call the function, which creates the clean data file. We open that file, read it and assert that it contains the expected lines.

Step 3: Teardown

```python
def test_on_raw_data():
    # Setup: create the raw data file
    preprocess(raw_data_file_path,
               clean_data_file_path
               )
    with open(clean_data_file_path) as f:
        lines = f.readlines()
    first_line = lines[0]
    assert first_line == "1801\t201411\n"
    second_line = lines[1]
    assert second_line == "2002\t333209\n"
    # Teardown: remove raw and clean data file
```

- Teardown brings environment to initial state.

13. Step 3: Teardown
Afterwards, we need to remove the clean and raw data file so that the next run of the test gets a clean environment. This step is called teardown, and it cleans any modification to the environment and brings it back to the initial state.

The new workflow

Old workflow

- assert

New workflow

- setup $\rightarrow$ assert $\rightarrow$ teardown

14. The new workflow
To summarize, instead of a sequence of assert statements, we have to follow the workflow: setup, assert and teardown.

15. Fixture
In pytest, the setup and teardown is placed outside the test, in a function called a fixture. A fixture is a function which has the pytest.fixture decorator. The first section is the setup. Then the function returns the data that the test needs. The test can access this data by calling the fixture passed as an argument.

Fixture

```python
import pytest


@pytest.fixture
def my_fixture():
    # Do setup here
    return data
```

16. Fixture
But instead of using the return keyword, the fixture function actually uses the yield keyword instead. The next section is the teardown. This section runs only when the test has finished executing.

```python
def test_something(my_fixture):
    ...
    data = my_fixture
    ...
```

Fixture

```python
import pytest

@pytest.fixture
def my_fixture():
    # Do setup here
    yield data    # Use yield instead of return
    # Do teardown here
```

```python
def test_something(my_fixture):
    ...
    data = my_fixture
    ...
```

17. Fixture example
Let's see an example of how this works for the test test_on_raw_data().

18. Fixture example
We create a fixture called raw_and_clean_data_file(). In setup, we create the paths to the raw and clean data file. Next, we write the raw data to the raw data file. Finally, we yield the paths as a tuple. The test calls the fixture and gets the paths required to call the preprocess() function. Then we proceed to the assert statements. In the teardown section, we remove both raw and clean data file using the os.remove() function.

Test

```
import os
import pytest

def test_on_raw_data():
```

Fixture

```python
@pytest.fixture
def raw_and_clean_data_file():
    raw_data_file_path = "raw.txt"
    clean_data_file_path = "clean.txt"
    with open(raw_data_file_path, "w") as f:
        f.write("1,801\t201,411\n"
                "1,767565,112\n"
                "2,002\t333,209\n"
                "1990\t782,911\n"
                "1,285\t389129\n"
                )
    yield raw_data_file_path, clean_data_file_path
    os.remove(raw_data_file_path)
    os.remove(clean_data_file_path)
```

Test

```python
import os
import pytest

def test_on_raw_data(raw_and_clean_data_file):
    raw_path, clean_path = raw_and_clean_data_file
    preprocess(raw_path, clean_path)
    with open(clean_data_file_path) as f:
        lines = f.readlines()
    first_line = lines[0]
    assert first_line == "1801\t201411\n"
    second_line = lines[1]
    assert second_line == "2002\t333209\n"
```

18. Fixture example
We create a fixture called raw_and_clean_data_file(). In setup, we create the paths to the raw and clean data file. Next, we write the raw data to the raw data file. Finally, we yield the paths as a tuple. The test calls the fixture and gets the paths required to call the preprocess() function. Then we proceed to the assert statements. In the teardown section, we remove both raw and clean data file using the os.remove() function.

The built-in tmpdir fixture

- **Setup**: create a temporary directory.
- **Teardown**: delete the temporary directory along with contents.

19. The built-in tmpdir fixture
There is a built-in pytest fixture called tmpdir, which is useful when dealing with files. This fixture creates a temporary directory during setup and deletes the temporary directory during teardown.

tmpdir and fixture chaining

- setup of `tmpdir()` → Setup of `raw_and_clean_data_file()` → test → teardown of `raw_and_clean_data_file()` → teardown of `tmpdir()` .

```python
@pytest.fixture
def raw_and_clean_data_file(tmpdir):
    raw_data_file_path = tmpdir.join("raw.txt")
    clean_data_file_path = tmpdir.join("clean.txt")
    with open(raw_data_file_path, "w") as f:
        f.write("1,801\t201,411\n"
                "1,767565,112\n"
                "2,002\t333,209\n"
                "1990\t782,911\n"
                "1,285\t389129\n"
                )
    yield raw_data_file_path, clean_data_file_path
    # No teardown code necessary
```

20. tmpdir and fixture chaining
We can pass this fixture as an argument to our fixture. This is called fixture chaining, which results in the setup of tmpdir to be called first, followed by the setup of our fixture. When the test finishes, the teardown of our fixture is called first, followed by the teardown of tmpdir. The tmpdir argument supports all os.path commands such as join. We use the join function of tmpdir to create the raw and clean data file inside the temporary directory. The rest of the setup looks identical. The awesome thing is: we can omit the teardown code in our fixture entirely, because the teardown of tmpdir will delete all files in the temporary directory when the test ends.

Let's practice setup and teardown using fixtures!

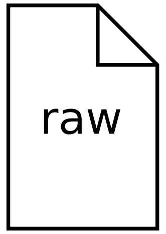
UNIT TESTING FOR DATA SCIENCE IN PYTHON

Mocking

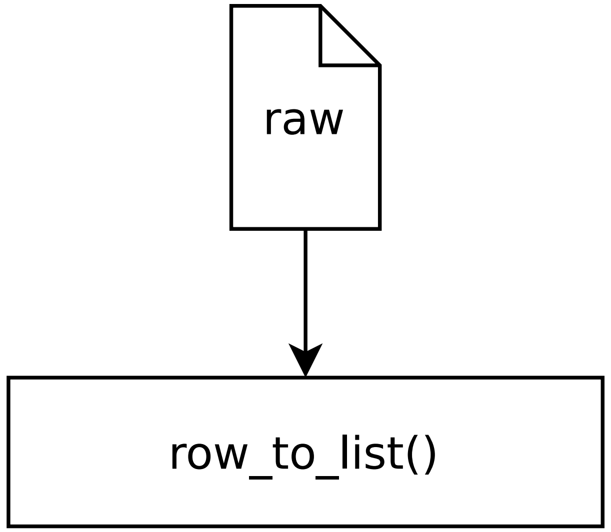
UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

The preprocessing function

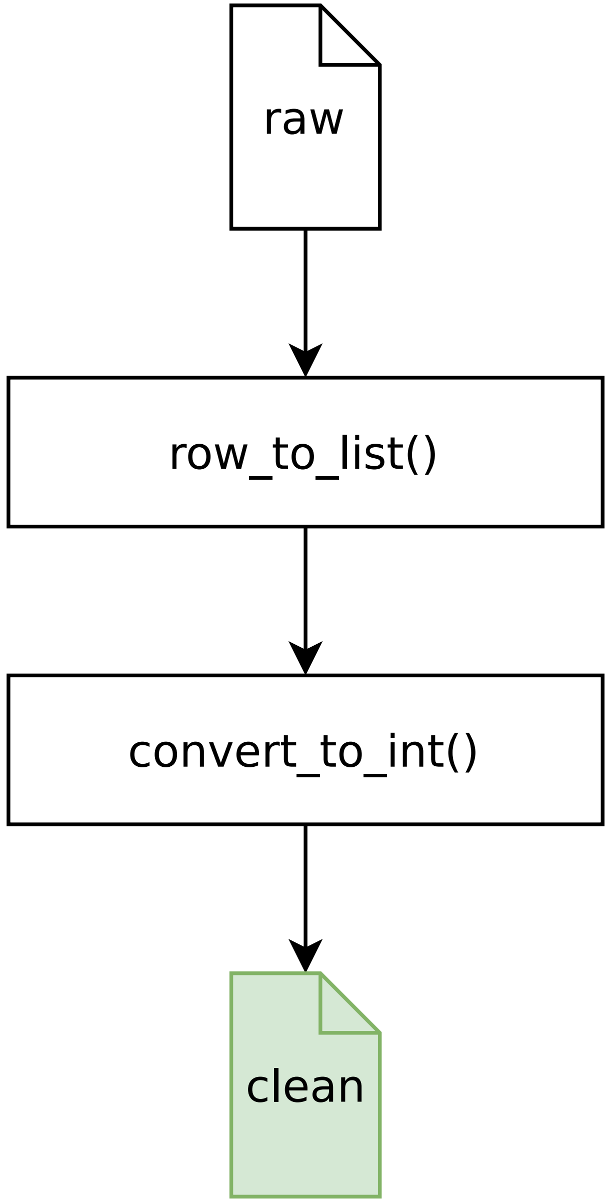


The preprocessing function



1. Mocking
In the previous lesson, we tested the preprocess() function.
2. The preprocessing function
preprocess() applies
3. The preprocessing function
row_to_list() and
4. The preprocessing function
convert_to_int() sequentially to the raw data file to create a clean data file.
5. Test result depend on dependencies
If the tests for preprocess() were to pass, row_to_list() and convert_to_int() must also work as expected.

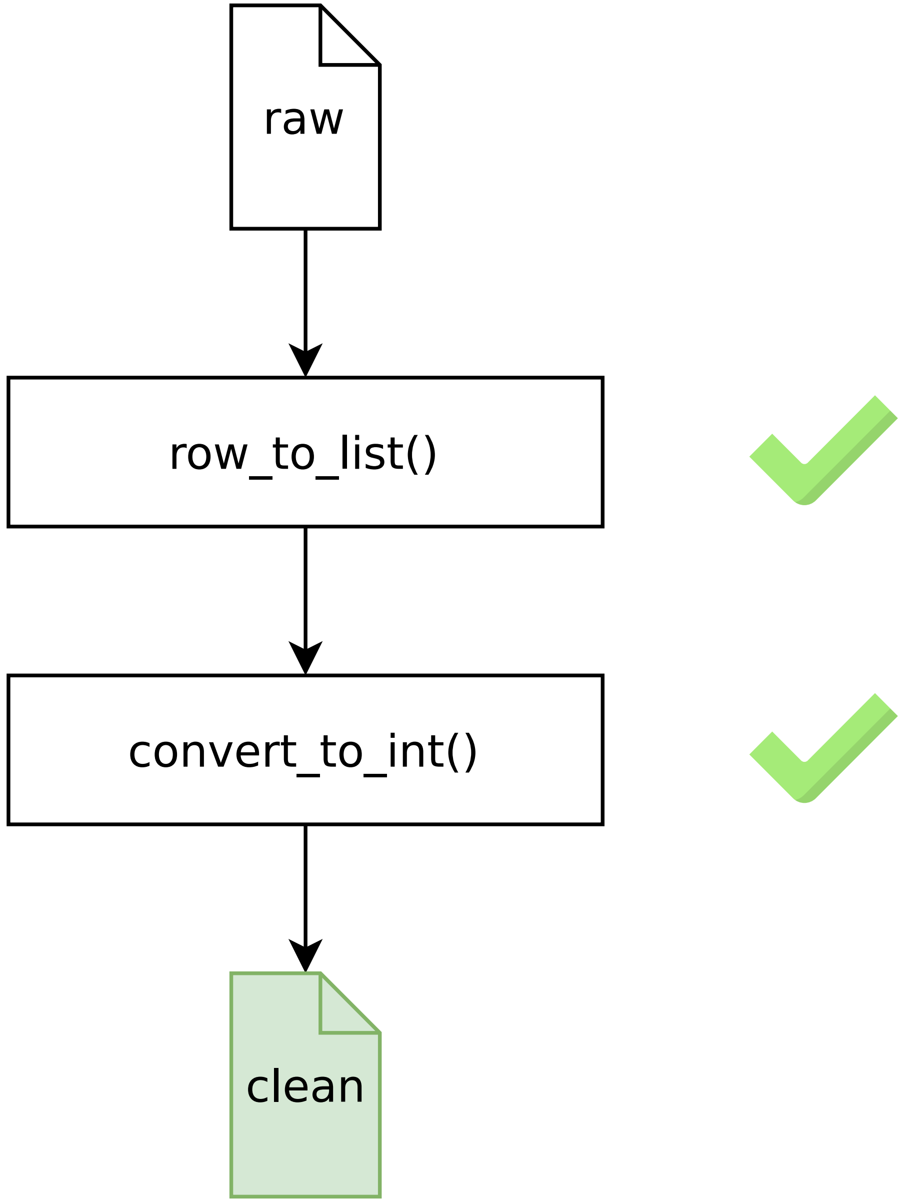
The preprocessing function




```
pytest -k "TestPreprocess"
```

```
=============== test session starts ================
...
collected 21 items / 20 deselected / 1 selected

data/test_preprocessing_helpers.py .            [100%]

===== 1 passed, 20 deselected in 0.61 seconds ======
```

