Beyond assertion: setup and teardown

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

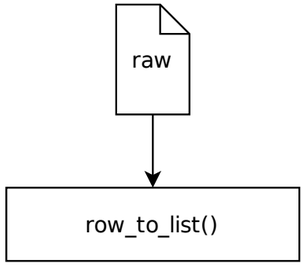
The preprocessing function

```
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
```

The preprocessing function

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

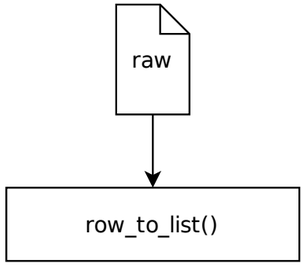



```
1,801    201,411
1,767565,112         # dirty row, no tab
2,002    333,209
1990    782,911
1,285    389129
```

The preprocessing function

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

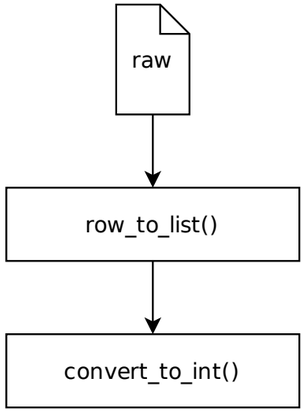
```
1,801    201,411
2,002    333,209
1990    782,911
1,285    389129
```



The preprocessing function

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

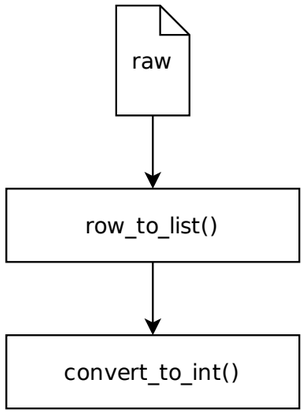
```
1,801    201,411
2,002    333,209
1990    782,911      # dirty row, no comma
1,285    389129      # dirty row, no comma
```



The preprocessing function

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

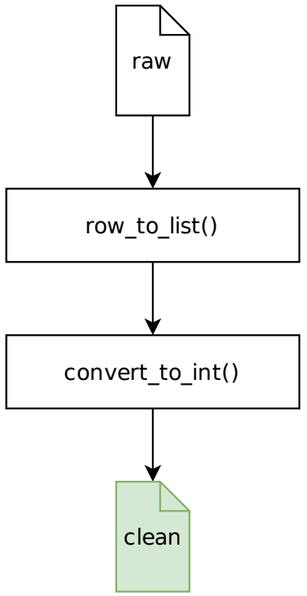
```
1,801    201,411
2,002    333,209
```



The preprocessing function

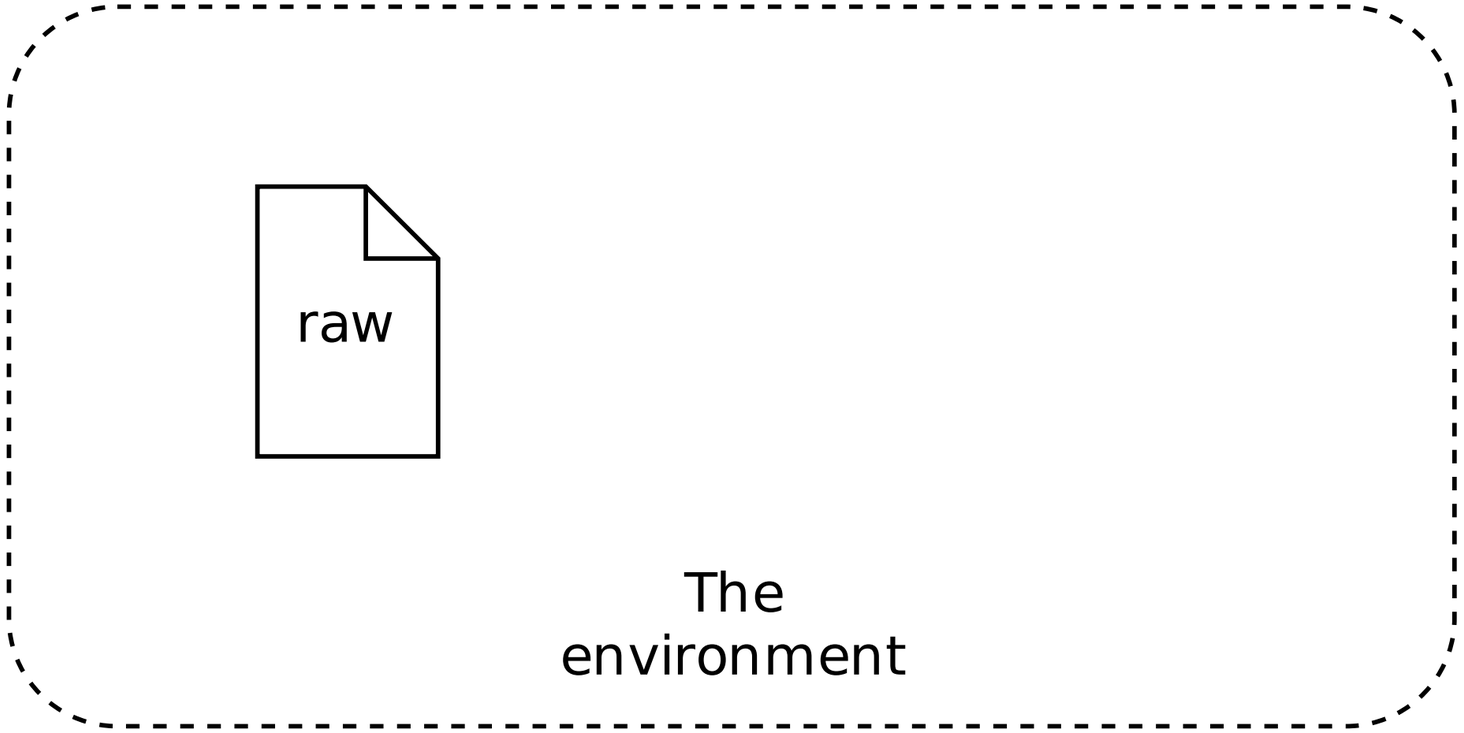
```
def preprocess(raw_data_file_path,
               clean_data_file_path
               ):
    ...
```

```
1801    201411
2002    333209
```



