

# Why unit test?

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# How can we test an implementation?

```
def my_function(argument):  
    ...
```



```
my_function(argument_1)
```

```
return_value_1
```

```
my_function(argument_2)
```

```
return_value_2
```

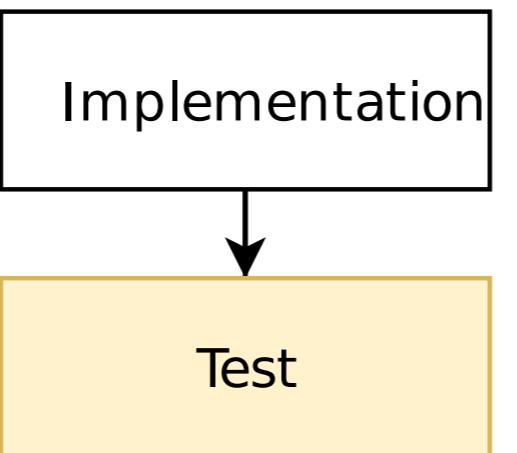
```
my_function(argument_3)
```

```
return_value_3
```

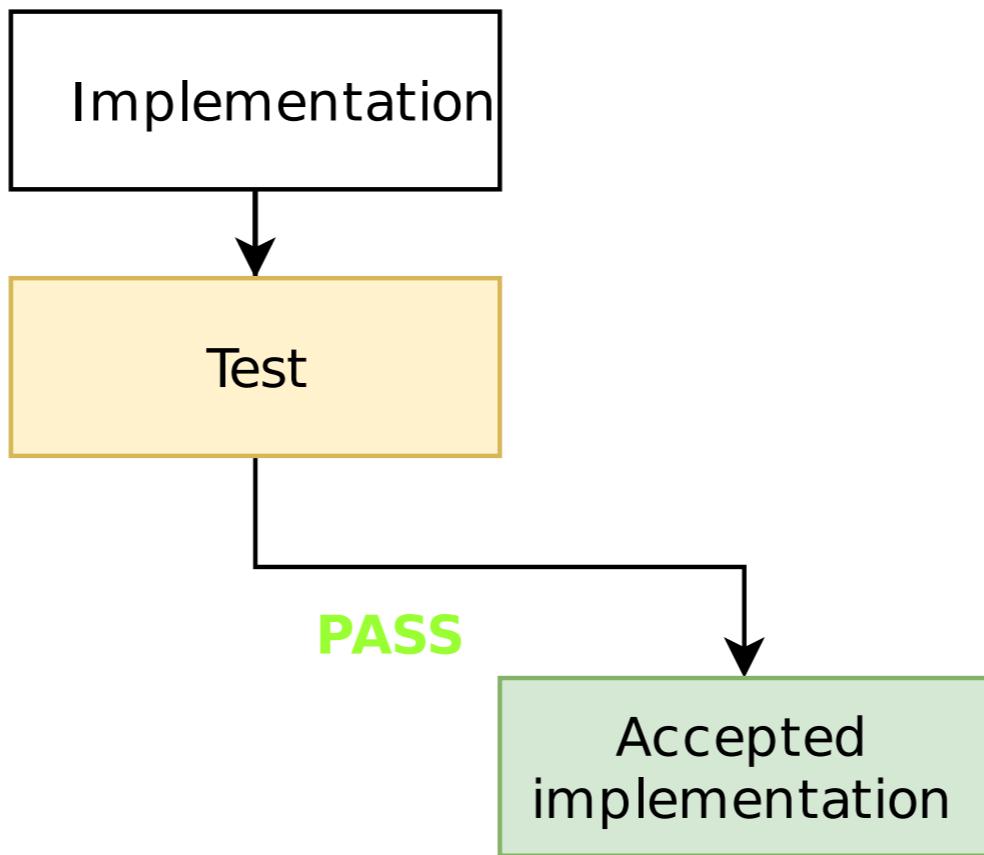
# Life cycle of a function

Implementation

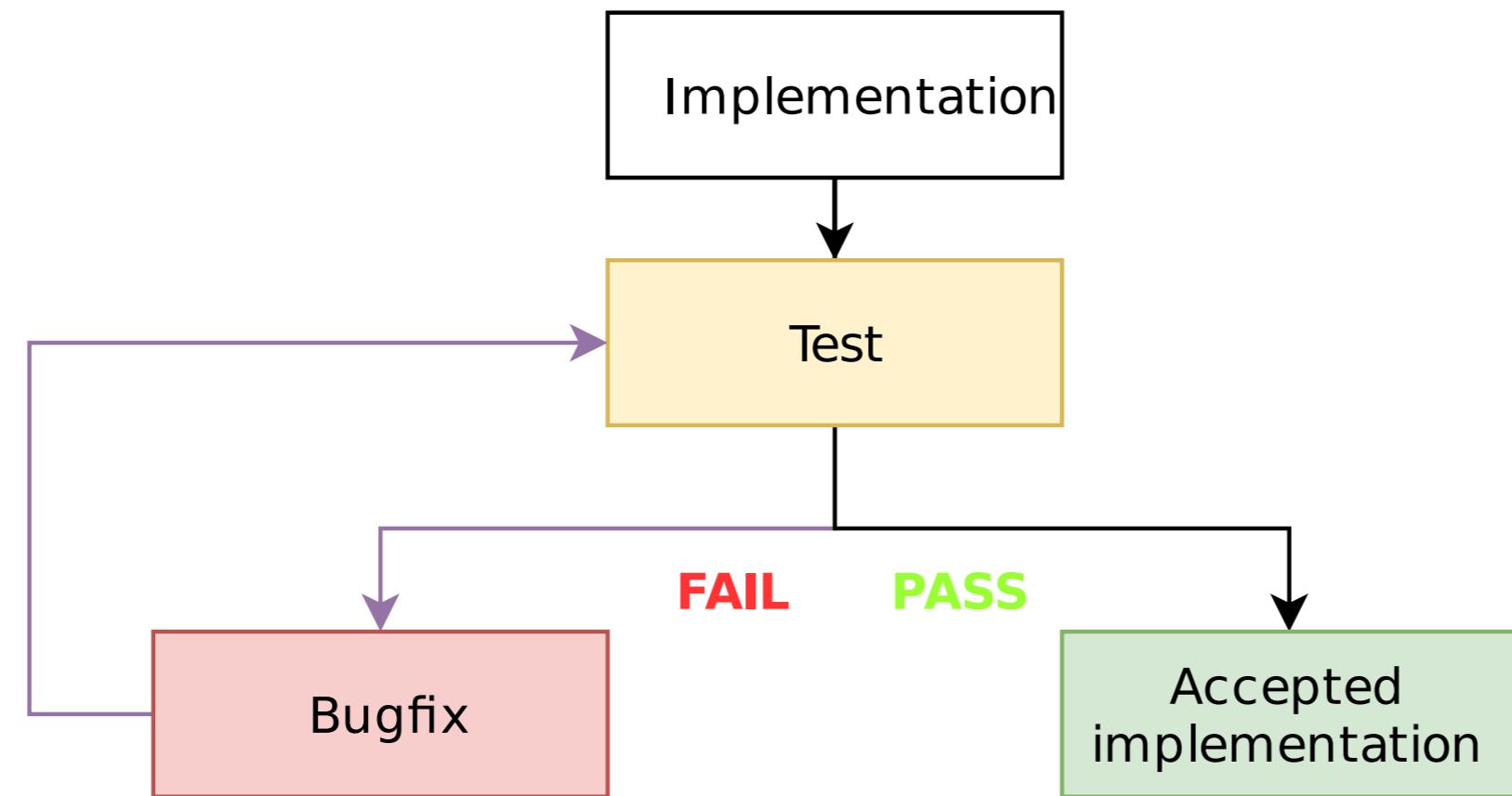
# Life cycle of a function



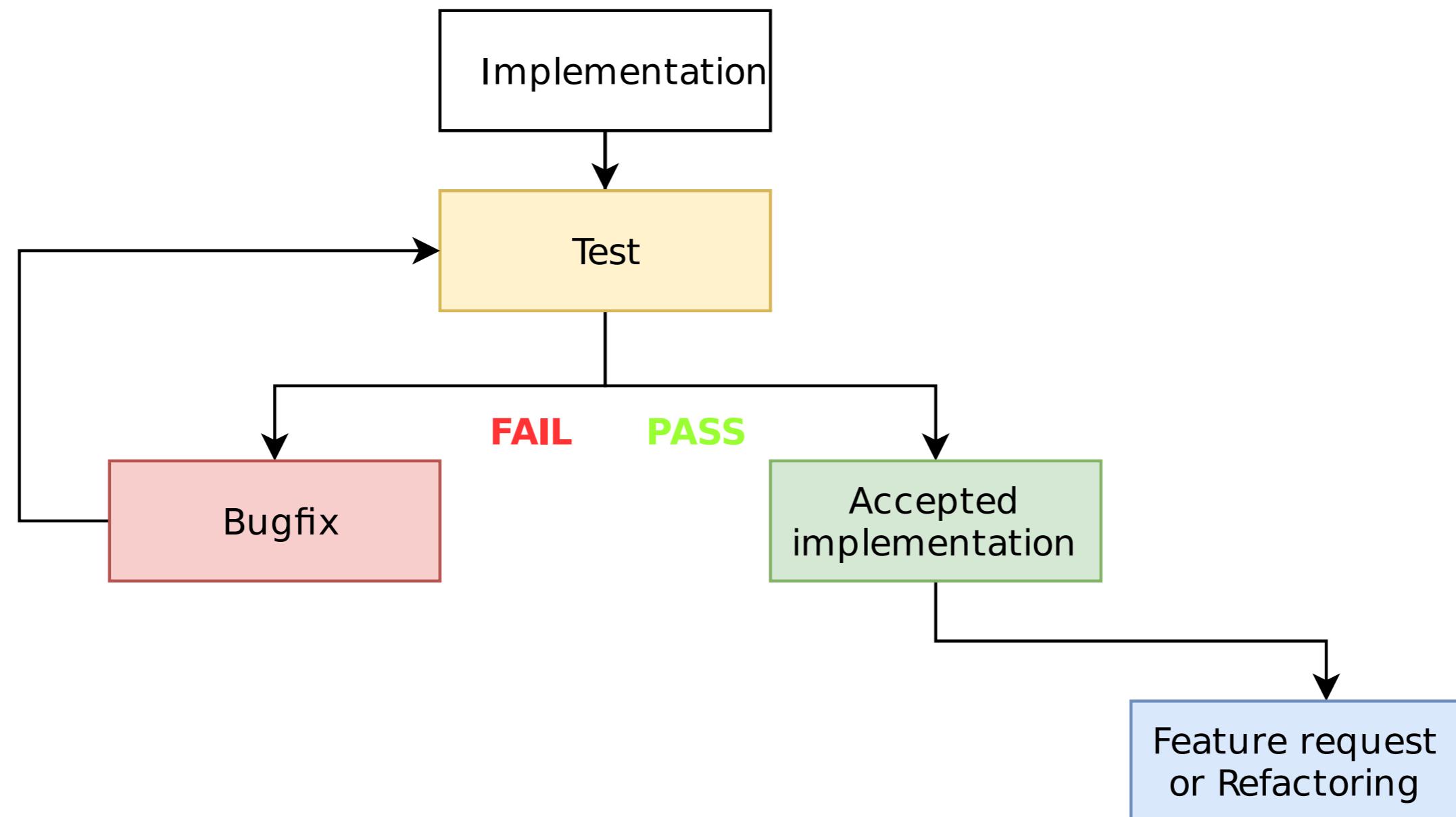
# Life cycle of a function



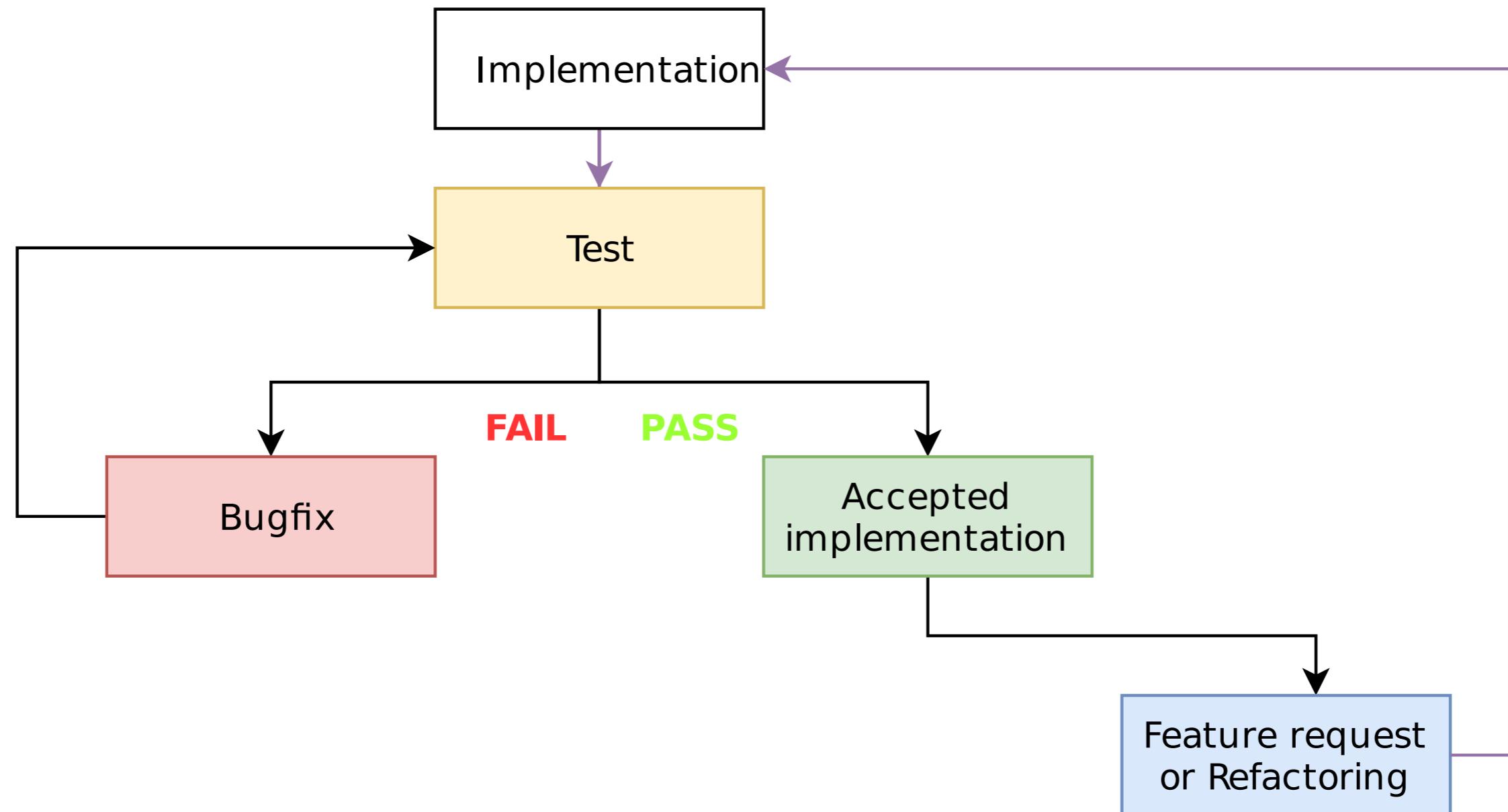
# Life cycle of a function



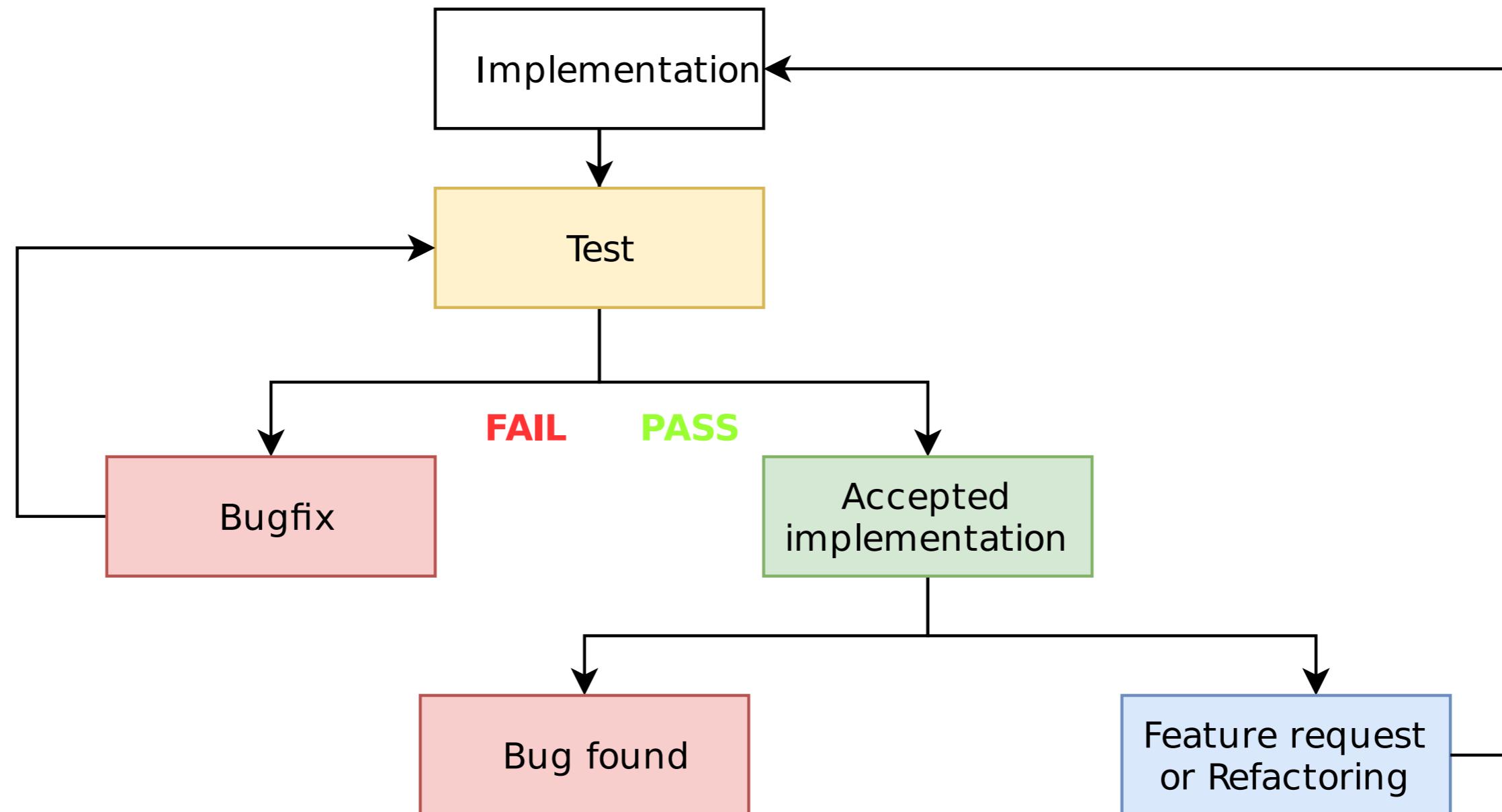
# Life cycle of a function



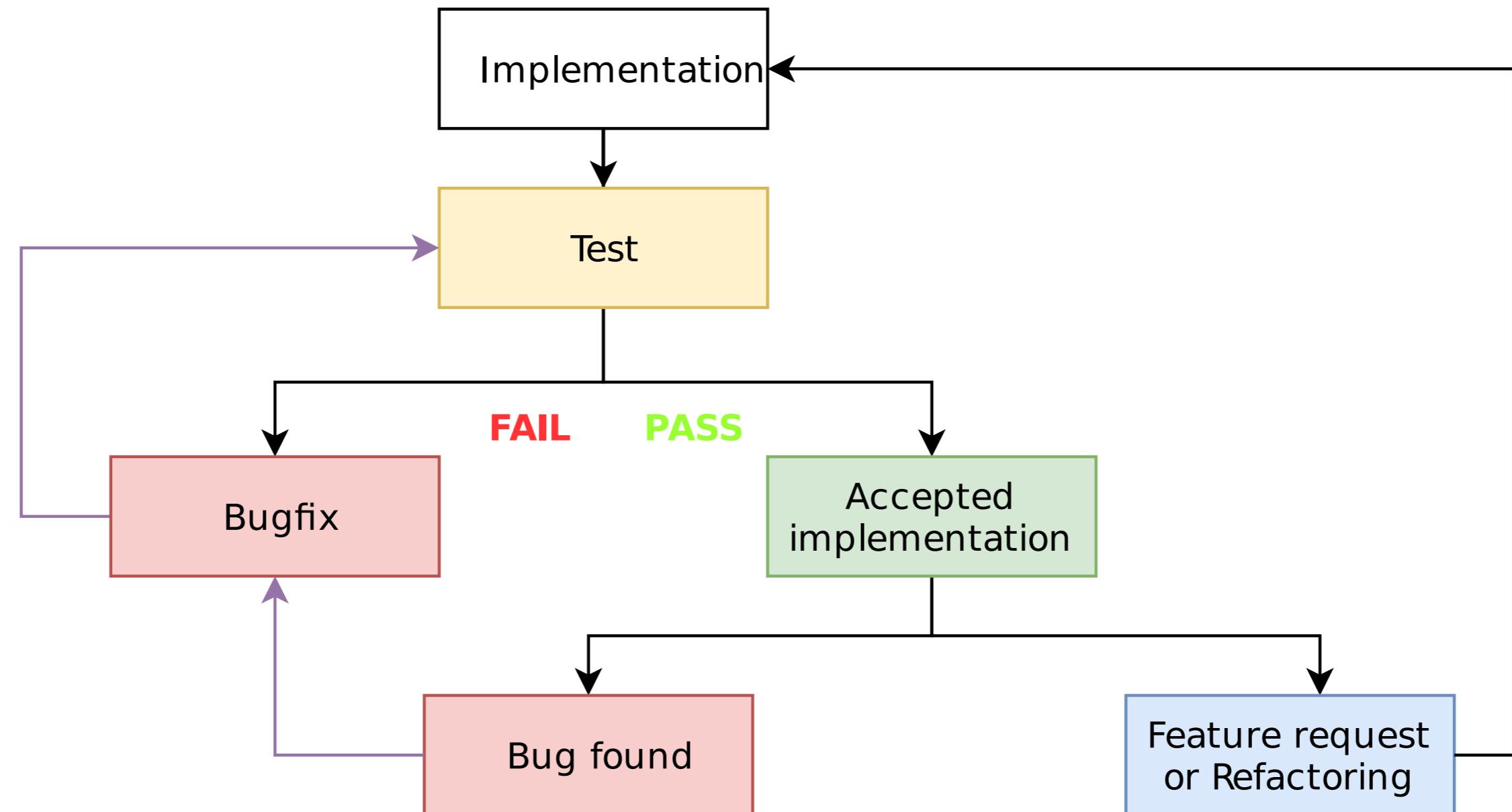
# Life cycle of a function



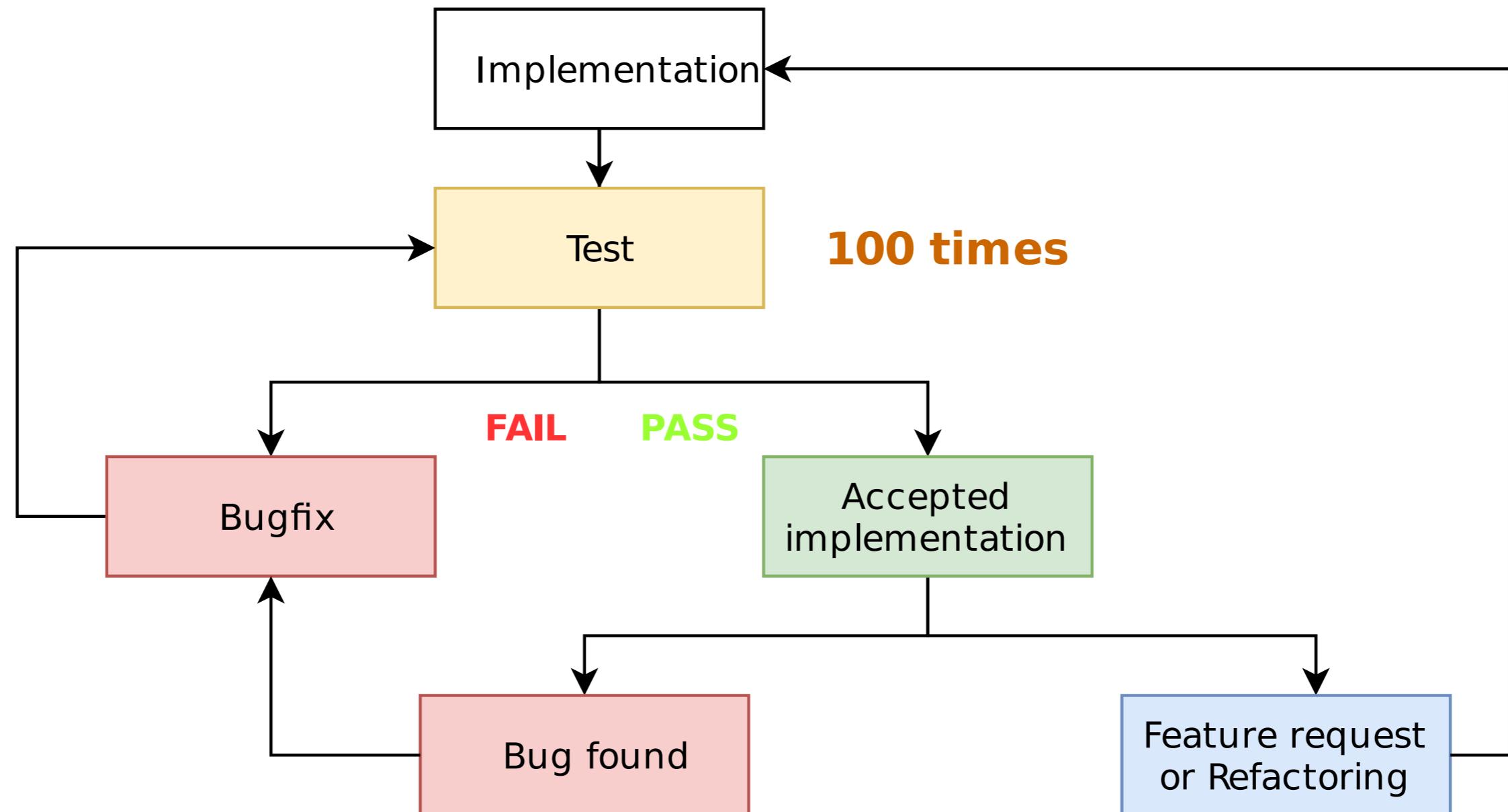
# Life cycle of a function



# Life cycle of a function



# Life cycle of a function



# Example

```
def row_to_list(row):  
    ...
```

area (sq. ft.)	price (dollars)
2,081	314,942
1,059	186,606
293,410	
1,148	206,186
1,506	248,419
1,210	214,114
1,697	277,794
1,268	194,345
2,318	372,162
1,463238,765	
1,468	239,007

File: housing\_data.txt

# Data format

```
def row_to_list(row):  
    ...
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]

area (sq. ft.)	price (dollars)
2,081	314,942
1,059	186,606
	293,410
1,148	206,186
1,506	248,419
1,210	214,114
1,697	277,794
1,268	194,345
2,318	372,162
1,463238,765	
1,468	239,007

File: housing\_data.txt

# Data isn't clean

```
def row_to_list(row):  
    ...
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]
"\t293,410\n"	Invalid	None

area (sq. ft.)	price (dollars)
2,081	314,942
1,059	186,606
293,410	<-- row with missing area
1,148	206,186
1,506	248,419
1,210	214,114
1,697	277,794
1,268	194,345
2,318	372,162
1,463238,765	
1,468	239,007

File: housing\_data.txt

# Data isn't clean

```
def row_to_list(row):  
    ...
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]
"\t293,410\n"	Invalid	None
"1,463238,765\n"	Invalid	None

```
area (sq. ft.)  price (dollars)  
2,081        314,942  
1,059        186,606  
          293,410      <-- row with missing area  
1,148        206,186  
1,506        248,419  
1,210        214,114  
1,697        277,794  
1,268        194,345  
2,318        372,162  
1,463238,765      <-- row with missing tab  
1,468        239,007
```

File: housing\_data.txt

# Time spent in testing this function

```
def row_to_list(row):  
    ...
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]
"\t293,410\n"	Invalid	None
"1,463238,765\n"	Invalid	None

```
row_to_list("2,081\t314,942\n")
```

```
[ "2,081", "314,942" ]
```