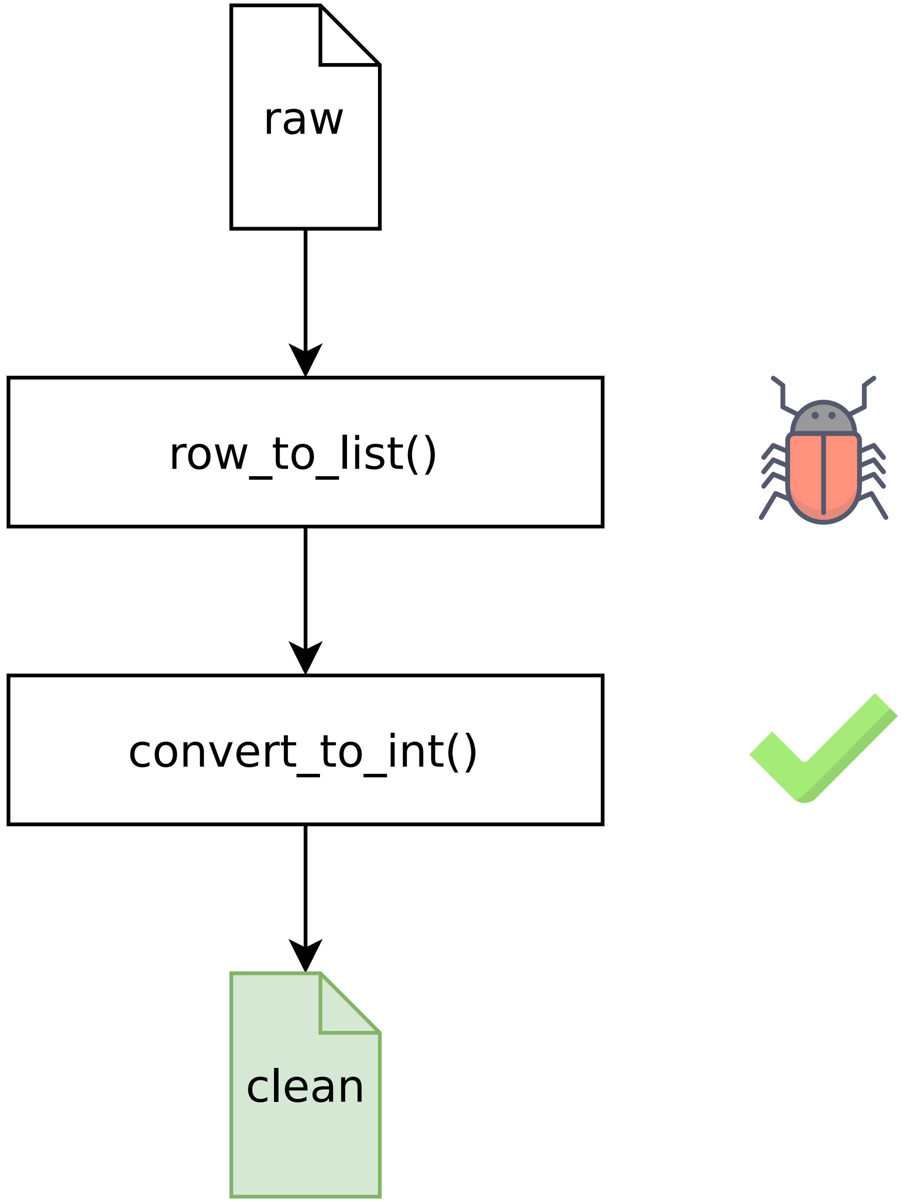
Test result depend on dependencies




```
pytest -k "TestPreprocess"
```

```
=============== test session starts ================
...
collected 21 items / 20 deselected / 1 selected

data/test_preprocessing_helpers.py .          [100%]

===== 1 passed, 20 deselected in 0.61 seconds ======
```

Test result depend on dependencies




```
pytest -k "TestPreprocess"
```

```
=============== test session starts ================
...
collected 21 items / 20 deselected / 1 selected

data/test_preprocessing_helpers.py F         [100%]

===================== FAILURES ======================
_________ TestPreprocess.test_on_raw_data __________

    def test_on_raw_data(self, raw_and_clean_data_file):
        raw_path, clean_path = raw_and_clean_data_file
        preprocess(raw_path, clean_path)
        with open(clean_path, "r") as f:
            lines = f.readlines()
>       first_line = lines[0]
E       IndexError: list index out of range

data/test_preprocessing_helpers.py:121: IndexError
===== 1 failed, 20 deselected in 0.68 seconds ======
```

Test result depends on dependencies

Test result should indicate bugs in

- function under test i.e. `preprocess()`.
- not dependencies e.g. `row_to_list()` or `convert_to_int()`.

6. Test result depend on dependencies
If any of them has a bug, the tests for preprocess() will not pass, even if preprocess() has no bugs.
7. Test result depends on dependencies
But the test results should be indicative of the bugs in the function under test, and not bugs in any of its dependencies.

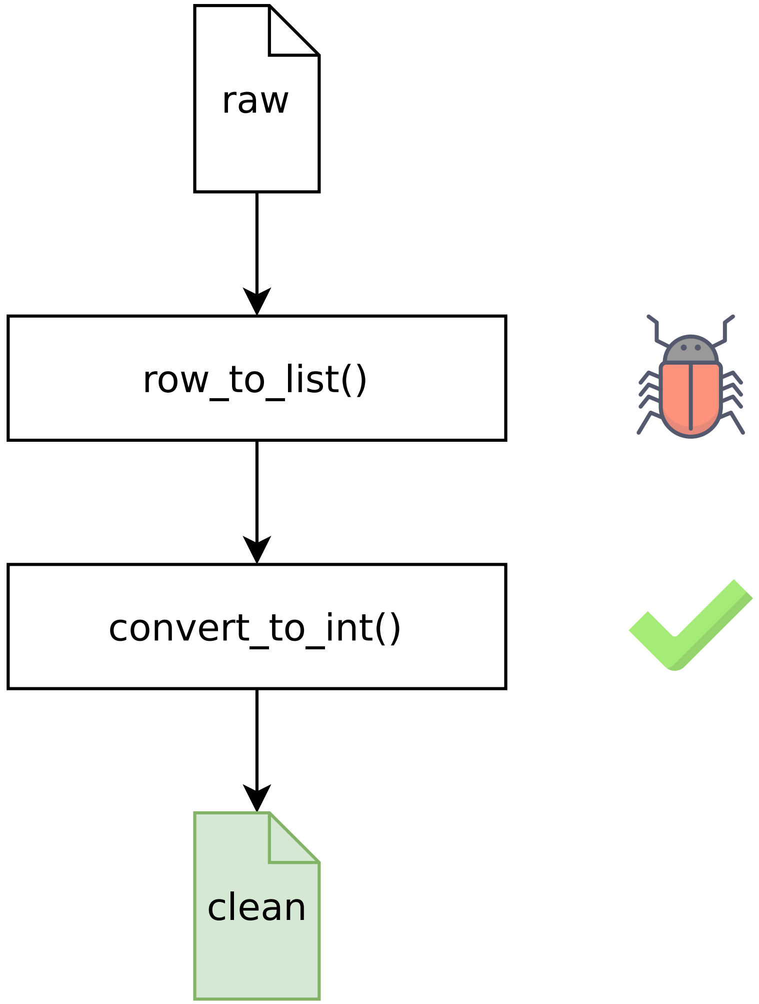
Mocking: testing functions independently of dependencies

Packages for mocking in `pytest`

- `pytest-mock` : Install using `pip install pytest-mock` .
- `unittest.mock` : Python standard library package.

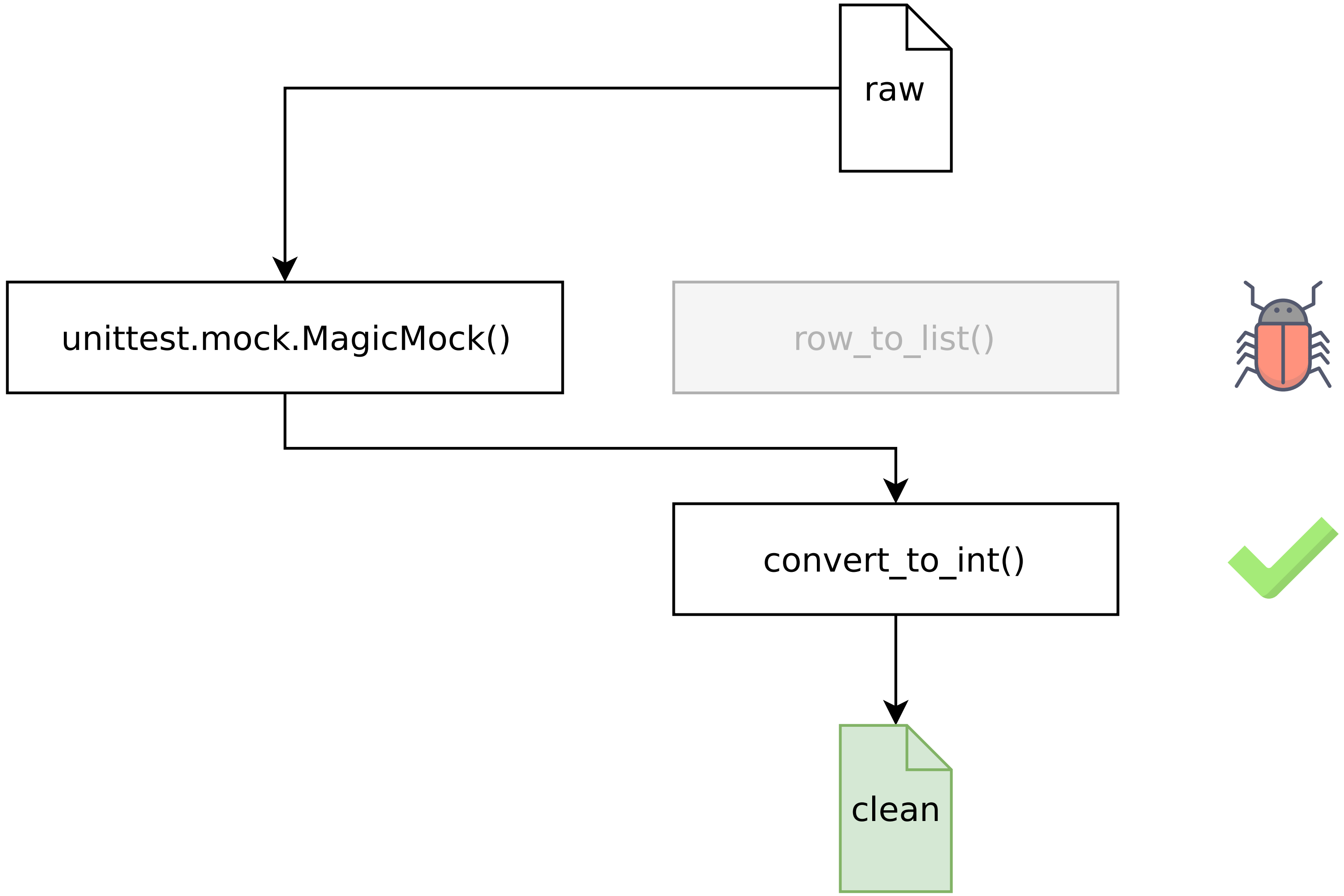
8. Mocking: testing functions independently of dependencies
In this lesson, we will learn a trick which will allow us to test a function independently of its dependencies. This very useful trick is called mocking. To use mocking in pytest, we will need two packages. The first one is pytest-mock and we can install it using pip. The second one is a standard library package called unittest.mock.

MagicMock() and mocker.patch()



9. MagicMock() and mocker.patch()
The basic idea of mocking
10. MagicMock() and mocker.patch()
is to replace potentially buggy dependencies such as row_to_list() with the object unittest.mock.MagicMock(), but only during testing. This replacement is done using a fixture called mocker, and calling its patch method right at the beginning of the test test_on_raw_data(), which we wrote in the last lesson.

MagicMock() and mocker.patch()




```python
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(...)
```

11. MagicMock() and mocker.patch()
The first argument of the mocker.patch method is the fully qualified name of the dependency including module name, as registered by the function under test.
12. MagicMock() and mocker.patch()
preprocess() knows row_to_list() as data.preprocessing_helpers.row_to_list, so that's what we will use here. The mocker.patch method returns the MagicMock object which we store in the variable row_to_list_mock.

MagicMock() and mocker.patch()

- Theoretical structure of `mocker.patch()`

```
mocker.patch("<dependency name with module name>")
```

```
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(...)
```

MagicMock() and mocker.patch()

- Theoretical structure of `mocker.patch()`

```
mocker.patch("data.preprocessing_helpers.row_to_list")
```

```
unittest.mock.MagicMock()
```

```
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(
        "data.preprocessing_helpers.row_to_list"
      )
```

Making the MagicMock() bug-free

- Raw data

```
1,801	201,411
1,767565,112
2,002	333,209
1990	782,911
1,285	389129
```

```python
def row_to_list_bug_free(row):
    return_values = {
        "1,801\t201,411\n": ["1,801", "201,411"],
        "1,767565,112\n": None,
        "2,002\t333,209\n": ["2,002", "333,209"],
        "1990\t782,911\n": ["1990", "782,911"],
        "1,285\t389129\n": ["1,285", "389129"],
        }
    return return_values[row]
```

```python
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(
        "data.preprocessing_helpers.row_to_list"
        )
    row_to_list_mock.side_effect = row_to_list_bug_free
```

13. Making the MagicMock() bug-free
During the test, row_to_list_mock can be programmed to behave like a bug-free replacement of row_to_list(). We call the bug free version of row_to_list() as row_to_list_bug_free(). Note that this only needs to be bug-free in the context of the test, and therefore, can be much simpler than the actual row_to_list() function. In the test, we use the following raw data file, which we already saw in the last lesson. The row_to_list_bug_free() simply needs to return the correct result for these five rows. Therefore, we create a dictionary containing the correct results for these five rows and return the results from the dictionary. Then we set the side_effect attribute of the MagicMock() object to the bug-free version.