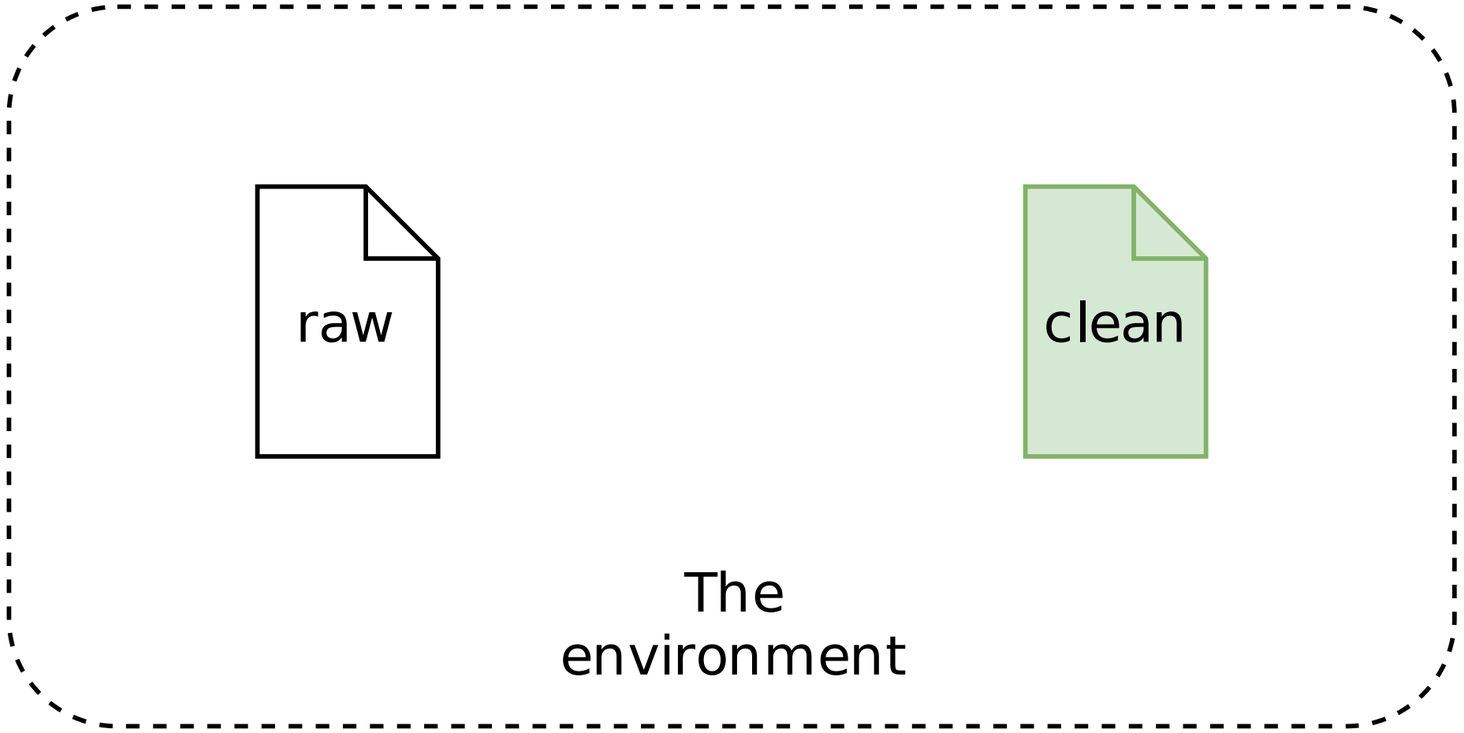
Environment preconditions

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



Environment modification

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



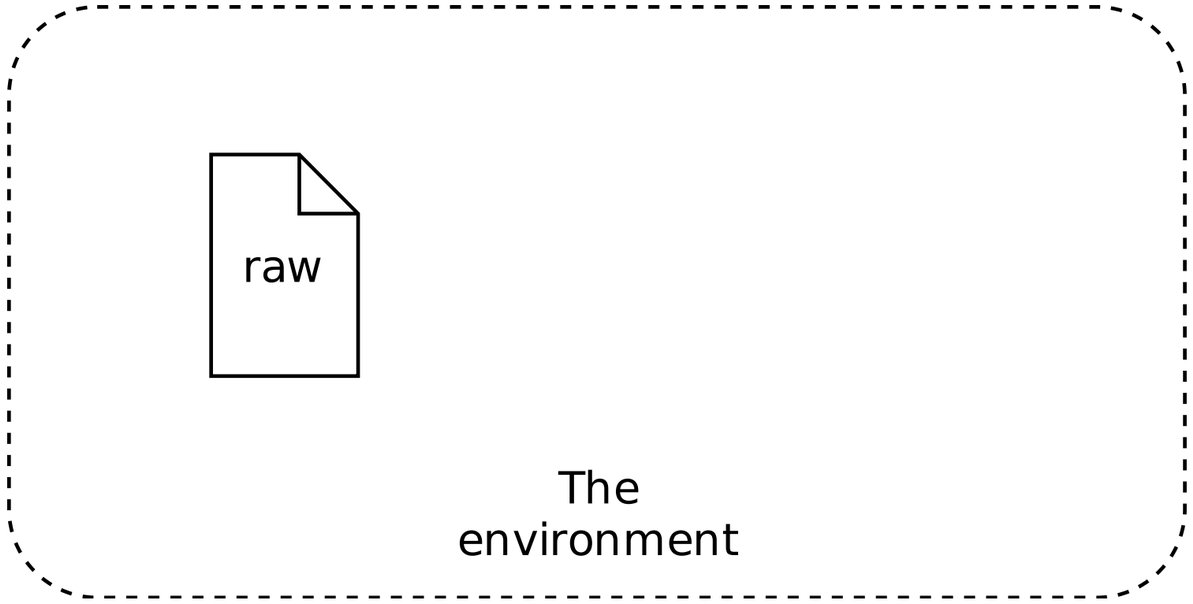
Testing the preprocessing function

```
def test_on_raw_data():
```

Step 1: Setup

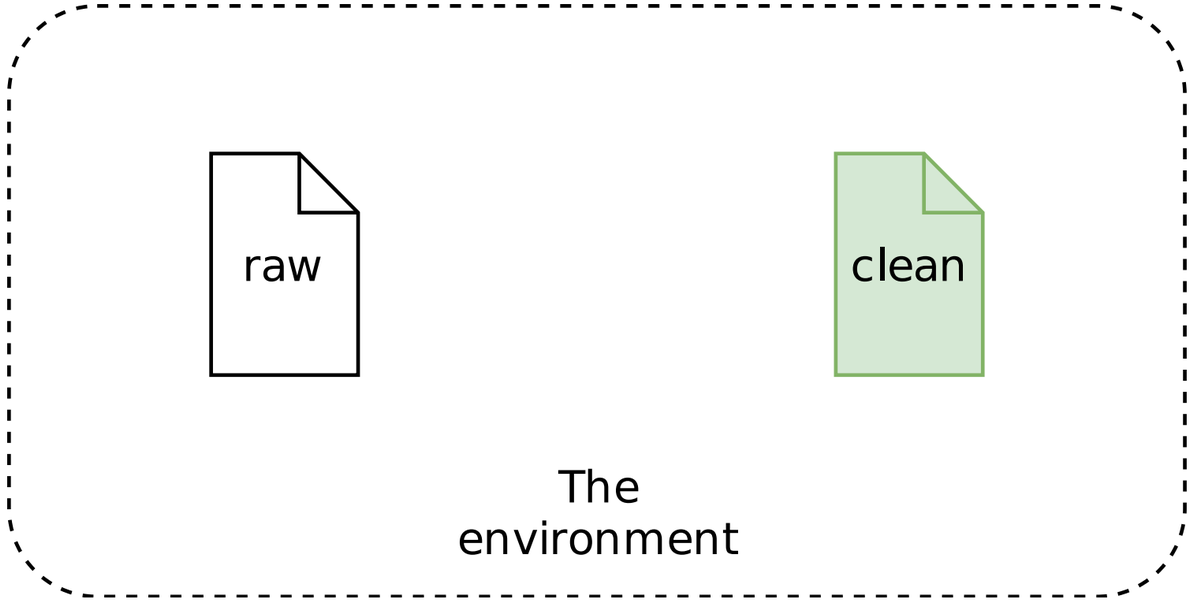
```
def test_on_raw_data():
    # Setup: create the raw data file
```

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



Step 2: Assert

```
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"
```



Step 3: Teardown

```
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.

The new workflow

Old workflow

- assert

New workflow

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

Fixture

```
import pytest

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

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

Fixture

```
import pytest

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

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

Test

```
import os
import pytest

def test_on_raw_data():
```

Fixture

```
@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_pat
    os.remove(raw_data_file_path)
    os.remove(clean_data_file_path)
```

Test

```
import os
import pytest

def test_on_raw_data(raw_and_clean_data_file):
    raw_path, clean_path = raw_and_clean_data_fil
    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"
```

The built-in tmpdir fixture

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

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()`.

```
@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
```

Let's practice setup and teardown using fixtures!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

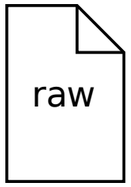
Mocking

UNIT TESTING FOR DATA SCIENCE IN PYTHON

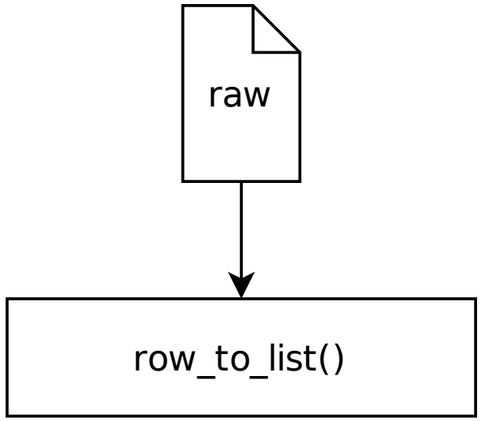
Dibya Chakravorty
Test Automation Engineer

datacamp

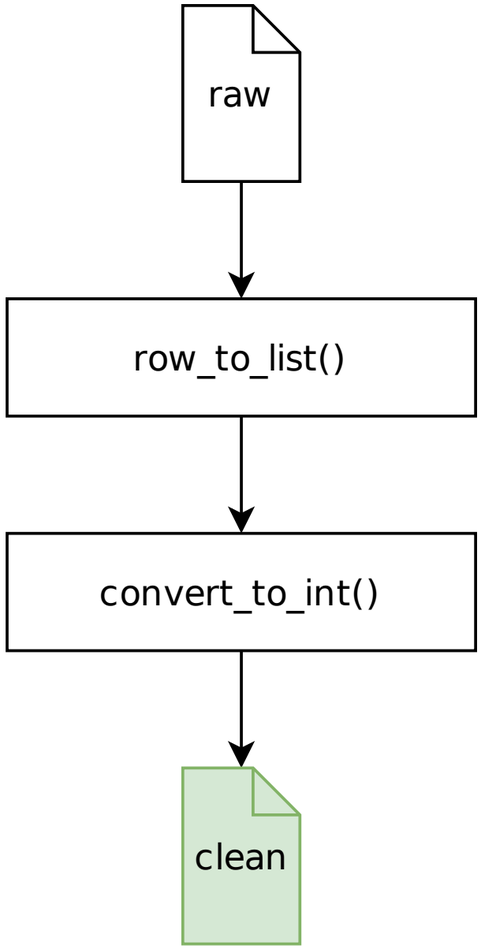
The preprocessing function



The preprocessing function



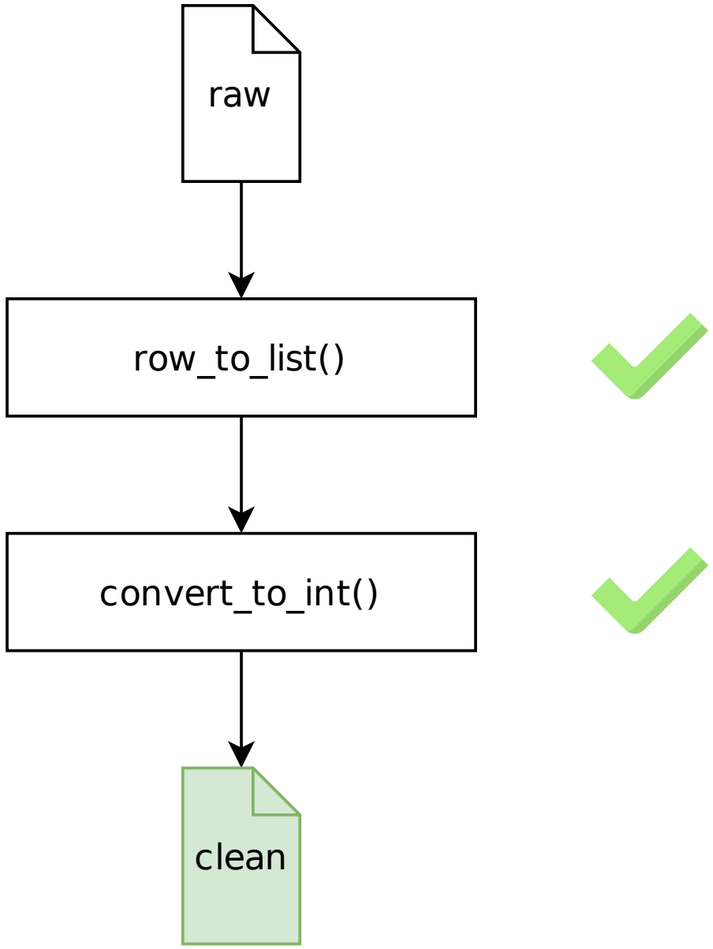
The preprocessing function