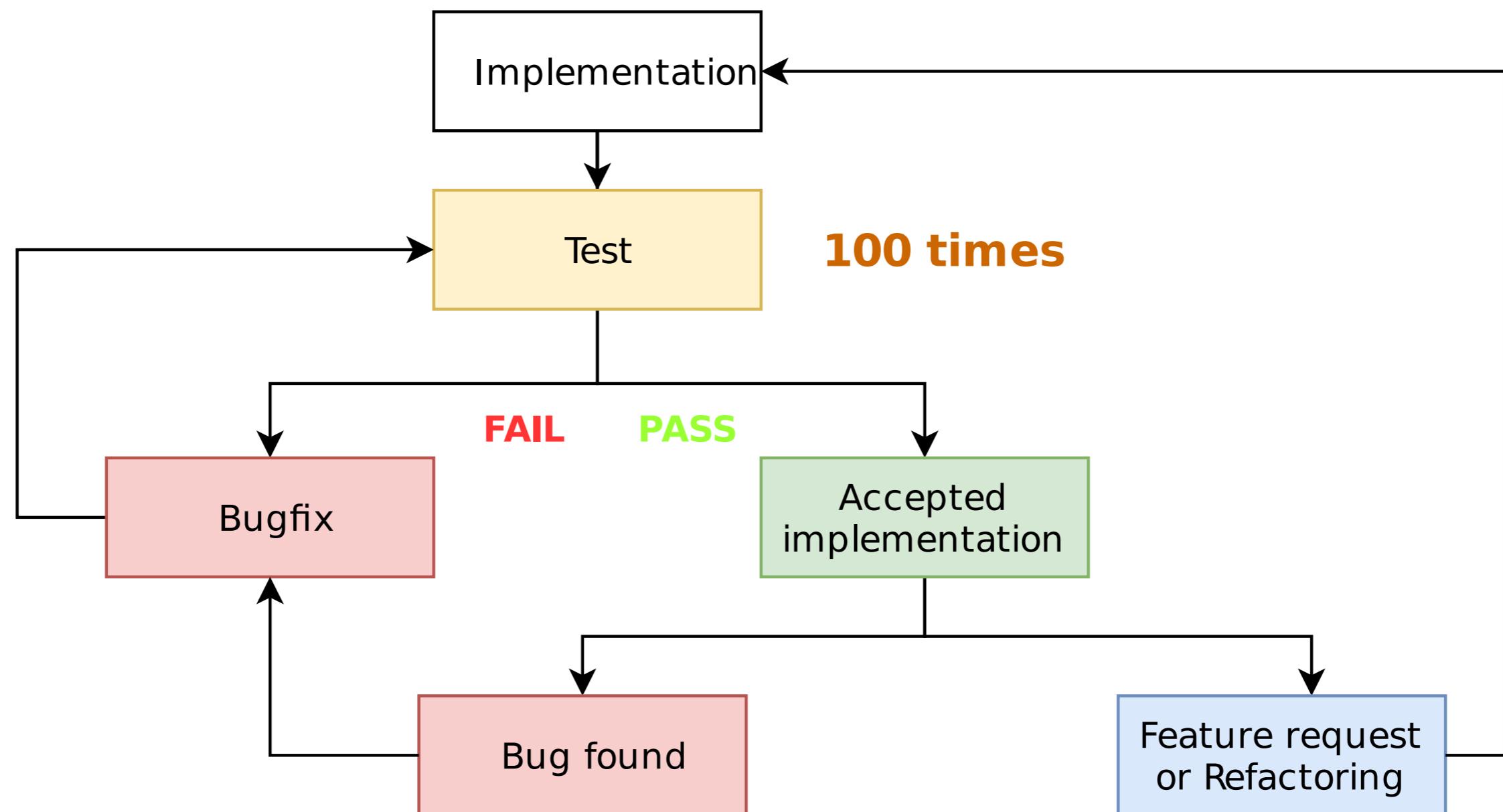
```
row_to_list("\t293,410\n")
```

```
None
```

```
row_to_list("1,463238,765\n")
```

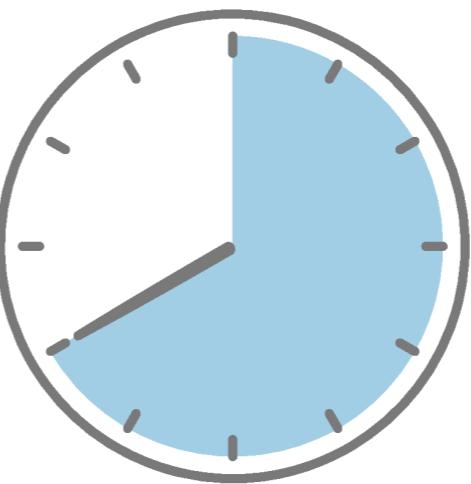
```
None
```

# Time spent in testing this function



# Time spent in testing this function

5 mins x 100 ~

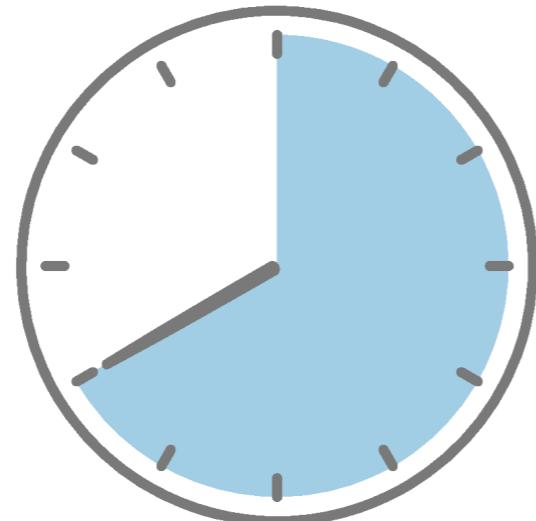


8 hours

# Manual testing vs. unit tests

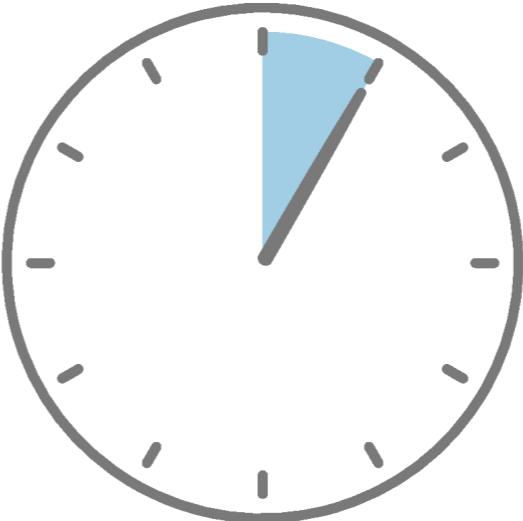
- Unit tests automate the repetitive testing process and saves time.

Manually testing on  
the interpreter



8 hours

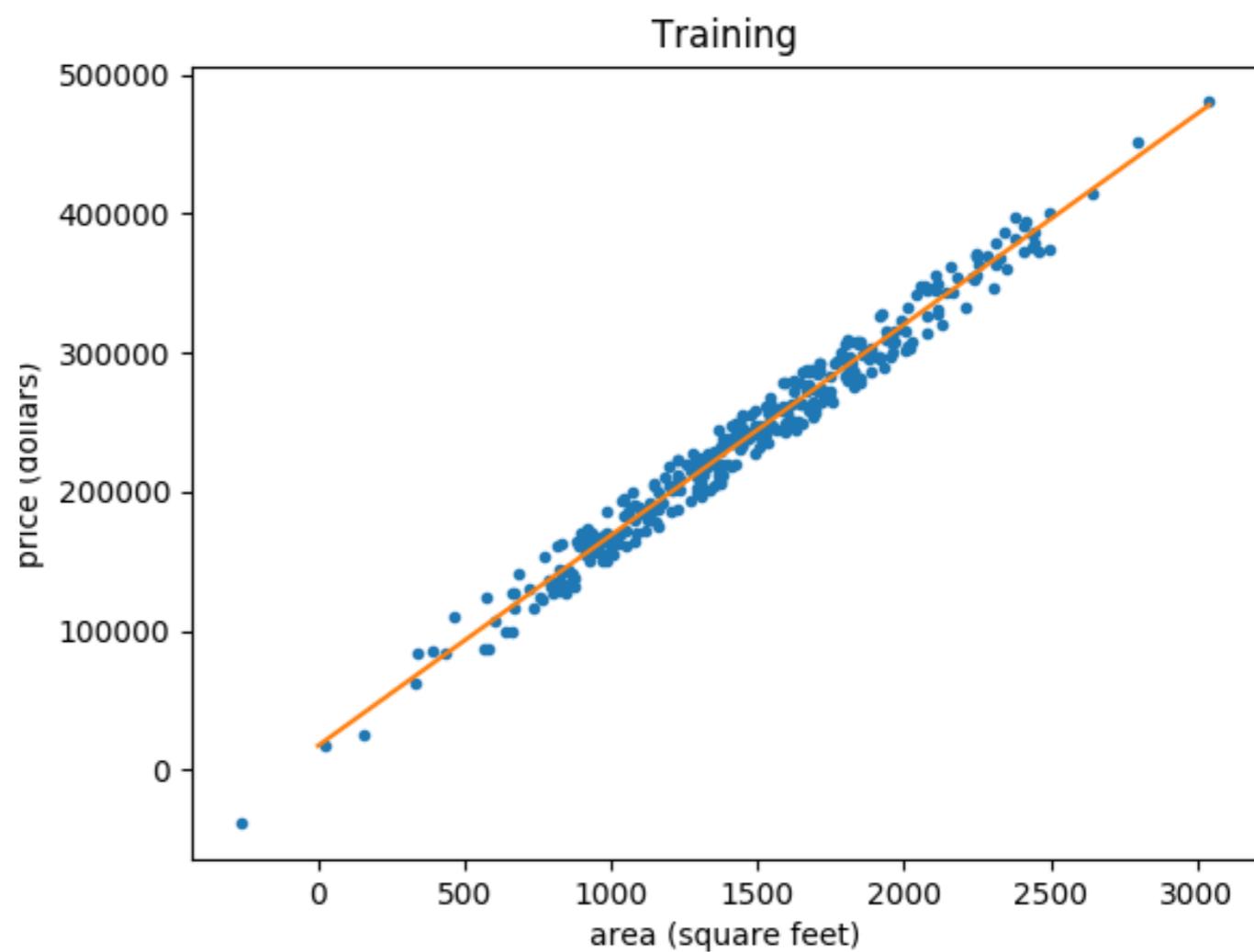
Unit tests



1 hour

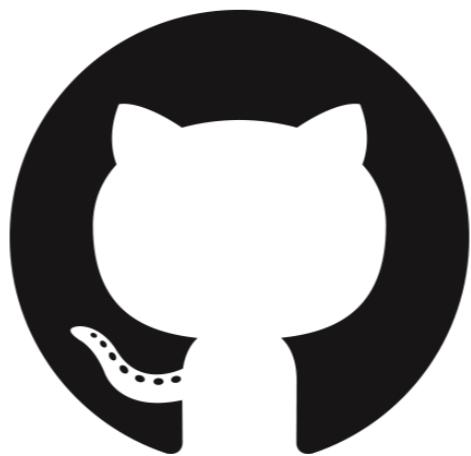
# Learn unit testing - with a data science spin

area (sq. ft.)	price (dollars)
2,081	314,942
1,059	186,606
	293,410
1,148	206,186
1,506	248,419
1,210	214,114
1,697	277,794
1,268	194,345
2,318	372,162
1,463	238,765
1,468	239,007



Linear regression of housing price against area

# GitHub repository of the course



- Implementation of functions like `row_to_list()`.

# Develop a complete unit test suite

```
data/  
src/  
|-- data/  
|-- features/  
|-- models/  
|-- visualization/
```

# Develop a complete unit test suite

```
data/  
src/  
|-- data/  
|-- features/  
|-- models/  
|-- visualization/  
tests/                      # Test suite  
|-- data/  
|-- features/  
|-- models/  
|-- visualization/
```

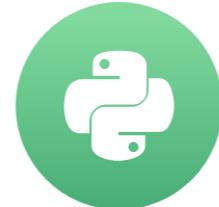
- Write unit tests for your own projects.

# Let's practice these concepts!

**UNIT TESTING FOR DATA SCIENCE IN PYTHON**

# Write a simple unit test using pytest

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# Testing on the console

```
row_to_list("2,081\t314,942\n")
```

```
[ "2,081", "314,942" ]
```

```
row_to_list("\t293,410\n")
```

```
None
```

```
row_to_list("1,463238,765\n")
```

```
None
```

- Unit tests improve this process.

# Python unit testing libraries

- pytest
- unittest
- nosetests
- doctest

We will use pytest!

- Has all essential features.
- Easiest to use.
- **Most popular.**



**pytest**

# Step 1: Create a file

- Create `test_row_to_list.py`.
- `test_` indicate unit tests inside (naming convention).
- Also called **test modules**.

# Step 2: Imports

Test module: `test_row_to_list.py`

```
import pytest  
import row_to_list
```

# Step 3: Unit tests are Python functions

Test module: test\_row\_to\_list.py

```
import pytest
import row_to_list

def test_for_clean_row():
```

# Step 3: Unit tests are Python functions

Test module: `test_row_to_list.py`

```
import pytest
import row_to_list

def test_for_clean_row():
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]

# Step 4: Assertion

Test module: `test_row_to_list.py`

```
import pytest
import row_to_list

def test_for_clean_row():
    assert ...
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]

# Theoretical structure of an assertion

```
assert boolean_expression
```

```
assert True
```

```
assert False
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

# Step 4: Assertion

Test module: `test_row_to_list.py`

```
import pytest
import row_to_list

def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == \
        ["2,081", "314,942"]
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	[ "2,081", "314,942" ]

# A second unit test

Test module: `test_row_to_list.py`

```
import pytest
import row_to_list

def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == \
        ["2,081", "314,942"]

def test_for_missing_area():
    assert row_to_list("\t293,410\n") is None
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	["2,081", "314,942"]
"\t293,410\n"	Invalid	None

# Checking for None values

Do this for checking if `var` is `None`.

```
assert var is None
```

Do *not* do this.

```
assert var == None
```

# A third unit test

Test module: `test_row_to_list.py`

```
import pytest
import row_to_list

def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == \
        ["2,081", "314,942"]

def test_for_missing_area():
    assert row_to_list("\t293,410\n") is None

def test_for_missing_tab():
    assert row_to_list("1,463238,765\n") is None
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	["2,081", "314,942"]
"\t293,410\n"	Invalid	None
"1,463238,765\n"	Invalid	None

# Step 5: Running unit tests

- Do this in the command line.

```
pytest test_row_to_list.py
```

# Running unit tests in DataCamp exercises

The screenshot shows a DataCamp exercise interface. On the left, there's a sidebar with the title "Running tests in the DataCamp exercise environment". Below it, a progress bar indicates "Instructions 1/2" and "50 XP". A bullet point says "Write tests.". The main area is titled "script.py" and contains the following Python code:

```
1 def test_for_clean_row():
2     assert row_to_list("2,081\t314,942") == ["2,081", "314,942"]
3
4 def test_for_missing_area():
5     assert row_to_list("\t293,410") is None
6
7 def test_for_missing_tab():
8     assert row_to_list("1,463238,765") is None
```

Below the code editor is an "IPython Shell" section with the prompt "In [1]: |". At the bottom right of the code editor are three buttons: a blue circular arrow icon, "Run Code", and "Submit Answer".

# Running unit tests in DataCamp exercises

The screenshot shows a DataCamp exercise interface. On the left, there's a sidebar with the title "Running tests in the DataCamp exercise environment". It includes a progress bar for "Instructions 2/2" (50 XP) and a "Submit Answer" button. A green box highlights the text: "Your tests were written to a test module `test_row_to_list.py`. Run the tests." Below this is a "Possible Answers" section. On the right, the main workspace shows a script named "script.py" with the following Python code:

```
1 def test_for_clean_row():
2     assert row_to_list("2,081\t314,942") == ["2,081", "314,942"]
3
4 def test_for_missing_area():
5     assert row_to_list("\t293,410") is None
6
7 def test_for_missing_tab():
8     assert row_to_list("1,463238,765") is None
```

Below the code editor is an "IPython Shell" tab with the prompt "In [1]: |". There are "Run Code" and "Reset" buttons next to the shell.

# Running unit tests in DataCamp exercises

The screenshot shows a DataCamp exercise interface. On the left, there's a sidebar with the title "Running tests in the DataCamp exercise environment", "Instructions 2/2", "50 XP", and a "Submit Answer" button. The main area has tabs for "script.py" and "IPython Shell". The "script.py" tab contains Python test code:

```
1 def test_for_clean_row():
2     assert row_to_list("2,081\t314,942") == ["2,081", "314,942"]
3
4 def test_for_missing_area():
5     assert row_to_list("\t293,410") is None
6
7 def test_for_missing_tab():
8     assert row_to_list("1,463238,765") is None
```

The "IPython Shell" tab shows the command "In [1]: !pytest test\_row\_to\_list.py" entered by the user.

# Running unit tests in DataCamp exercises

The screenshot shows a DataCamp exercise interface. On the left, there's a sidebar with the title "Running tests in the DataCamp exercise environment", "Instructions 2/2", "50 XP", and a "Submit Answer" button. The main area has tabs for "script.py" and "IPython Shell". The "script.py" tab shows Python test code:

```
1 def test_for_clean_row():
2     assert row_to_list("2,081\t314,942") == ["2,081", "314,942"]
3
4 def test_for_missing_area():
5     assert row_to_list("\t293,410") is None
6
7 def test_for_missing_tab():
8     assert row_to_list("1,463238,765") is None
```

The "IPython Shell" tab shows the command "In [1]: !pytest test\_row\_to\_list.py" with a green arrow pointing to the exclamation mark. A note below says "Notice the exclamation mark before the command."

# Next lesson: test result report

The screenshot shows a DataCamp exercise interface for a course on unit testing in Python. The left sidebar displays the title "Running tests in the DataCamp exercise environment" and "Instructions 2/2" with 50 XP available. A "Submit Answer" button is visible. The main area shows a script named "script.py" with the following code:

```
1 def test_for_clean_row():
2     assert row_to_list("2,081\t314,942") == ["2,081", "314,942"]
3
4 def test_for_missing_area():
5     assert row_to_list("\t293,410") is None
6
7 def test_for_missing_tab():
8     assert row_to_list("1,463238,765") is None
```

To the right of the code editor is an IPython Shell window containing the output of a pytest run:

```
In [1]: !pytest test_row_to_list.py
=====
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.8.0, pluggy-0.9
.0
rootdir: /tmp/tmpf4bvxdw_, infile:
plugins: mock-1.10.0
collecting
```

# Let's write some unit tests!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

# Understanding test result report

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# Unit tests for row\_to\_list()

Test module: test\_row\_to\_list.py

```
import pytest
import row_to_list

def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == \
        ["2,081", "314,942"]

def test_for_missing_area():
    assert row_to_list("\t293,410\n") is None

def test_for_missing_tab():
    assert row_to_list("1,463238,765\n") is None
```

Argument	Type	Return value
"2,081\t314,942\n"	Valid	["2,081", "314,942"]
"\t293,410\n"	Invalid	None
"1,463238,765\n"	Invalid	None

# Test result report

```
!pytest test_row_to_list.py
```

```
===== test session starts =====
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.8.0, pluggy-0.9.0
rootdir: /tmp/tmpvdblq9g7, inifile:
plugins: mock-1.10.0
collecting ...
collected 3 items

test_row_to_list.py .F. [100%]

===== FAILURES =====
----- test_for_missing_area -----
def test_for_missing_area():
```

# Section 1: general information

```
===== test session starts =====  
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.8.0, pluggy-0.9.0  
rootdir: /tmp/tmpvdblq9g7, infile:  
plugins: mock-1.10.0
```

# Section 2: Test result

```
collecting ...
collected 3 items

test_row_to_list.py .F.
```

[100%]

# Section 2: Test result

```
collecting ...  
collected 3 items
```

```
test_row_to_list.py .F. [100%]
```

Character	Meaning	When	Action
F	Failure	An exception is raised when running unit test.	Fix the function or unit test.

# Section 2: Test result

```
collecting ...
collected 3 items
```

```
test_row_to_list.py .F.
```

[100%]

Character	Meaning	When	Action
F	Failure	An exception is raised when running unit test.	Fix the function or unit test.

- assertion raises `AssertionError`

```
def test_for_missing_area():
    assert row_to_list("\t293,410") is None      # AssertionError from this line
```

# Section 2: Test result

```
collecting ...
collected 3 items
```

```
test_row_to_list.py .F.
```

[100%]

Character	Meaning	When	Action
F	Failure	An exception is raised when running unit test.	Fix the function or unit test.

- another exception

```
def test_for_missing_area():
    assert row_to_list("\t293,410") is None      # NameError from this line
```

# Section 2: Test result

```
collecting ...  
collected 3 items
```

```
test_row_to_list.py .F. [100%]
```

Character	Meaning	When	Action
F	Failure	An exception is raised when running unit test.	Fix the function or unit test.
.	Passed	No exception raised when running unit test	Everything is fine. Be happy!

# Section 3: Information on failed tests

```
===== FAILURES =====
----- test_for_missing_area -----
def test_for_missing_area():
>     assert row_to_list("\t293,410\n") is None
E     AssertionError: assert ['', '293,410'] is None
E     +  where ['', '293,410'] = row_to_list('\t293,410\n')

test_row_to_list.py:7: AssertionError
```

- The line raising the exception is marked by `>`.

```
>     assert row_to_list("\t293,410\n") is None
```

# Section 3: Information on failed tests

```
===== FAILURES =====
----- test_for_missing_area -----
def test_for_missing_area():
>     assert row_to_list("\t293,410\n") is None
E     AssertionError: assert ['', '293,410'] is None
E     +  where ['', '293,410'] = row_to_list('\t293,410\n')

test_row_to_list.py:7: AssertionError
```

- the exception is an `AssertionError`.

```
E     AssertionError: assert ['', '293,410'] is None
```

# Section 3: Information about failed tests

```
===== FAILURES =====
----- test_for_missing_area -----
def test_for_missing_area():
>     assert row_to_list("\t293,410\n") is None
E     AssertionError: assert ['', '293,410'] is None
E     +  where ['', '293,410'] = row_to_list('\t293,410\n')

test_row_to_list.py:7: AssertionError
```

- the line containing `where` displays return values.

```
E     +  where ['', '293,410'] = row_to_list('\t293,410\n')
```

# Section 4: Test result summary

```
===== 1 failed, 2 passed in 0.03 seconds =====
```

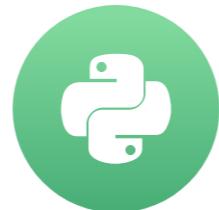
- Result summary from all unit tests that ran: **1 failed, 2 passed tests.**
- Total time for running tests: **0.03 seconds.**
  - Much faster than testing on the interpreter!

# Let's practice reading test result reports

UNIT TESTING FOR DATA SCIENCE IN PYTHON

# More benefits and test types

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# Unit tests serve as documentation

Test module: test\_row\_to\_list.py

```
import pytest
import row_to_list

def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == \
        [ "2,081", "314,942" ]

def test_for_missing_area():
    assert row_to_list("\t293,410\n") is None

def test_for_missing_tab():
    assert row_to_list("1,463238,765\n") is None
```

# Unit tests serve as documentation

Test module: `test_row_to_list.py`

```
import pytest
import row_to_list

def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == \
        [ "2,081", "314,942" ]

def test_for_missing_area():
    assert row_to_list("\t293,410\n") is None

def test_for_missing_tab():
    assert row_to_list("1,463238,765\n") is None
```

- Created from the test module

Argument	Return value
"2,081\t314,942\n"	[ "2,081", "314,942" ]
"\t293,410\n"	None
"1,463238,765\n"	None

# Guess function's purpose by reading unit tests

```
!cat test_row_to_list.py
```

The screenshot shows a DataCamp exercise interface. On the left, there is a sidebar with the DataCamp logo, course navigation buttons, and a progress bar. The main area has a title "Exercise" and a sub-section titled "Guess the function's purpose from the unit tests". Below this is a "Instructions" section with a "100 XP" badge. The central part of the interface features a code editor window titled "script.py" containing a single line of code: "1". To the right of the code editor is an "IPython Shell" window showing the command "In [1]: !cat test\_row\_to\_list.py". At the bottom of the interface are "Run Code" and "Submit Answer" buttons.

# Guess function's purpose by reading unit tests

```
!cat test_row_to_list.py
```

The screenshot shows a DataCamp exercise interface. On the left, there is a sidebar with the title "Guess the function's purpose from the unit tests". Below it are "Instructions" and "100 XP". The main area has tabs for "script.py" and "IPython Shell". In the "script.py" tab, the code is displayed:

```
script.py
1
Run Code Submit Answer

IPython Shell
In [1]: !cat test_row_to_list.py

import pytest
import row_to_list

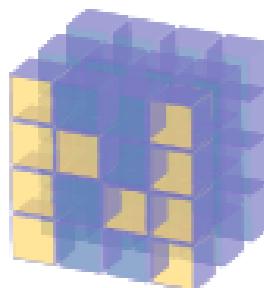
def test_for_clean_row():
    assert row_to_list("2,081\t314,942\n") == ["2,081", "314,942"]

def test_for_missing_area():
    assert row_to_list("\t293,410\n") == None

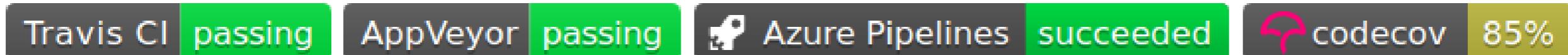
def test_for_missing_tab():
    assert row_to_list("1,463238,765\n") == None
```

# More trust

- Users can run tests and verify that the package works.



NumPy

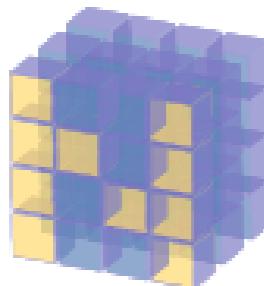


NumPy is the fundamental package needed for scientific computing with Python.

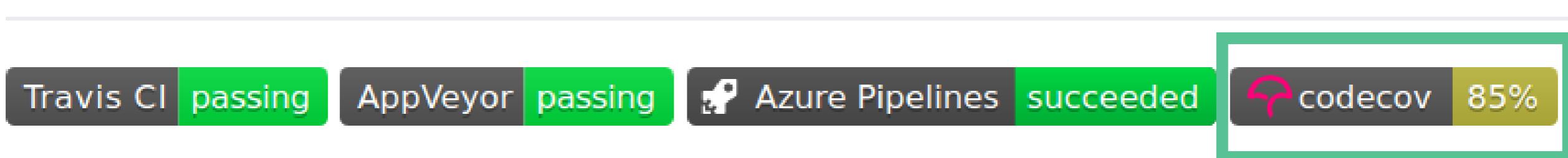
- **Website (including documentation):** <https://www.numpy.org>

# More trust

- Users can run tests and verify that the package works.



NumPy



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

- **Website (including documentation):** <https://www.numpy.org>

# More trust

- Users can run tests and verify that the package works.