Side effect

- Raw data

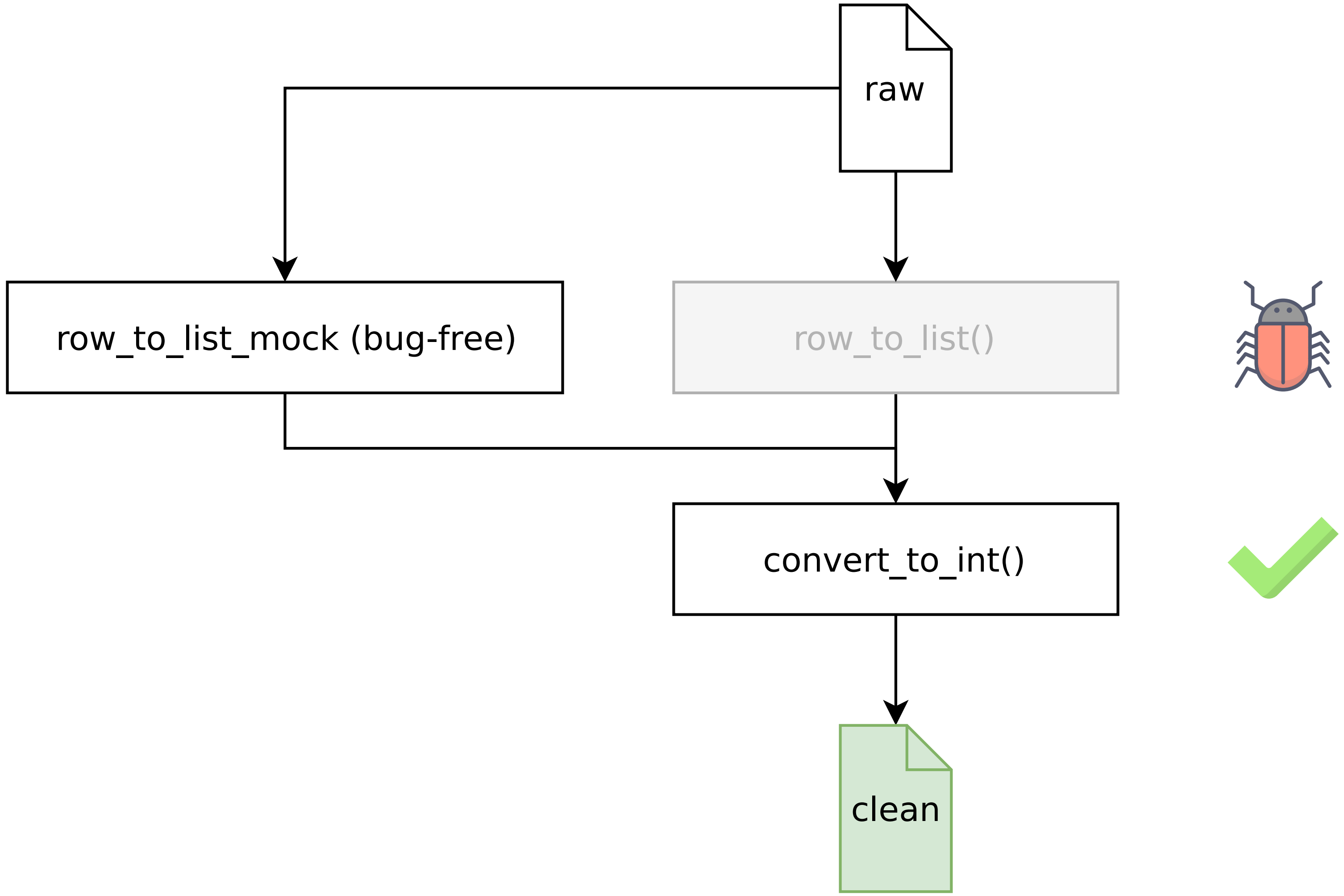
```
1,801	201,411
1,767565,112
2,002	333,209
1990	782,911
1,285	389129
```

```python
def row_to_list_bug_free():
    return_values = {
        "1,801\t201,411\n": ["1,801", "201,411"],
        "1,767565,112\n": None,
        "2,002\t333,209\n": ["2,002", "333,209"],
        "1990\t782,911\n": ["1990", "782,911"],
        "1,285\t389129\n": ["1,285", "389129"],
        }
    return return_values[row]
```

```python
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(
        "data.preprocessing_helpers.row_to_list",
        side_effect = row_to_list_bug_free
        )
```

14. Side effect
We can also set the side_effect attribute by passing side_effect as a keyword argument to mocker.patch. mocker.patch treats any keyword argument that it does not recognize as an attribute of the returned MagicMock() object and sets the attribute value accordingly.

Bug free replacement of dependency




```
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(
        "data.preprocessing_helpers.row_to_list",
        side_effect = row_to_list_bug_free
        )
    preprocess(raw_path, clean_path)
```

15. Bug free replacement of dependency
From this point on, when we call preprocess() in the test, the bug-free mock of row_to_list() will be used, and the test will not encounter any bugs.

Checking the arguments

- `call_args_list` attribute returns a list of arguments that the mock was called with

```
row_to_list_mock.call_args_list
```

```
[call("1,801\t201,411\n"),
 call("1,767565,112\n"),
 call("2,002\t333,209\n"),
 call("1990\t782,911\n"),
 call("1,285\t389129\n")
 ]
```

```python
def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(
        "data.preprocessing_helpers.row_to_list",
        side_effect = row_to_list_bug_free
        )
    preprocess(raw_path, clean_path)
```

16. Checking the arguments
We can also check if preprocess() is calling row_to_list() with the correct arguments. The call_args_list attribute is a list of all the arguments that row_to_list_mock was called with, wrapped in a convenience object called call().

Checking the arguments

- `call_args_list` attribute returns a list of arguments that the mock was called with

```
row_to_list_mock.call_args_list
```

```
[call("1,801\t201,411\n"),
 call("1,767565,112\n"),
 call("2,002\t333,209\n"),
 call("1990\t782,911\n"),
 call("1,285\t389129\n")
 ]
```

17. Checking the arguments
This convenience object can be imported from unittest.mock, and we import it at the top of the test. In the test, we can assert that the call_args_list attribute is the expected list containing the five rows of the raw data file in the correct order.

```python
from unittest.mock import call

def test_on_raw_data(raw_and_clean_data_file,
                     mocker,
                     ):
    raw_path, clean_path = raw_and_clean_data_file
    row_to_list_mock = mocker.patch(
        "data.preprocessing_helpers.row_to_list",
        side_effect = row_to_list_bug_free
        )
    preprocess(raw_path, clean_path)
    assert row_to_list_mock.call_args_list == [
        call("1,801\t201,411\n"),
        call("1,767565,112\n"),
        call("2,002\t333,209\n"), call("1990\t782,911\n"),
        call("1,285\t389129\n")
        ]
```

Dependency buggy, function bug-free, test still passes!

```
pytest -k "TestRowToList"
```

```
========================== test session starts ============================
collected 21 items / 14 deselected / 7 selected

data/test_preprocessing_helpers.py .....FF                          [100%]

================================ FAILURES =================================
________________ TestRowToList.test_on_normal_argument_1 __________________

...
________________ TestRowToList.test_on_normal_argument_2 __________________

...
============ 2 failed, 5 passed, 14 deselected in 0.70 seconds ============
```

18. Dependency buggy, function bug-free, test still passes!
We have prepared a scenario where row_to_list() contains a bug but preprocess() doesn't. If we run the tests for both functions, we see that the some tests for row_to_list() fail,
19. Dependency buggy, function bug-free, test still passes!
but the test for preprocess() passes. That's exactly the behavior we wanted!

Dependency buggy, function bug-free, test still passes!

```
pytest -k "TestPreprocess"
```

```
=========================== test session starts ============================
collected 21 items / 20 deselected / 1 selected

data/test_preprocessing_helpers.py .                                [100%]

================= 1 passed, 20 deselected in 0.63 seconds ==================
```

Let's practice mocking!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Testing models

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

Functions we have tested so far

- `preprocess()`
- `get_data_as_numpy_array()`
- `split_into_training_and_testing_sets()`

Raw data to clean data

```python
from data.preprocessing_helpers import preprocess
from features.as_numpy import get_data_as_numpy_array
from models.train import (
  split_into_training_and_testing_sets
  )

preprocess("data/raw/housing_data.txt",
           "data/clean/clean_housing_data.txt"
           )
```

```
data
|-- raw
|    |-- housing_data.txt
|-- clean
|
src
tests
```

`data/raw/housing_data.txt`

```
2,081    314,942
1,059    186,606
    293,410     <-- row with missing area
1,148    206,186
...
```

Raw data to clean data

```python
from data.preprocessing_helpers import preprocess
from features.as_numpy import get_data_as_numpy_array
from models.train import (
  split_into_training_and_testing_sets
  )

preprocess("data/raw/housing_data.txt",
           "data/clean/clean_housing_data.txt"
           )
```

```
data
|-- raw
|     |-- housing_data.txt
|-- clean
|     |-- clean_housing_data.txt
src
tests
```

`data/clean/clean_housing_data.txt`

```
2081    314942
1059    186606
1148    206186
...
```

Clean data to NumPy array

```
from data.preprocessing_helpers import preprocess
from features.as_numpy import get_data_as_numpy_array
from models.train import (
  split_into_training_and_testing_sets
  )

preprocess("data/raw/housing_data.txt",
           "data/clean/clean_housing_data.txt"
           )
data = get_data_as_numpy_array(
    "data/clean/clean_housing_data.txt", 2
    )
```

```
get_data_as_numpy_array(
    "data/clean/clean_housing_data.txt", 2
    )
```

```
array([[  2081., 314942.],
       [  1059., 186606.],
       [  1148., 206186.]
       ...
       ]
      )
```

Splitting into training and testing sets

```python
from data.preprocessing_helpers import preprocess
from features.as_numpy import get_data_as_numpy_array
from models.train import (
  split_into_training_and_testing_sets
  )

preprocess("data/raw/housing_data.txt",
           "data/clean/clean_housing_data.txt"
           )
data = get_data_as_numpy_array(
    "data/clean/clean_housing_data.txt", 2
    )
training_set, testing_set = (
    split_into_training_and_testing_sets(data)
    )
```

```python
split_into_training_and_testing_sets(data)
```

```
(array([[1148, 206186],     # Training set (3/4)
        [2081, 314942],
        ...
        ]
       ),
 array([[1059, 186606]      # Testing set (1/4)
        ...
        ]
       )
)
```

6. Splitting into training and testing sets
Finally, we can apply split_into_training_and_testing_set() to randomly split this NumPy array row-wise in the ratio 3:1. Three fourth of the data will be used for training a linear regression model. The rest will be used to test the model.

Functions are well tested - thanks to you!

The linear regression model

```python
def train_model(training_set):
```

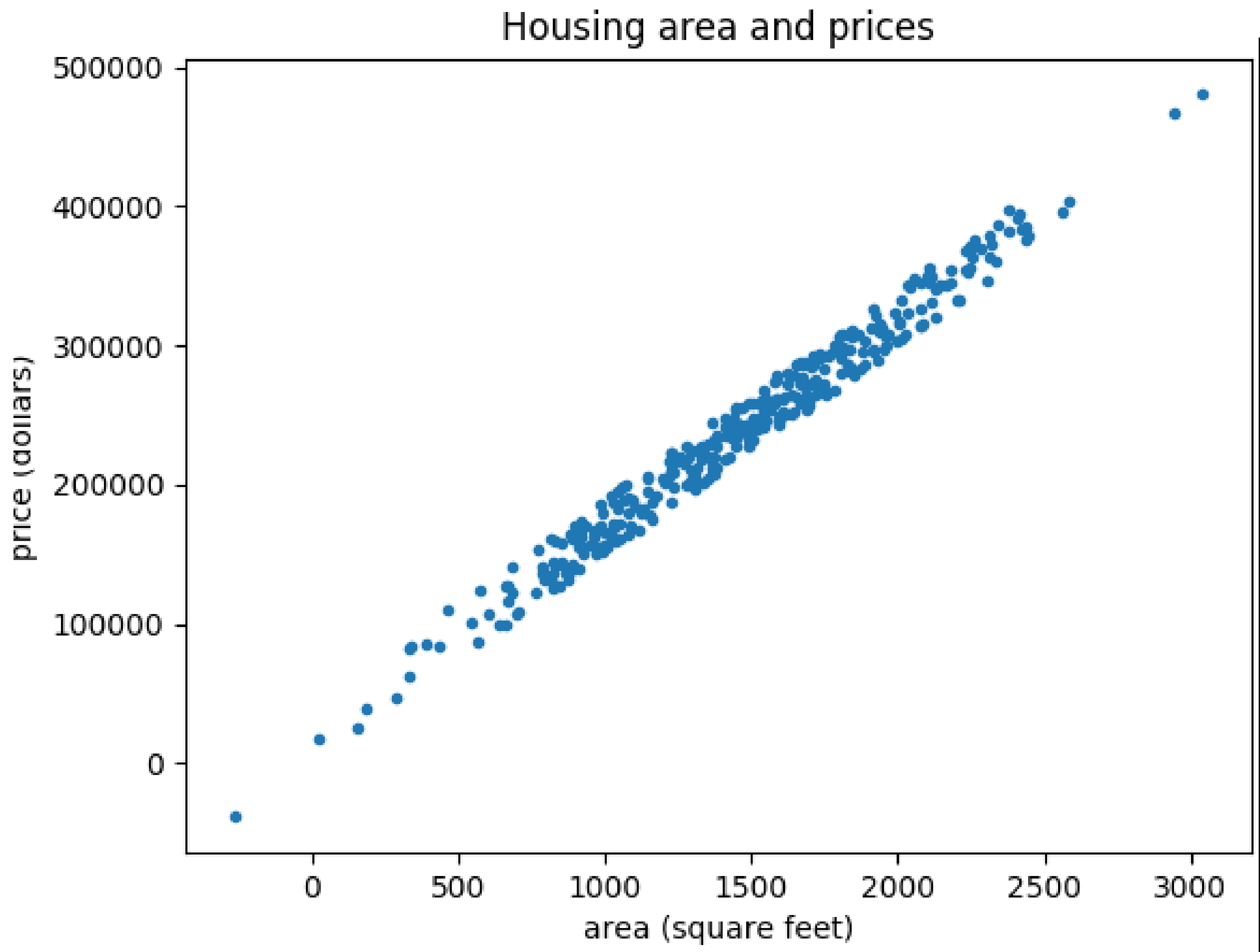
8. The linear regression model
It's time now to train a linear regression model using the function train_model(), which takes the training set as the only argument. The training set has areas in the first column and prices in the second column.