```
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 ======
```

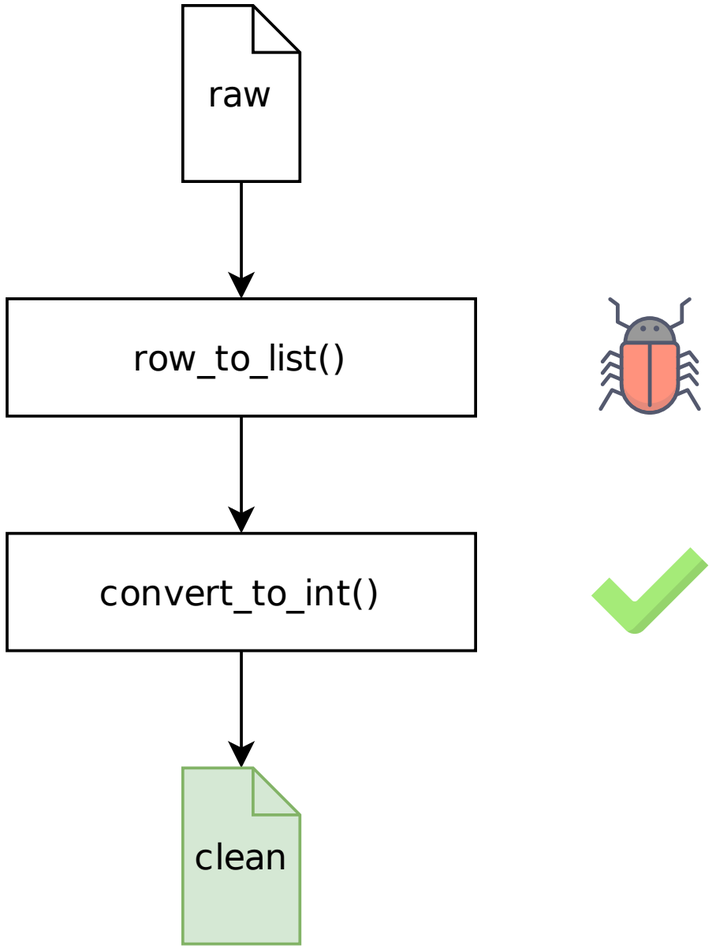
Test result depend on dependencies




```
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

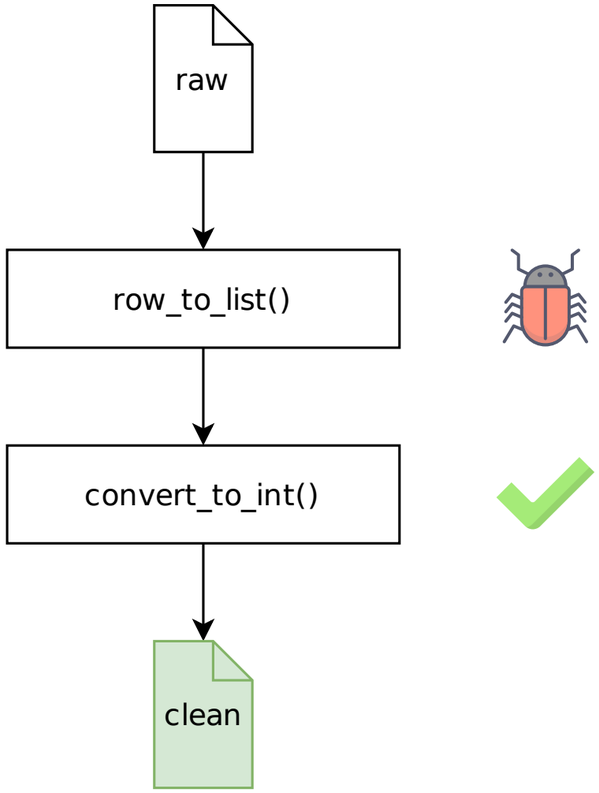
- function under test i.e. `preprocess()`.
- not dependencies e.g. `row_to_list()` or `convert_to_int()`.

Mocking: testing functions independently of dependencies

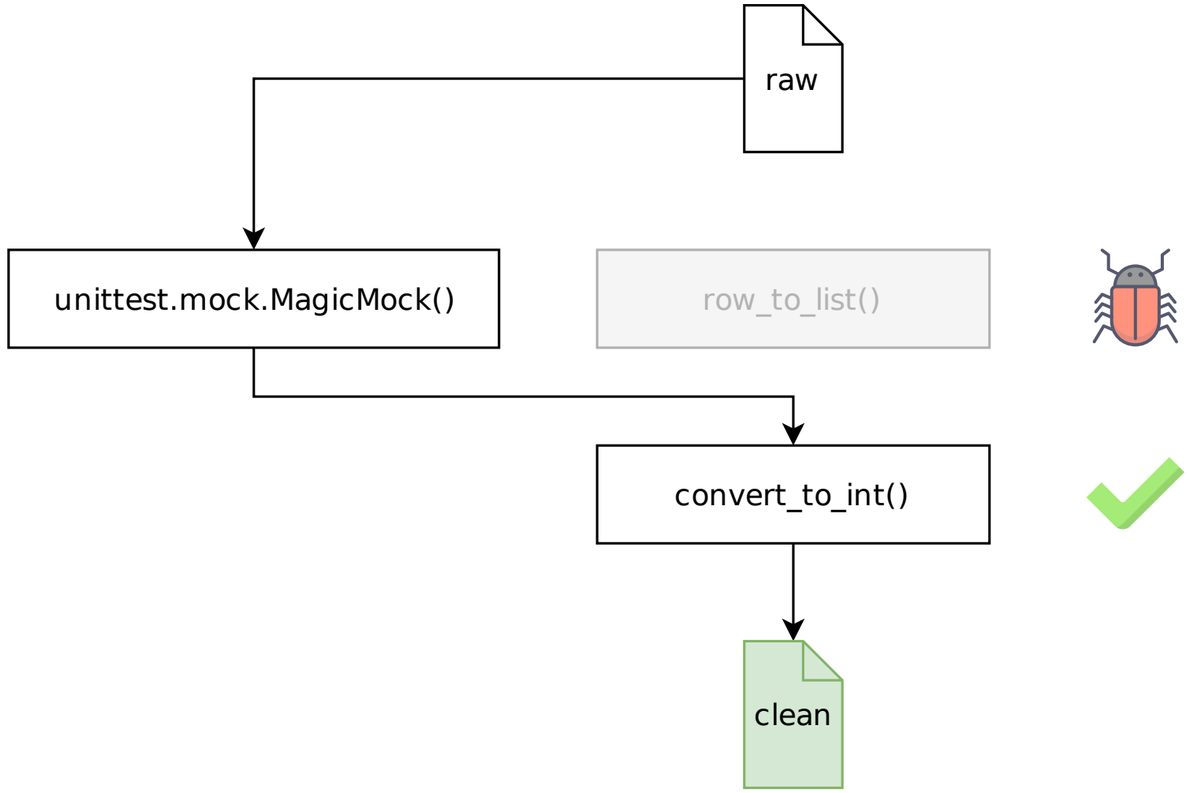
Packages for mocking in `pytest`

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

MagicMock() and mocker.patch()



MagicMock() and mocker.patch()




```
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("<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
```

```
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]
```

```
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
```