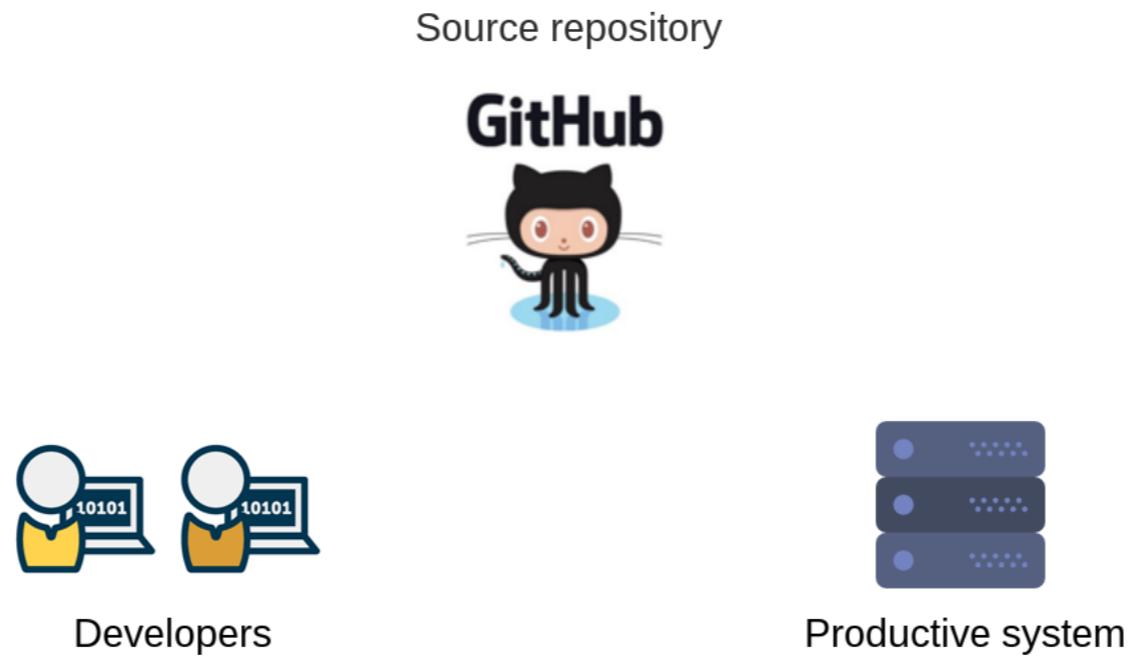
NumPy



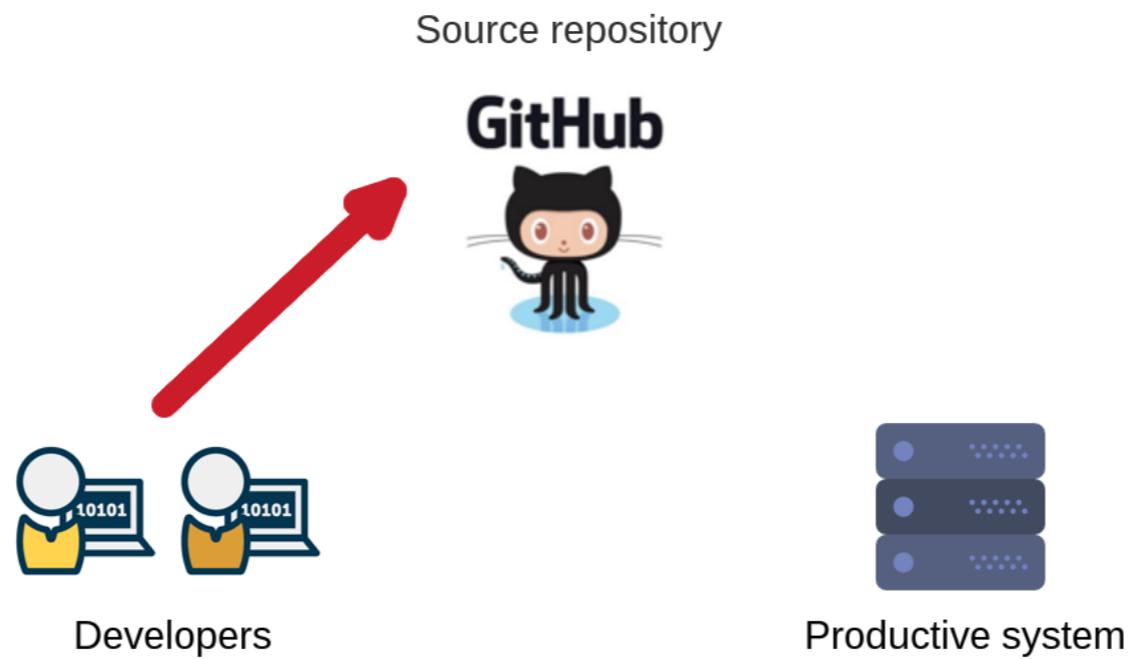
NumPy is the fundamental package needed for scientific computing with Python.

- **Website (including documentation):** <https://www.numpy.org>

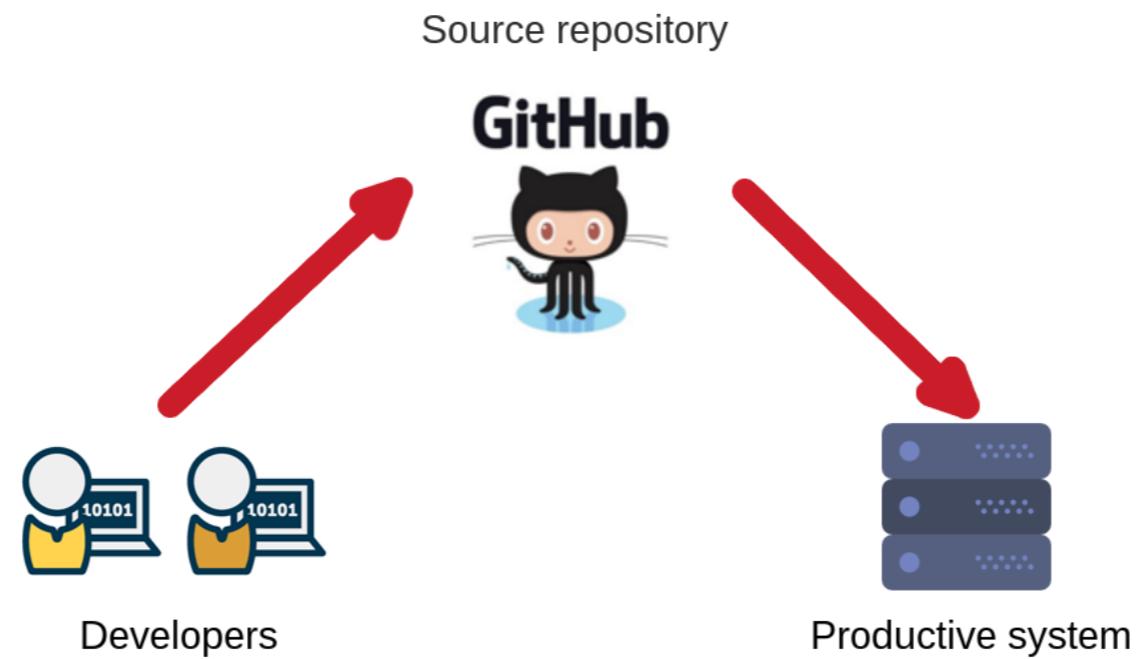
# Reduced downtime



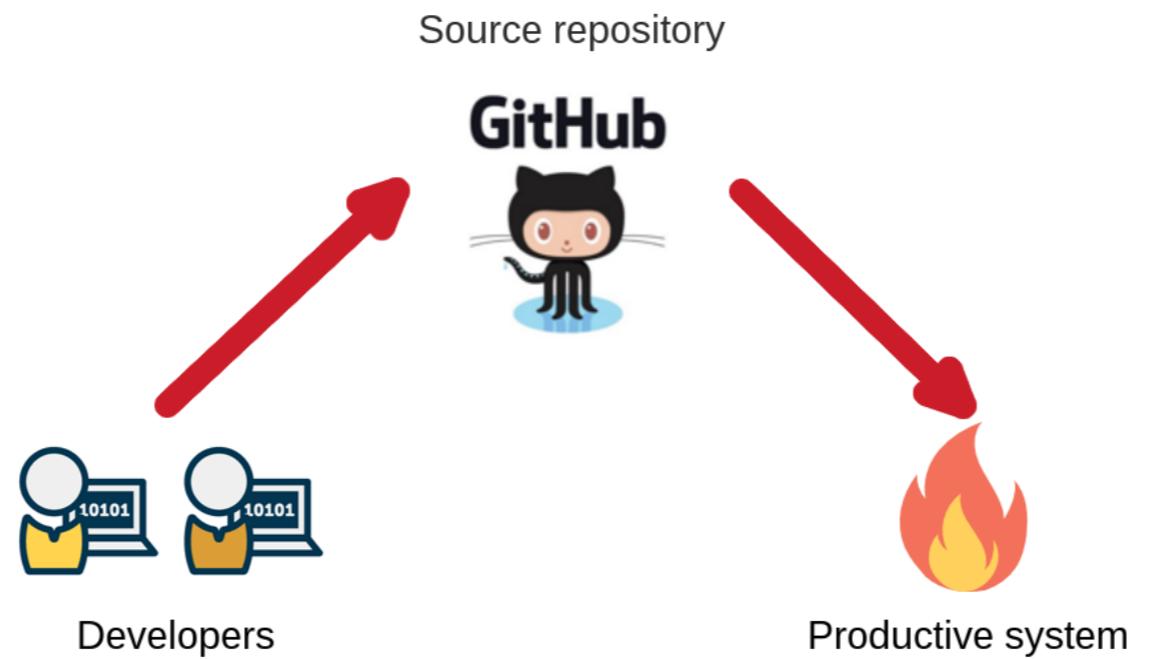
# Reduced downtime



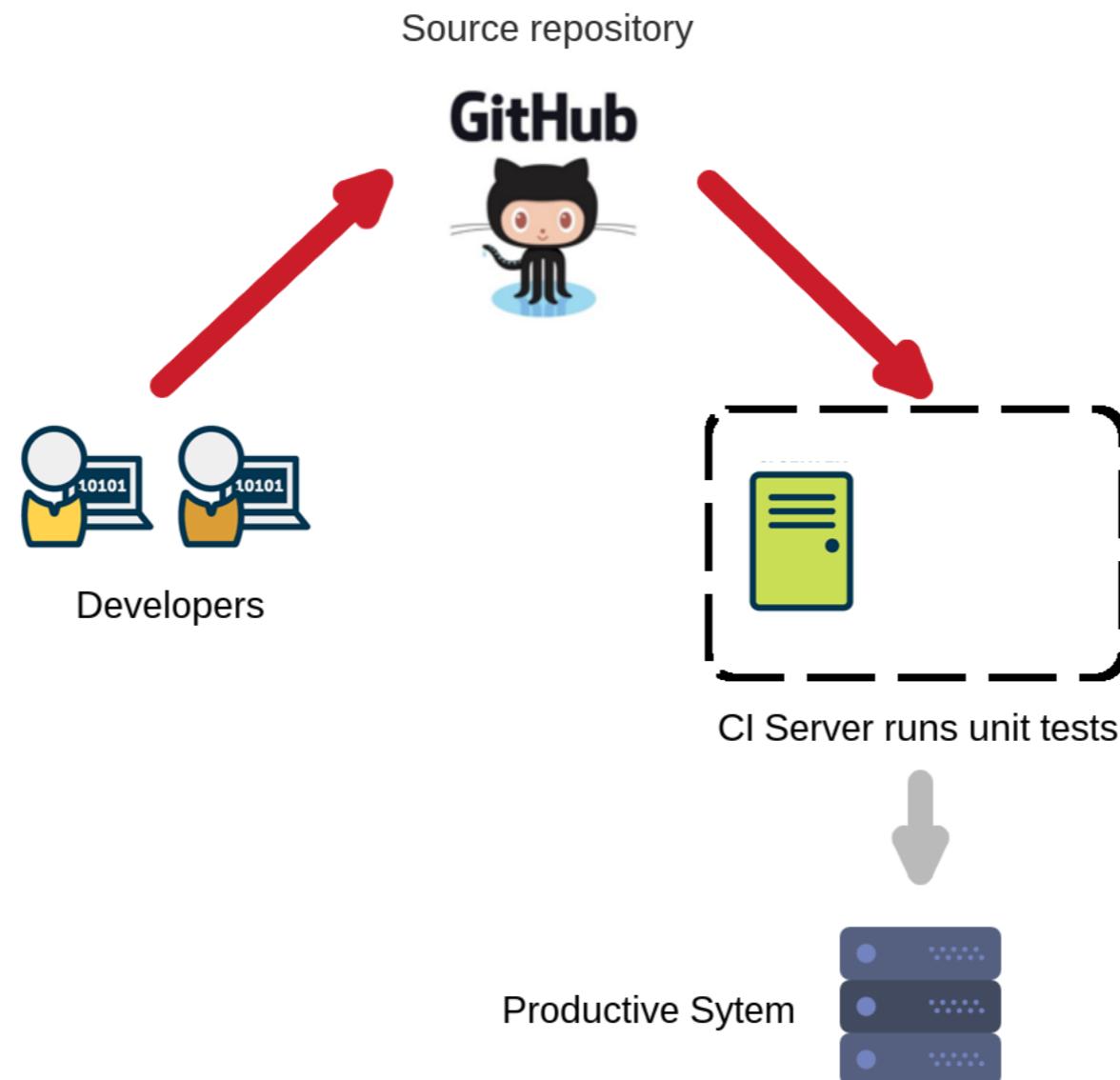
# Reduced downtime



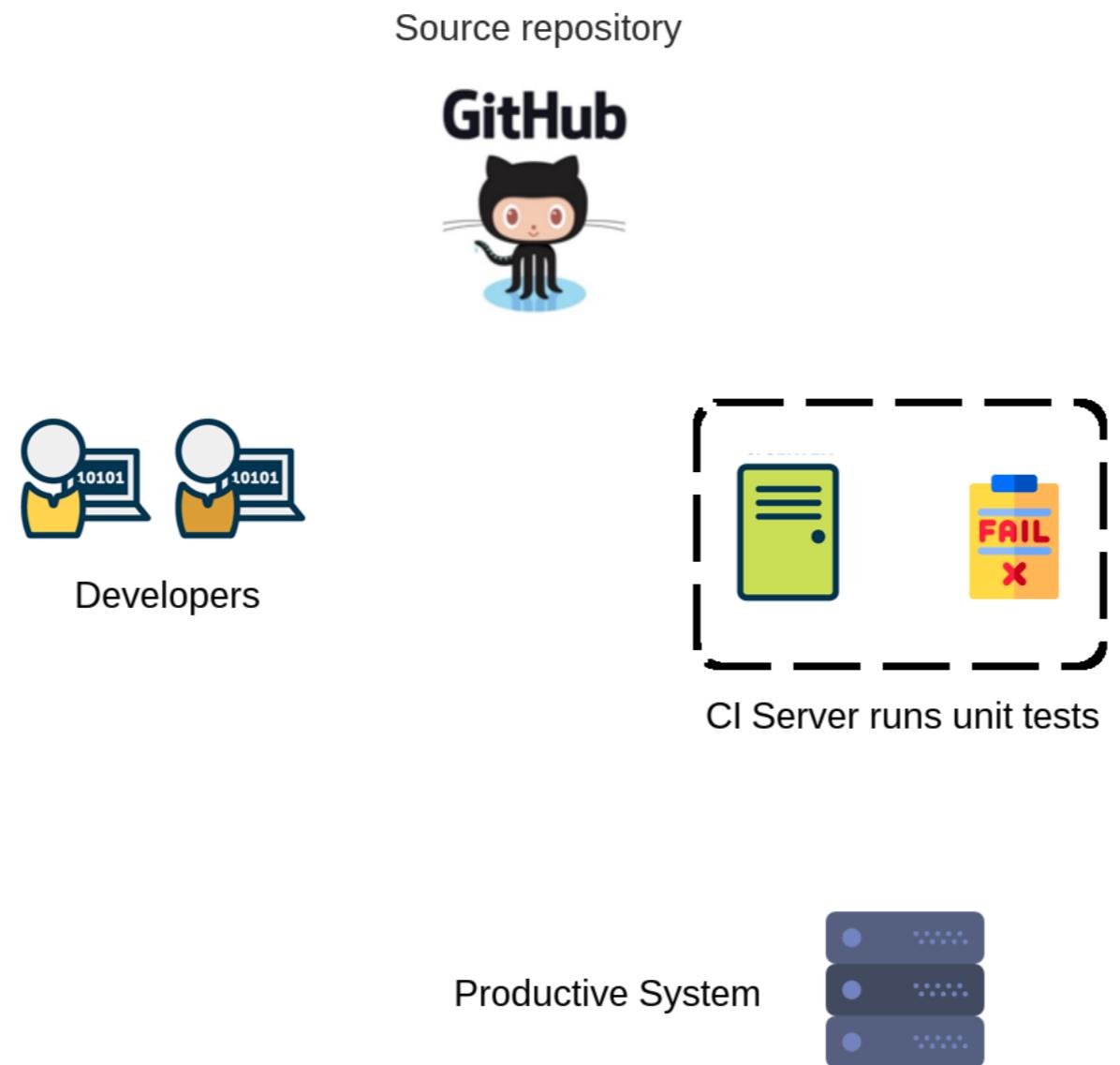
# Reduced downtime



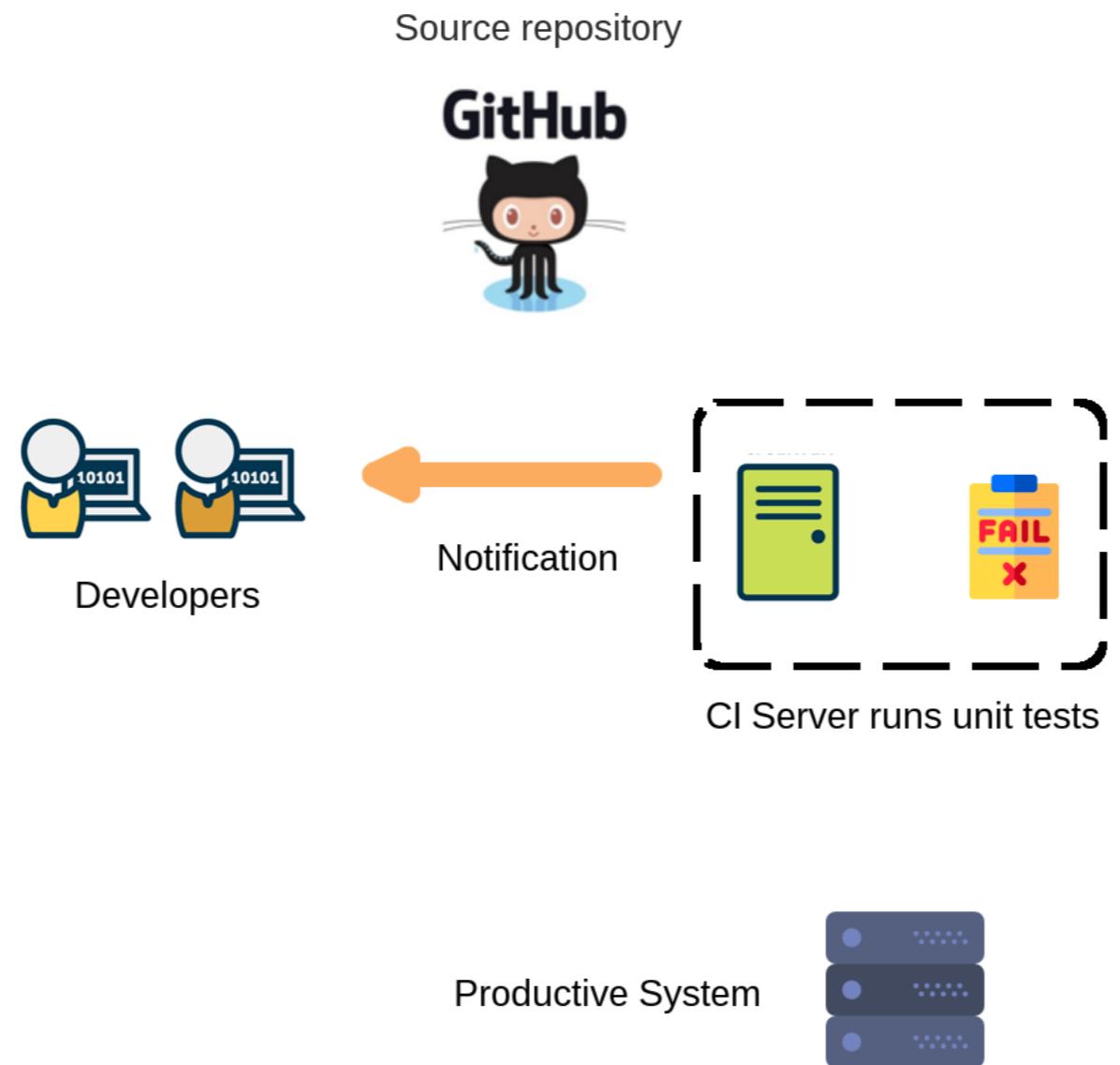
# Reduced downtime



# Reduced downtime



# Reduced downtime



# All benefits

- Time savings.
- Improved documentation.
- More trust.
- Reduced downtime.

# Tests we already wrote

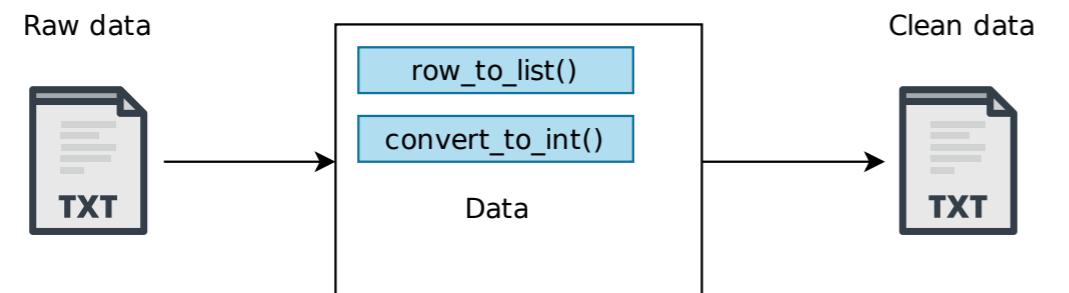
row\_to\_list()