The linear regression model

```
from scipy.stats import linregress


def train_model(training_set):
    slope, intercept, _, _, _ = linregress(training_set[:, 0], training_set[:, 1])
    return slope, intercept
```

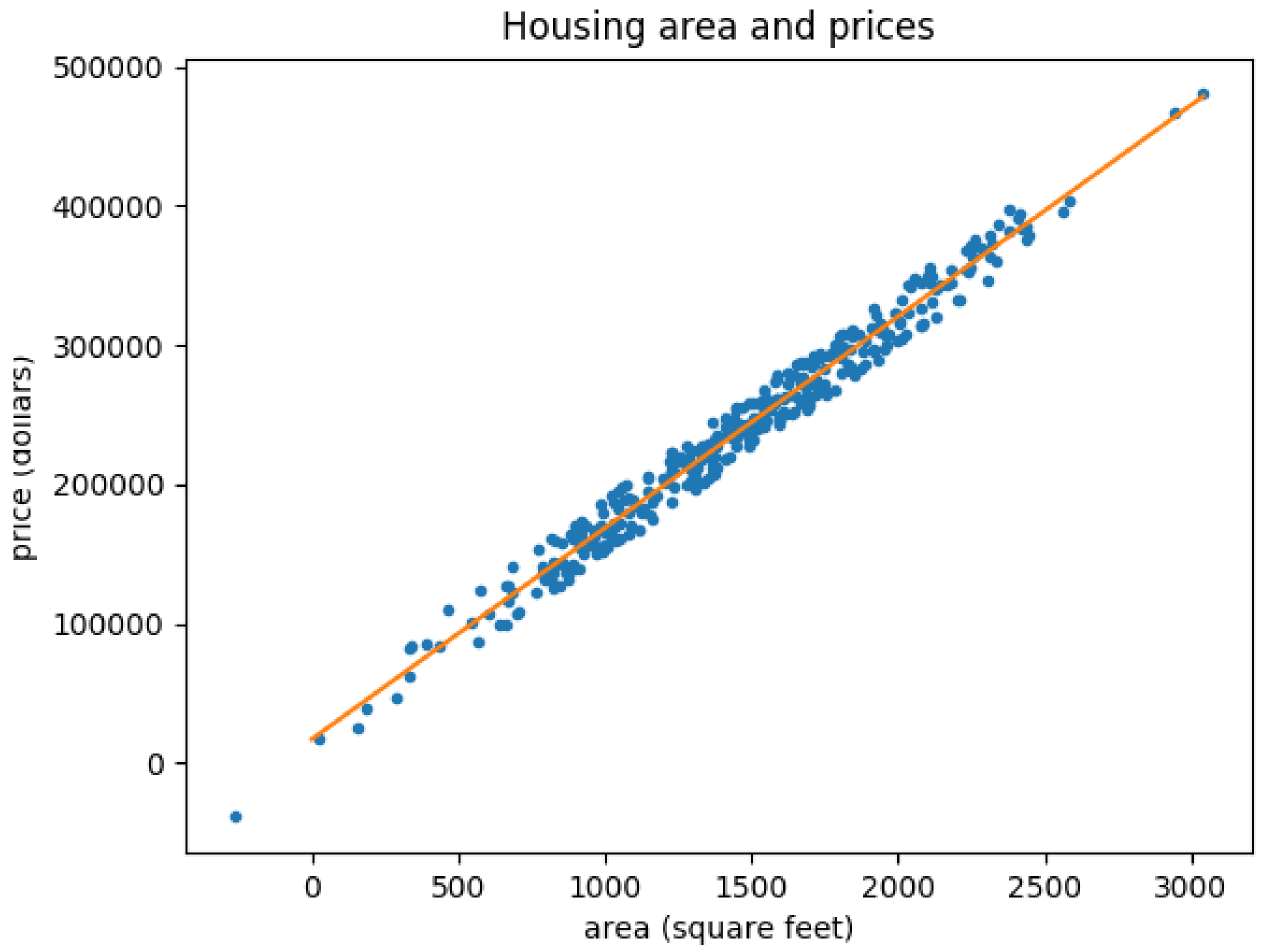
9. The linear regression model
The linregress() function from scipy.stats is used to perform linear regression on the two columns. It returns the slope and intercept of the best fit line. It also returns three other quantities related to linear regression, but since we don't need them, we simply use the dummy variable underscore three times.

Return values difficult to compute manually



10. Return values difficult to compute manually
The train_model() function is different from the functions that we have tested so far.
11. Return values difficult to compute manually
Since it performs a complicated linear regression procedure,
12. Return values difficult to compute manually
we cannot easily predict the best fit line.
And if we don't know the expected return value, we cannot test the function!

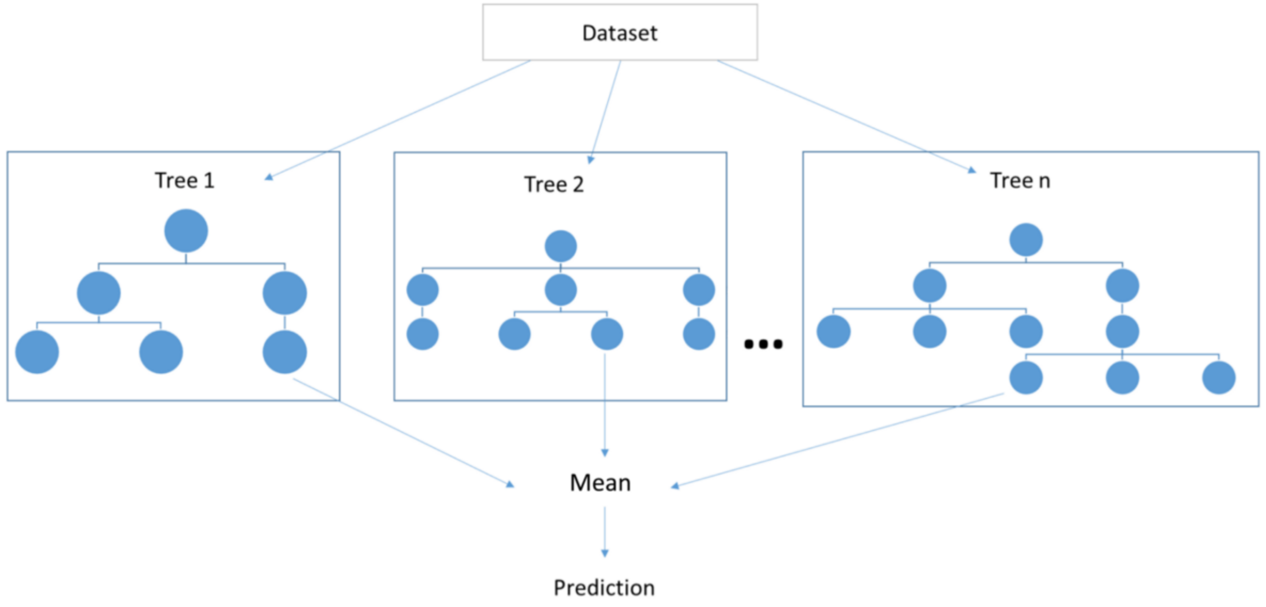
Return values difficult to compute manually

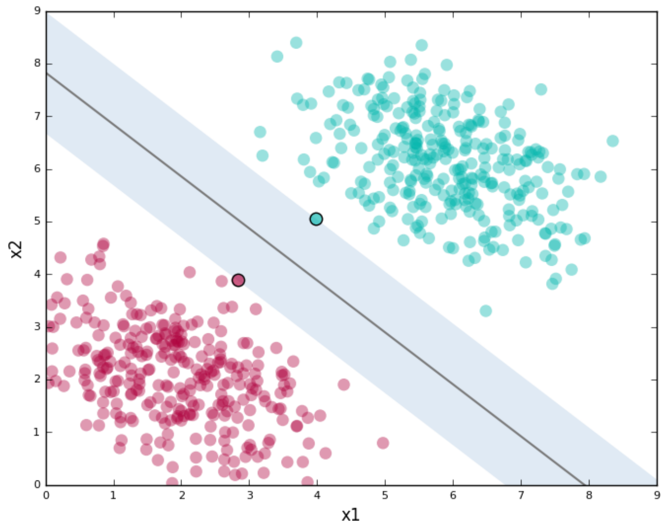


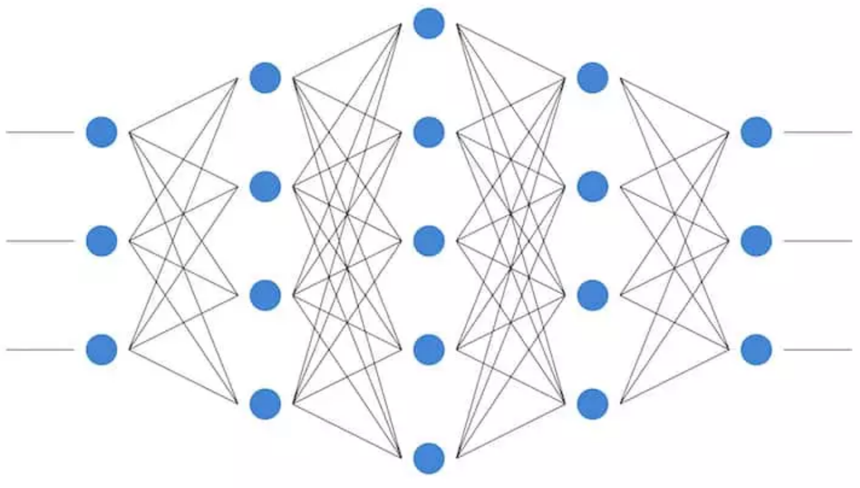
Return values difficult to compute manually

- Cannot test `train_model()` without knowing expected return values.

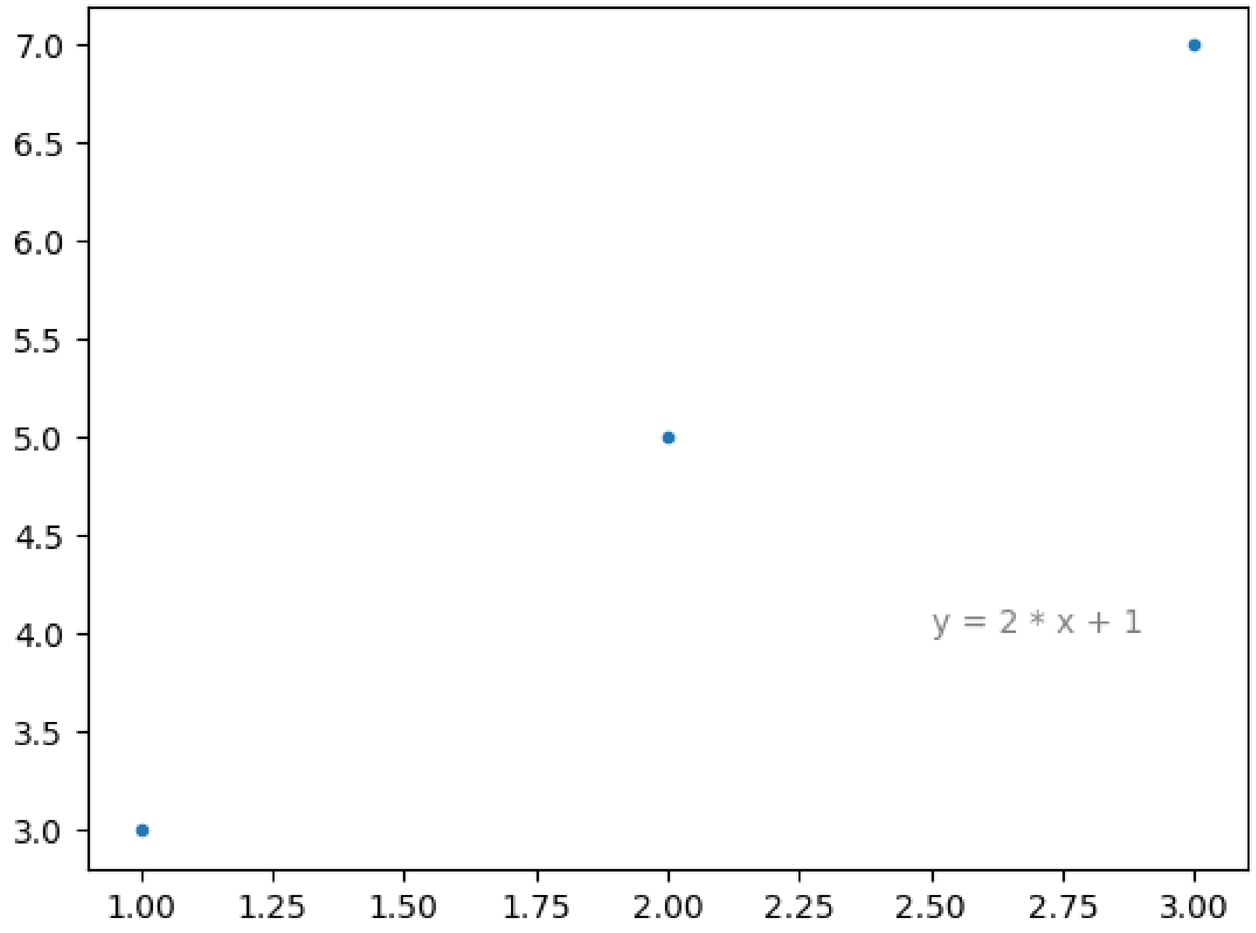
True for all data science models





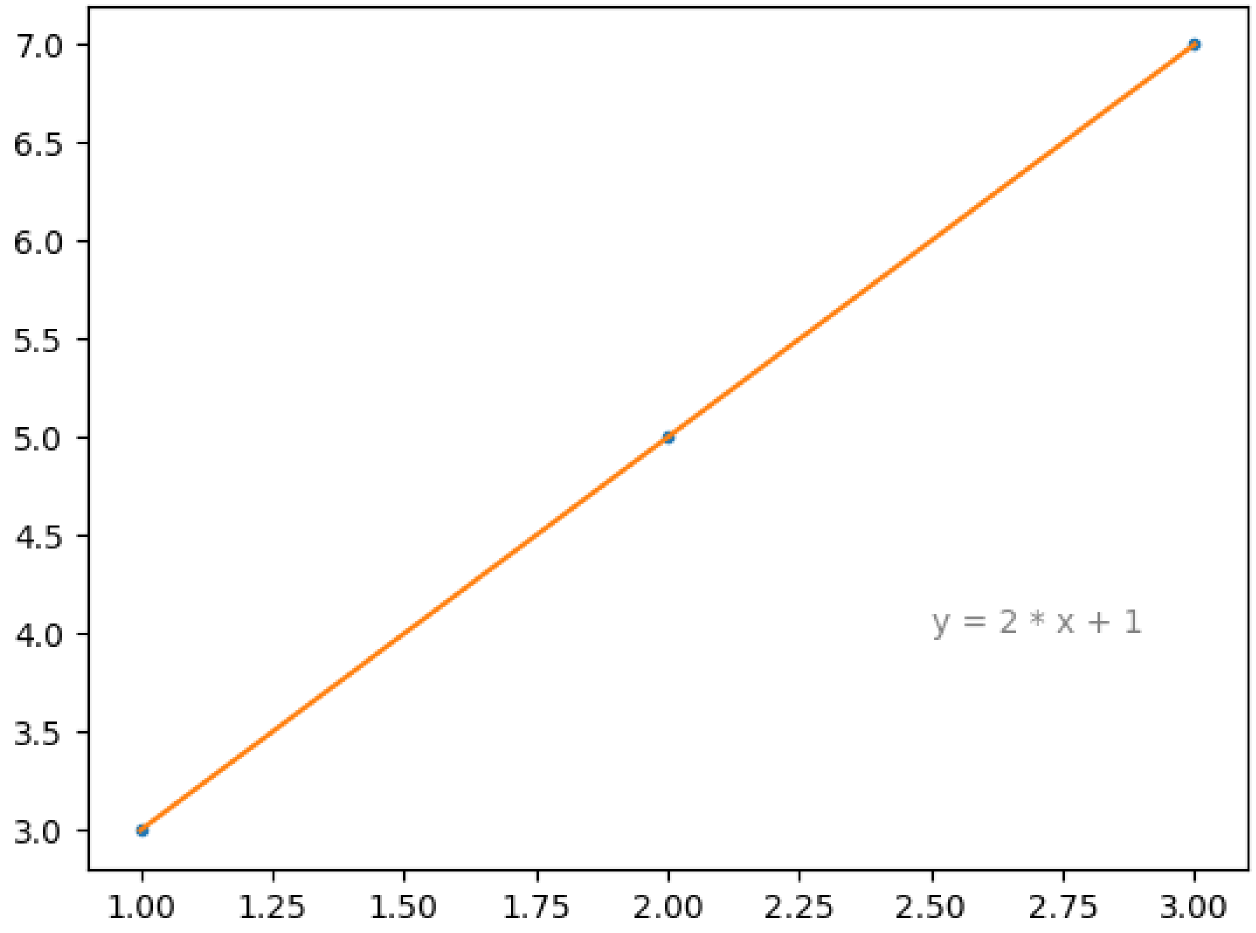


Trick 1: Use dataset where return value is known




```
import pytest
import numpy as np
from models.train import train_model

def test_on_linear_data():
    test_argument = np.array([[1.0, 3.0],
                              [2.0, 5.0],
                              [3.0, 7.0]
                              ]
                             )
```