Side effect

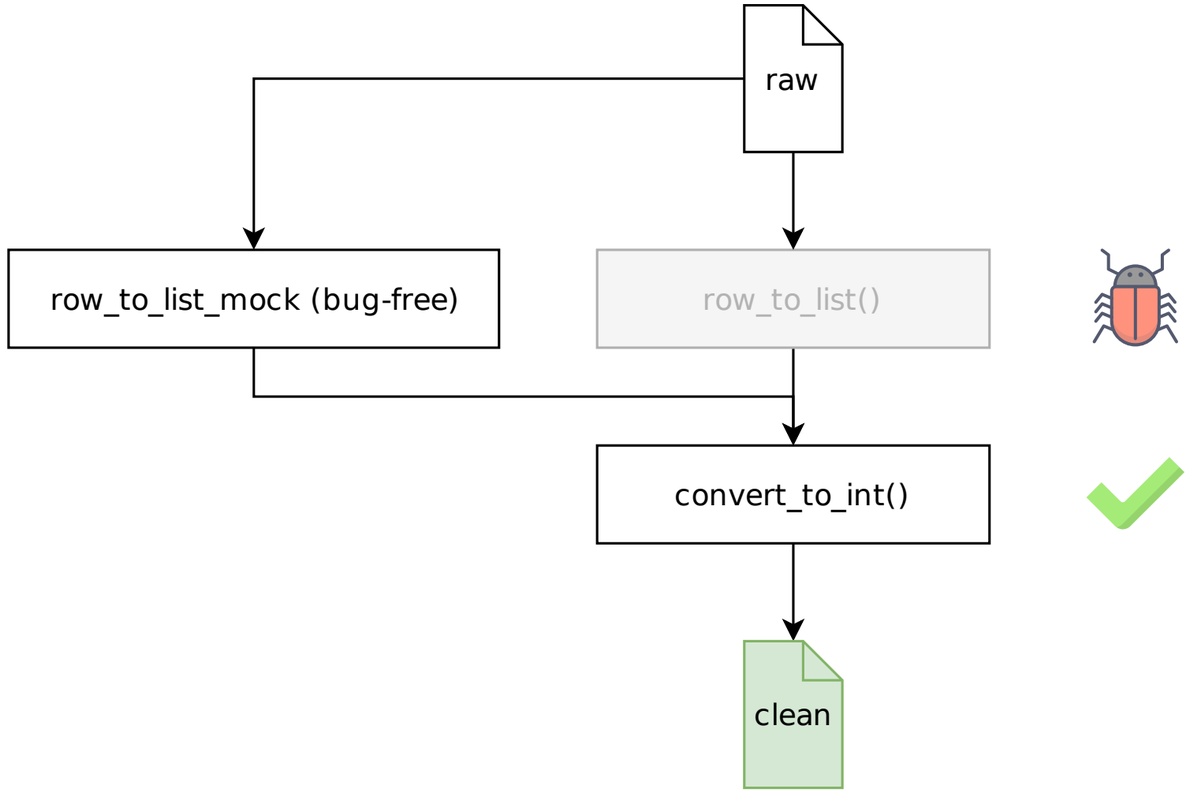
- Raw data

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

```
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]
```

```
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
        )
```

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)
```

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")
 ]
```

```
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)
```

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")
 ]
```

```
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 =============
```

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

datacamp

Functions we have tested so far

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

Raw data to clean data

```
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

```
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

```
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)
    )
```

```
split_into_training_and_testing_sets(data)
```

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

Functions are well tested - thanks to you!

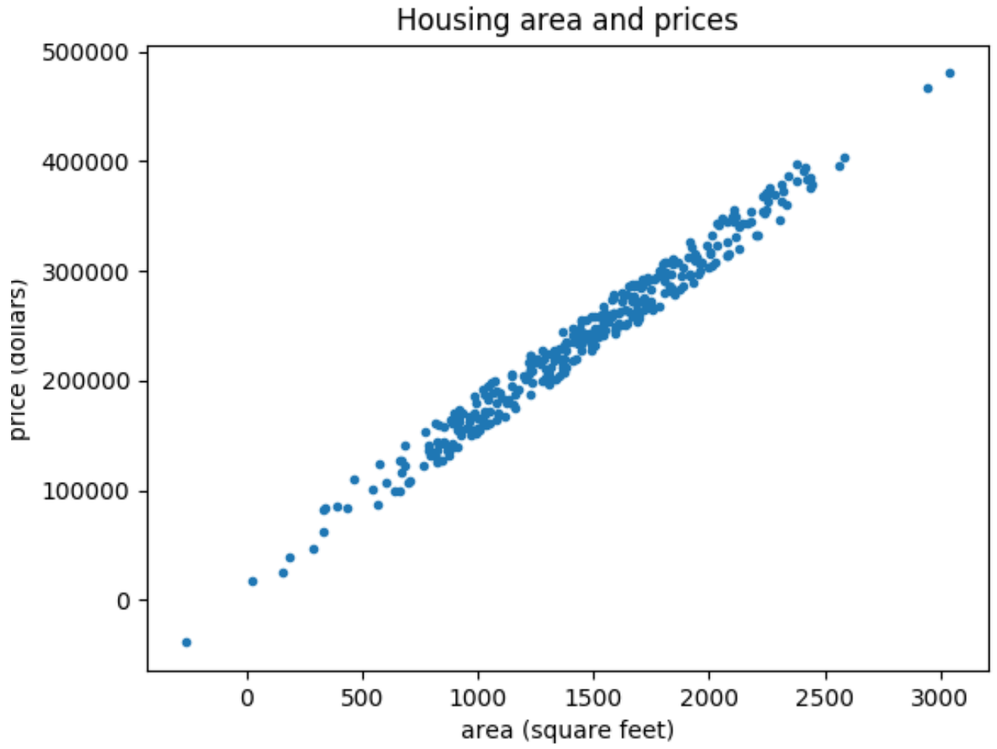
The linear regression model

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

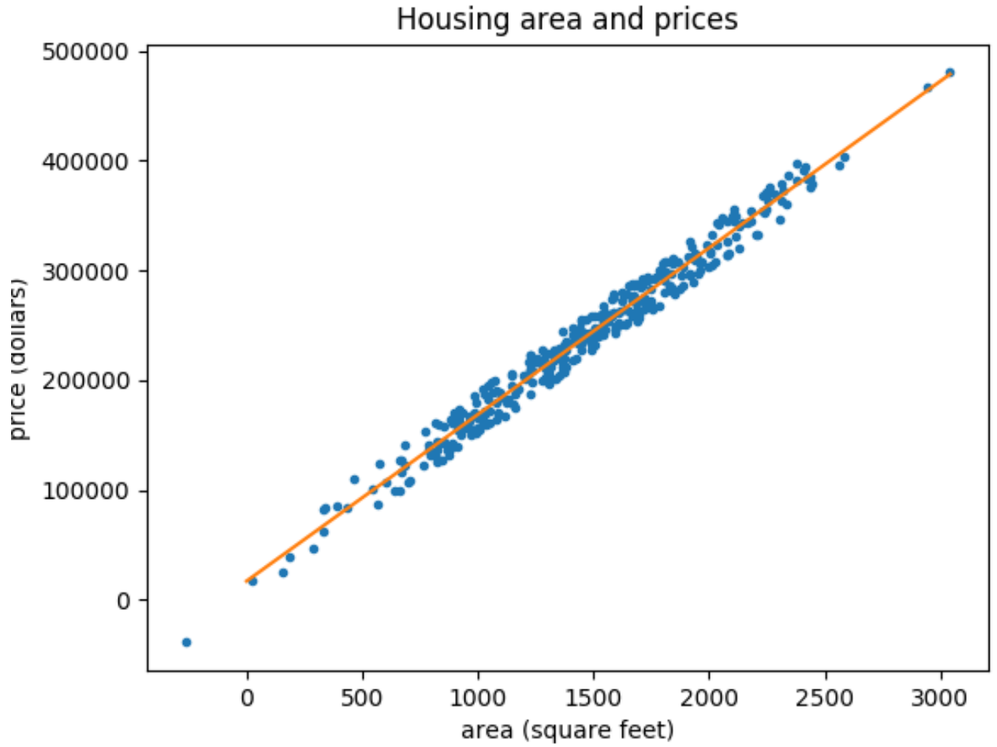
The linear regression model

```
from scipy.stats import linregress


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

Return values difficult to compute manually



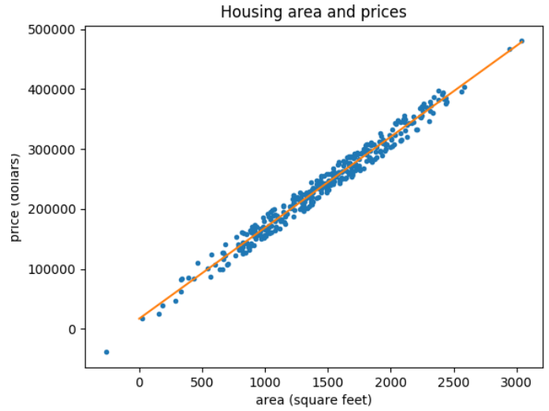
Return values difficult to compute manually

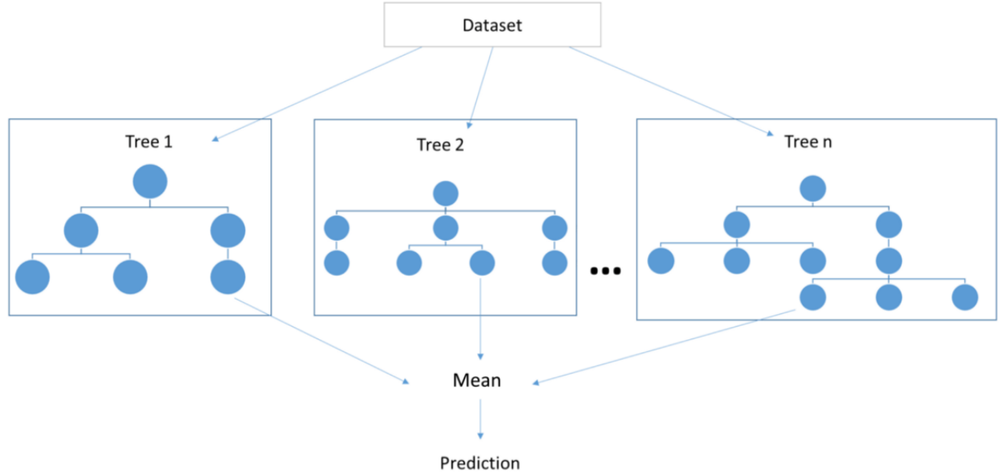


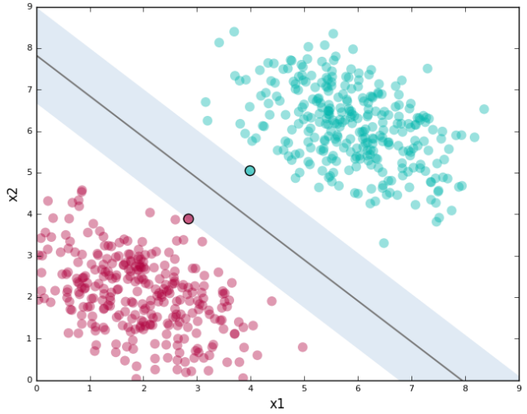
Return values difficult to compute manually