# Tests we already wrote

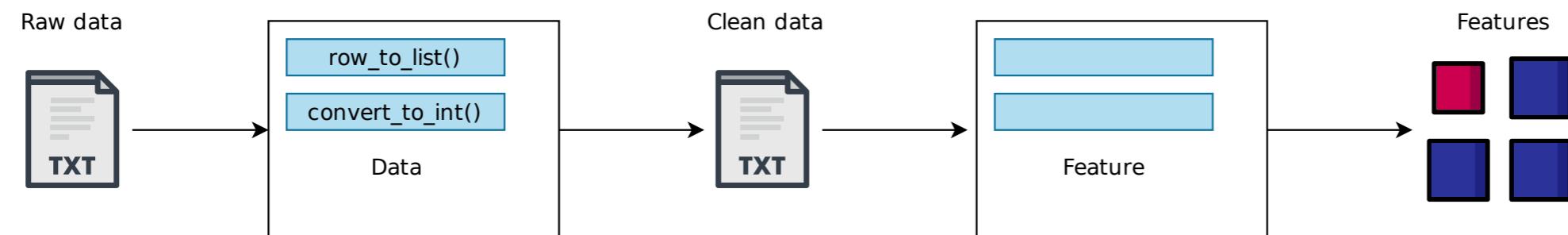
`row_to_list()`

`convert_to_int()`

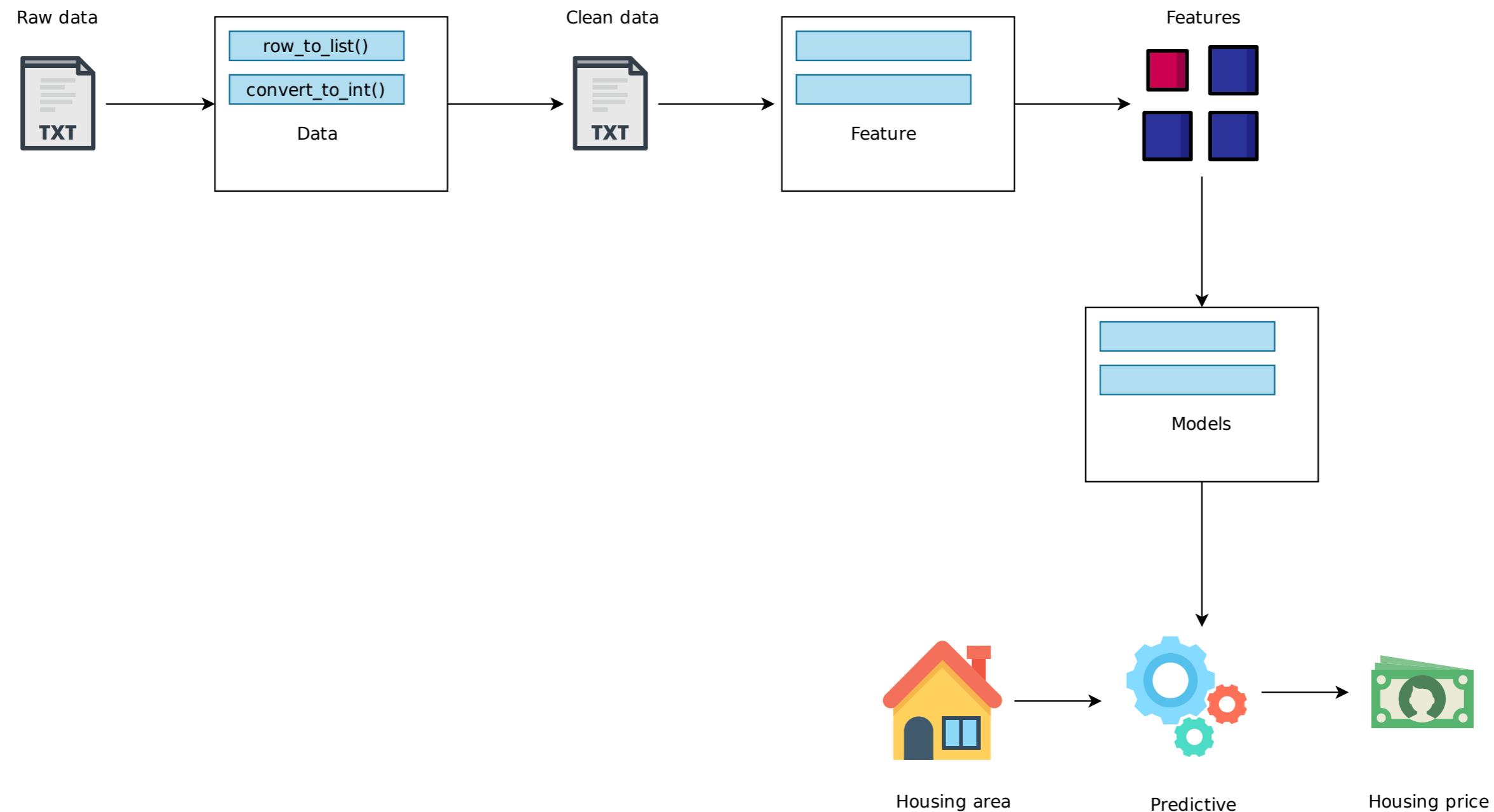
# Data module



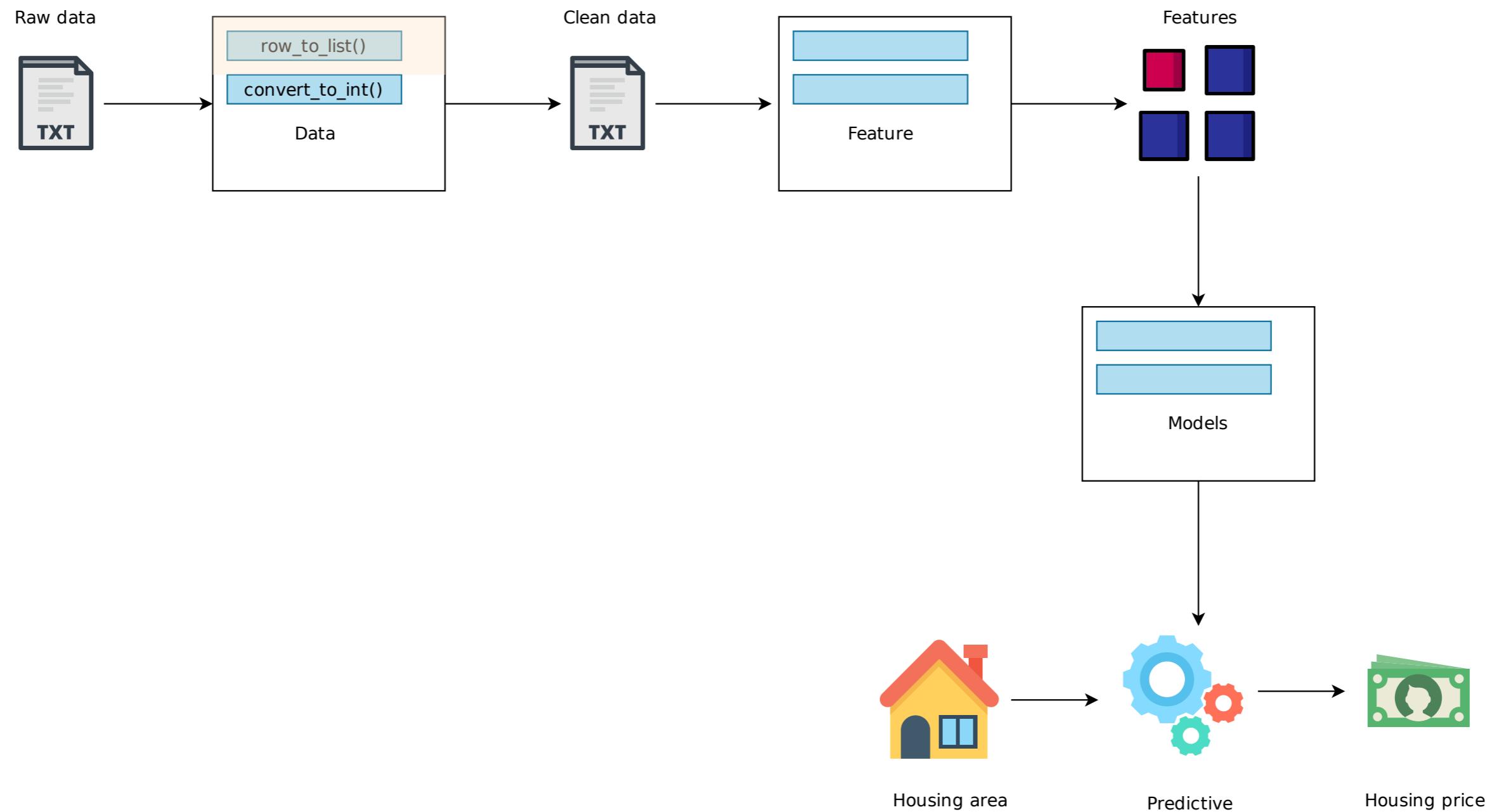
# Feature module



# Models module



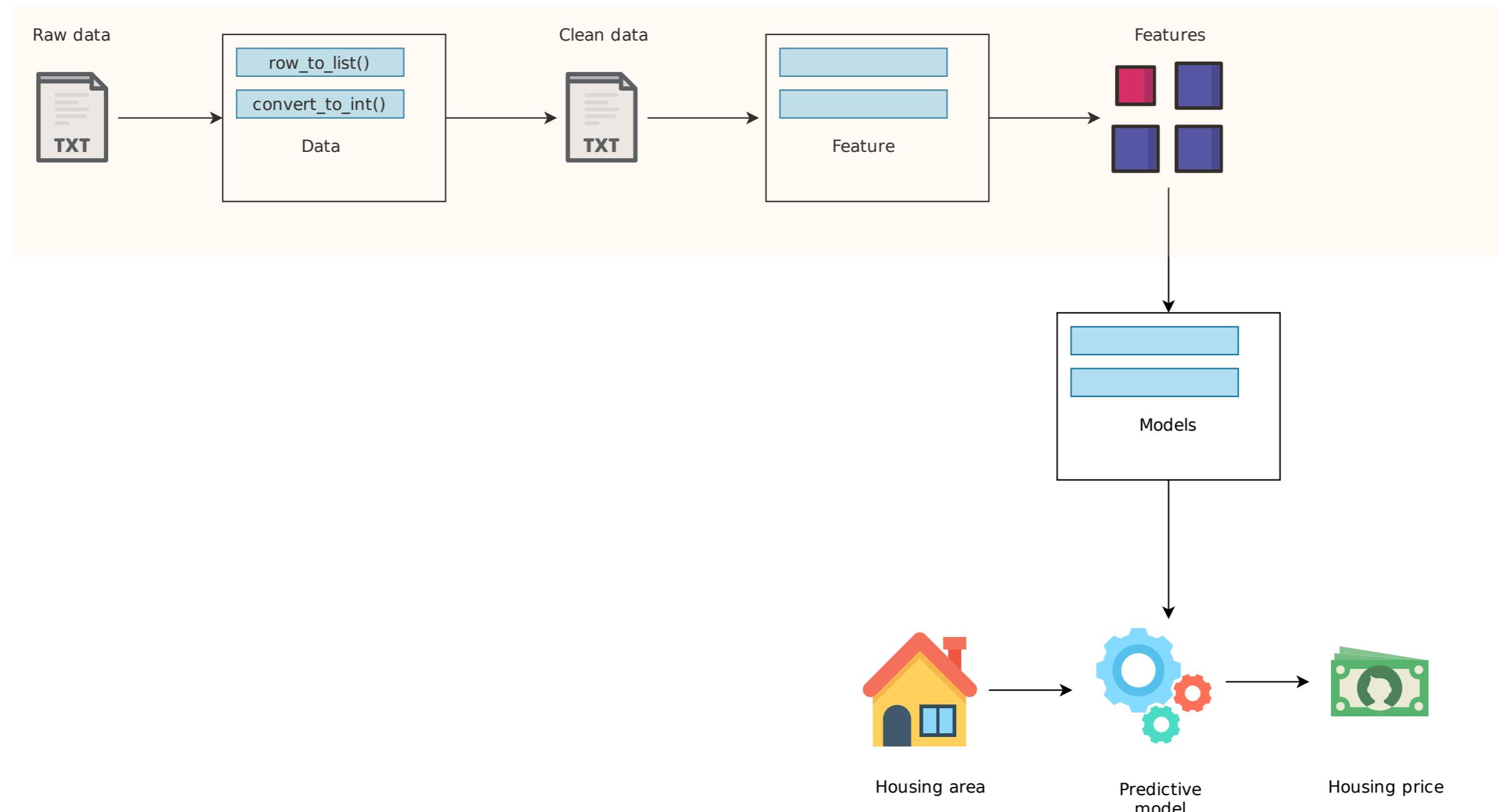
# Unit test



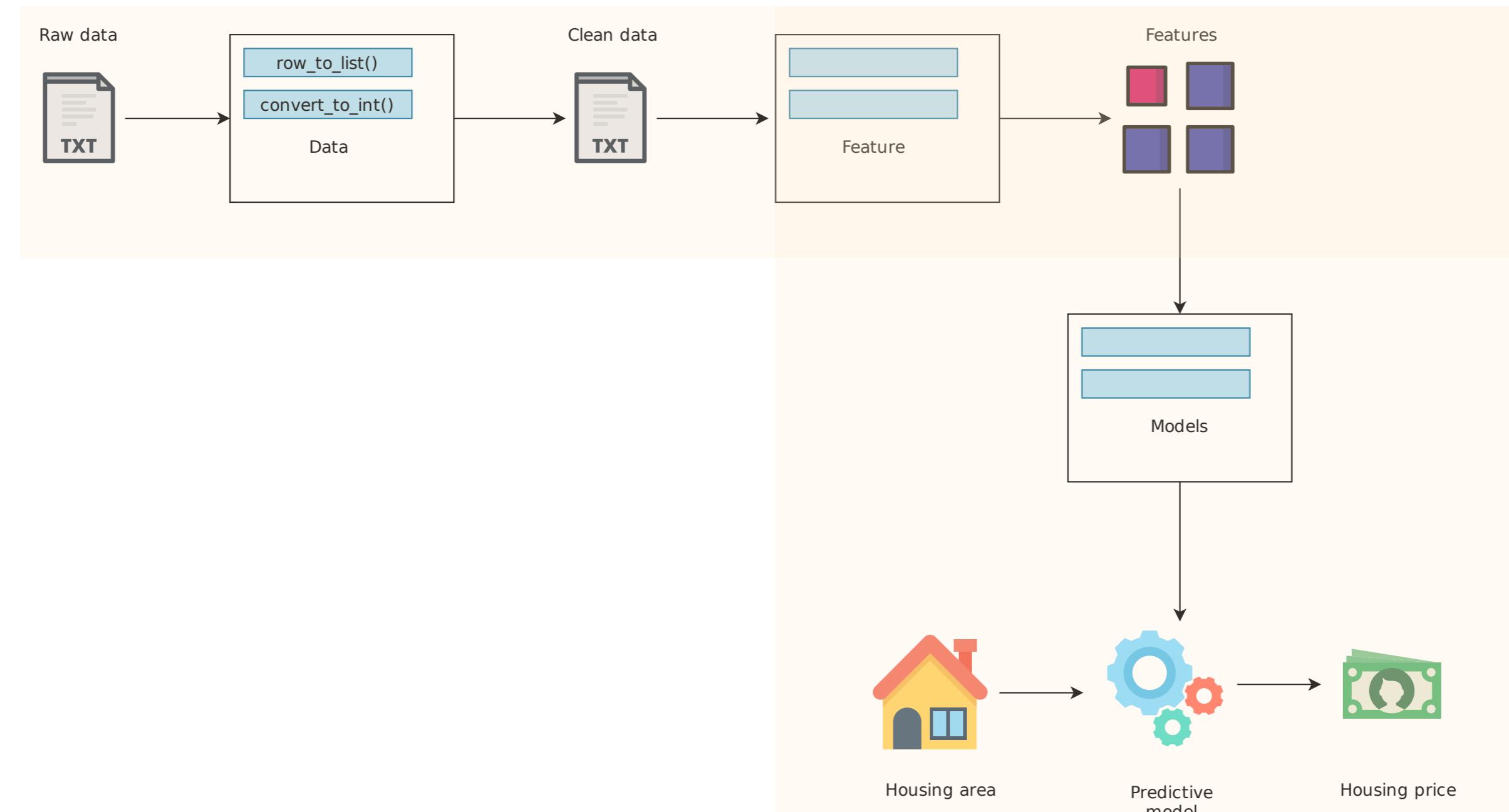
# What is a unit?

- Small, independent piece of code.
- Python function or class.

# Integration test



# End to end test



# This course focuses on unit tests

- Writing unit tests is the best way to learn pytest.

# In Chapter 2...

- Learn more pytest.
- Write more advanced unit tests.
- Work with functions in the `features` and `models` modules.



# Let's practice these concepts!

**UNIT TESTING FOR DATA SCIENCE IN PYTHON**

# Mastering assert statements

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# Theoretical structure of an assertion

```
assert boolean_expression
```

# The optional message argument

```
assert boolean_expression, message
```

```
assert 1 == 2, "One is not equal to two!"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: One is not equal to two!
```

```
assert 1 == 1, "This will not be printed since assertion passes"
```

# Adding a message to a unit test

- test module: `test_row_to_list.py`

```
import pytest  
...  
  
def test_for_missing_area():  
    assert row_to_list("\t293,410\n") is None
```

# Adding a message to a unit test

- test module: `test_row_to_list.py`

```
import pytest  
...  
  
def test_for_missing_area():  
    assert row_to_list("\t293,410\n") is None
```

- test module: `test_row_to_list.py`

```
import pytest  
...  
  
def test_for_missing_area_with_message():  
    actual = row_to_list("\t293,410\n")  
    expected = None  
  
    message = ("row_to_list('\t293,410\n') "  
              "returned {0} instead "  
              "of {1}").format(actual, expected)  
  
    assert actual is expected, message
```

# Test result report with message

- `test_on_missing_area()` output on failure

```
E      AssertionError: assert ['', '293,410'] is None
E      +  where ['', '293,410'] = row_to_list('\t293,410\n')
```

- `test_on_missing_area_with_message()` output on failure

```
>      assert actual is expected, message
E      AssertionError: row_to_list('\t293,410\n') returned ['', '293,410'] instead
          of None
E      assert ['', '293,410'] is None
```

# Recommendations

- Include a message with assert statements.
- Print values of any variable that is relevant to debugging.

# Beware of float return values!

```
0.1 + 0.1 + 0.1 == 0.3
```

False



# Beware of float return values!

```
0.1 + 0.1 + 0.1
```

```
0.3000000000000004
```

# Don't do this

```
assert 0.1 + 0.1 + 0.1 == 0.3, "Usual way to compare does not always work with floats!"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Usual way to compare does not always work with floats!
```

# Do this

- Use `pytest.approx()` to wrap expected return value.

```
assert 0.1 + 0.1 + 0.1 == pytest.approx(0.3)
```

# NumPy arrays containing floats

```
assert np.array([0.1 + 0.1, 0.1 + 0.1 + 0.1]) == pytest.approx(np.array([0.2, 0.3]))
```

# Multiple assertions in one unit test

```
convert_to_int("2,081")
```

```
2081
```

# Multiple assertions in one unit test

- test module: test\_convert\_to\_int.py

```
import pytest  
...  
  
def test_on_string_with_one_comma():  
    assert convert_to_int("2,081") == 2081
```

- test\_module: test\_convert\_to\_int.py

```
import pytest  
...  
  
def test_on_string_with_one_comma():  
    return_value = convert_to_int("2,081")  
    assert isinstance(return_value, int)  
    assert return_value == 2081
```

- Test will pass only if both assertions pass.

# Let's practice writing assert statements!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

# Testing for exceptions instead of return values

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty  
Test Automation Engineer



# Example

```
import numpy as np  
example_argument = np.array([[2081, 314942],  
                           [1059, 186606],  
                           [1148, 206186],  
                           ]  
                           )  
split_into_training_and_testing_sets(example_argument)
```

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

# Example

```
import numpy as np  
example_argument = np.array([[2081, 314942],      # must be two dimensional  
                           [1059, 186606],  
                           [1148, 206186],  
                           ]  
                           )  
split_into_training_and_testing_sets(example_argument)
```

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

# Example

```
import numpy as np  
example_argument = np.array([2081, 314942, 1059, 186606, 1148, 206186])      # one dimensional  
split_into_training_and_testing_sets(example_argument)
```

ValueError: Argument data array must be two dimensional. Got 1 dimensional array instead!

# Unit testing exceptions

## Goal

Test if `split_into_training_and_testing_set()` raises `ValueError` with one dimensional argument.

```
def test_valueerror_on_one_dimensional_argument():
    example_argument = np.array([2081, 314942, 1059, 186606, 1148, 206186])
    with pytest.raises(ValueError):
```

# Theoretical structure of a with statement

```
with ----:  
    print("This is part of the context")    # any code inside is the context
```

# Theoretical structure of a with statement

```
with context_manager:  
    print("This is part of the context")    # any code inside is the context
```

# Theoretical structure of a with statement

```
with context_manager:  
    # <--- Runs code on entering context  
    print("This is part of the context")    # any code inside is the context  
    # <--- Runs code on exiting context
```



# Theoretical structure of a with statement

```
with pytest.raises(ValueError):  
    # <--- Does nothing on entering the context  
    print("This is part of the context")  
    # <--- If context raised ValueError, silence it.  
    # <--- If the context did not raise ValueError, raise an exception.
```

# Theoretical structure of a with statement

```
with pytest.raises(ValueError):  
    raise ValueError      # context exits with ValueError  
    # <--- pytest.raises(ValueError) silences it
```

```
with pytest.raises(ValueError):  
    pass      # context exits without raising a ValueError  
    # <--- pytest.raises(ValueError) raises Failed
```

```
Failed: DID NOT RAISE <class 'ValueError'>
```

# Unit testing exceptions

```
def test_valueerror_on_one_dimensional_argument():
    example_argument = np.array([2081, 314942, 1059, 186606, 1148, 206186])
    with pytest.raises(ValueError):
        split_into_training_and_testing_sets(example_argument)
```

- If function raises expected `ValueError`, test will pass.
- If function is buggy and does not raise `ValueError`, test will fail.

# Testing the error message

```
ValueError: Argument data array must be two dimensional. Got 1 dimensional array instead!
```

# Testing the error message

```
def test_valueerror_on_one_dimensional_argument():
    example_argument = np.array([2081, 314942, 1059, 186606, 1148, 206186])
    with pytest.raises(ValueError) as exception_info:      # store the exception
        split_into_training_and_testing_sets(example_argument)

    # Check if ValueError contains correct message
    assert exception_info.match("Argument data array must be two dimensional.
                                  "Got 1 dimensional array instead!
                                  )
```

- `exception_info` stores the `ValueError`.
- `exception_info.match(expected_msg)` checks if `expected_msg` is present in the actual error message.

**Let's practice unit  
testing exceptions.**

**UNIT TESTING FOR DATA SCIENCE IN PYTHON**

# The well tested function

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# Example

```
import numpy as np  
example_argument_value = np.array([[2081, 314942],  
                                  [1059, 186606],  
                                  [1148, 206186],  
                                  ]  
                                )  
split_into_training_and_testing_sets(example_argument_value)
```

```
(array([[1148, 206186],      # Training array  
       [2081, 314942],  
       ]  
     ),  
 array([[1059, 186606]])  # Testing array  
 )
```

# Test for length, not value

```
import numpy as np  
  
example_argument_value = np.array([[2081, 314942],  
                                  [1059, 186606],  
                                  [1148, 206186],  
                                  ]  
                                )  
  
split_into_training_and_testing_sets(example_argument_value)
```

```
(array([[1148, 206186],      # Training array has int(0.75 * example_argument_value.shape[0]) rows  
       [2081, 314942],  
       ]  
     ),  
 array([[1059, 186606]])  # Rest of the rows go to the testing array  
 )
```

# Test arguments and expected return values

Number of rows (argument)	Number of rows (training array)	Number of rows (testing array)
8	$\text{int}(0.75 * 8) = 6$	$8 - \text{int}(0.75 * 8) = 2$

# Test arguments and expected return values

Number of rows (argument)	Number of rows (training array)	Number of rows (testing array)
8	$\text{int}(0.75 * 8) = 6$	$8 - \text{int}(0.75 * 8) = 2$
10	$\text{int}(0.75 * 10) = 7$	$10 - \text{int}(0.75 * 10) = 3$

# Test arguments and expected return values

Number of rows (argument)	Number of rows (training array)	Number of rows (testing array)
8	$\text{int}(0.75 * 8) = 6$	$8 - \text{int}(0.75 * 8) = 2$
10	$\text{int}(0.75 * 10) = 7$	$10 - \text{int}(0.75 * 10) = 3$
23	$\text{int}(0.75 * 23) = 17$	$23 - \text{int}(0.75 * 23) = 6$

# How many arguments to test?

Input array number of rows	Training array number of rows	Testing array number of rows
8	$\text{int}(0.75 * 8) = 6$	$8 - \text{int}(0.75 * 8) = 2$
10	$\text{int}(0.75 * 10) = 7$	$10 - \text{int}(0.75 * 10) = 3$
23	$\text{int}(0.75 * 23) = 17$	$23 - \text{int}(0.75 * 23) = 6$
...	...	...
...	...	...
...	...	...

# Test argument types

Test for these argument types

- Bad arguments.
- Special arguments.
- Normal arguments.

# Test argument types

Test for these argument types

- Bad arguments. ✓
- Special arguments.
- Normal arguments.

# Test argument types

Test for these argument types

- Bad arguments. ✓
- Special arguments. ✓
- Normal arguments.

# Test argument types

Test for these argument types

- Bad arguments. ✓
- Special arguments. ✓
- Normal arguments. ✓

# The well tested function

Test for these argument types

- Bad arguments. ✓
- Special arguments. ✓
- Normal arguments. ✓



# Type I: Bad arguments

- When passed bad arguments, function raises an exception.

# Type I: Bad arguments (one dimensional array)

- When passed bad arguments, function raises an exception.

Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	ValueError

Example: `np.array([845.0, 31036.0, 1291.0, 72205.0])`

# Type I: Bad arguments (array with only one row)

- When passed bad arguments, function raises an exception.

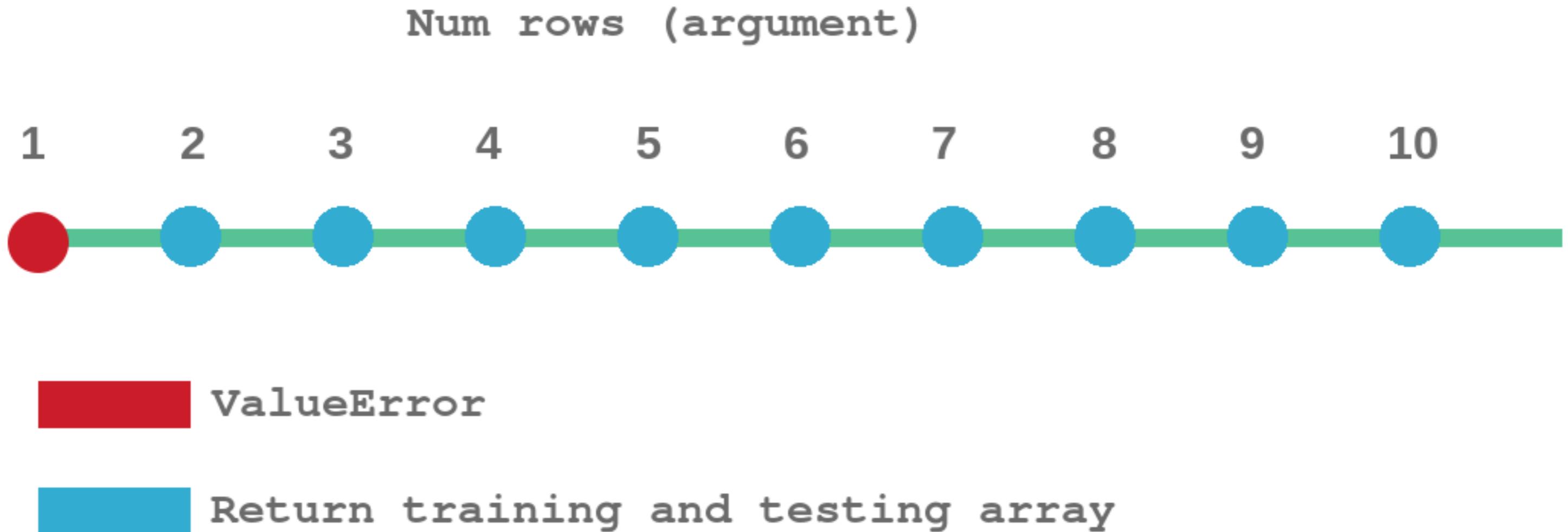
Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	ValueError
Contains 1 row	Bad	-	-	ValueError

Example: `np.array([[845.0, 31036.0]])`

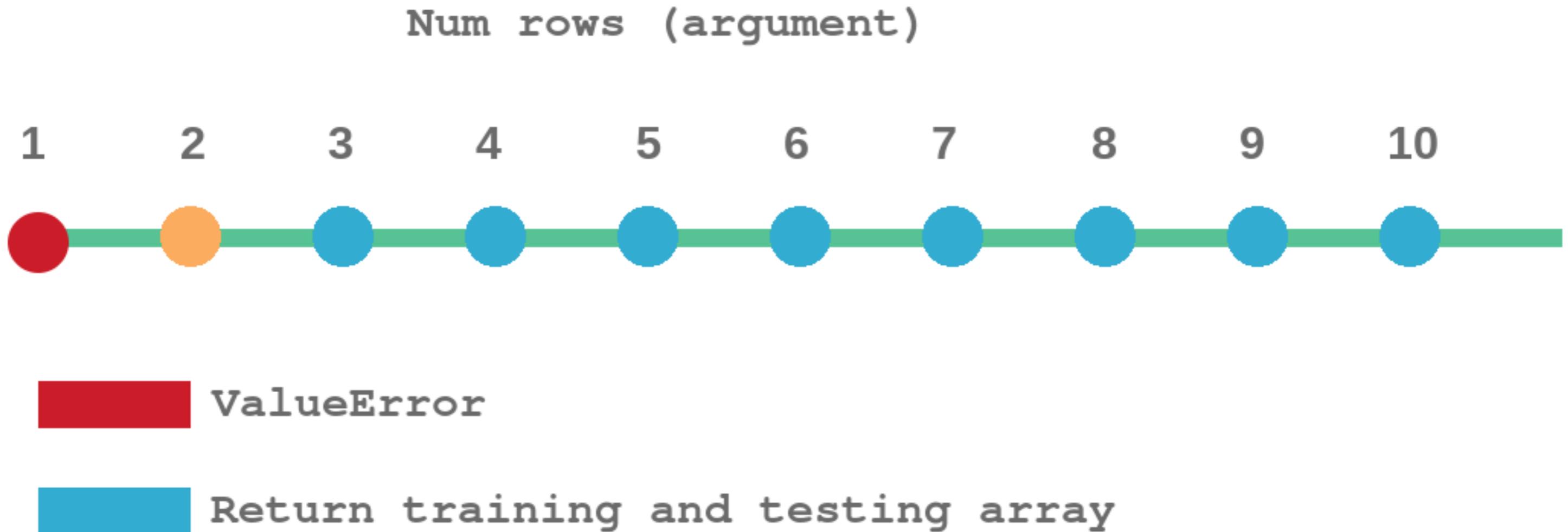
# Type II: Special arguments

- Boundary values.
- For some argument values, function uses special logic.

# Boundary values



# Boundary values



# Test arguments table

Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	ValueError
Contains 1 row	Bad	-	-	ValueError
Contains 2 rows	Special	$\text{int}(0.75 * 2) = 1$	$2 - \text{int}(0.75 * 2) = 1$	-

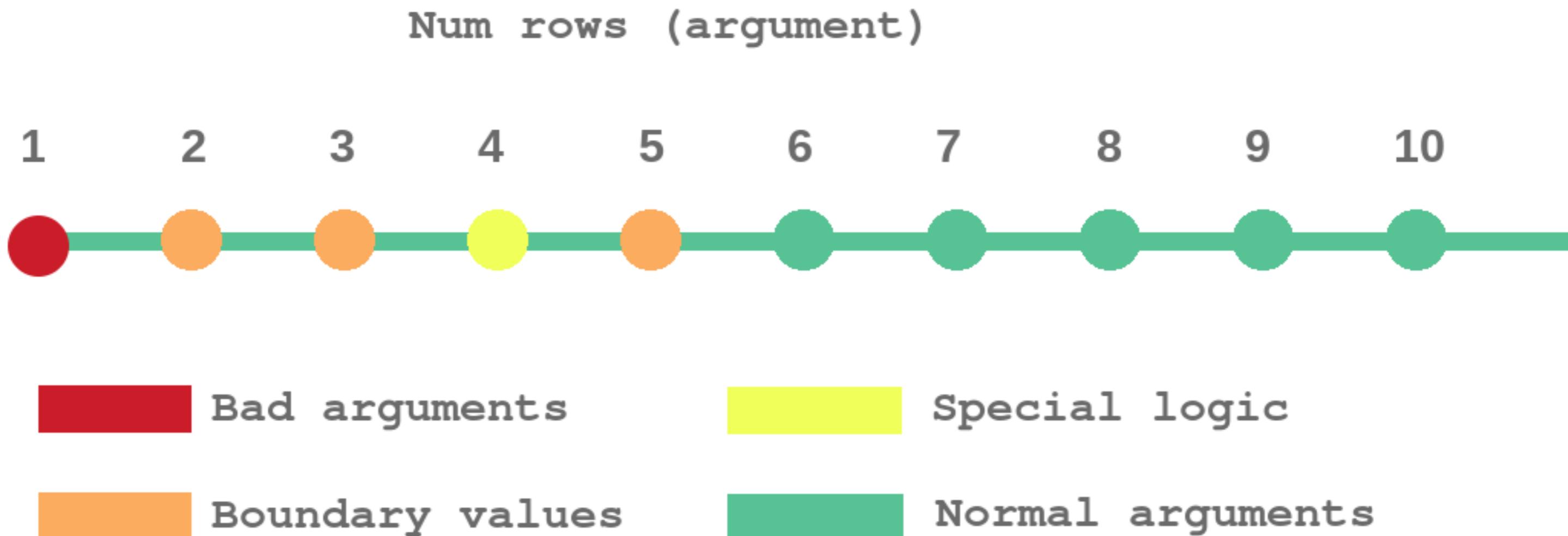
# Arguments triggering special logic

Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	ValueError
Contains 1 row	Bad	-	-	ValueError
Contains 2 rows	Special	$\text{int}(0.75 * 2) = 1$	$2 - \text{int}(0.75 * 2) = 1$	-
Contains 4 rows		$\text{int}(0.75 * 4) = 3$	$4 - \text{int}(0.75 * 4) = 1$	-

# Arguments triggering special logic

Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	ValueError
Contains 1 row	Bad	-	-	ValueError
Contains 2 rows	Special	$\text{int}(0.75 * 2) = 1$	$2 - \text{int}(0.75 * 2) = 1$	-
Contains 4 rows	Special	32	±2	-

# Normal arguments



Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	ValueError
Contains 1 row	Bad	-	-	ValueError
Contains 2 rows	Special	<code>int(0.75 * 2) = 1</code>	<code>2 - int(0.75 * 2) = 1</code>	-
Contains 3 rows	Special	<code>int(0.75 * 3) = 2</code>	<code>3 - int(0.75 * 3) = 1</code>	-
Contains 4 rows	Special	<code>32</code>	<code>42</code>	-
Contains 5 rows	Special	<code>int(0.75 * 5) = 3</code>	<code>5 - int(0.75 * 5) = 2</code>	-
Contains 6 rows	Normal	<code>int(0.75 * 6) = 4</code>	<code>6 - int(0.75 * 6) = 2</code>	-
Contains 8 rows	Normal	<code>int(0.75 * 8) = 6</code>	<code>8 - int(0.75 * 6) = 2</code>	-

```
split_into_training_and_testing_sets()
```



# Caveat

- Not all functions have bad or special arguments.
  - In this case, simply ignore these class of arguments.