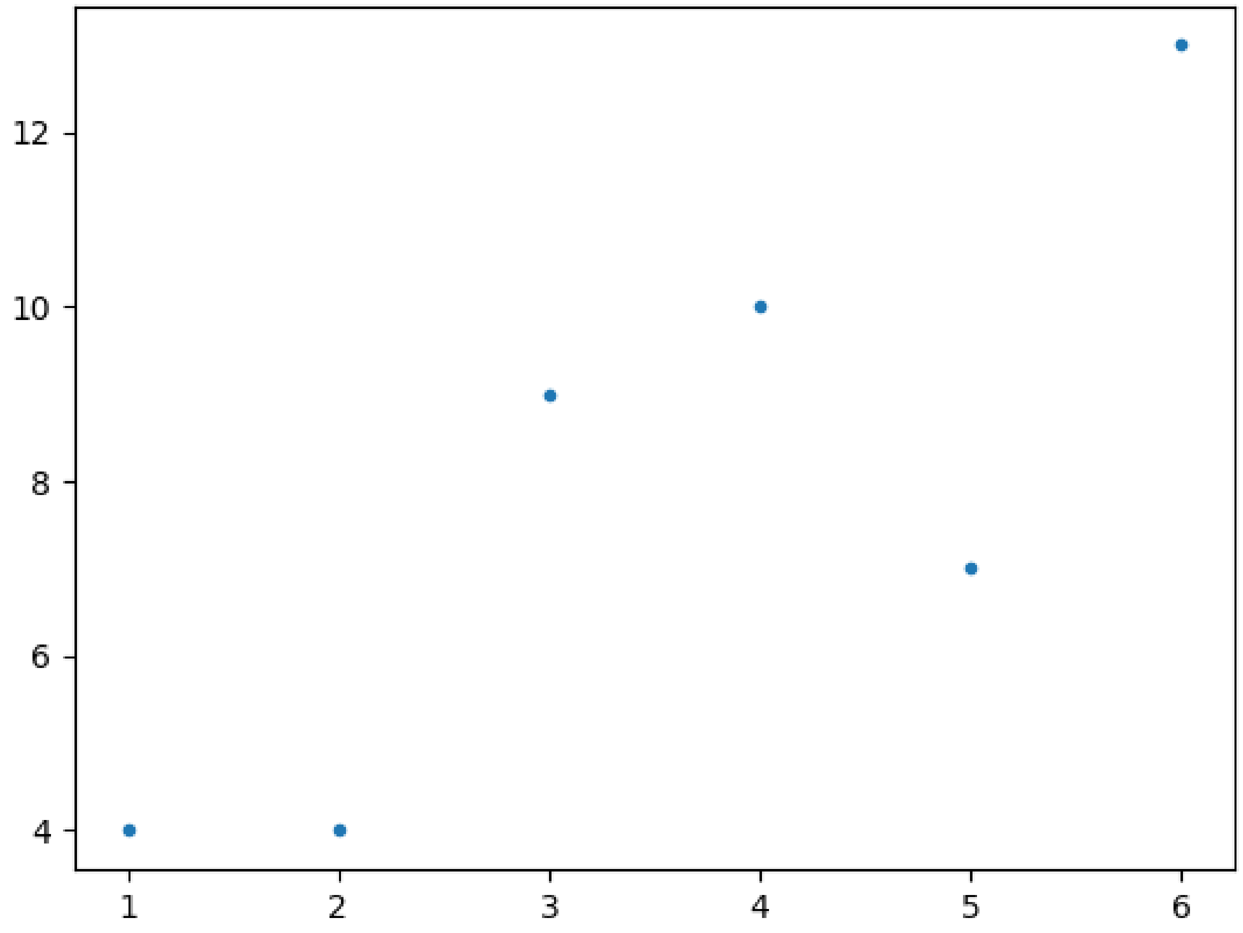
Trick 1: Use dataset where return value is known




```python
import pytest
import numpy as np
from models.train import train_model

def test_on_linear_data():
    test_argument = np.array([[1.0, 3.0],
                              [2.0, 5.0],
                              [3.0, 7.0]
                              ]
                             )
    expected_slope = 2.0
    expected_intercept = 1.0
    slope, intercept = train_model(test_argument)
    assert slope == pytest.approx(expected_slope)
    assert intercept == pytest.approx(
        expected_intercept
    )
```

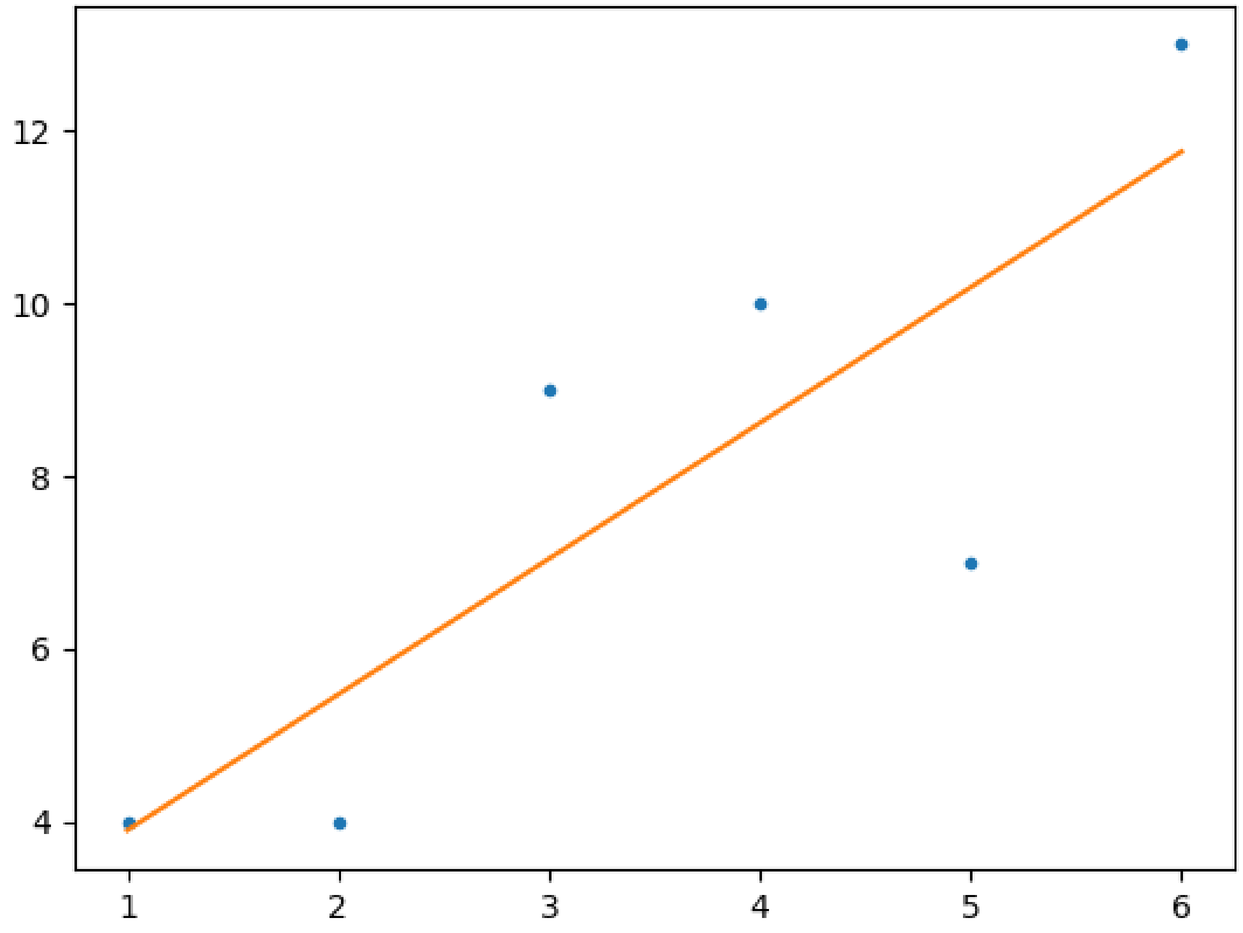
14. Trick 1: Use dataset where return value is known
The trick is to use an artificial or well-known training set, where it is easy to manually compute the return value. In the case of linear regression, one such training dataset is a linear data set. In the test test_on_linear_data(), we use such a dataset which follows the equation price equals two times area plus one.

Trick 2: Use inequalities




```python
import numpy as np
from models.train import train_model

def test_on_positively_correlated_data():
    test_argument = np.array([[1.0, 4.0], [2.0, 4.0],
                              [3.0, 9.0], [4.0, 10.0],
                              [5.0, 7.0], [6.0, 13.0],
                              ]
                             )
```

Trick 2: Use inequalities




```python
import numpy as np
from models.train import train_model

def test_on_positively_correlated_data():
    test_argument = np.array([[1.0, 4.0], [2.0, 4.0],
                              [3.0, 9.0], [4.0, 10.0],
                              [5.0, 7.0], [6.0, 13.0],
                              ]
                             )
    slope, intercept = train_model(test_argument)
    assert slope > 0
```

Recommendations

- Do not leave models untested just because they are complex.
- Perform as many sanity checks as possible.

18. Recommendations
We shouldn't leave our model untested just because it is complex. Perform all sorts of sanity checks. This will save us lots of debugging effort in the long run.

Using the model

```python
from data.preprocessing_helpers import preprocess
from features.as_numpy import get_data_as_numpy_array
from models.train import (
  split_into_training_and_testing_sets, train_model
  )

preprocess("data/raw/housing_data.txt",
           "data/clean/clean_housing_data.txt"
           )
data = get_data_as_numpy_array(
    "data/clean/clean_housing_data.txt", 2
    )
training_set, testing_set = (
    split_into_training_and_testing_sets(data)
    )
slope, intercept = train_model(training_set)
```

```python
train_model(training_set)
```

```
151.78430060614986 17140.77537937442
```

19. Using the model
Once the training function has been tested, we use it to find the best fit line for the housing data.

20. Testing model performance
The next step is to test the model using the model_test() function. It takes the testing set as the first argument. It also takes the slope and intercept returned by the model, and checks the performance of the model on the testing set. It returns a quantity called the r squared, which expresses how well the model fits the testing set. The value of r squared usually ranges from 0 to 1. It is 1 when the fit is perfect, it is 0 if there's no fit. It is hard to compute r squared in the general case. Therefore, we will have to use the recommendations of this lesson to test this function

Testing model performance

```python
def model_test(testing_set, slope, intercept):
    """Return r^2 of fit"""
```

- Returns a quantity r^2.
- Indicates how well the model performs on unseen data.
- Usually, $0 \leq r^2 \leq 1$.
- $r^2 = 1$ indicates perfect fit.
- $r^2 = 0$ indicates no fit.
- Complicated to compute r^2 manually.

Let's practice writing sanity tests!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Testing plots

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

Pizza without cheese!

This lesson: testing matplotlib visualizations

matplotlib

Version 3.1.0

The plotting function

```
data/
src/
|-- data/
|-- features/
|-- models/
|-- visualization
|   |-- __init__.py
tests/
```

The plotting function

- `plots.py`

```
def get_plot_for_best_fit_line(slope,
                               intercept,
                               x_array,
                               y_array,
                               title
                               ):
    """
    slope: slope of best fit line
    intercept: intercept of best fit line
    """
```

```
data/
src/
|-- data/
|-- features/
|-- models/
|-- visualization
|   |-- __init__.py
|   |-- plots.py
tests/
```

5. The plotting function
The package has a Python module called plots.py. The module contains a function called get_plot_for_best_fit_line(), which we are going to test. It takes the slope and the intercept of the best fit line as arguments.
6. The plotting function
Other arguments include x_array and y_array, which hold the housing area and prices data respectively, either from the training set or the testing set.