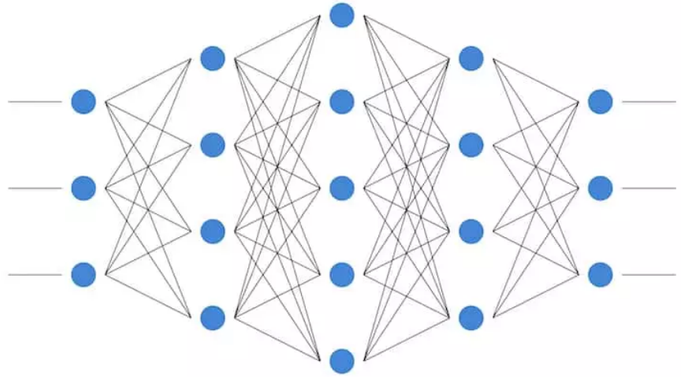
- Cannot test `train_model()` without knowing expected return values.

True for all data science models

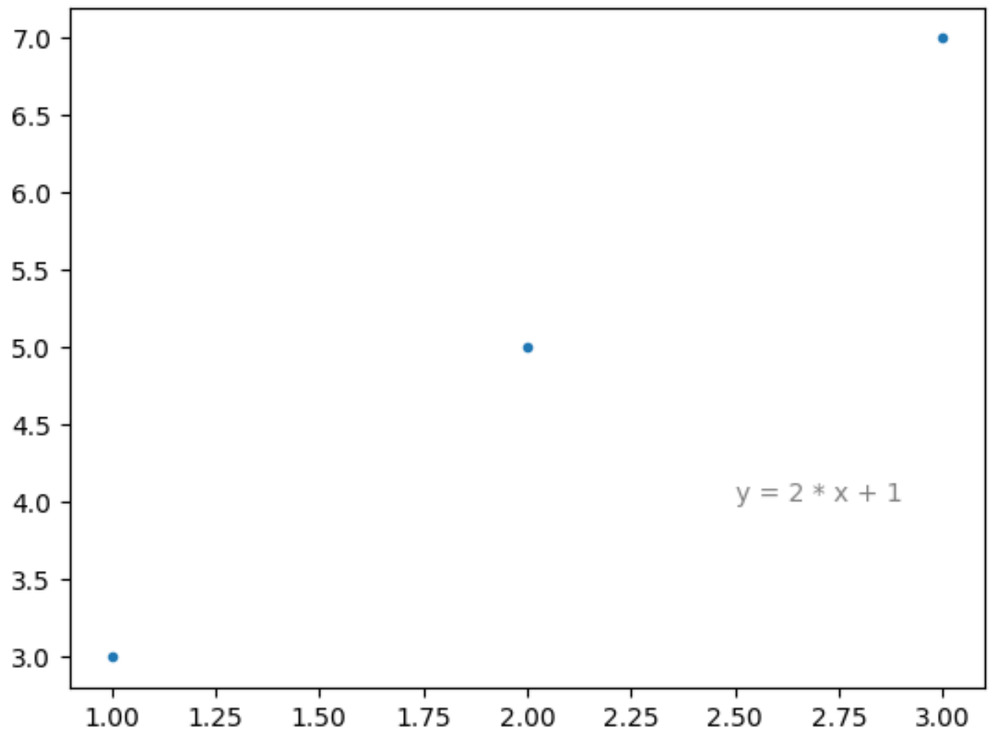






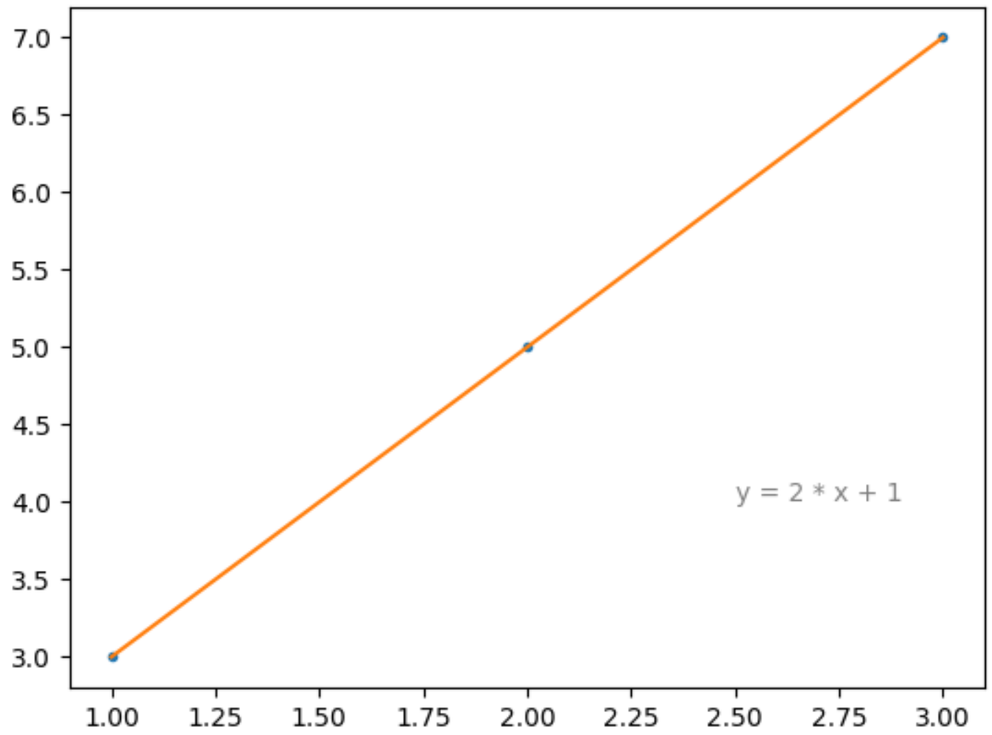


Trick 1: Use dataset where return value is known



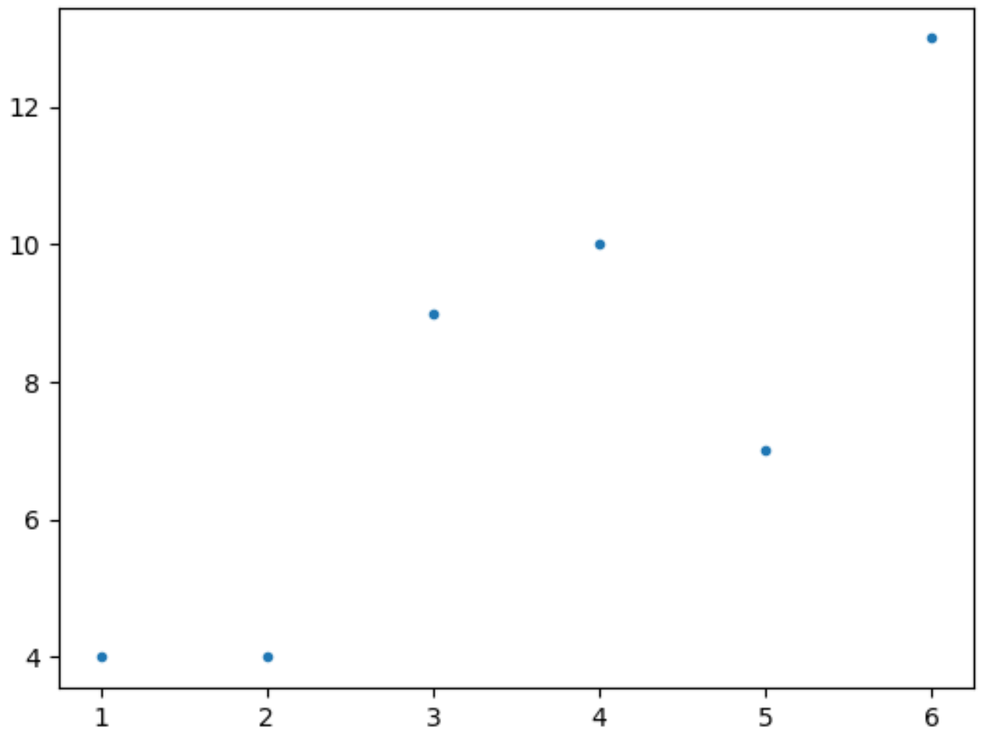

```
import pytest
import numpy as np
from models.train import train_model

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



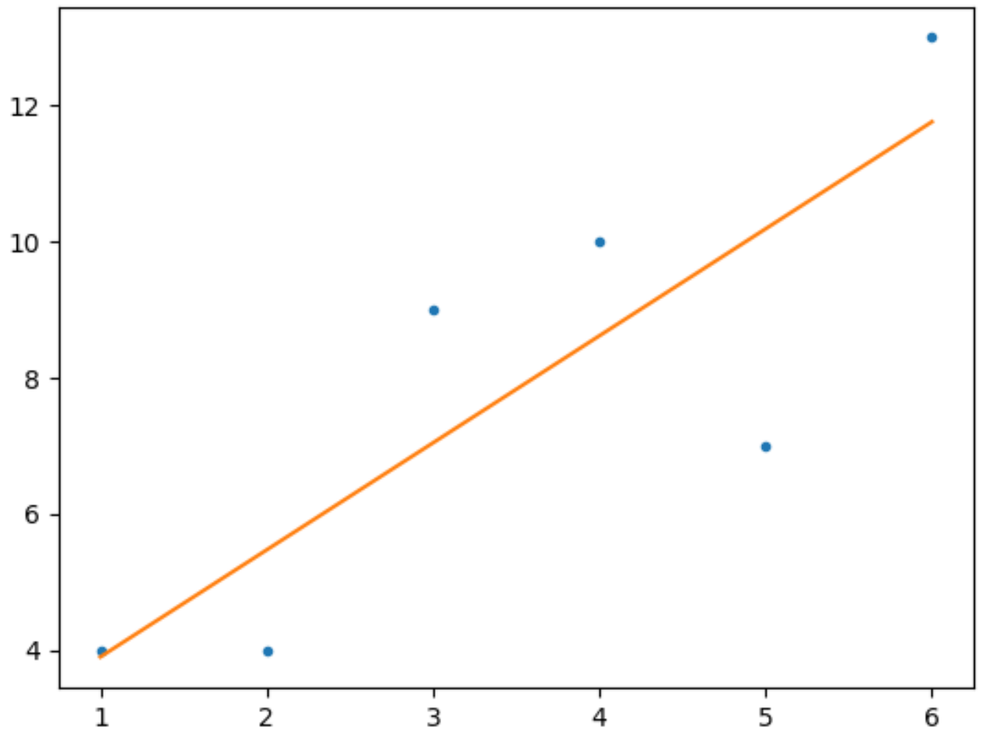