**Let's apply this to  
other functions!**

**UNIT TESTING FOR DATA SCIENCE IN PYTHON**

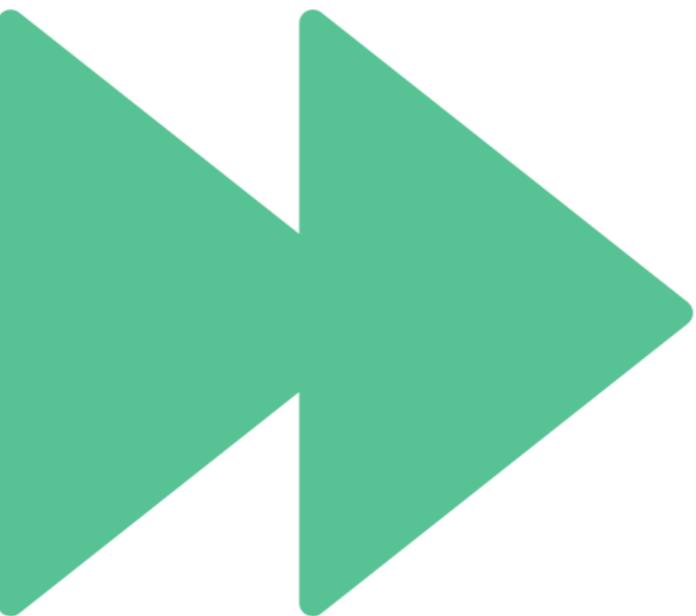
# Test Driven Development (TDD)

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

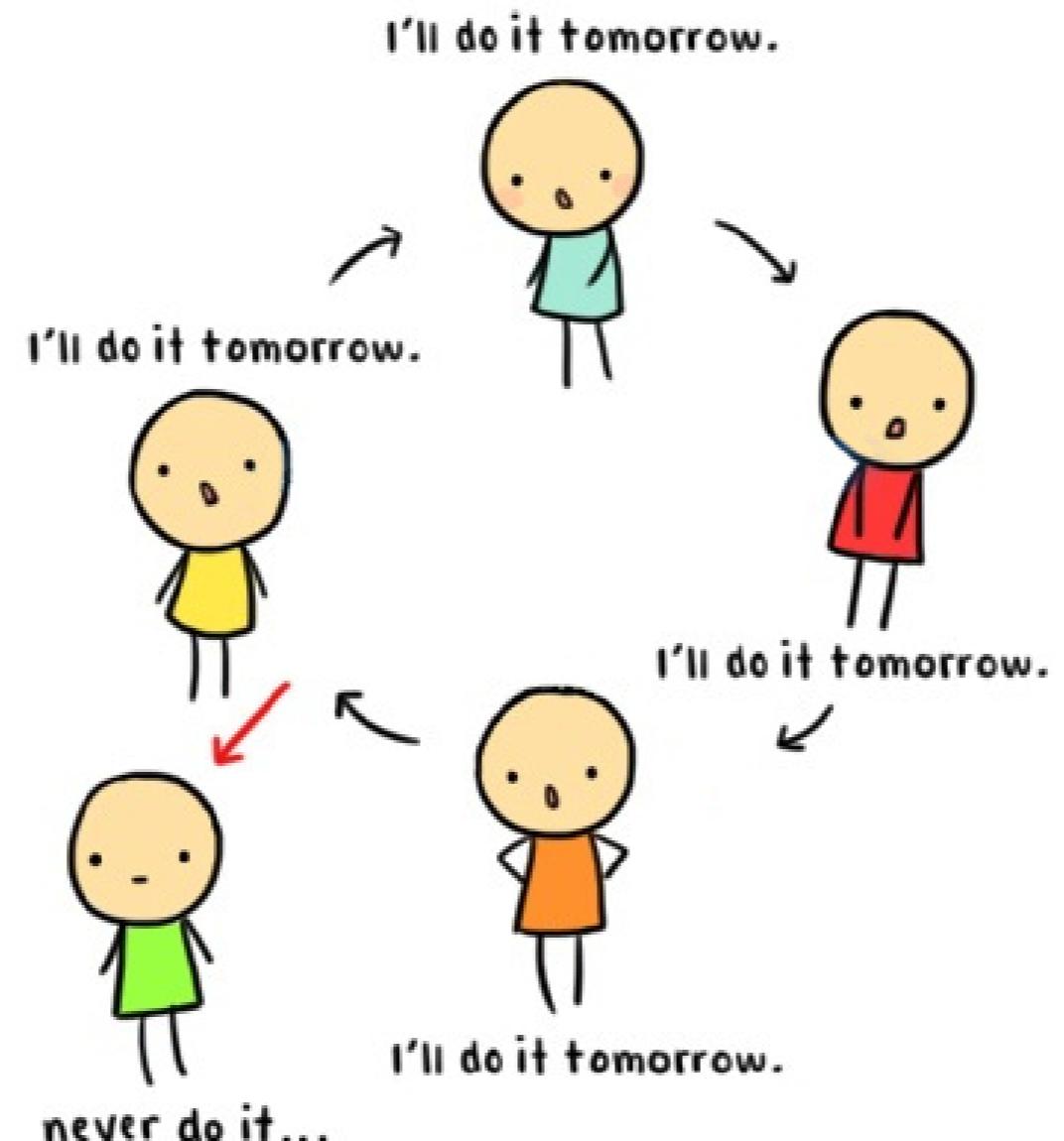
# Writing unit tests is often skipped



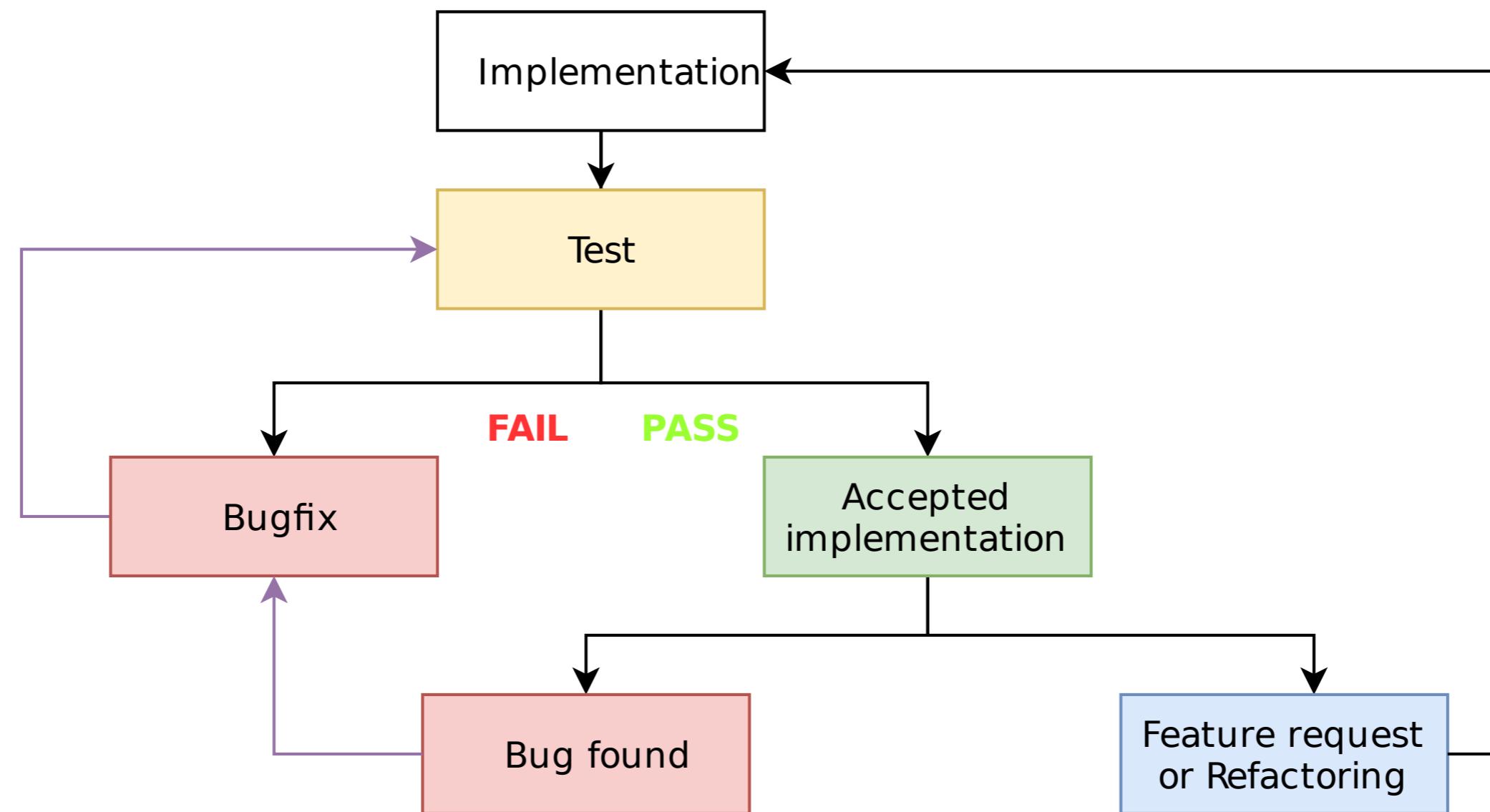
# Usual priorities in the industry

1. Feature development.
2. Unit testing.

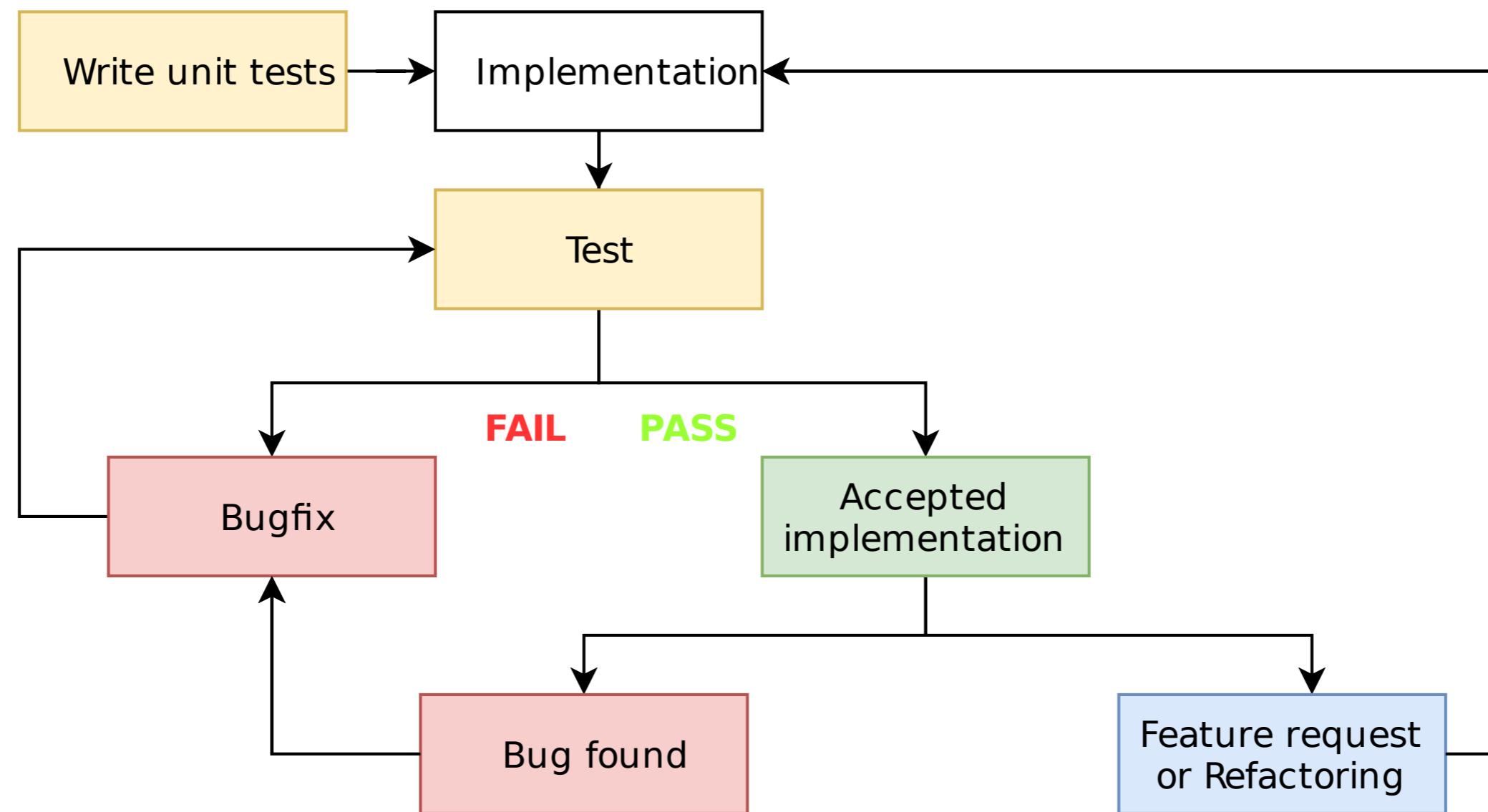
# Unit tests never get written



# Test Driven Development (TDD)



# Test Driven Development (TDD)



# Write unit tests before implementation!

- Unit tests *cannot* be deprioritized.
- Time for writing unit tests factored in implementation time.
- Requirements are clearer and implementation easier.



# In the coding exercises...

- We will use TDD to develop `convert_to_int()`.

```
convert_to_int("2,081")
```

```
2081
```

# Step 1: Write unit tests and fix requirements

Test module: `test_convert_to_int.py`

```
import pytest

def test_with_no_comma():
    ...

def test_with_one_comma():
    ...

def test_with_two_commas():
    ...
```

# Step 2: Run tests and watch it fail

```
!pytest test_convert_to_int.py
```

```
===== test session starts =====
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.8.0, pluggy-0.11.0
rootdir: /tmp/tmpbhadho_b, configparser:
plugins: mock-1.10.0
collecting ...
collected 6 items

test_convert_to_int.py FFFFFFFF [100%]

===== 6 failed in 0.06 seconds =====
```

# Step 3: Implement function and run tests again

```
def convert_to_int():  
    ...
```

```
!pytest test_convert_to_int.py
```

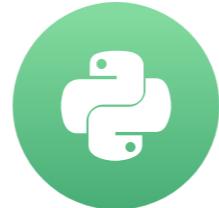
```
===== test session starts =====  
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.8.0, pluggy-0.11.0  
rootdir: /tmp/tmp793ds6mt, infile:  
plugins: mock-1.10.0  
collecting ...  
collected 6 items  
test_convert_to_int.py ..... [100%]  
  
===== 6 passed in 0.03 seconds =====
```

# Let's apply TDD!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

# How to organize a growing set of tests?

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# What you've done so far

16

- `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()
```

# 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
```

# Python module and test module correspondence

- `my_module.py`  $\iff$  `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/
|-- data/                            # Test suite: all tests live here
|   |-- __init__.py
|   |-- test.preprocessing_helpers.py # Corresponds to module src/data/preprocessing_helpers.py
|-- features/
|   |-- __init__.py
|-- models/
    |-- __init__.py
```

# 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()
    ...

...
```

# Test class

# 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
```

# 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
```

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

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

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

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

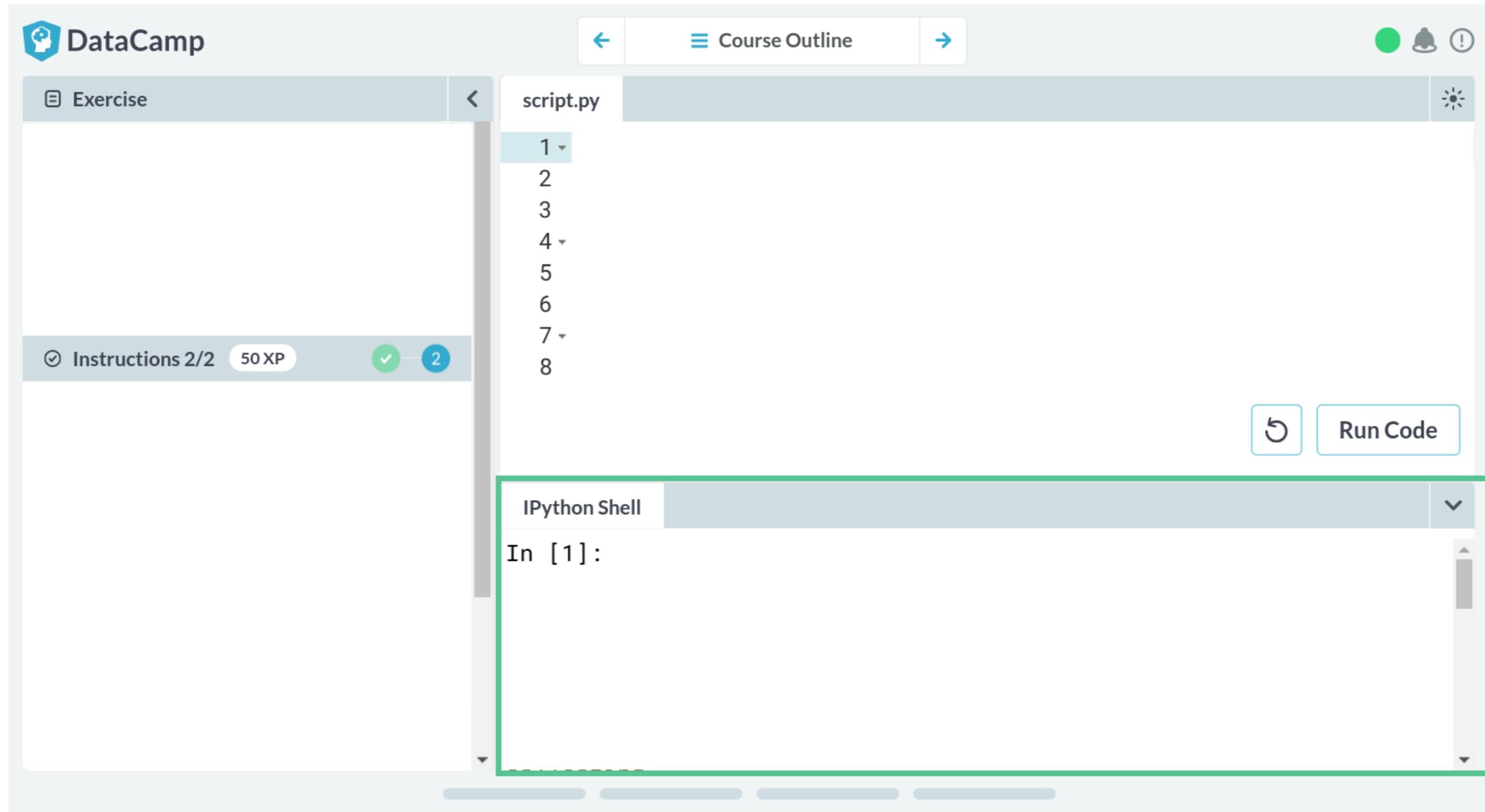
# Final test directory structure

```
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
```

# Test directory is well organized!



# IPython console's working directory is tests



The screenshot shows a DataCamp exercise interface. At the top, there's a navigation bar with the DataCamp logo, course outline links, and user status indicators. Below the navigation bar, the main area is titled "Exercise" and contains a file named "script.py". The code editor shows lines 1 through 8 of the script. To the right of the code editor are two buttons: "Run Code" and a refresh icon. At the bottom of the interface is an "IPython Shell" window, which is highlighted with a green border. The shell window displays the text "In [1]:", indicating it is ready for input.

# IPython console's working directory is tests

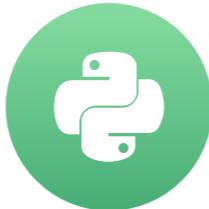
```
src/
|-- data/
|   |-- __init__.py
|   |-- preprocessing_helpers.py
|-- features/
|   |-- __init__.py
|   |-- as_numpy.py
|-- models/
|   |-- __init__.py
|   |-- train.py
tests/                               # This is IPython console's working directory from now on
|-- data/
|   |-- __init__.py
|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

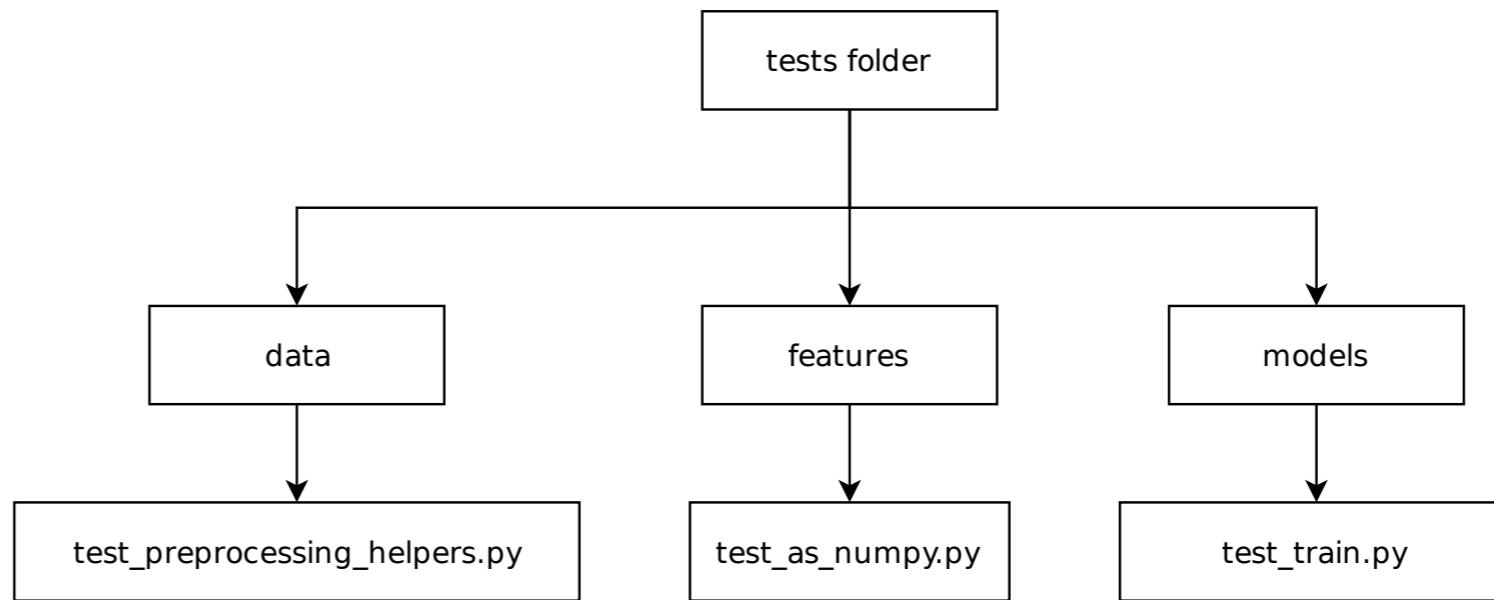


Dibya Chakravorty  
Test Automation Engineer

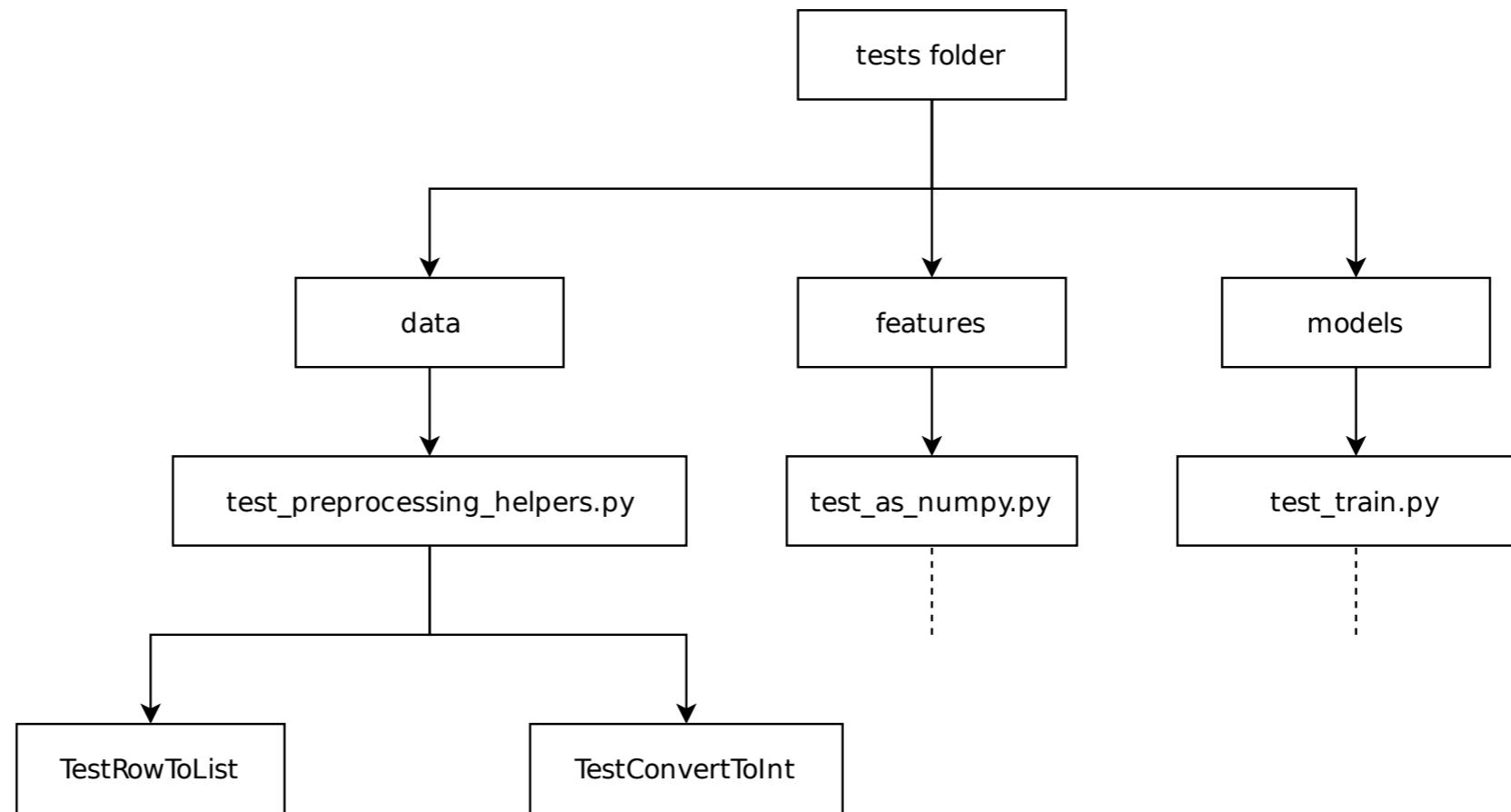
# Test organization

tests folder

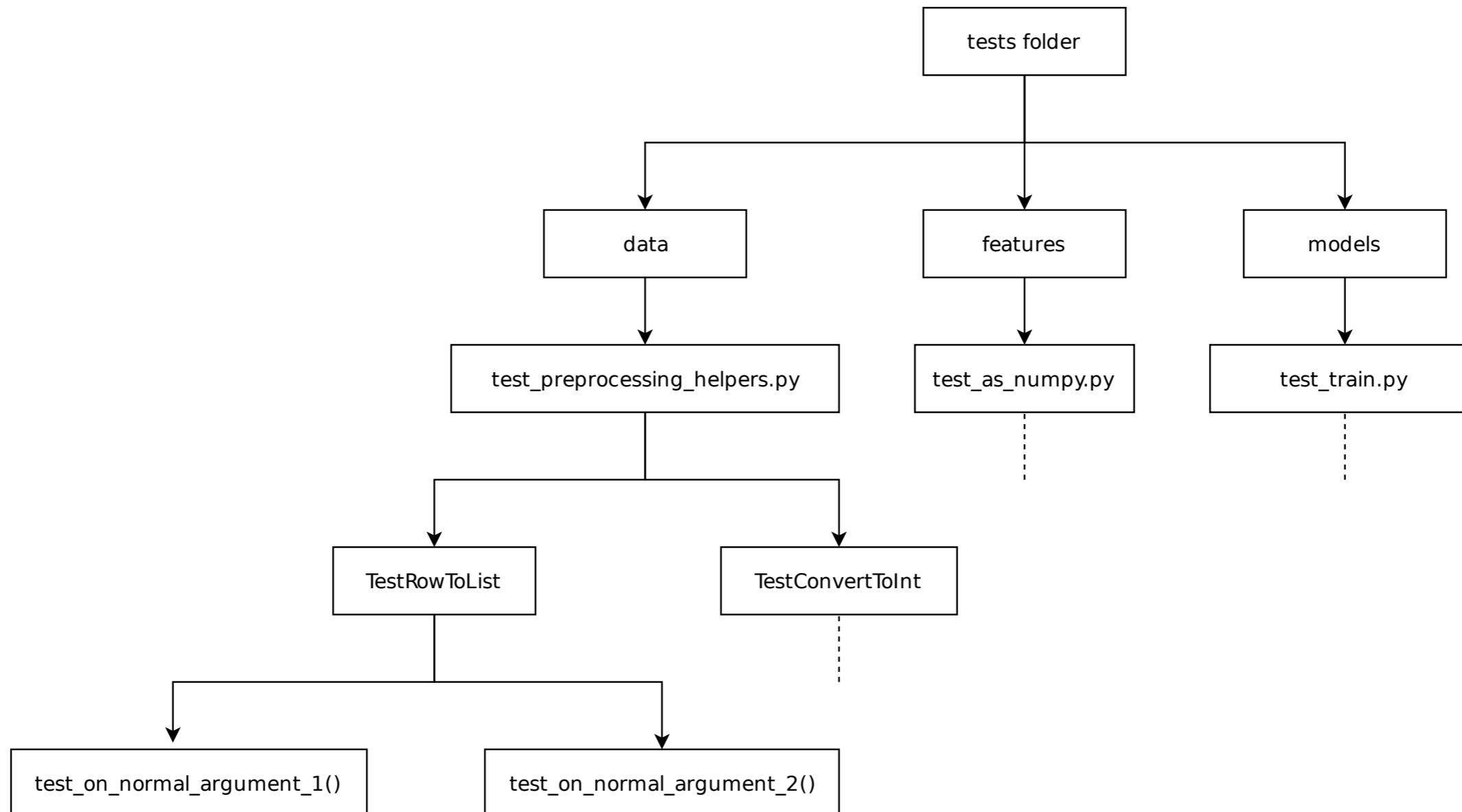
# Test organization



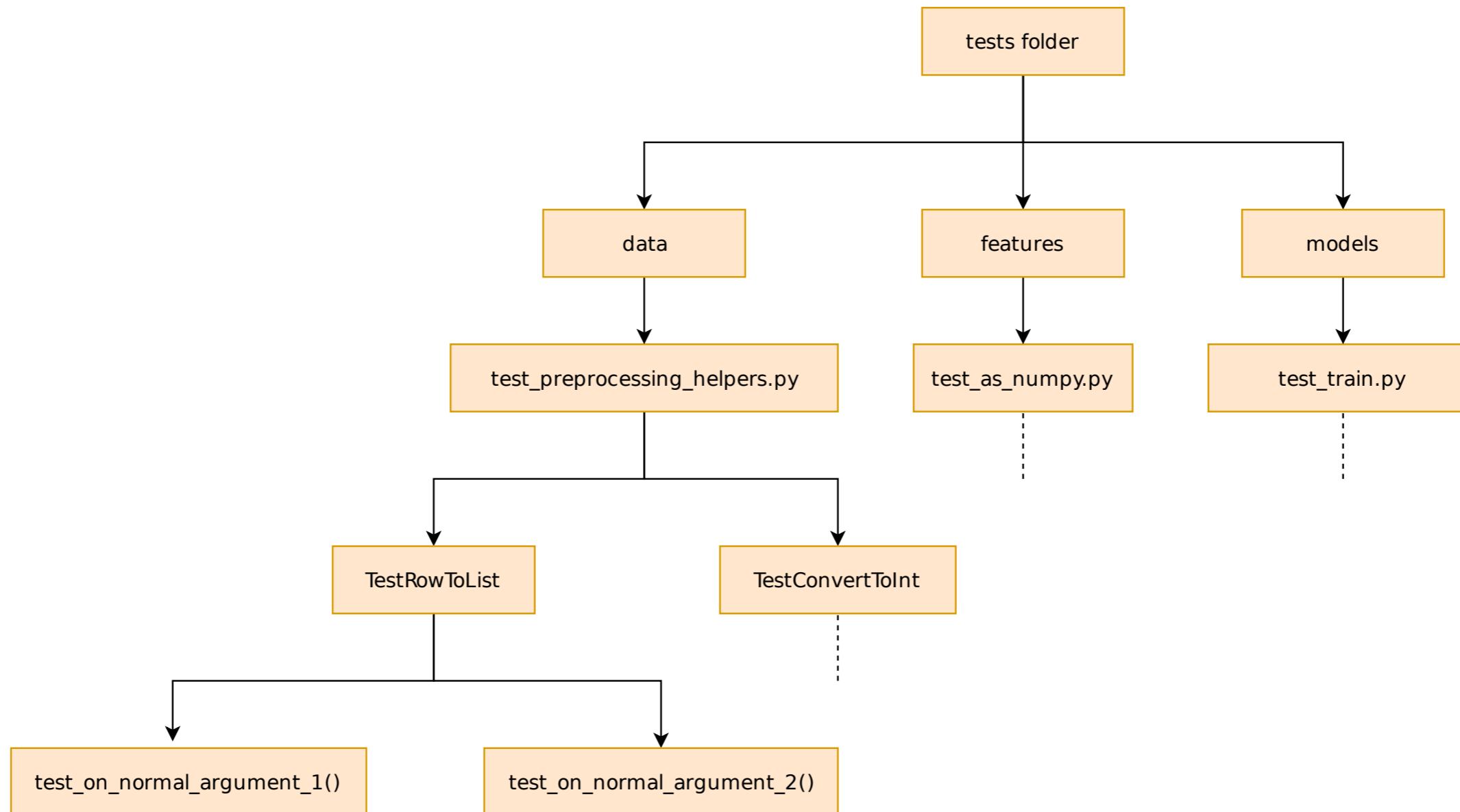
# Test organization



# Test organization



# Running all tests



# Running all tests