The plotting function

- `plots.py`

```
def get_plot_for_best_fit_line(slope,
                               intercept,
                               x_array,
                               y_array,
                               title
                               ):
    """
    slope: slope of best fit line
    intercept: intercept of best fit line
    x_array: array containing housing areas
    y_array: array containing housing prices
    """
```

```
data/
src/
|-- data/
|-- features/
|-- models/
|-- visualization
|   |-- __init__.py
|   |-- plots.py
tests/
```

The plotting function

- `plots.py`

```python
def get_plot_for_best_fit_line(slope,
                               intercept,
                               x_array,
                               y_array,
                               title
                               ):
    """
    slope: slope of best fit line
    intercept: intercept of best fit line
    x_array: array containing housing areas
    y_array: array containing housing prices
    title: title of the plot
    """
```

```
data/
src/
|-- data/
|-- features/
|-- models/
|-- visualization
|   |-- __init__.py
|   |-- plots.py
tests/
```

The plotting function

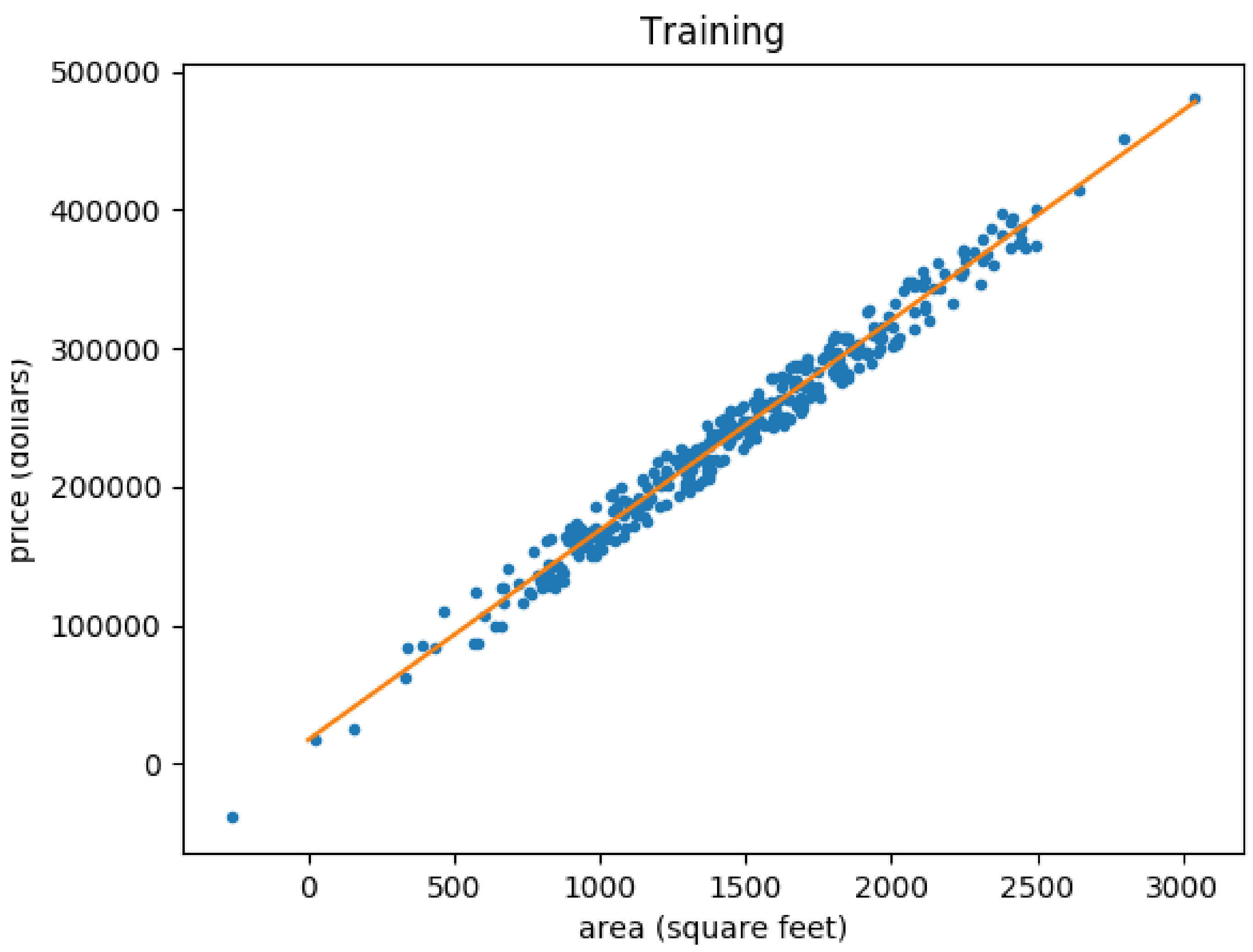
- `plots.py`

```
def get_plot_for_best_fit_line(slope,
                               intercept,
                               x_array,
                               y_array,
                               title
                               ):
    """
    slope: slope of best fit line
    intercept: intercept of best fit line
    x_array: array containing housing areas
    y_array: array containing housing prices
    title: title of the plot

    Returns: matplotlib.figure.Figure()
    """
```

```
data/
src/
|-- data/
|-- features/
|-- models/
|-- visualization
|   |-- __init__.py
|   |-- plots.py
tests/
```

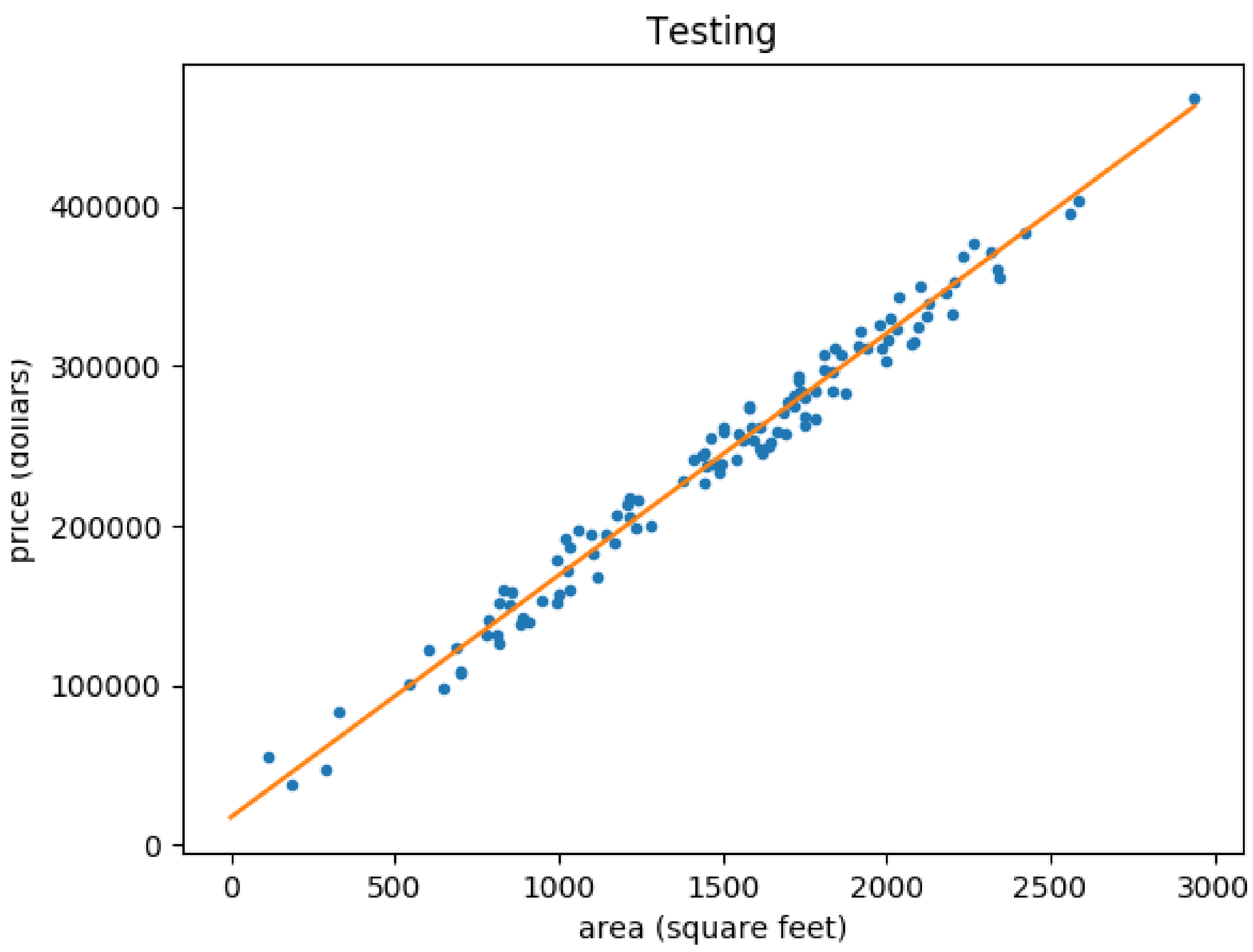
Training plot

```python
...
from visualization import get_plot_for_best_fit_line

preprocess(...)
data = get_data_as_numpy_array(...)
training_set, testing_set = (
    split_into_training_and_testing_sets(data)
    )
slope, intercept = train_model(training_set)
get_plot_for_best_fit_line(slope, intercept,
    training_set[:, 0], training_set[:, 1],
    "Training"
    )
```



Testing plot

```python
...
from visualization import get_plot_for_best_fit_line

preprocess(...)
data = get_data_as_numpy_array(...)
training_set, testing_set = (
    split_into_training_and_testing_sets(data)
    )
slope, intercept = train_model(training_set)
get_plot_for_best_fit_line(slope, intercept,
    training_set[:, 0], training_set[:, 1],
    "Training"
    )
get_plot_for_best_fit_line(slope, intercept,
    testing_set[:, 0], testing_set[:, 1], "Testing"
    )
```

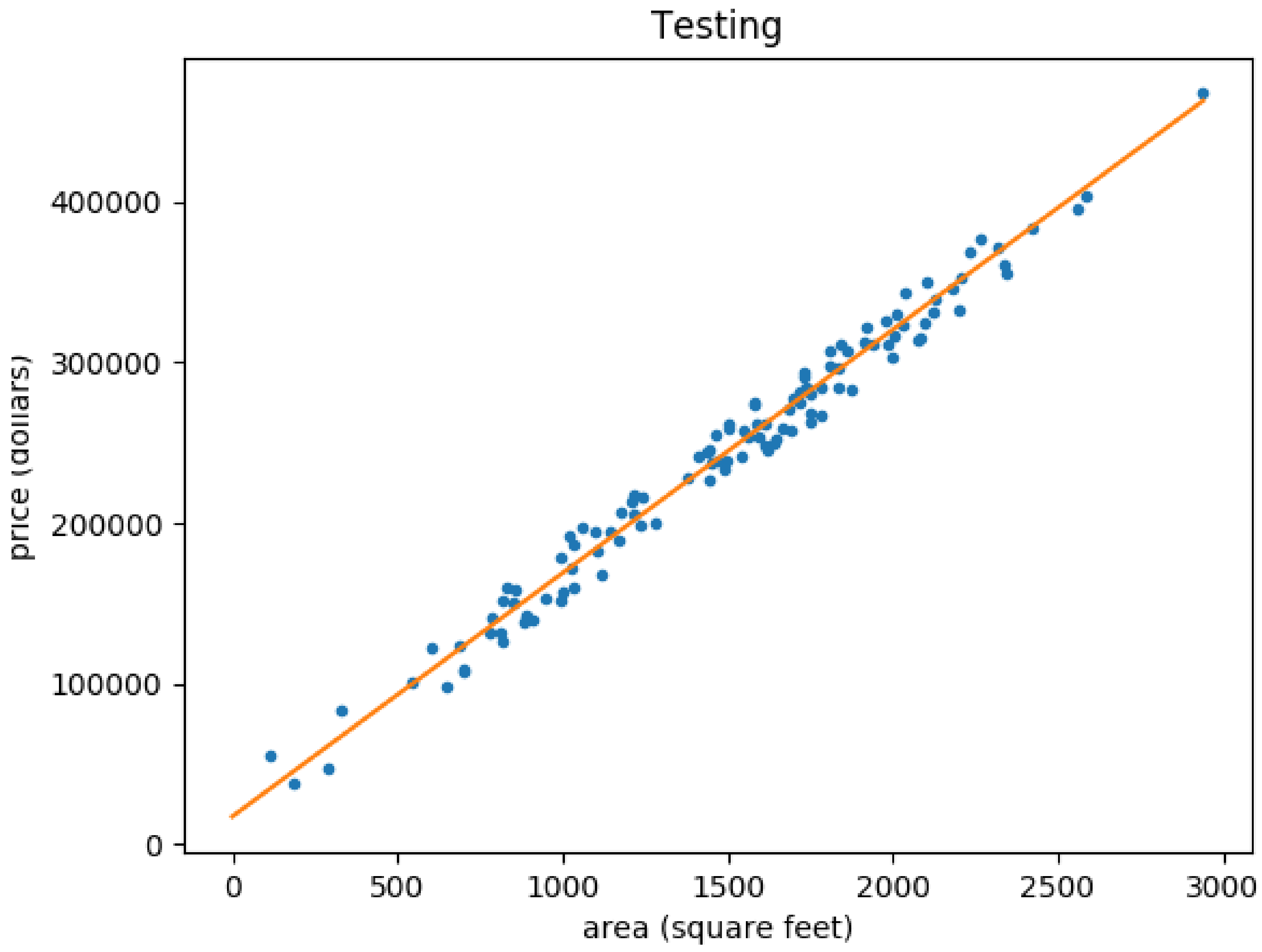


Don't test properties individually

`matplotlib.figure.Figure()`

- Axes
  - configuration
  - style
- Data
  - style
- Annotations
  - style
- ...

11. Don't test properties individually
The return value of the plotting function is a matplotlib.figure.Figure() object. This object has tons of properties, for example, the axes and its configuration and style, the plotted data and its style, annotations and its style etc. Due to the sheer number of properties, it is not advisable to test each of them individually.