```
import pytest
import numpy as np
from models.train import train_model

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
        )
```

Trick 2: Use inequalities




```
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




```
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.

Using the model

```
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)
```

```
train_model(training_set)
```

```
151.78430060614986 17140.77537937442
```

Testing model performance

```
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

datacamp

Pizza without cheese!

This lesson: testing matplotlib visualizations

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/
```

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`

```
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

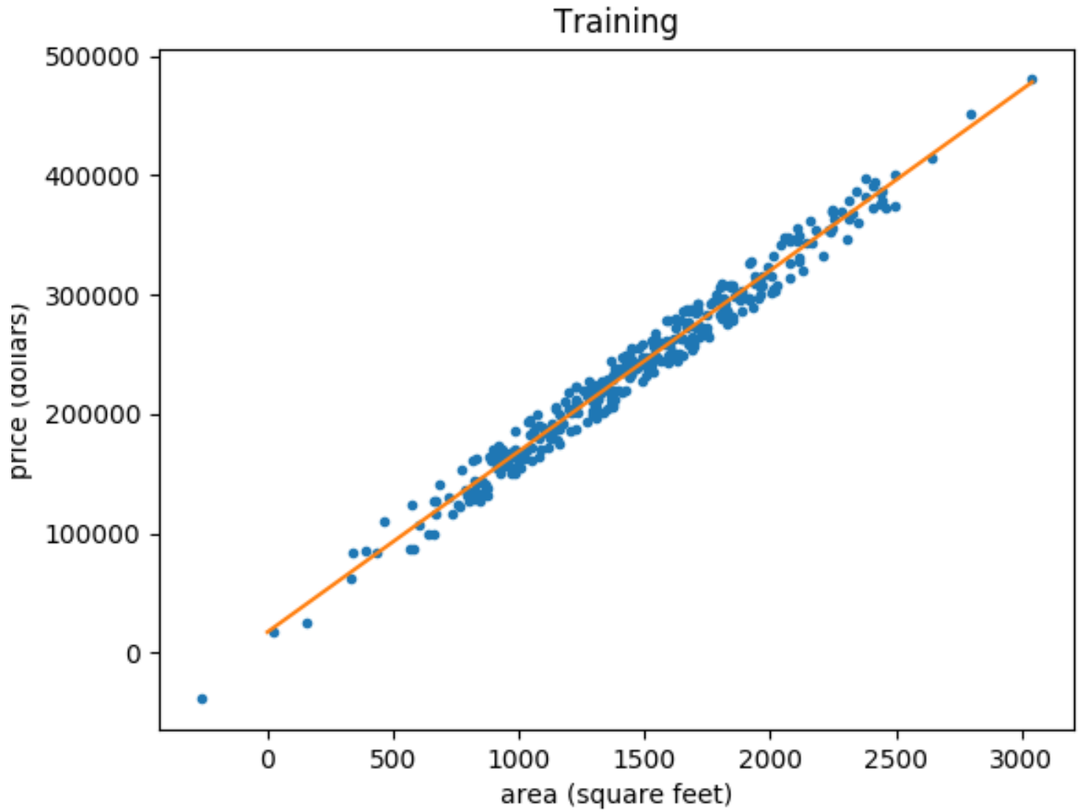
- `plots.py`

```
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/
```

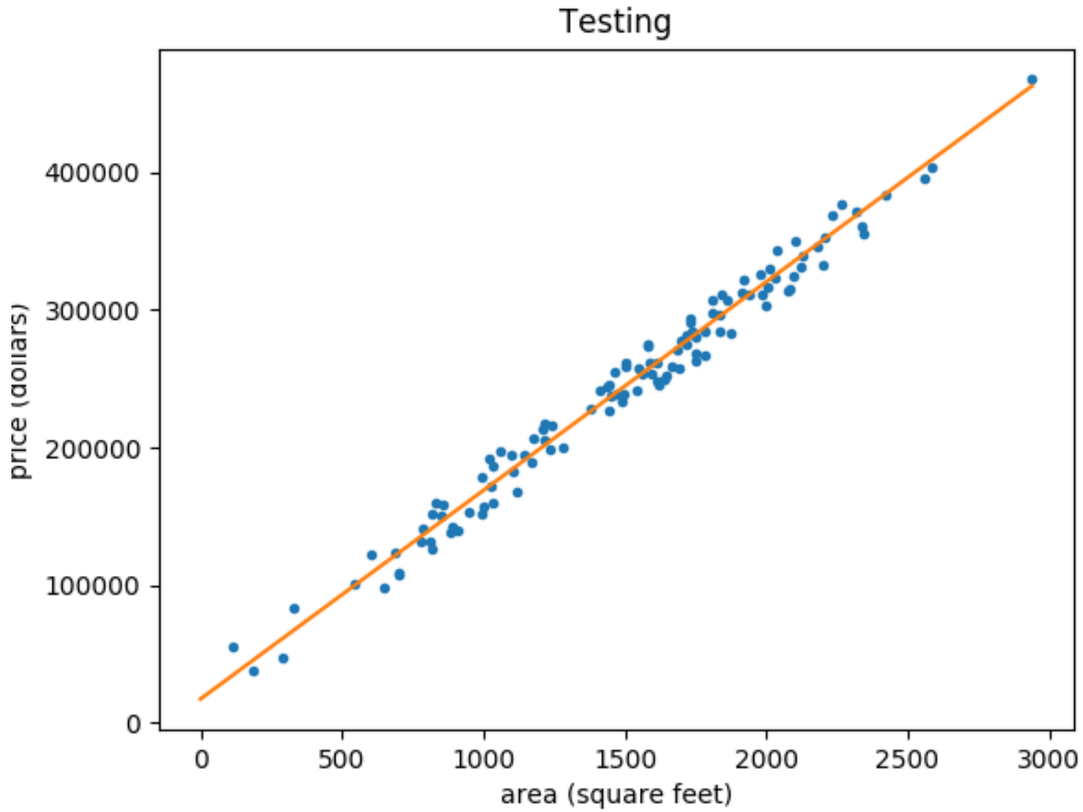
Training plot

```
...
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

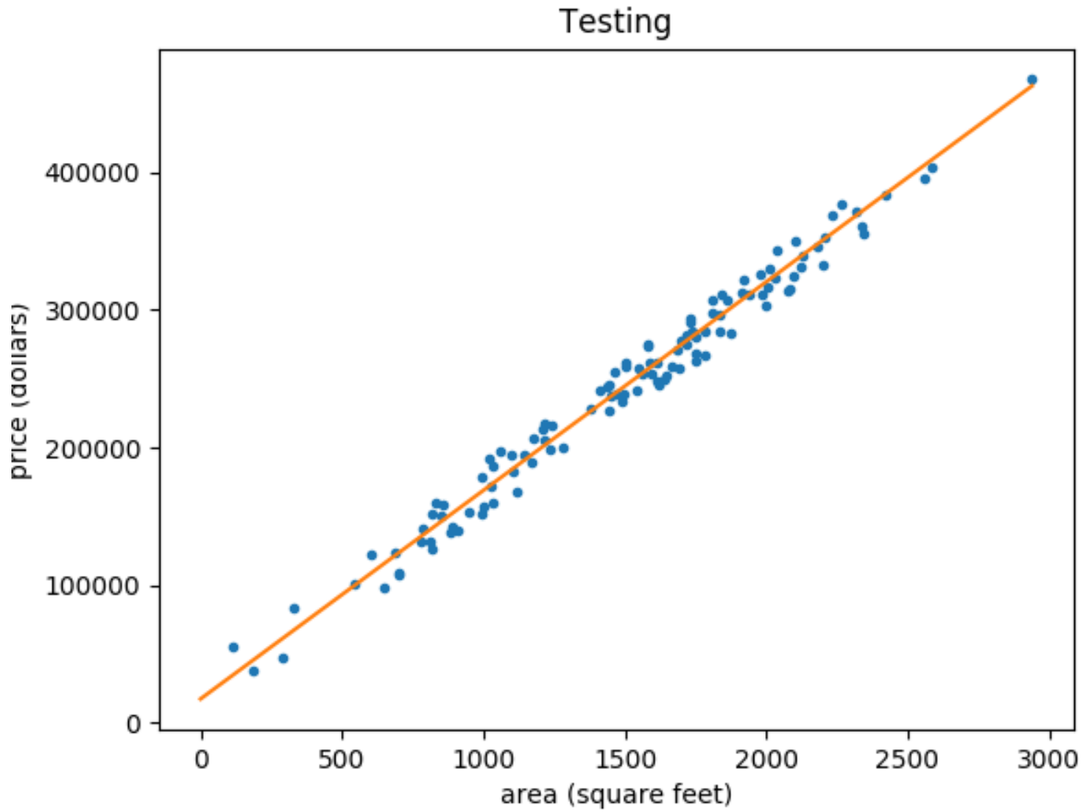
```
...
from visualization import get_plot_for_best_fit_line