```
cd tests  
pytest
```

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

# Running all tests

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

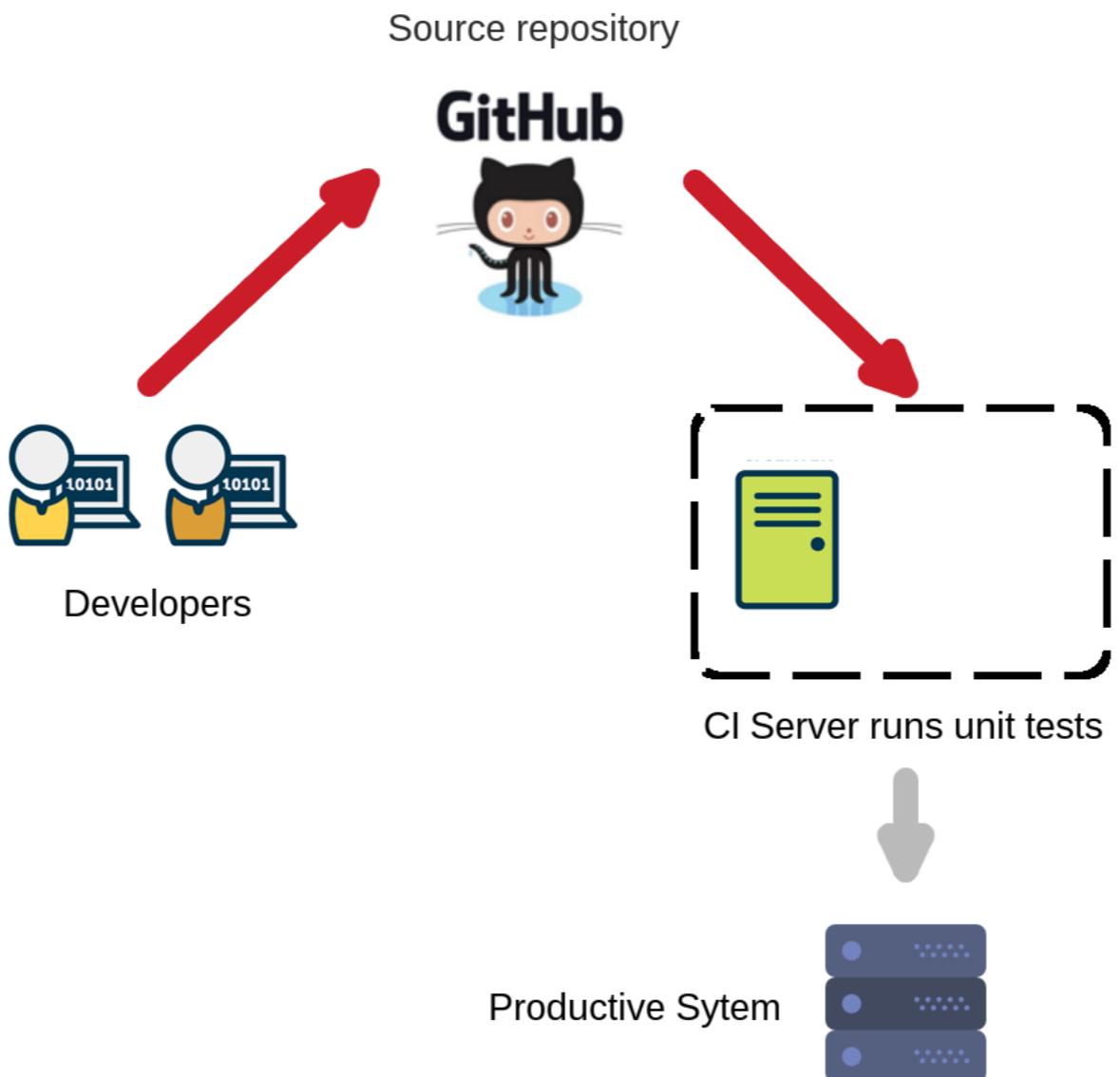
===== FAILURES =====
---- TestRowToList.test_on_one_tab_with_missing_value ----

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

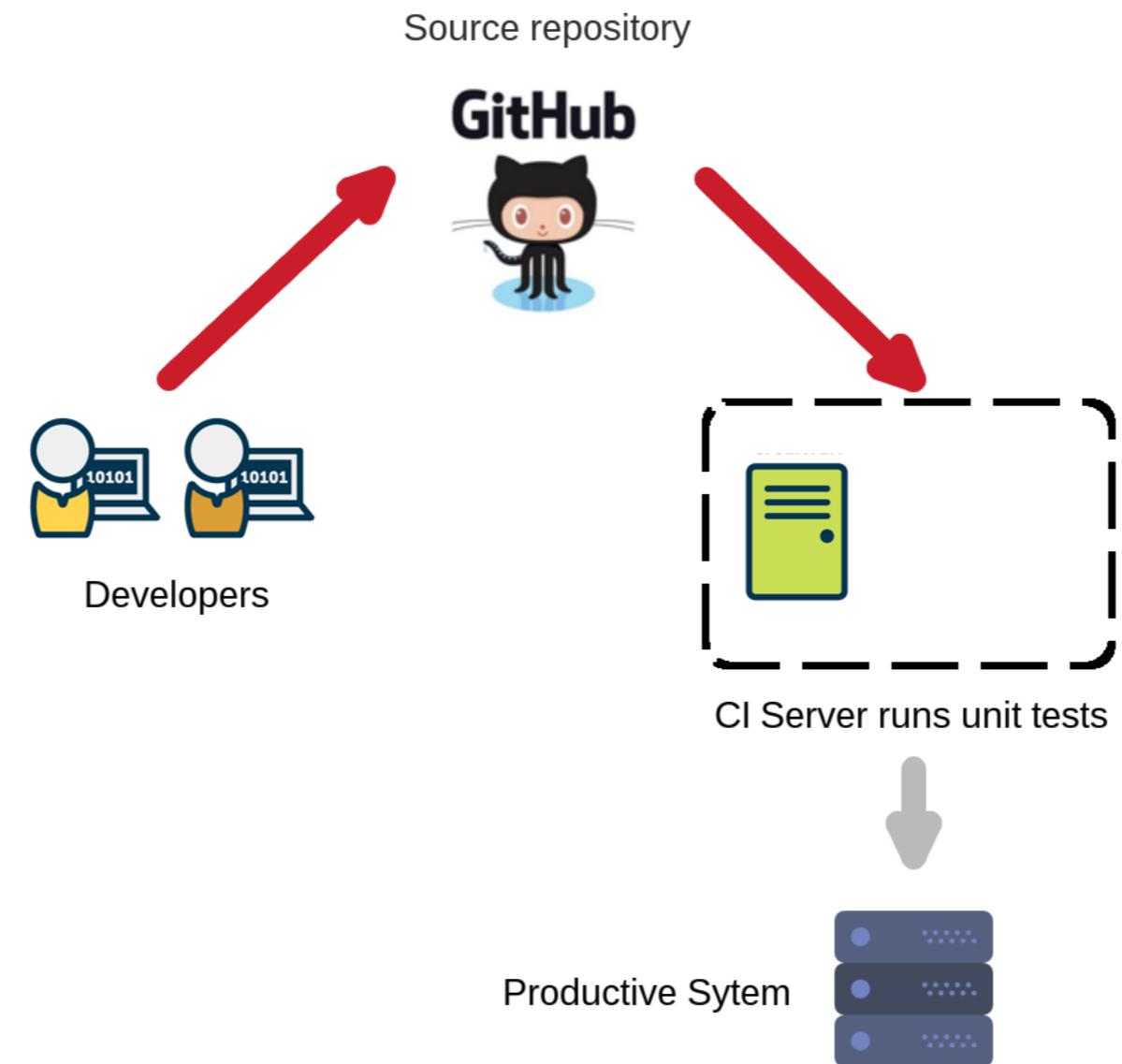
    def test_on_one_tab_with_missing_value(self):    # (1, 1) boundary value
        actual = row_to_list("\t4,567\n")
>       assert actual is None, "Expected: None, Actual: {}".format(actual)
E       AssertionError: Expected: None, Actual: ['', '4,567']
E       assert ['', '4,567'] is None

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
```

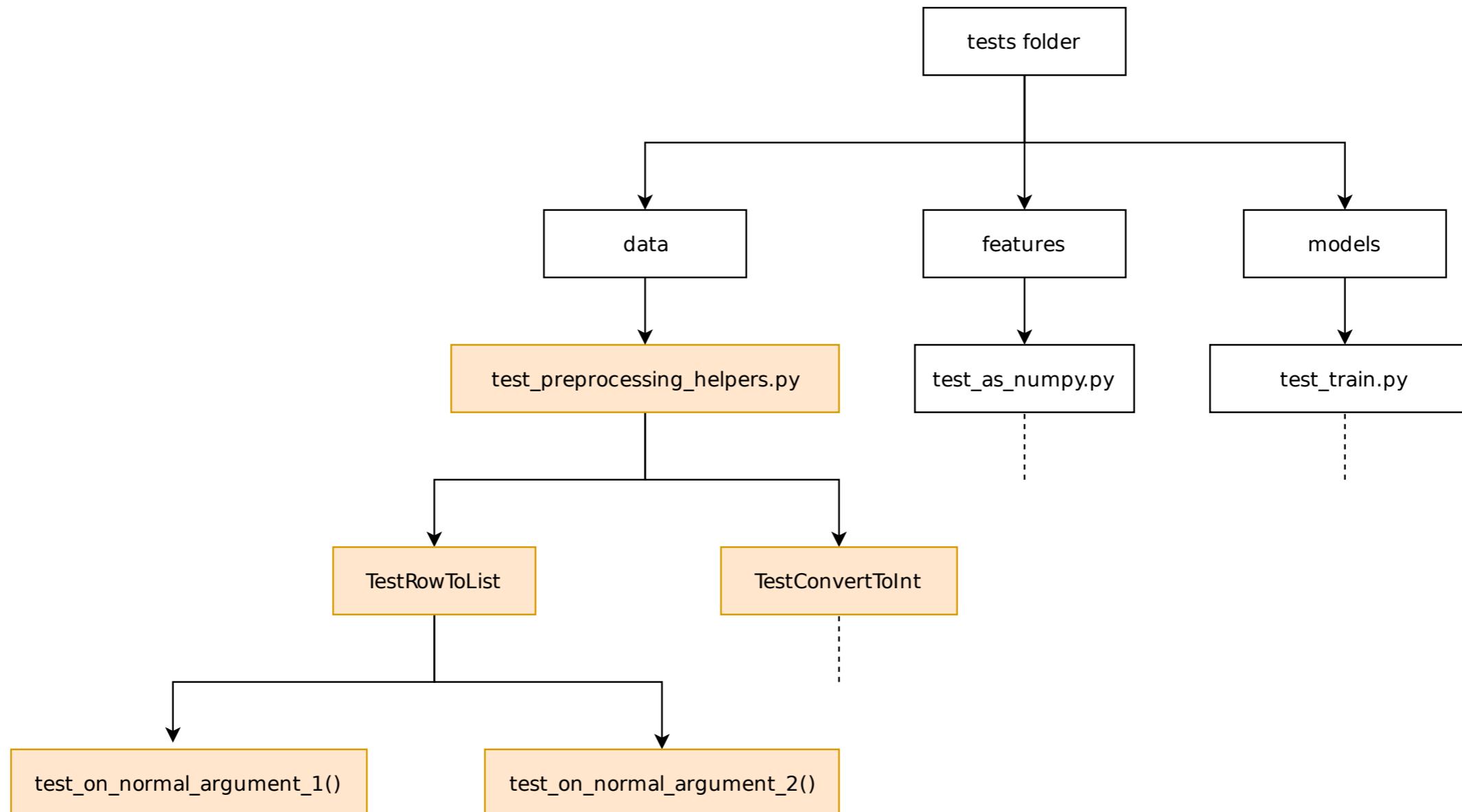
```
===== test session starts =====
data/test_preprocessing_helpers.py .....F

===== 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: {}".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 =====
```

# Running tests in a test module



# Running tests in a test module

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

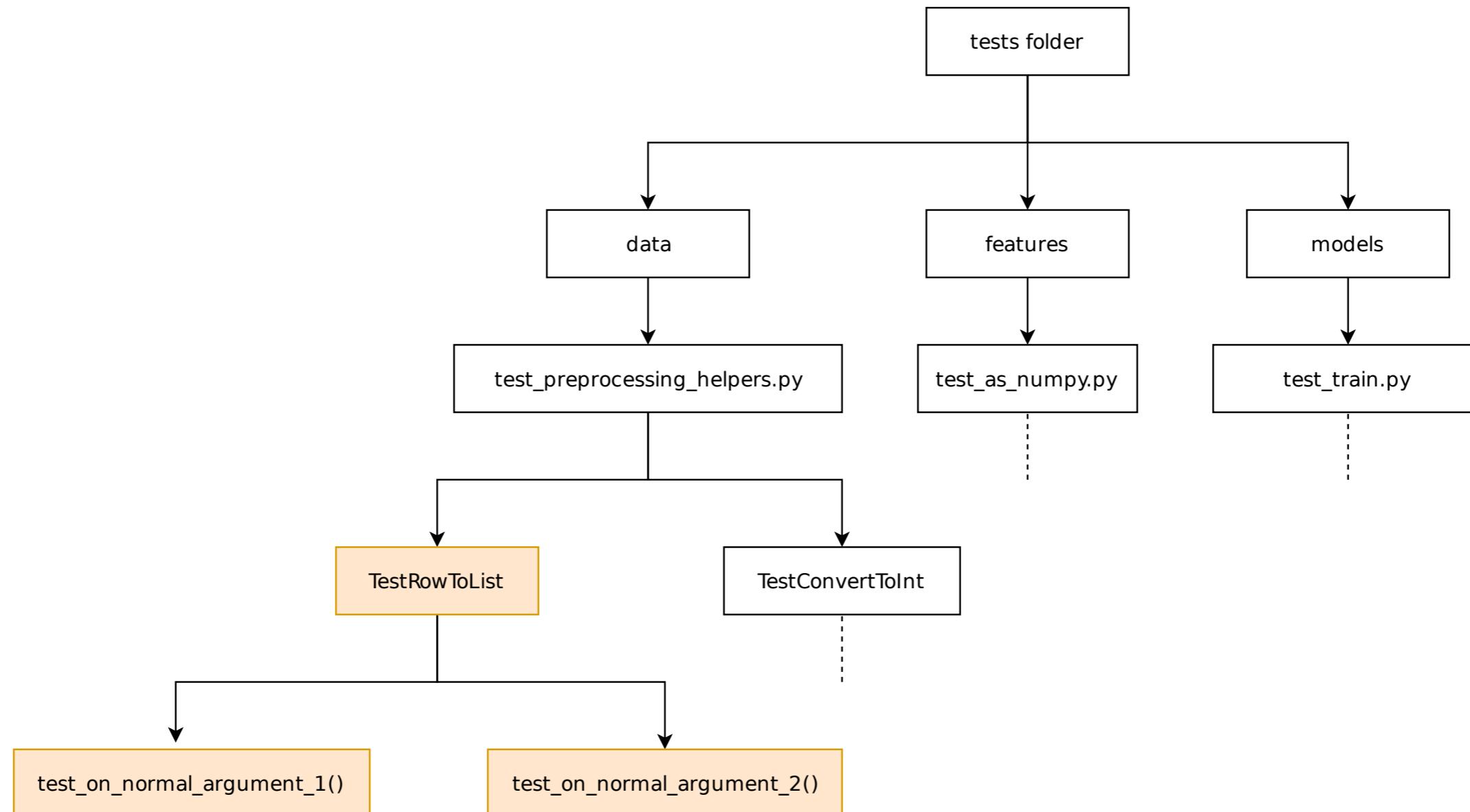
```
data/test_preprocessing_helpers.py .....F.... [100%]

=====
 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: {}".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>

# 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: {}".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 =====
```

# 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
```

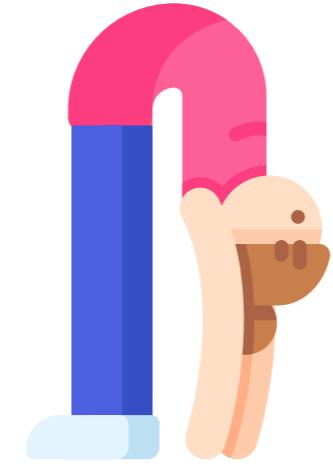
```
data/test_preprocessing_helpers.py F [100%]
```

```
===== 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: {}".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.

# 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

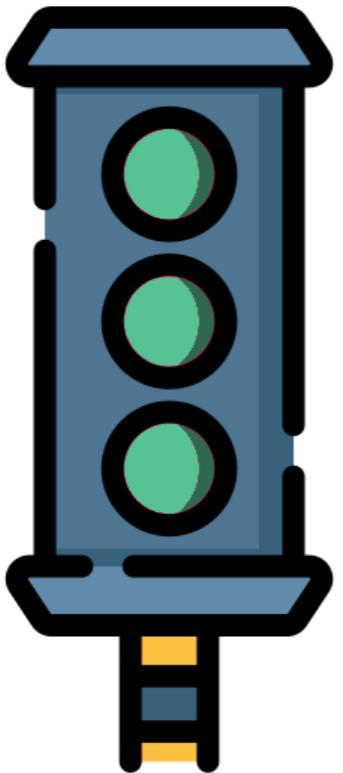
# Expected failures and conditional skipping

UNIT TESTING FOR DATA SCIENCE IN PYTHON

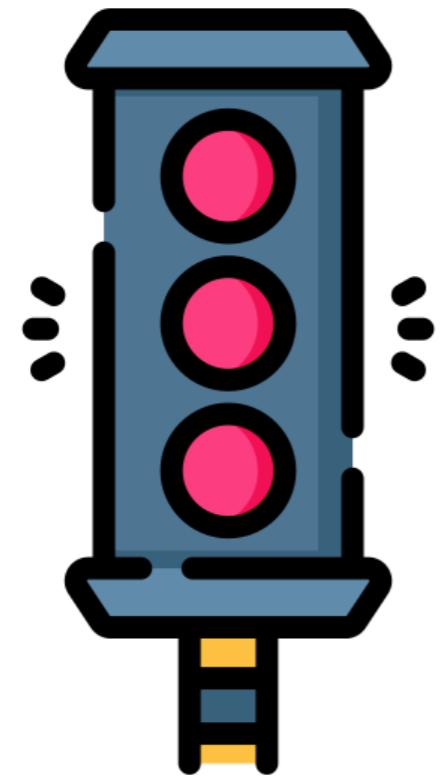
Dibya Chakravorty  
Test Automation Engineer



# Test suite is green when all tests pass



# Test suite is red when any test fails



# 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):
        ...
```

# The test fails, of course!

pytest

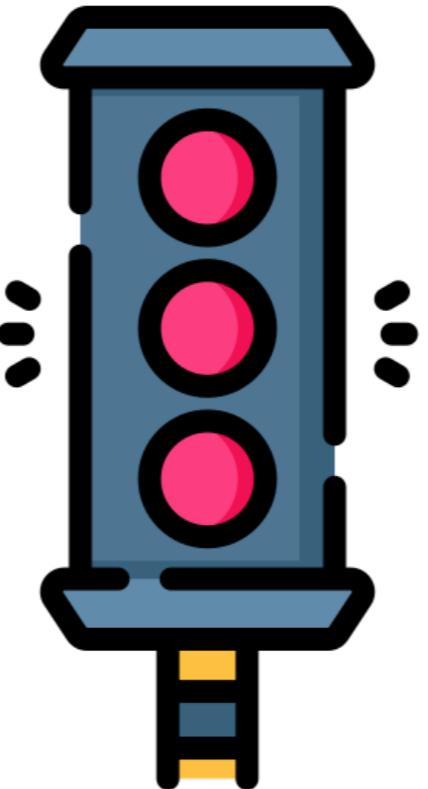
```
===== test session starts =====
data/test_preprocessing_helpers.py ..... [ 76%]
features/test_as_numpy.py . [ 82%]
models/test_train.py ..F [100%]

===== 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



# xfail: marking tests as "expected to fail"

```
import pytest

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

# xfail: marking tests as "expected to fail"

```
import pytest

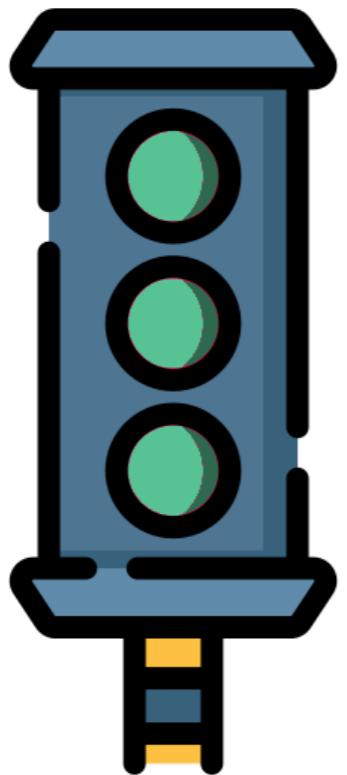
class TestTrainModel(object):
    @pytest.mark.xfail
    def test_on_linear_data(self):
        ...
```

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.