Testing strategy for plots

Testing strategy for plots

One-time baseline generation

Testing

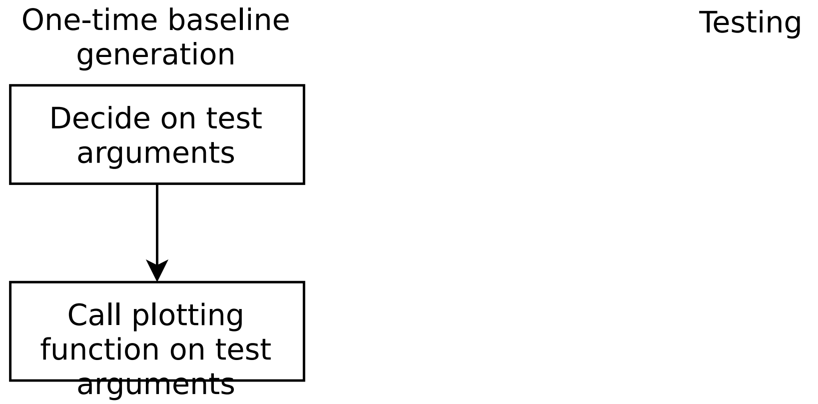
One-time baseline generation

One-time baseline generation

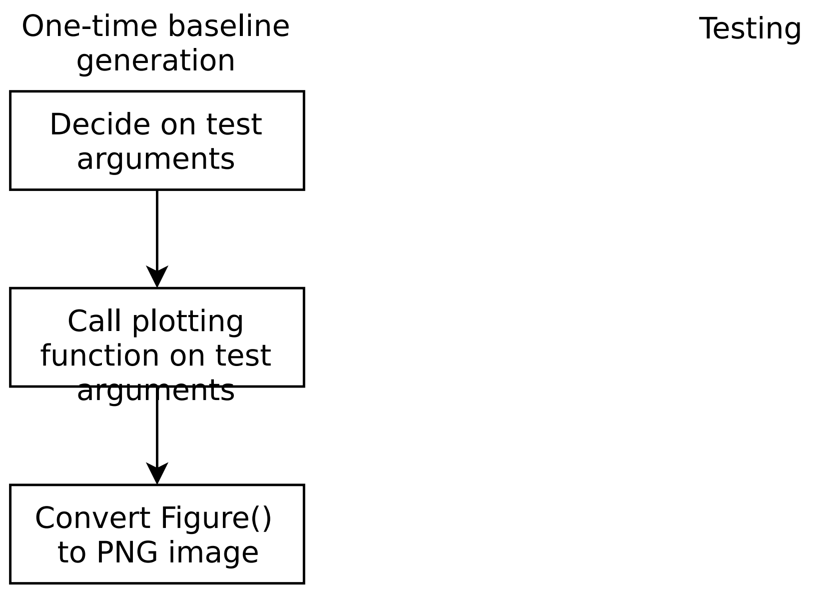
Testing

Decide on test arguments

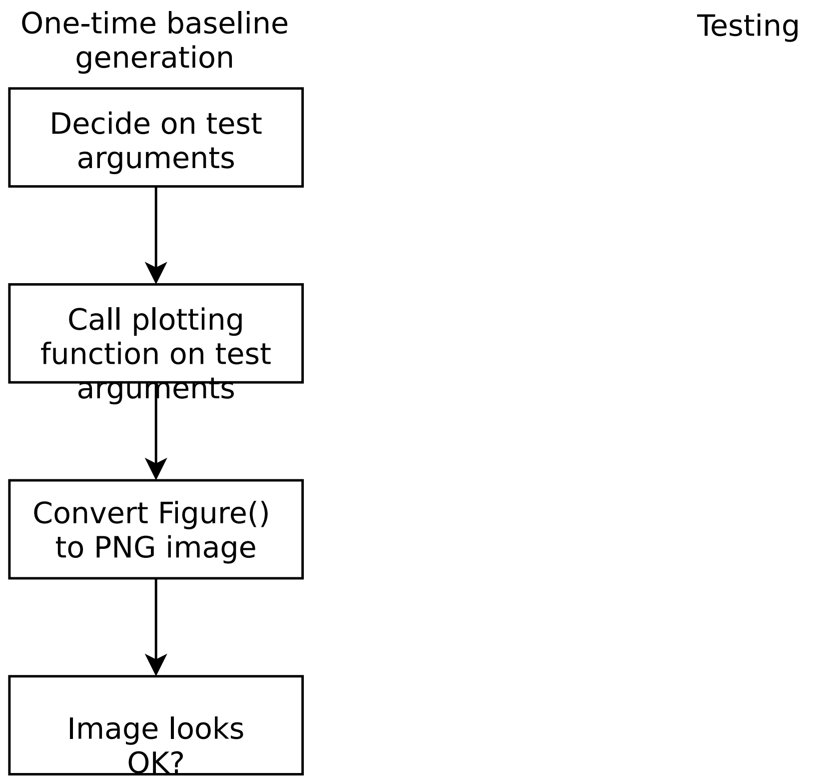
One-time baseline generation



One-time baseline generation

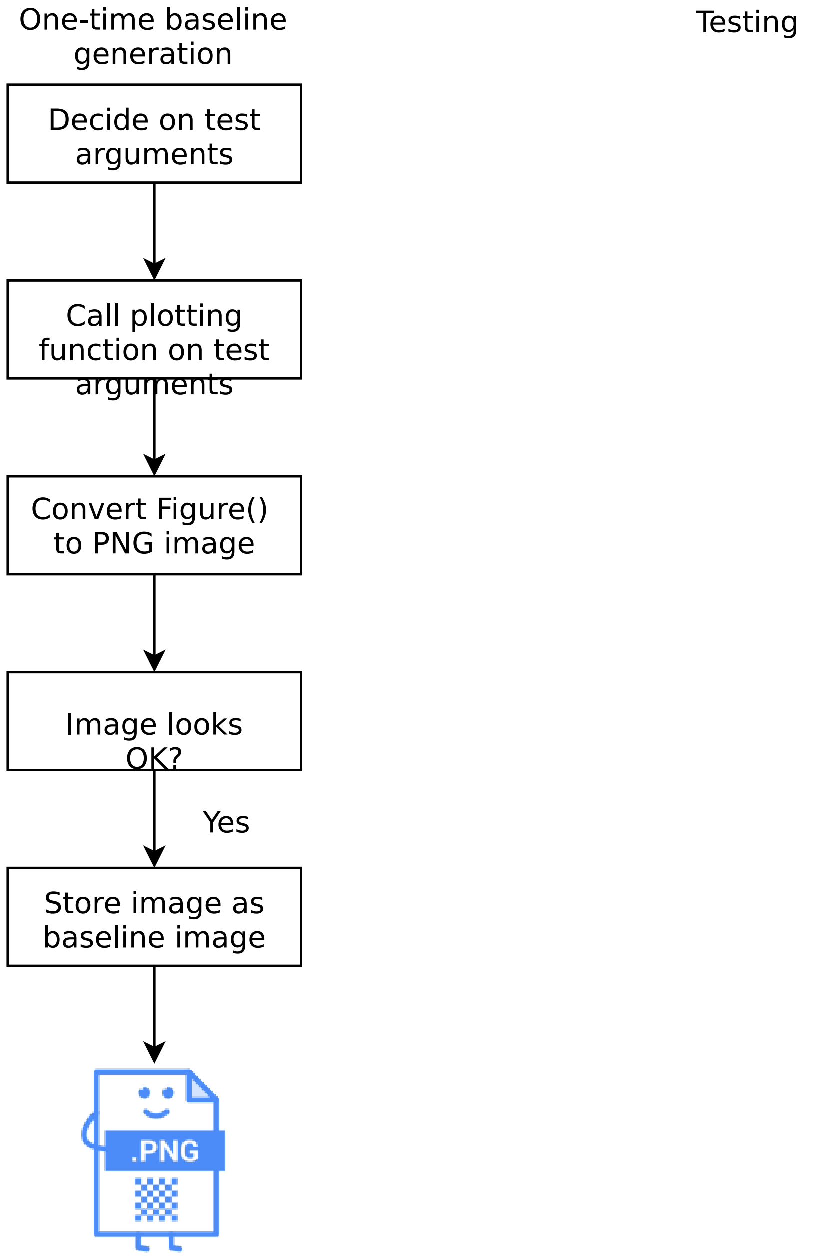


One-time baseline generation

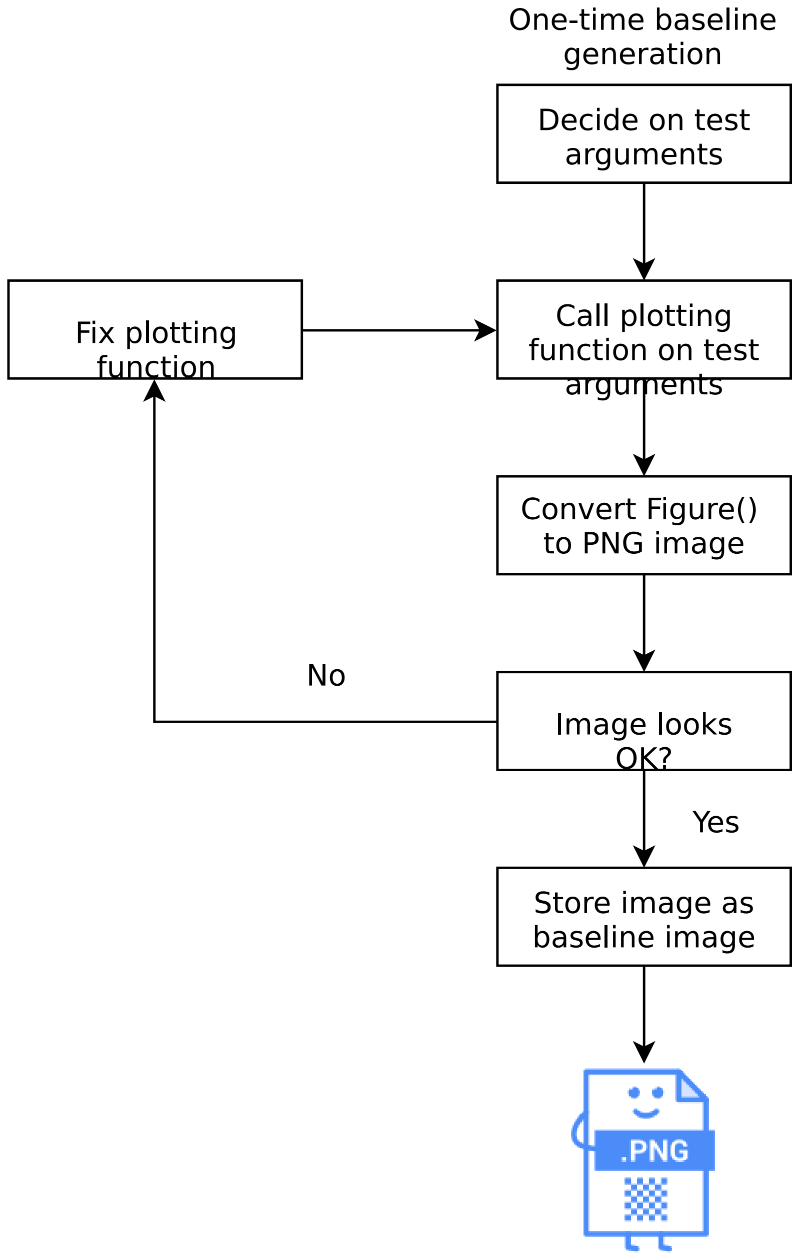


One-time baseline generation

12. Testing strategy for plots
Instead, we will use a shortcut using the human eye.
13. Testing strategy for plots
The idea involves two steps - a one-time baseline generation and testing.
14. One-time baseline generation
To generate the one-time baseline, we decide on a set of test arguments for the plotting function.
15. One-time baseline generation
Then we call the plotting function with these test arguments
16. One-time baseline generation
and convert the returned matplotlib.figure.Figure() object into a PNG image.
17. One-time baseline generation
We inspect this image manually
18. One-time baseline generation
and verify that it looks as expected. If it does, we store this image as a baseline image.
19. One-time baseline generation
If it doesn't, we modify the function until it does.