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"
    )
```



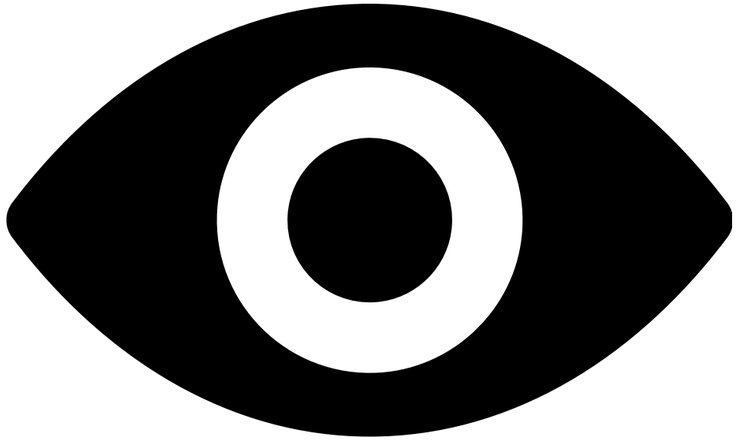
Don't test properties individually

`matplotlib.figure.Figure()`

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



Testing strategy for plots



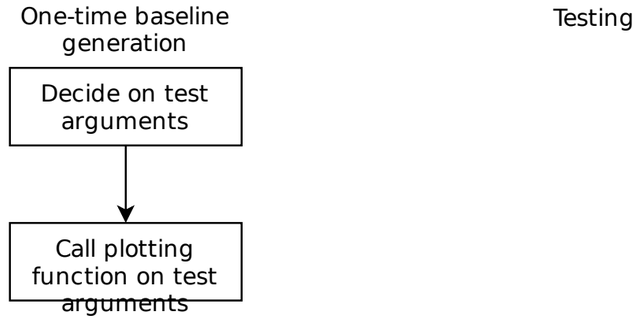
Testing strategy for plots

One-time baseline generation

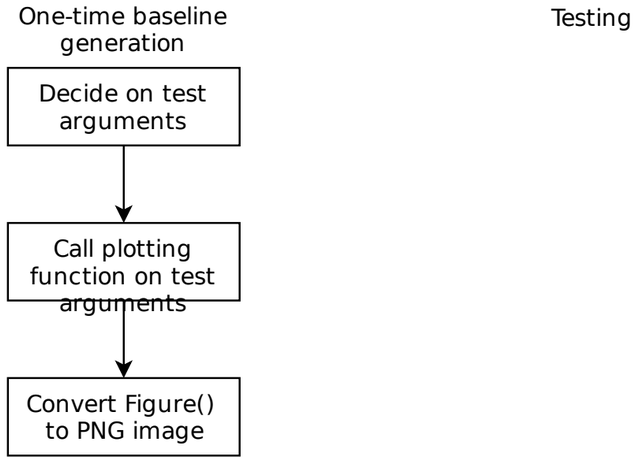
Testing

One-time baseline generation

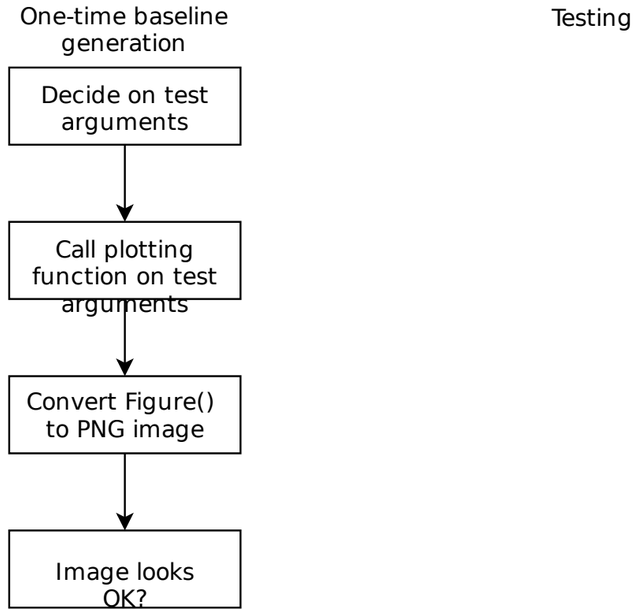
One-time baseline generation



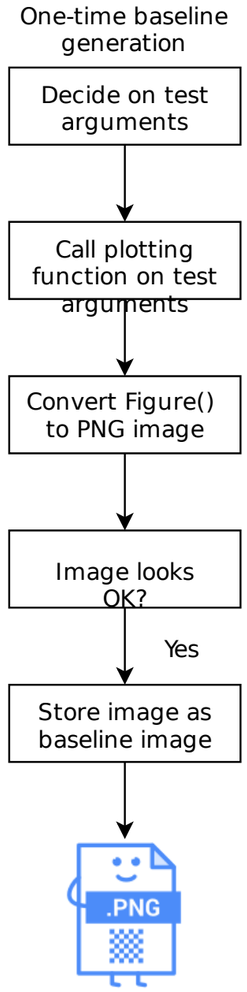
One-time baseline generation



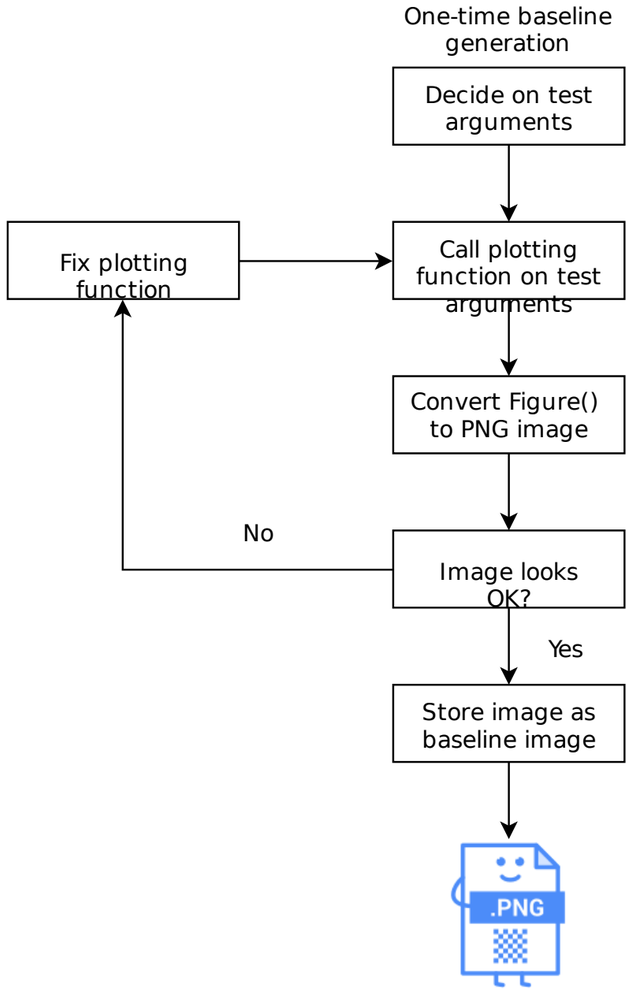
One-time baseline generation



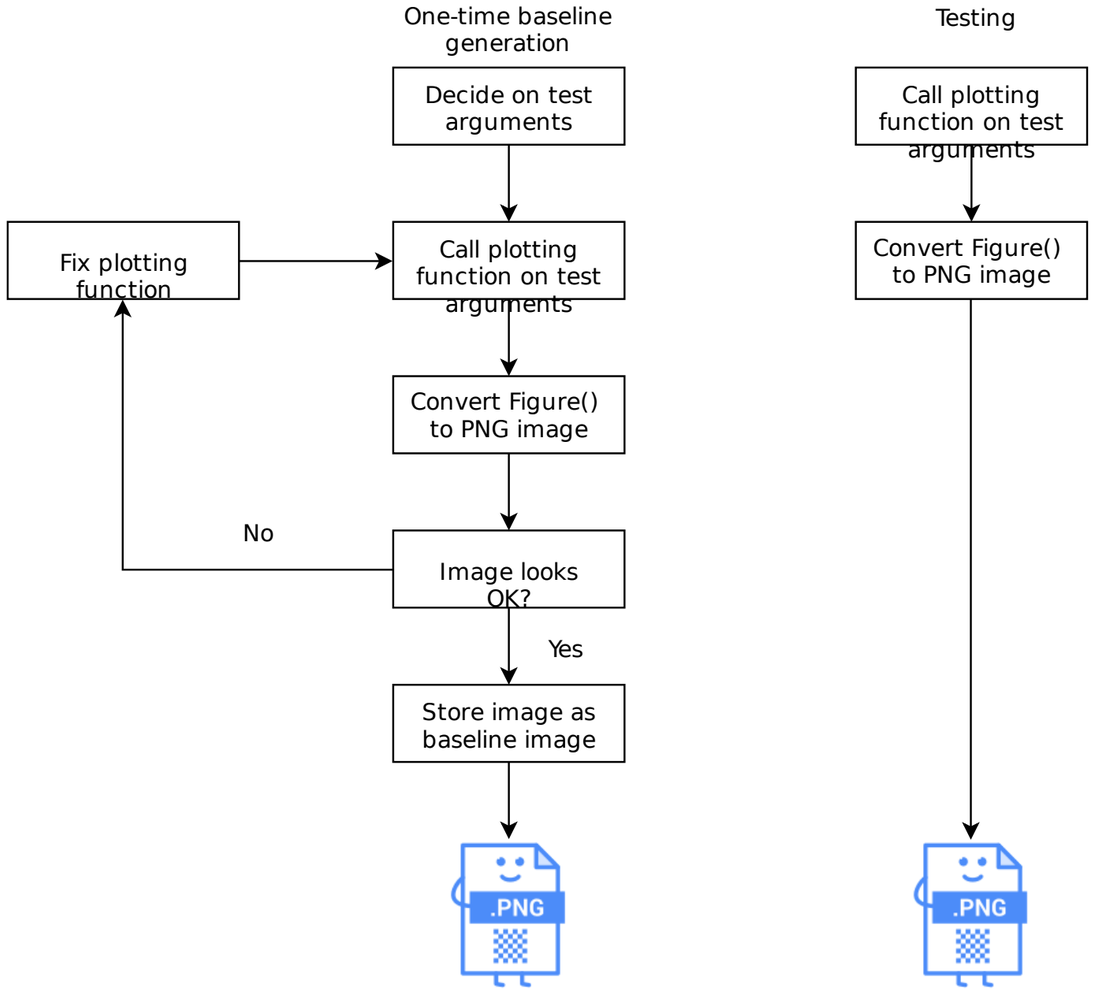
One-time baseline generation



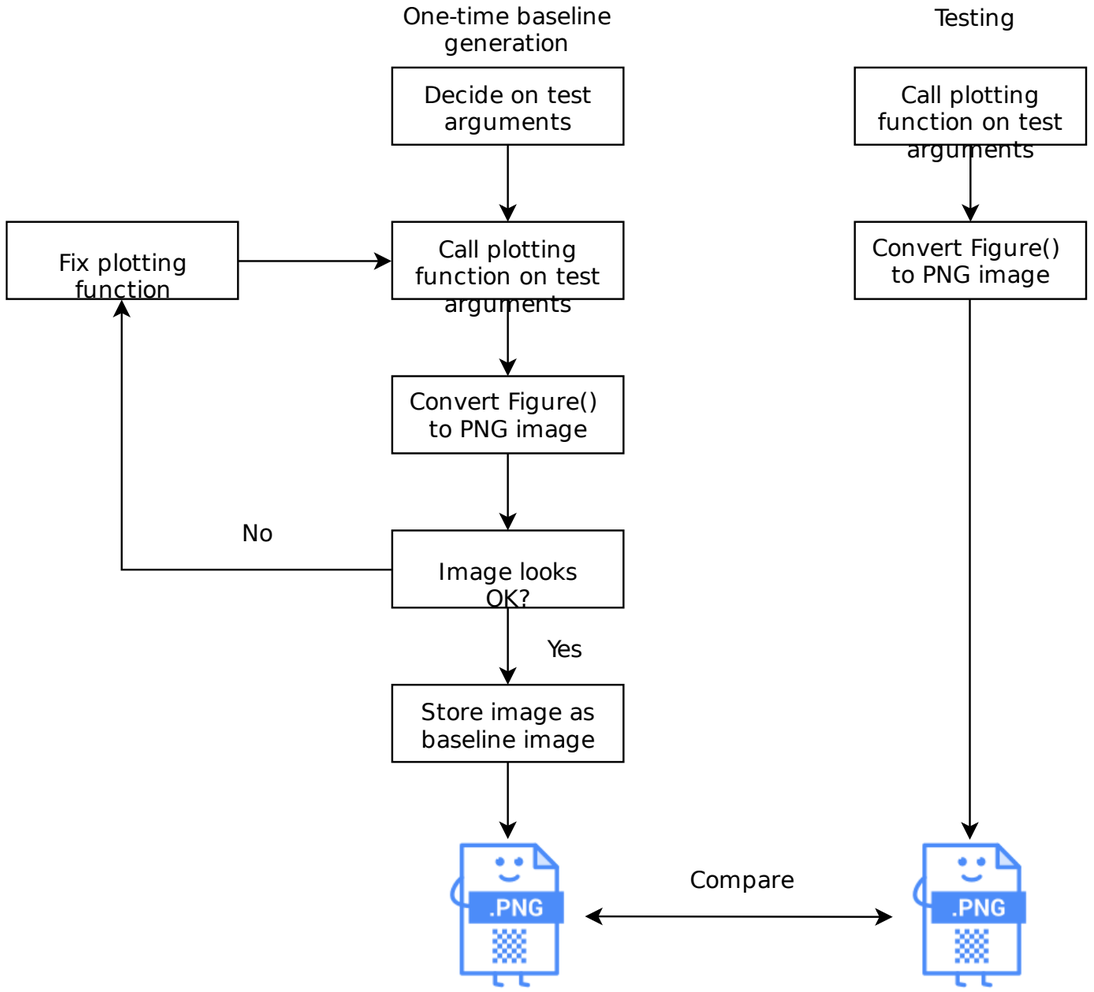
One-time baseline generation



Testing


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

Testing


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

pytest-mpl

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