```
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
```

# 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
```

- 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
```

# The reason argument

```
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
```

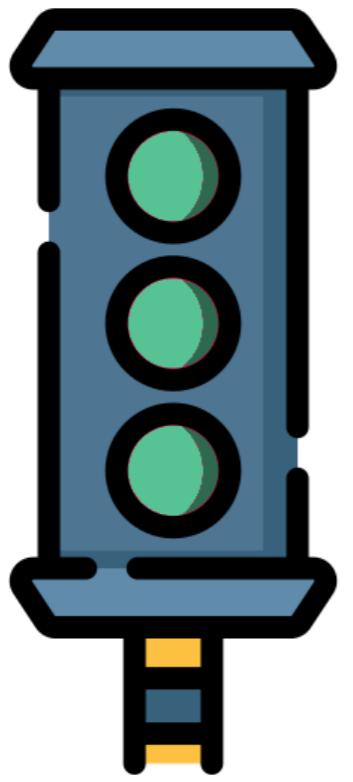
# 1 skipped, 1 xfailed

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 =====
```



# Showing reason in the test result report

```
pytest -r
```

# The -r option

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

# Showing reason for skipping

```
pytest -rs
```

```
===== 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):
        ...
```

# 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):
        ...
```

# Showing reason for xfail

```
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
```

```
===== 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, infile:
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/x failing 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):
```

```
    ...
```

# Let's practice xfailing and skipping!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

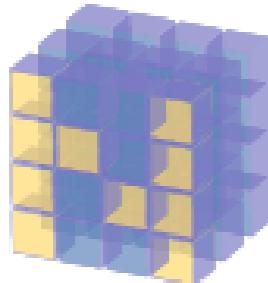
# Continuous integration and code coverage

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty  
Test Automation Engineer



# Code coverage and build status badges



NumPy

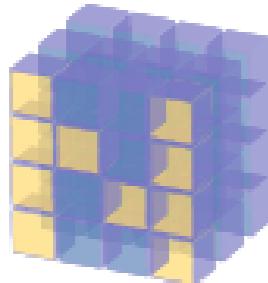
---

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>

# Code coverage and build status badges



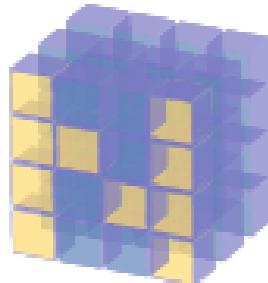
NumPy



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>

# Code coverage and build status badges



NumPy



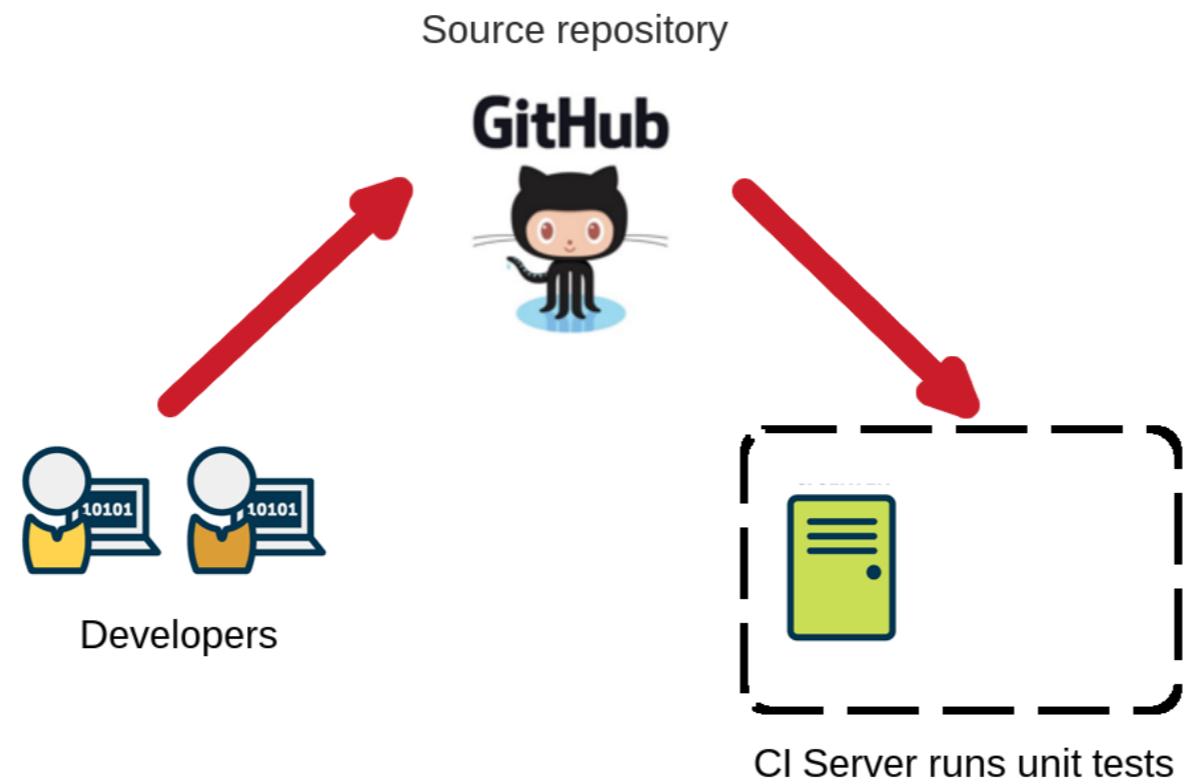
NumPy is the fundamental package needed for scientific computing with Python.

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

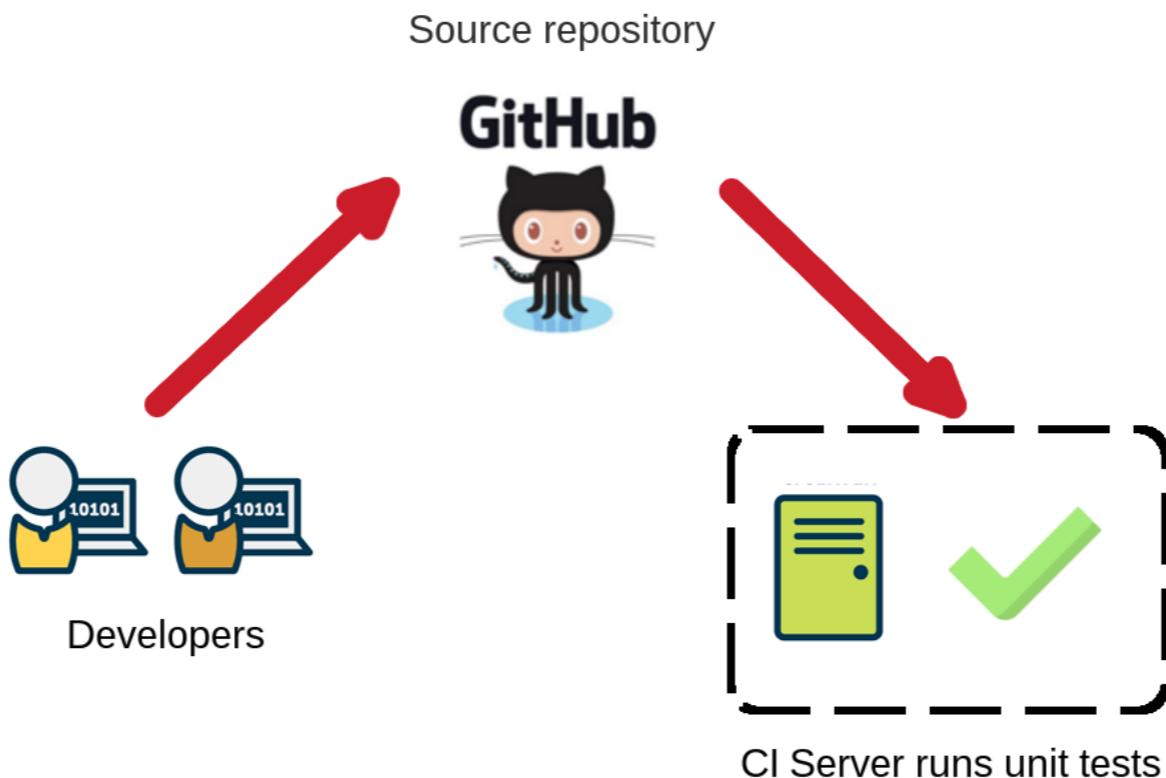
# The build status badge

build passing

# The build status badge

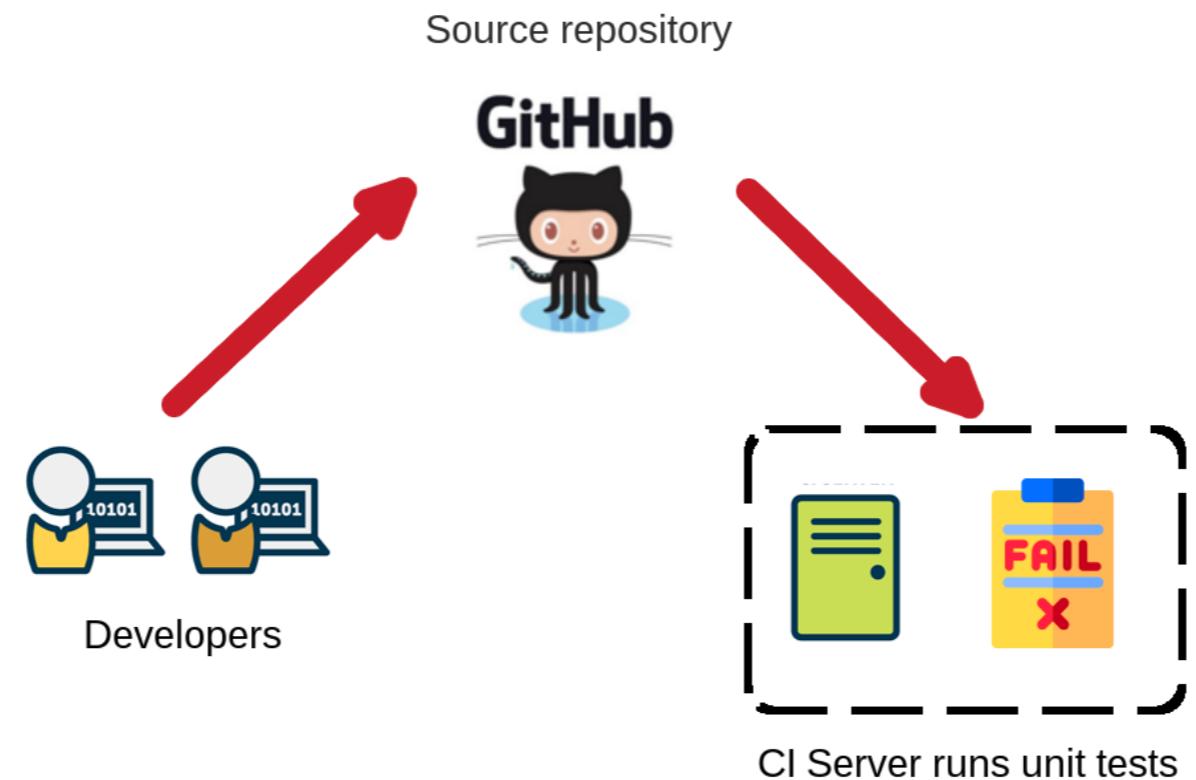


# Build passing = Stable project



build passing

# Build failing = Unstable project



build failing

# CI server



# Travis CI

# Step 1: Create a configuration file

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

# Step 1: Create a configuration file

- Contents of `.travis.yml`.

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

# Step 2: Push the file to GitHub

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

# Step 3: Install the Travis CI app

The screenshot shows a GitHub user profile for 'gutfeeling'. The top navigation bar has tabs for 'Pull requests', 'Issues', 'Marketplace' (which is highlighted in green), and 'Explore'. The left sidebar shows repositories like 'datacamp/courses-unit-testing-i...', 'gutfeeling/univariate-linear-regr...', and 'gutfeeling/practical\_rl\_for\_coders'. The main area displays activity feeds for repositories forked from 'gutfeeling/beginner\_nlp'. A pink callout box for 'GitHub Sponsors Matching Fund' is visible on the right.

sofiaa forked sofiaa/beginner\_nlp from gutfeeling/beginner\_nlp yesterday

**gutfeeling/beginner\_nlp**  
A curated list of beginner resources in Natural Language Processing

★ 367 Updated Jun 11

sofiaa and akashgudadhe1 starred 1 repository yesterday

**gutfeeling/beginner\_nlp**  
A curated list of beginner resources in Natural Language Processing

★ 367 Updated Jun 11

akashgudadhe1 forked akashgudadhe1/beginner\_nlp from gutfeeling/beginner\_nlp yesterday

**gutfeeling/beginner\_nlp**

Discover repositories

- nickdavidhaynes/spacy-cld**  
Language detection extension for spaCy 2.0+  
Python ★ 87
- PacktPublishing/Python-Machine-Learning-Cookbook**  
Code files for Python-Machine-Learning-Cookbook  
Python ★ 259
- sloria/textblob-fr**  
French language support for TextBlob.

# Step 3: Install the Travis CI app

Screenshot of the GitHub Marketplace search results for "travis".

The search bar contains "travis".

The results section shows 1 result for "travis":

- Travis CI** (verified icon) - Test and deploy with confidence

Navigation buttons: Previous, Next.

Categories listed on the left:

- API management
- Chat
- Code quality
- Code review
- Continuous integration
- Dependency management
- Deployment
- IDEs
- Learning

# Step 3: Install the Travis CI app

## Pricing and setup

### Open Source

We offer free CI for Open Source projects

\$0

### ONE

[Free Trial](#)

Unlimited builds, 1 job at a time. Ideal for hobby and small projects.

\$69

/ month

### THREE

[Free Trial](#)

Unlimited builds, 3 jobs at a time. Best suited for small teams.

\$199

/ month

### SIX

[Free Trial](#)

Unlimited builds, 6 jobs at a time. Great for growing teams.

\$349

/ month

Travis CI

### Open Source



We offer free CI for Open Source projects

- ✓ Unlimited public repositories
- ✓ Unlimited collaborators

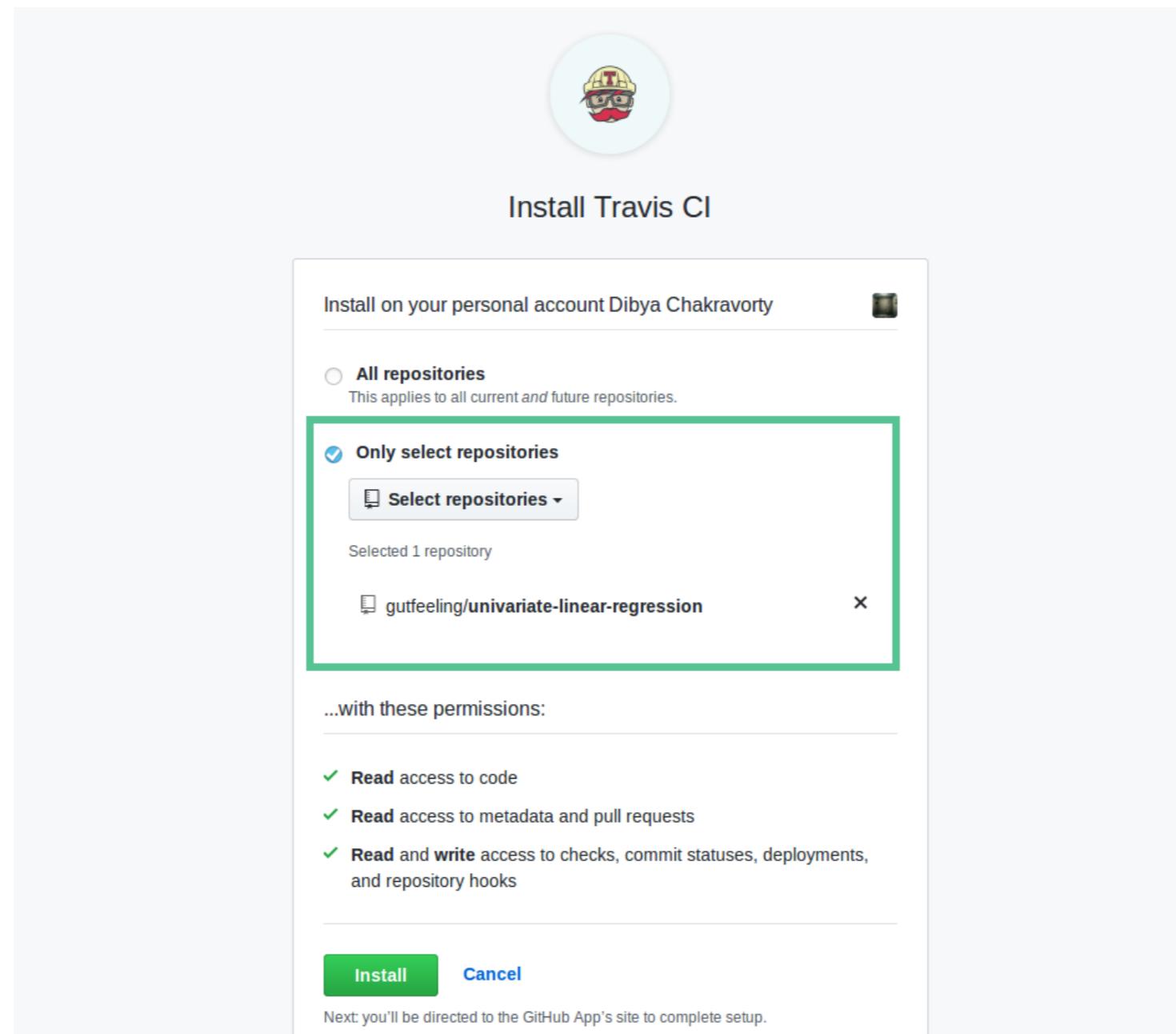
Account: [gutfeeling](#) ▾

[Install it for free](#)

Next: Confirm your installation location.

Travis CI is provided by a third-party and is governed by separate [terms of service](#), [privacy policy](#), and [support contact](#).

# Step 3: Install the Travis CI app



# Step 3: Install the Travis CI app

Travis CI   About Us   Plans & Pricing   Enterprise   Help



We're so glad you're here!

Please sign in to view your repositories.

Sign in with GitHub

# Every commit leads to a build

Travis CI  Dashboard Changelog Documentation Help 

Search all repositories 

My Repositories Running (1/1) +

gutfeeling / univariate-linear-regression  build passing

Current Branches Build History Pull Requests More options 

gutfeeling/univariate-linear-reg # 13 

Duration: 28 sec

master Remove useless comment #13 started  
Commit 8f6c815   
Compare 3695237..8f6c815   
Branch master   
Dibya Chakravorty

Python: 3.6

 Job log  View config

# Step 4: Showing the build status badge

The screenshot shows the Travis CI dashboard for the repository `gutfeeling / univariate-linear-regression`. The build status is shown as `build passing` in a green box. The build details show a job for branch `master` with commit `8f6c815`, started at #13, running for 28 seconds. The build is currently running. The repository has 13 pull requests. The sidebar shows the repository's URL and duration.

Travis CI Dashboard for `gutfeeling / univariate-linear-regression`

Current Build Status: `build passing`

Build Details:

- Job: master Remove useless comment
- Commit: 8f6c815
- Pull Request: #13 started
- Duration: 28 sec
- Branch: master
- User: Dibya Chakravorty
- Python: 3.6

Repository Statistics:

- My Repositories: 1
- Running (1/1)
- Duration: 28 sec
- Branches: 1
- Build History: 13
- Pull Requests: 13

Actions:

- Job log
- View config
- Cancel build

# Step 4: Showing the build status badge

The screenshot shows the Travis CI interface for a repository named "gutfeeling/univariate-linear-regression". The repository has 13 branches, a duration of 42 seconds, and was finished less than a minute ago. A modal window titled "Status Image" is open, showing configuration for the "master" branch. The "MARKDOWN" section is highlighted with a green border. The code snippet inside the box is:

```
[![Build Status](https://travis-ci.com/gutfeeling/univariate-linear-regression.svg?branch=master)](https://travis-ci.com/gutfeeling/univariate-linear-regression)
```

Below the modal, there are links for "Job log" and "View config".

# Step 4: Showing the build status badge

The screenshot shows a GitHub commit history and a README.md file. The commit history lists nine commits from a user named 'gutfeeling'. The most recent commit is 'Latest commit e08b854 now' and it removes a codecov badge. The README.md file contains a green rectangular badge with the text 'build passing'.

Commit	Message	Time
Remove codecov badge	Latest commit e08b854 now	
Add tests	2 months ago	
Fix jupyter notebook	2 months ago	
remove fake data generator	4 days ago	
Better test for float valued string	4 days ago	
Improved gitignore	3 months ago	
Remove comments	4 days ago	
Remove codecov badge	now	
Remove useless comment	3 minutes ago	

**README.md**

build passing

# Code coverage



- 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
```

# 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`

```
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

The screenshot shows the GitHub Marketplace search results for "codecov". The search bar at the top contains the query "codecov". To the left, there is a sidebar titled "Categories" with links to various tools: API management, Chat, Code quality, Code review, Continuous integration, Dependency management, Deployment, IDEs, and Learning. The main search results area displays one result: "Codecov" with a green checkmark icon, described as "Group, merge, archive and compare coverage reports". Below the result are "Previous" and "Next" navigation buttons.

Marketplace / Search results

Categories

API management

Chat

Code quality

Code review

Continuous integration

Dependency management

Deployment

IDEs

Learning

Search or jump to... / Pull requests Issues Marketplace Explore

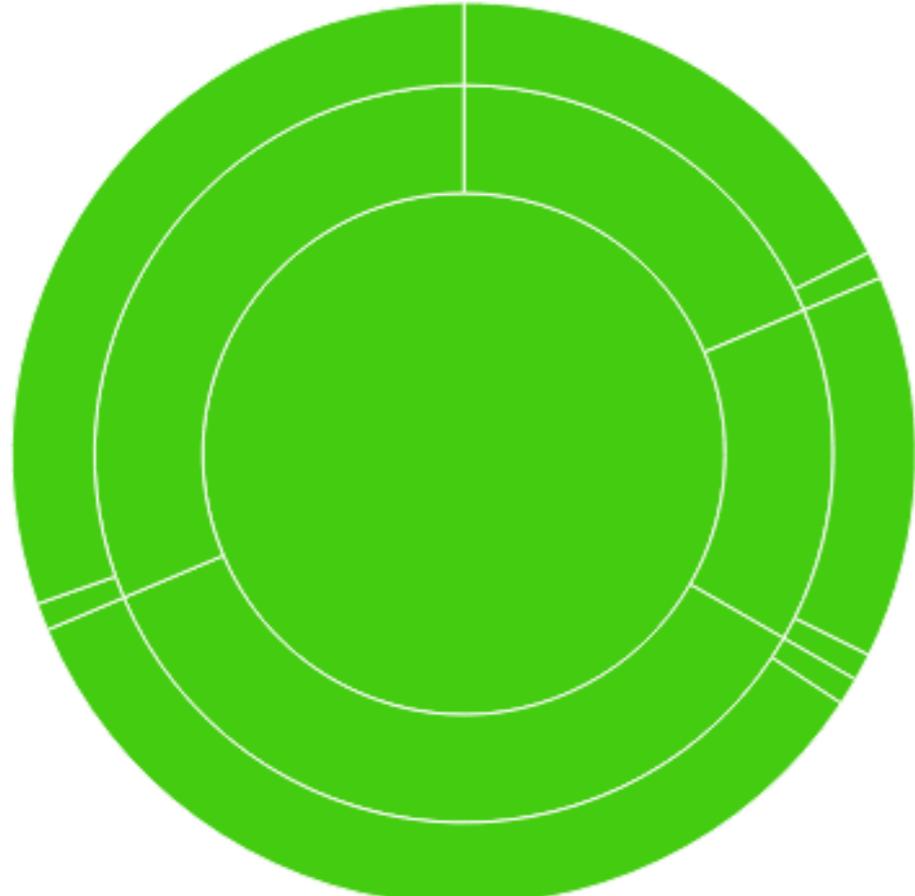
(codecov) 1 result for "codecov"

Codecov ✓ Group, merge, archive and compare coverage reports

Previous Next

# Commits lead to coverage report at codecov.io

COVERAGE SUNBURST



A large green sunburst chart representing code coverage data, divided into several segments.

ALL RECENT COMMITS

Commit Message	Author	Date	Branch	SHA	CI Status
Fix side effects and call order in mocks	gutfeeling	7 days ago	master	298091d	CI Passed
Add pytest-mock	gutfeeling	7 days ago	master	02e714a	CI Passed
Add test for preprocessing function	gutfeeling	7 days ago	master	b8dfc76	CI Passed
Add baseline plots and add pytest-mpl requirement	gutfeeling	7 days ago	master	752e0e0	CI Passed
Merge remote-tracking branch 'origin/master'	gutfeeling	7 days ago	master	7ef5ef2	CI Passed
Add codecov badge	gutfeeling	10 days ago	master	960fa3f	CI Passed

Browse Report

Browse Report

Browse Report

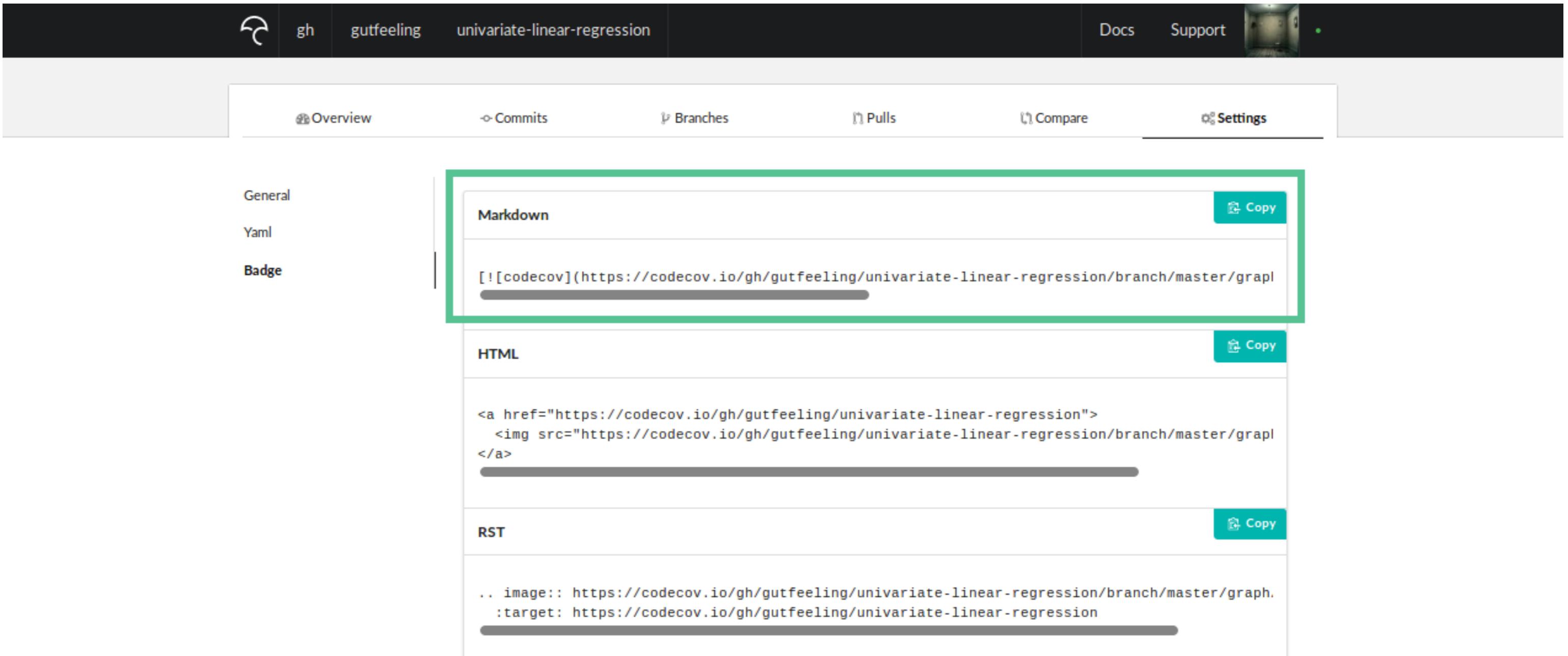
Browse Report

Browse Report

Browse Report

[View all recent commits](#)

# Step 3: Showing the badge in GitHub



The screenshot shows a GitHub repository page for 'gutfeeling/univariate-linear-regression'. The 'Settings' tab is selected. On the left, there's a sidebar with 'General', 'Yaml', and 'Badge' sections. The 'Badge' section is highlighted with a green box. It contains three tabs: 'Markdown', 'HTML', and 'RST'. The 'Markdown' tab shows the code: `[![codecov](https://codecov.io/gh/gutfeeling/univariate-linear-regression/branch/master/graph/badge.svg)](https://codecov.io/gh/gutfeeling/univariate-linear-regression/branch/master/graph)`. The 'HTML' tab shows: `<a href="https://codecov.io/gh/gutfeeling/univariate-linear-regression"></a>`. The 'RST' tab shows: `.. image:: https://codecov.io/gh/gutfeeling/univariate-linear-regression/branch/master/graph/badge.svg  
: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%

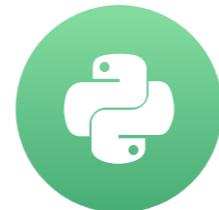
This repository holds the code for the DataCamp course Unit Testing for Data Science in Python by Dibya Chakravorty.

# Let's practice CI and code coverage!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

# Beyond assertion: setup and teardown

UNIT TESTING FOR DATA SCIENCE IN PYTHON



Dibya Chakravorty  
Test Automation Engineer

# The preprocessing function

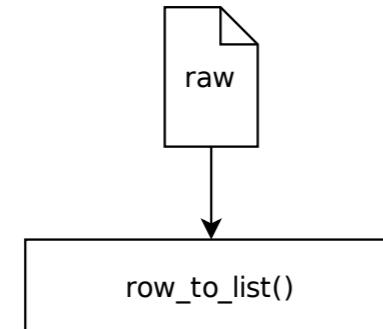
```
def preprocess(raw_data_file_path,  
              clean_data_file_path  
              ):  
    ...
```



1,801	201,411
1,767	565,112
2,002	333,209
1990	782,911
1,285	389129

# The preprocessing function

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

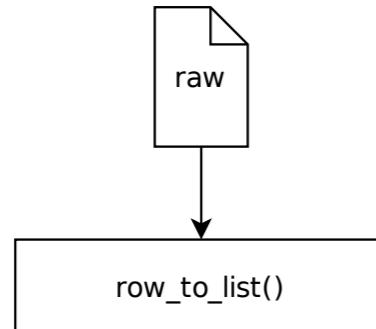


1,801	201,411
1,767565,112	# dirty row, no tab
2,002	333,209
1990	782,911
1,285	389129

# The preprocessing function

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

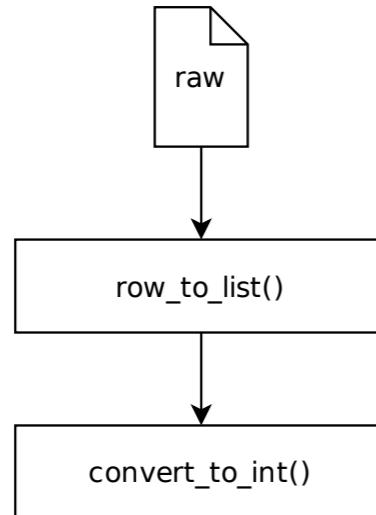
1,801	201,411
2,002	333,209
1990	782,911
1,285	389129



# 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

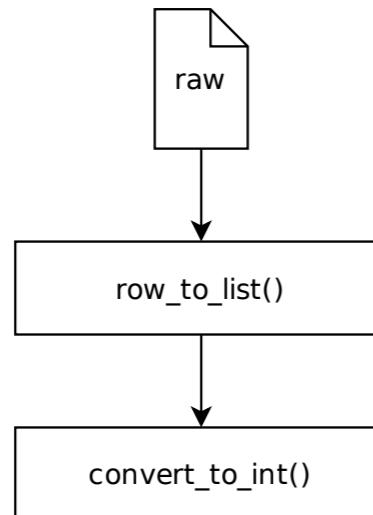


# The preprocessing function

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

...

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

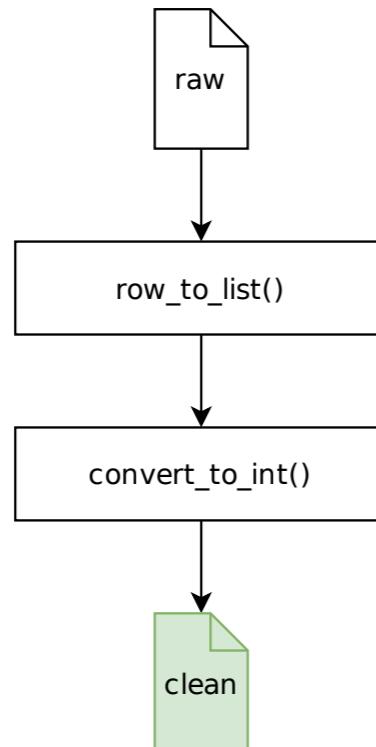


# The preprocessing function

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

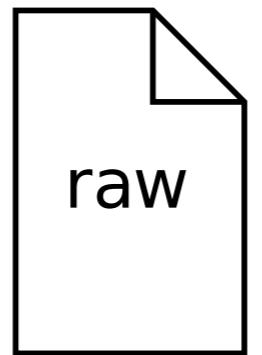
...

1801	201411
2002	333209



# Environment preconditions

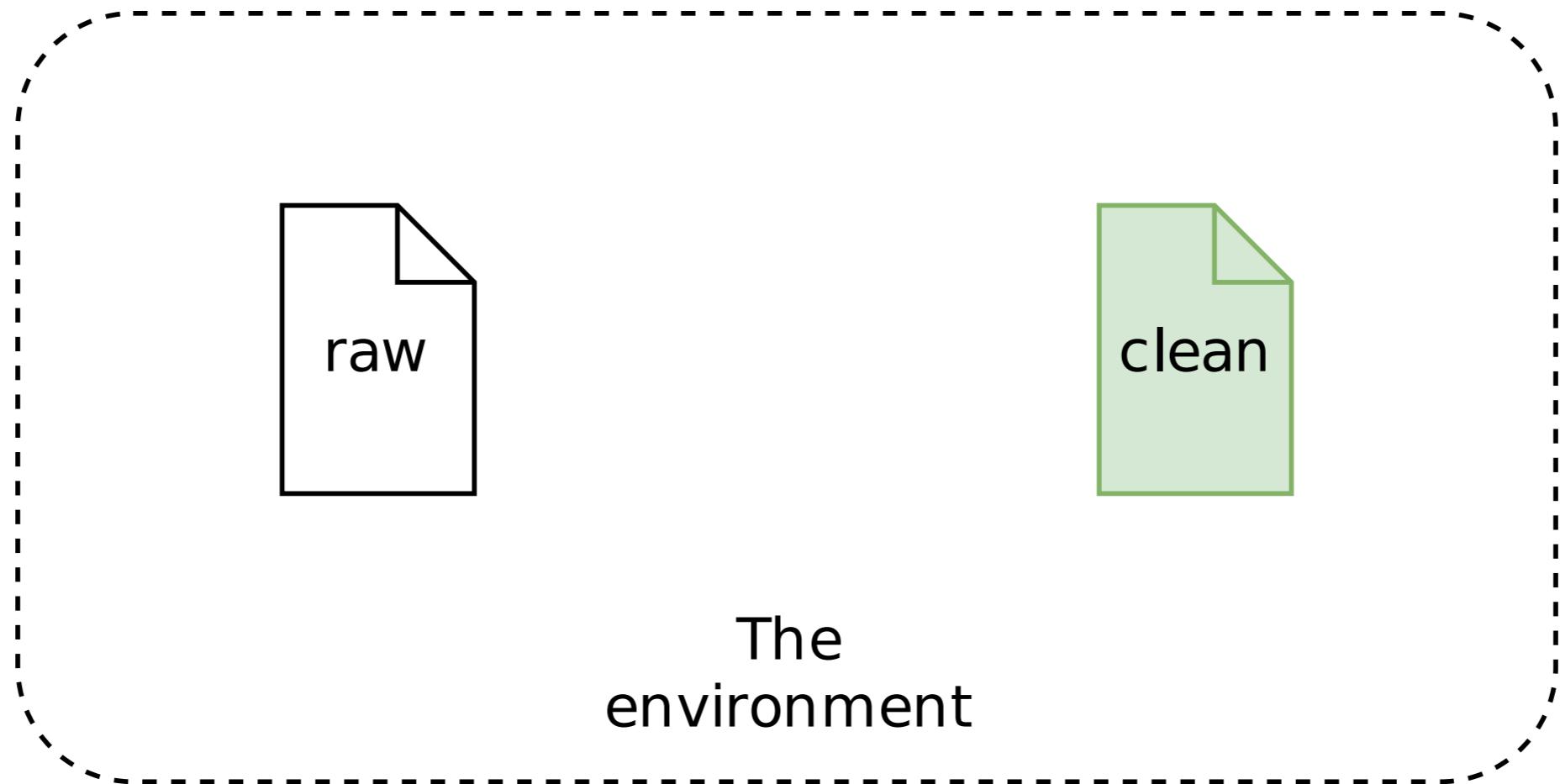
- `preprocess()` needs a raw data file in the environment to run.



The  
environment

# Environment modification

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



# Testing the preprocessing function

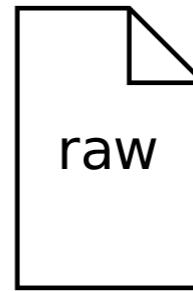
```
def test_on_raw_data():
```

The  
environment

# 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.



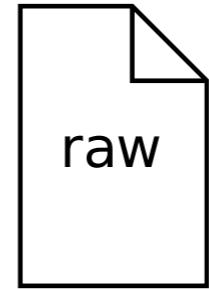
The  
environment

# 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"
```