One-time baseline generation

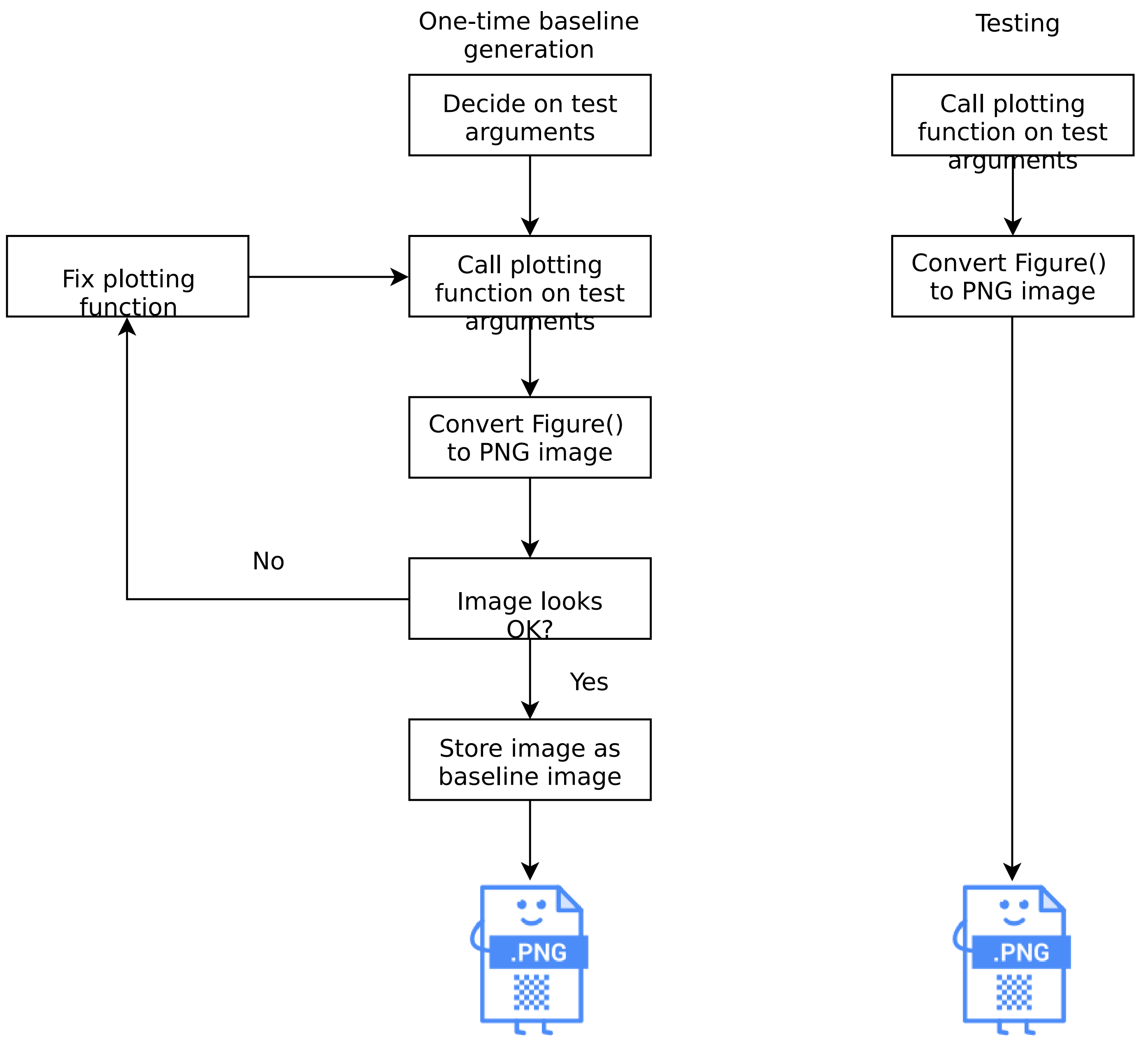


Testing

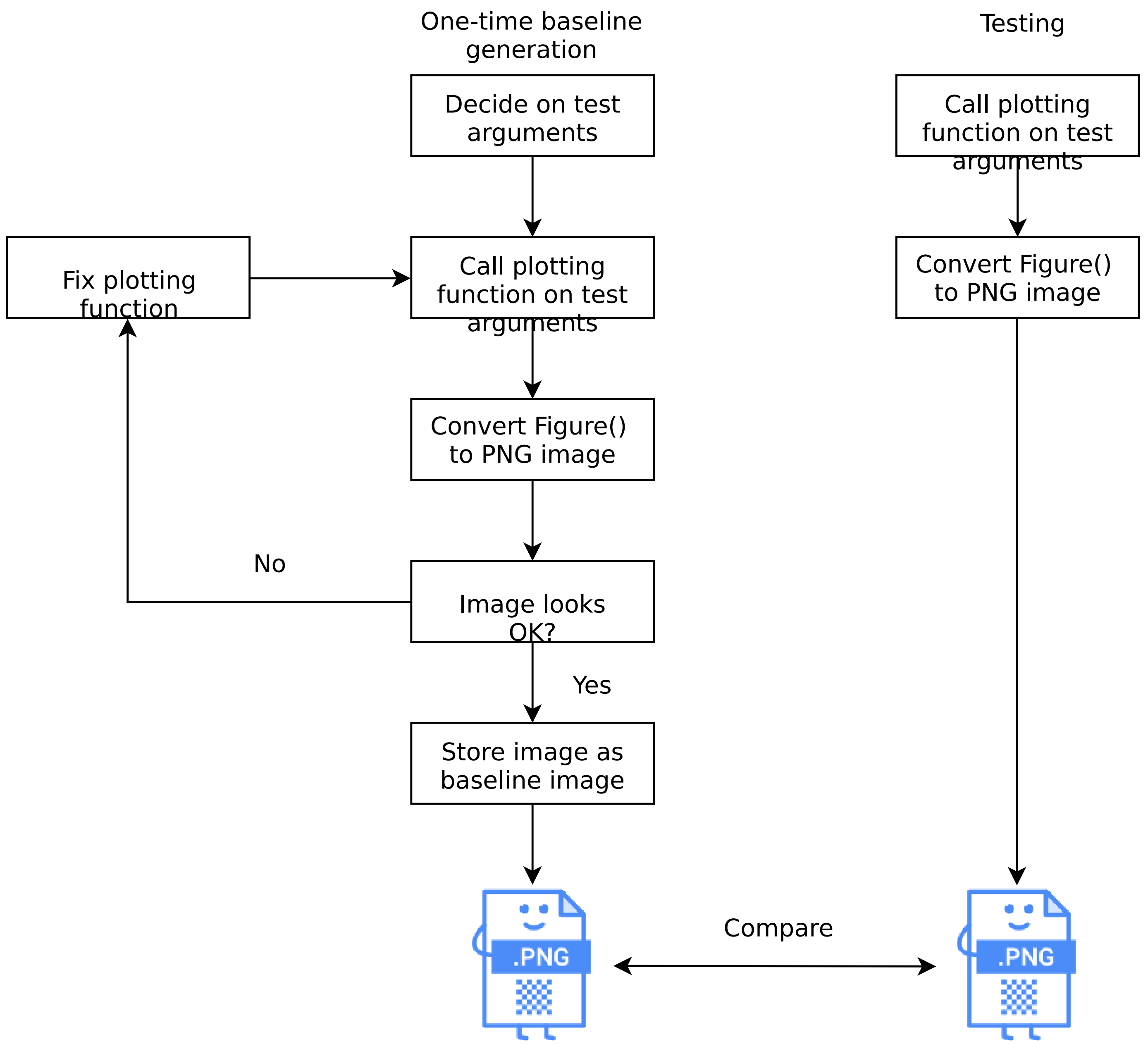
20. Testing
The testing step involves generating a PNG image for the test arguments that we decided upon earlier

21. Testing
and comparing the image with the stored baseline image.

Testing


One-time baseline generation
Decide on test arguments
Fix plotting function
Call plotting function on test arguments
Convert Figure() to PNG image
No
Image looks OK?
Yes
Store image as baseline image
.PNG
Testing
Call plotting function on test arguments
Convert Figure() to PNG image
.PNG

Testing


One-time baseline generation
Testing
Decide on test arguments
Call plotting function on test arguments
Fix plotting function
Call plotting function on test arguments
Convert Figure() to PNG image
Convert Figure() to PNG image
No
Image looks OK?
Yes
Store image as baseline image
.PNG
Compare
.PNG

pytest-mpl

- Knows how to ignore OS related differences.
- Makes it easy to generate baseline images.

```
pip install pytest-mpl
```

22. pytest-mpl
Since images generated on different operating systems look slightly different, we need to use a pytest plugin called pytest-mpl for image comparisons. This library knows how to ignore the operating system related differences and makes it easy to generate baseline images. We install it using pip.

An example test

```python
import pytest
import numpy as np
from visualization import get_plot_for_best_fit_line


def test_plot_for_linear_data():
    slope = 2.0
    intercept = 1.0
    x_array = np.array([1.0, 2.0, 3.0])    # Linear data set
    y_array = np.array([3.0, 5.0, 7.0])
    title = "Test plot for linear data"
    return get_plot_for_best_fit_line(slope, intercept, x_array, y_array, title)
```

23. An example test
To illustrate how this package works, we will write a test called test_plot_for_linear_data(). For this test, we have decided on a simple linear data set. Instead of an assert statement, the test returns the matplotlib figure returned by the function under test.

An example test

23. An example test
To illustrate how this package works, we will write a test called test_plot_for_linear_data(). For this test, we have decided on a simple linear data set. Instead of an assert statement, the test returns the matplotlib figure returned by the function under test.

```
import pytest
import numpy as np
from visualization import get_plot_for_best_fit_line

@pytest.mark.mpl_image_compare    # Under the hood baseline generation and comparison
def test_plot_for_linear_data():
    slope = 2.0
    intercept = 1.0
    x_array = np.array([1.0, 2.0, 3.0])    # Linear data set
    y_array = np.array([3.0, 5.0, 7.0])
    title = "Test plot for linear data"
    return get_plot_for_best_fit_line(slope, intercept, x_array, y_array, title)
```

Generating the baseline image

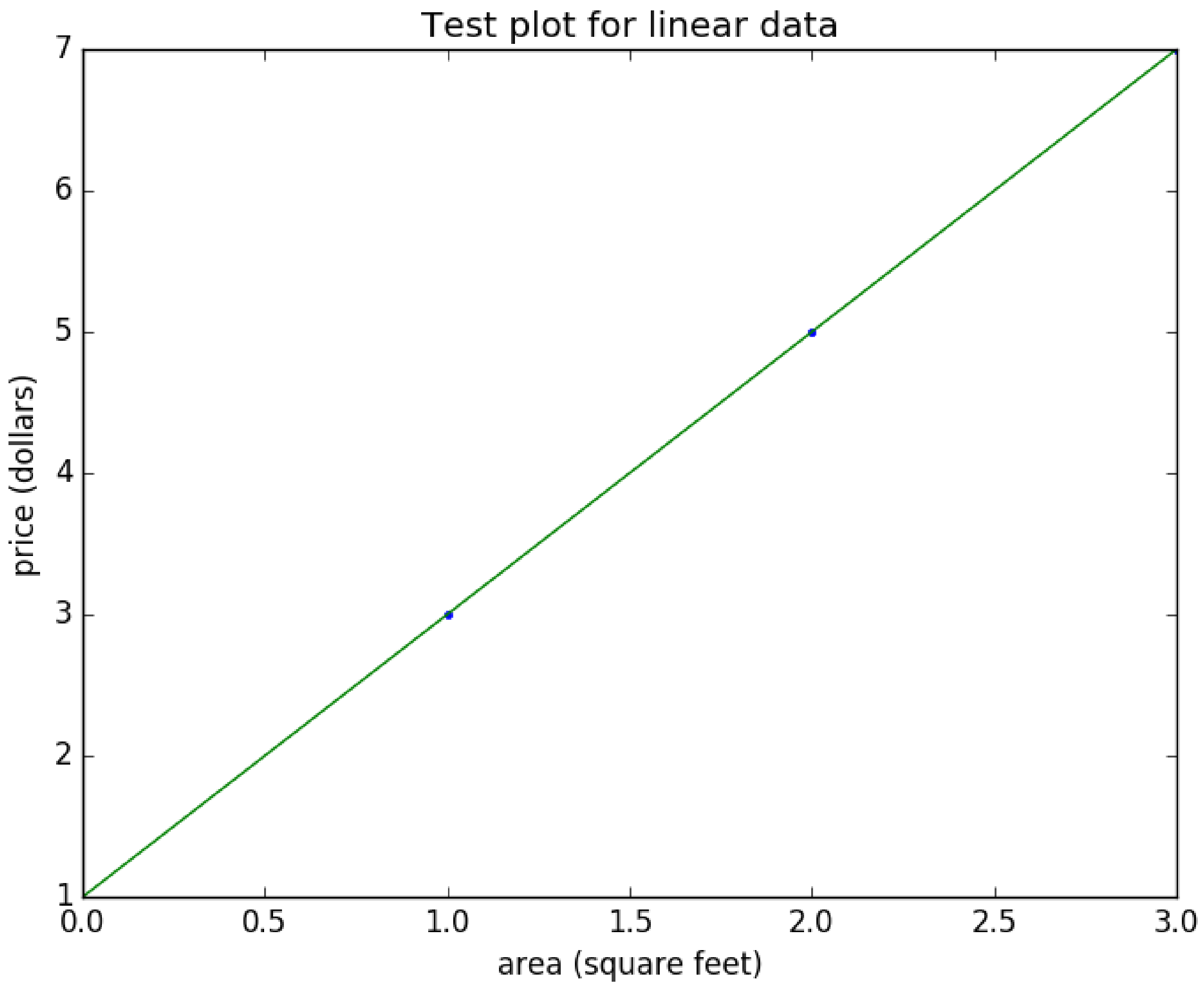
Generate baseline image

```
!pytest -k "test_plot_for_linear_data"
        --mpl-generate-path
        visualization/baseline
```

```
data/
src/
tests/
|-- data/
|-- features/
|-- models/
|-- visualization
    |-- __init__.py
    |-- test_plots.py    # Test module
    |-- baseline         # Contains baselines
```

25. Generating the baseline image
pytest expects baseline images to be stored in a folder called baseline relative to the test module test_plots.py. To generate the baseline image, we run the test with the command line argument --mpl-generate-path and enter the path to the baseline folder as argument. This will create the baseline image.

Verify the baseline image




```
data/
src/
tests/
|-- data/
|-- features/
|-- models/
|-- visualization
    |-- __init__.py
    |-- test_plots.py    # Test module
    |-- baseline         # Contains baselines
        |-- test_plot_for_linear_data.png
```

Run the test

```
!pytest -k "test_plot_for_linear_data" --mpl
```

```
====================== test session starts ======================
...
collected 24 items / 23 deselected / 1 selected

visualization/test_plots.py .                               [100%]

============= 1 passed, 23 deselected in 0.68 seconds =============
```

26. Verify the baseline image
Then we open the baseline image and confirm that it looks as expected.
27. Run the test
The next time we run the test, we must use the --mpl option with the pytest command. This will make pytest compare the baseline image with the actual one. If they are identical, then the test will pass.

Reading failure reports

```
!pytest -k "test_plot_for_linear_data" --mpl
```

```
============================= FAILURES ==============================
_______ TestGetPlotForBestFitLine.test_plot_for_linear_data _______
Error: Image files did not match.
  RMS Value: 11.191347848524174
  Expected:
    /tmp/tmplcbtsb10/baseline-test_plot_for_linear_data.png
  Actual:
    /tmp/tmplcbtsb10/test_plot_for_linear_data.png
  Difference:
    /tmp/tmplcbtsb10/test_plot_for_linear_data-failed-diff.png
  Tolerance:
    2
============= 1 failed, 36 deselected in 1.13 seconds =============
```

28. Reading failure reports
If they are not identical, the test will fail and pytest will save the baseline image, the actual image and an image containing the pixelwise difference to a temporary directory. The paths to these images will be printed in the failures section of the test result report, as we see here. Looking at these images helps in debugging the function.

Yummy!

Let's test plots!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Congratulations

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

You've written so many tests

1

You've written so many tests

5

You've written so many tests

10

You've written so many tests

25

You learned a lot

- Testing saves time and effort.
- `pytest`
  - Testing return values and exceptions.
  - Running tests and reading the test result report.
- Best practices
  - Well tested function using normal, special and bad arguments.
  - TDD, where tests get written before implementation.
  - Test organization and management.
- Advanced skills
  - Setup and teardown with fixtures, mocking.
  - Sanity tests for data science models.
  - Plot testing.

Code for this course

https://github.com/gutfeeling/univariate-linear-regression

9. Code for this course
Since this was a course on testing, you didn't always get to see how the functions under test were implemented. If you are interested about that, the entire code for this course is available in this GitHub repository.

Icon sources

Icons made by the following authors from **flaticon.com**.

- Freepik
- Smashicons
- Vectors Market
- Kiranshastry
- Dimitry Miroliubov
- Creaticca Creative Agency
- Gregor Cresnar

Image sources

1. https://chibird.com/post/20998191414/i-make-a-lot-of-procrastination-drawings-theyre
2. http://www.dekoleidenschaft.de/ratgeber/10-tipps-fuer-mehr-ordnung-im-kleiderschrank/
3. http://me-monaco.me/paper-storage-box-with-lid/
4. https://towardsdatascience.com/random-forests-and-decision-trees-from-scratch-in-python-3e4fa5ae4249
5. https://towardsdatascience.com/demystifying-support-vector-machines-8453b39f7368
6. https://www.bbc.co.uk/bbcthree/article/b290ff0e-1d75-43b1-8ff1-a9ac80d4d842

I wish you all the best!