```
pip install pytest-mpl
```

An example test

```
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)
```

An example 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)
```

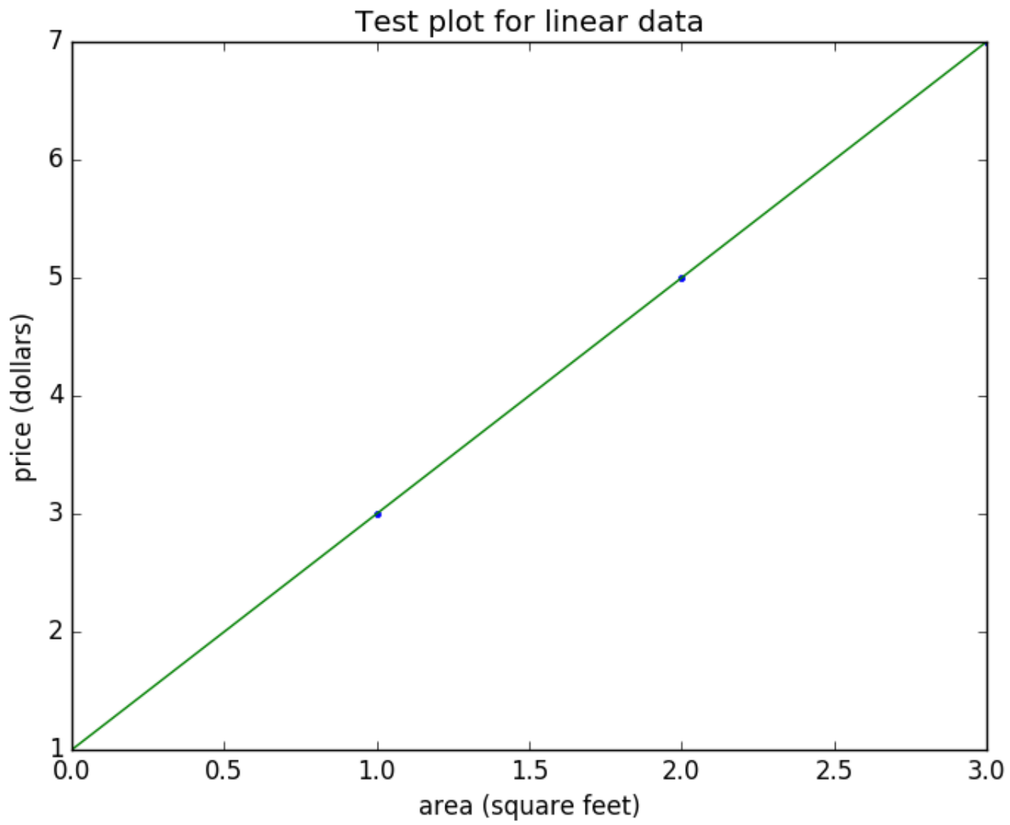
Generating the baseline image

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
```

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 =============
```

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 =============
```

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

datacamp

You've written so many tests

You've written so many tests

You've written so many tests

10

You've written so many tests

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

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!

UNIT TESTING FOR DATA SCIENCE IN PYTHON