The  
environment

# 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
```

The  
environment

- Teardown brings environment to initial state.

# The new workflow

## Old workflow

- assert

## New workflow

- setup → assert → 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_path
    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_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"
```

# 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**

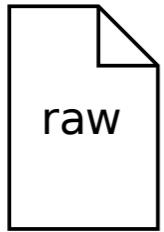
# Mocking

UNIT TESTING FOR DATA SCIENCE IN PYTHON

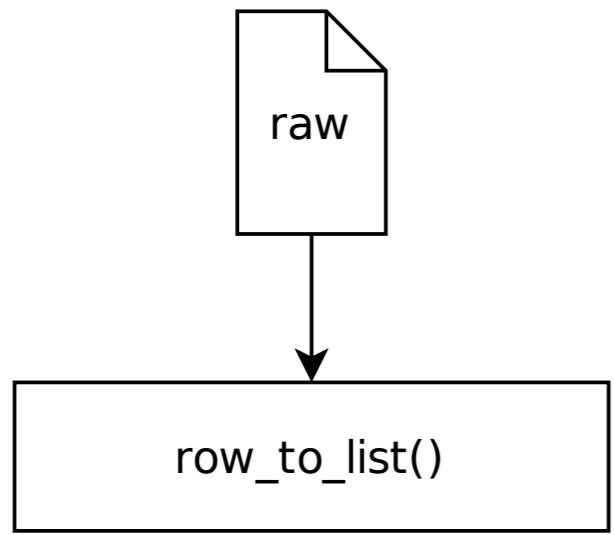


**Dibya Chakravorty**  
Test Automation Engineer

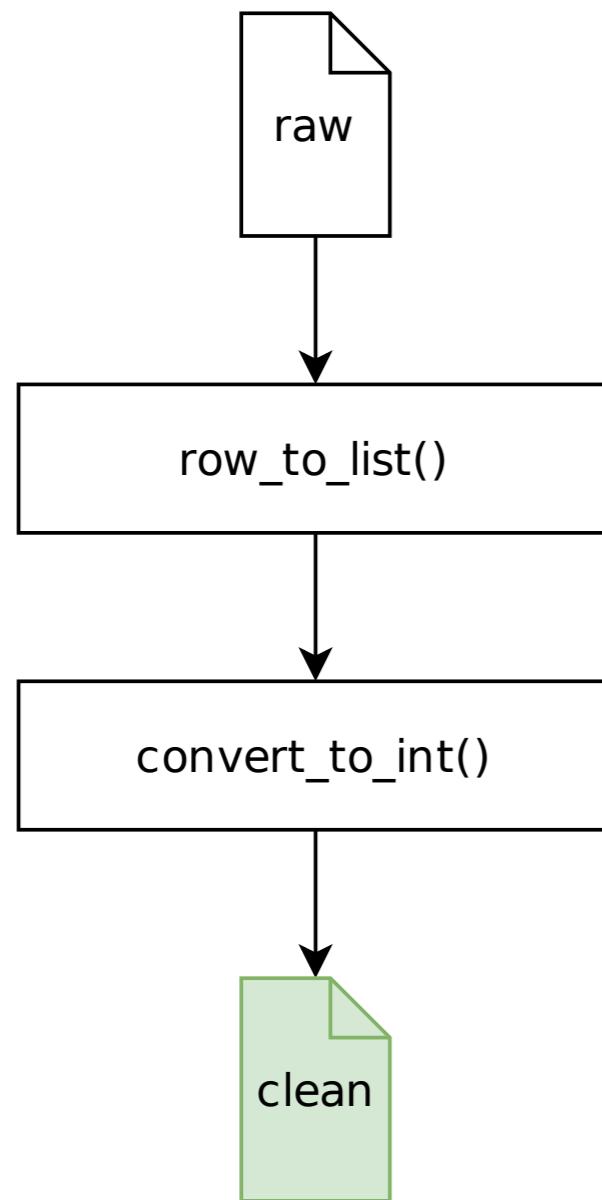
# The preprocessing function



# The preprocessing function



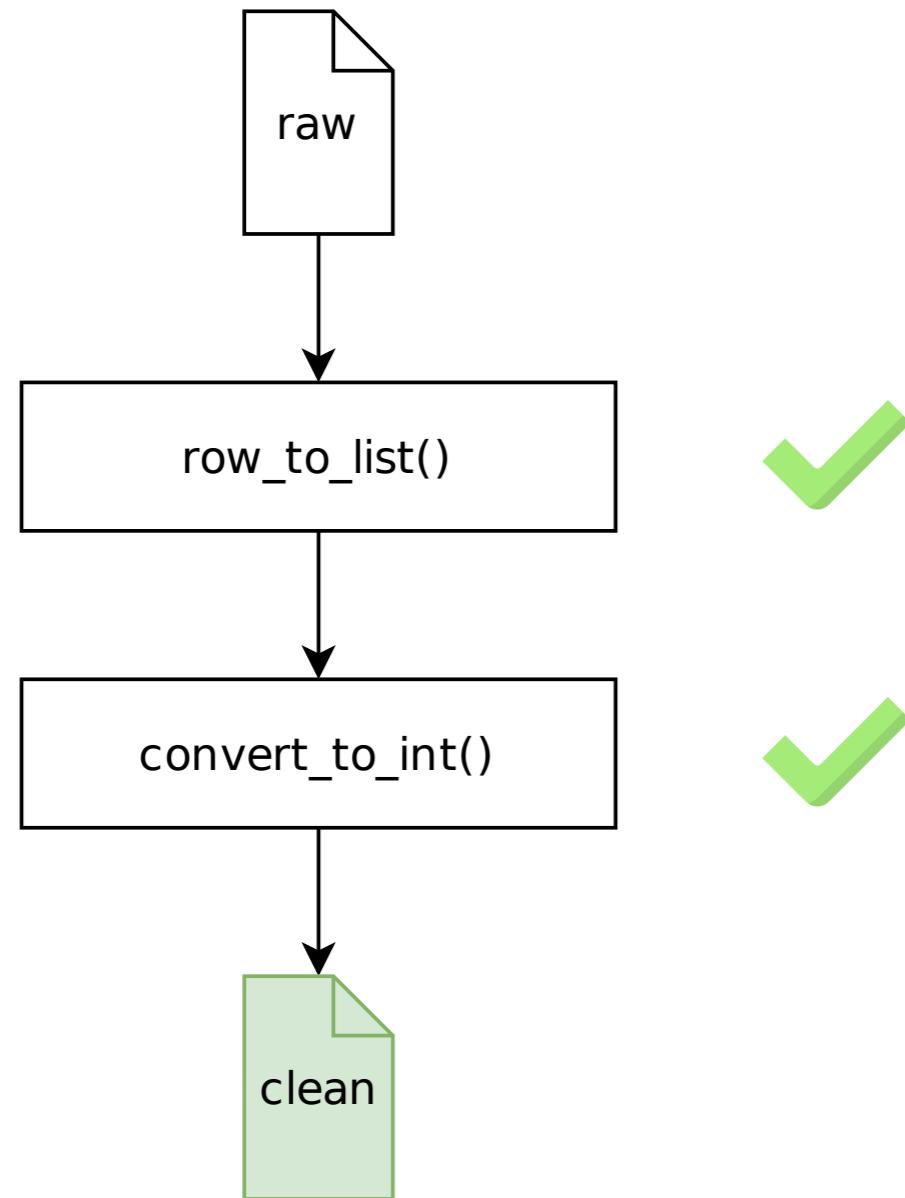
# The preprocessing function



```
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 =====
```

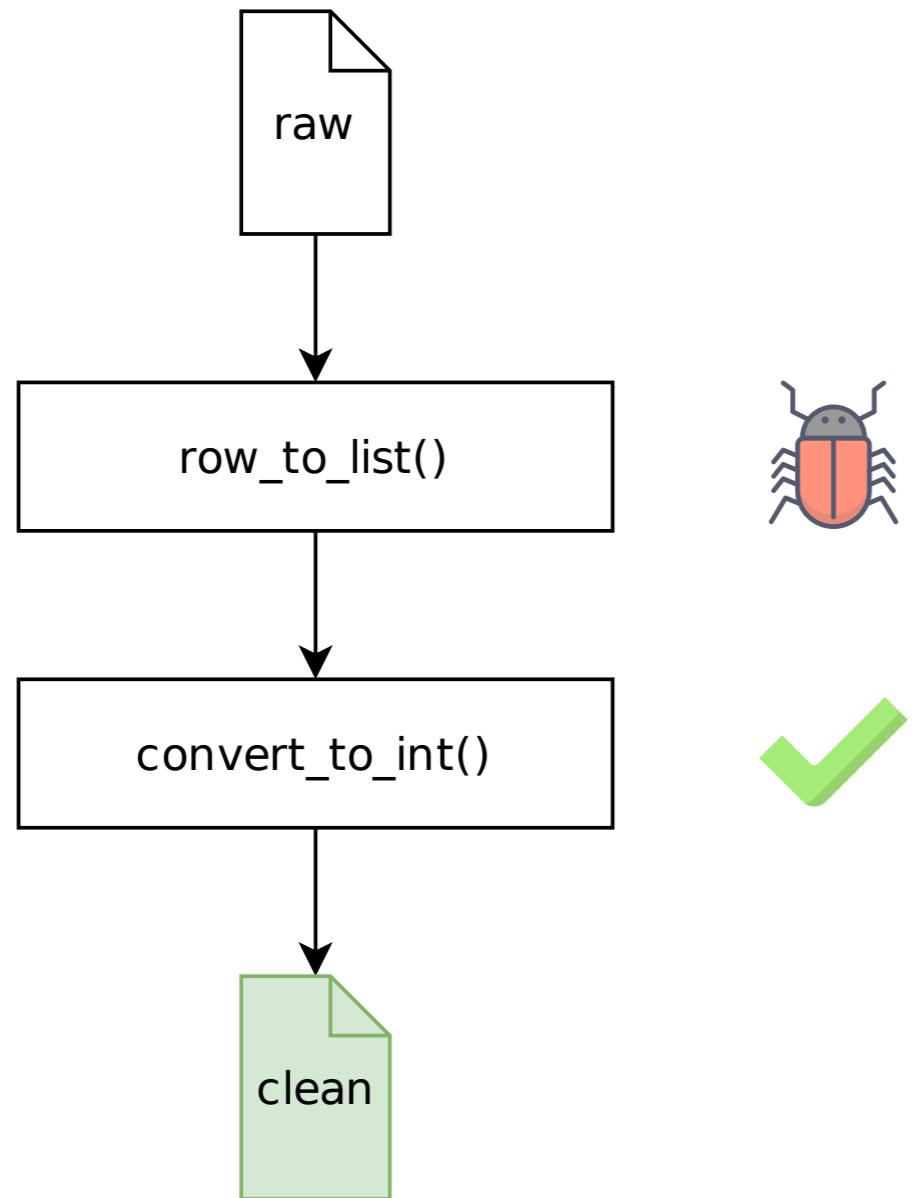
# Test result depend on dependencies



```
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 0.60 seconds
```

# Test result depends on dependencies

Test result should indicate bugs in

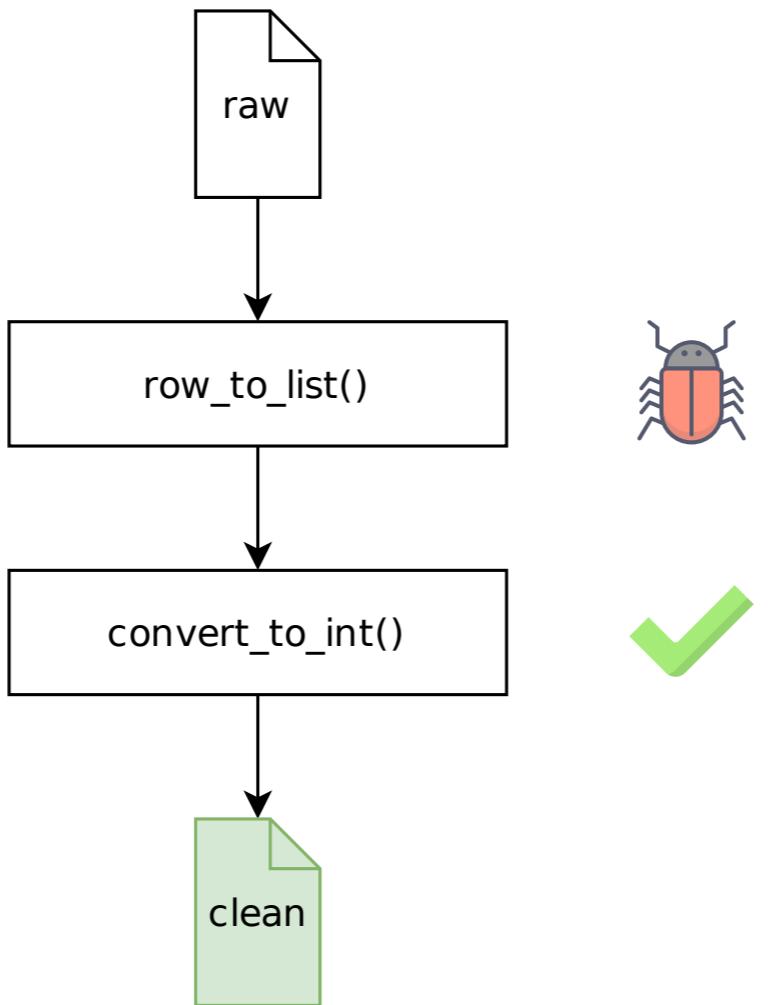
- function under test i.e. `preprocess()` .
- not dependencies e.g. `row_to_list()` or `convert_to_int()` .

# Mocking: testing functions independently of dependencies

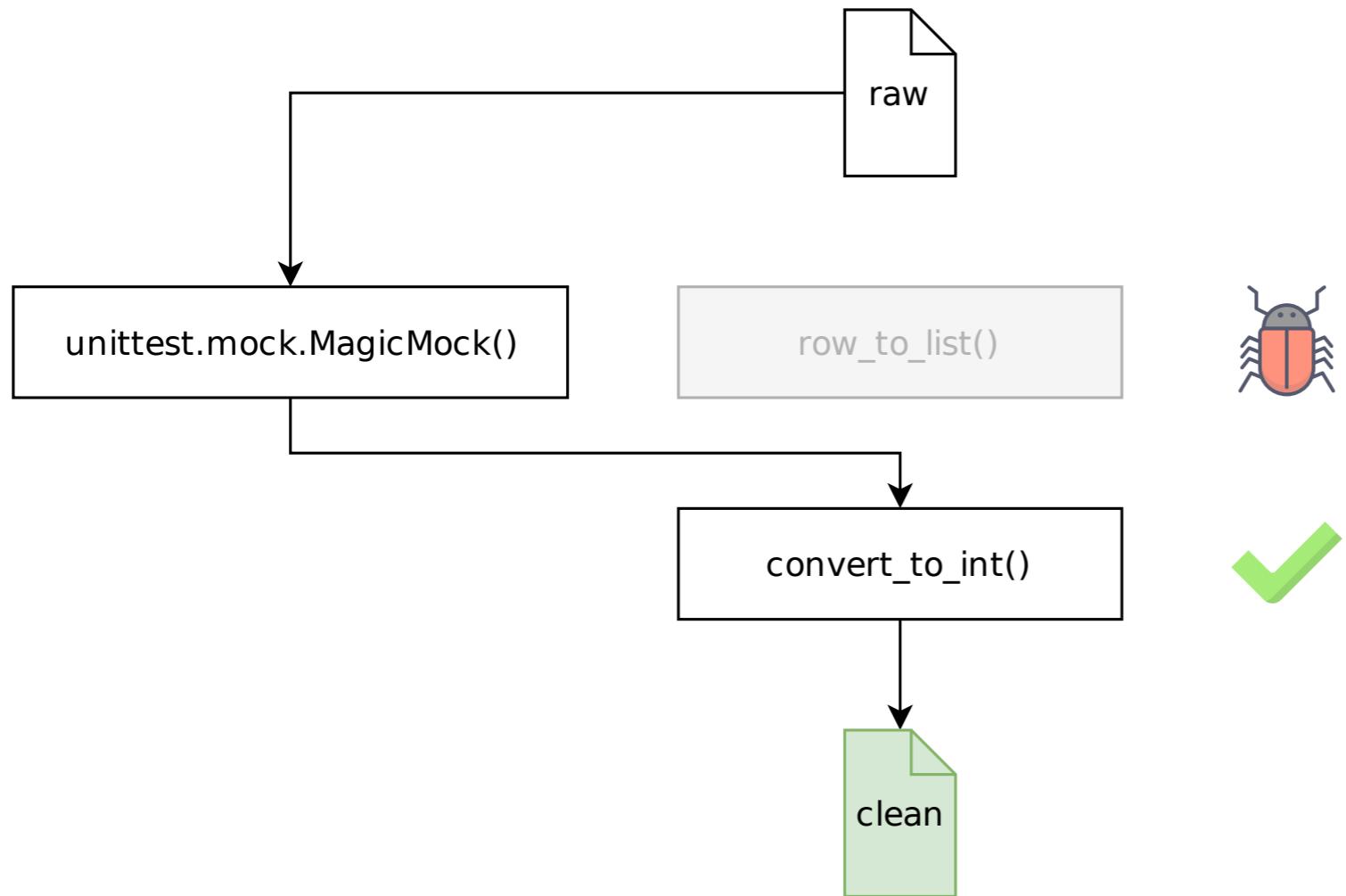
Packages for mocking in `pytest`

- `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

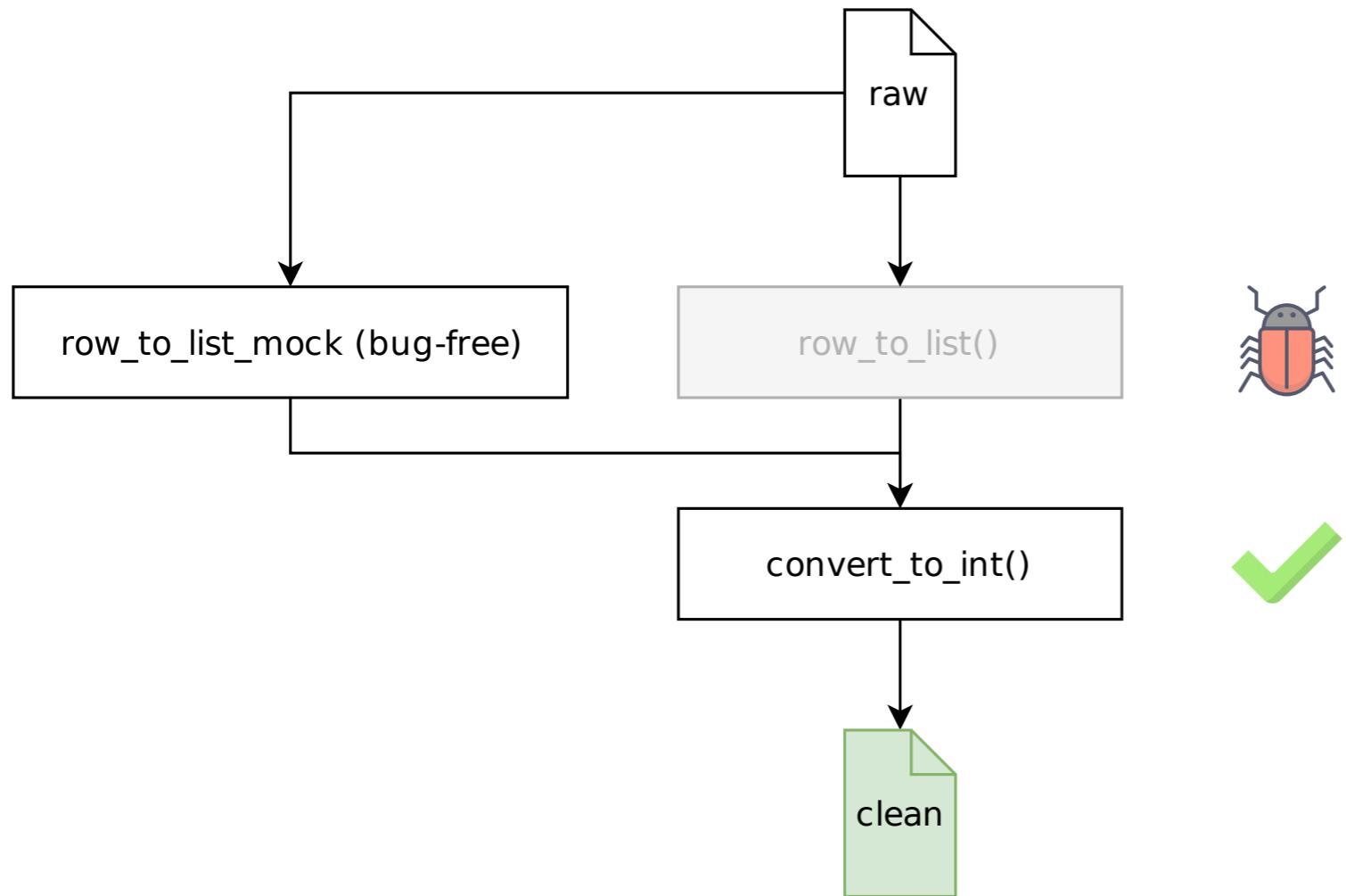
- Raw data

```
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%]

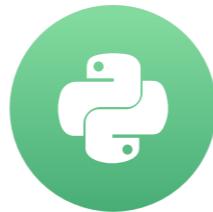
===== 1 passed, 20 deselected in 0.63 seconds =====
```

# 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

# 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

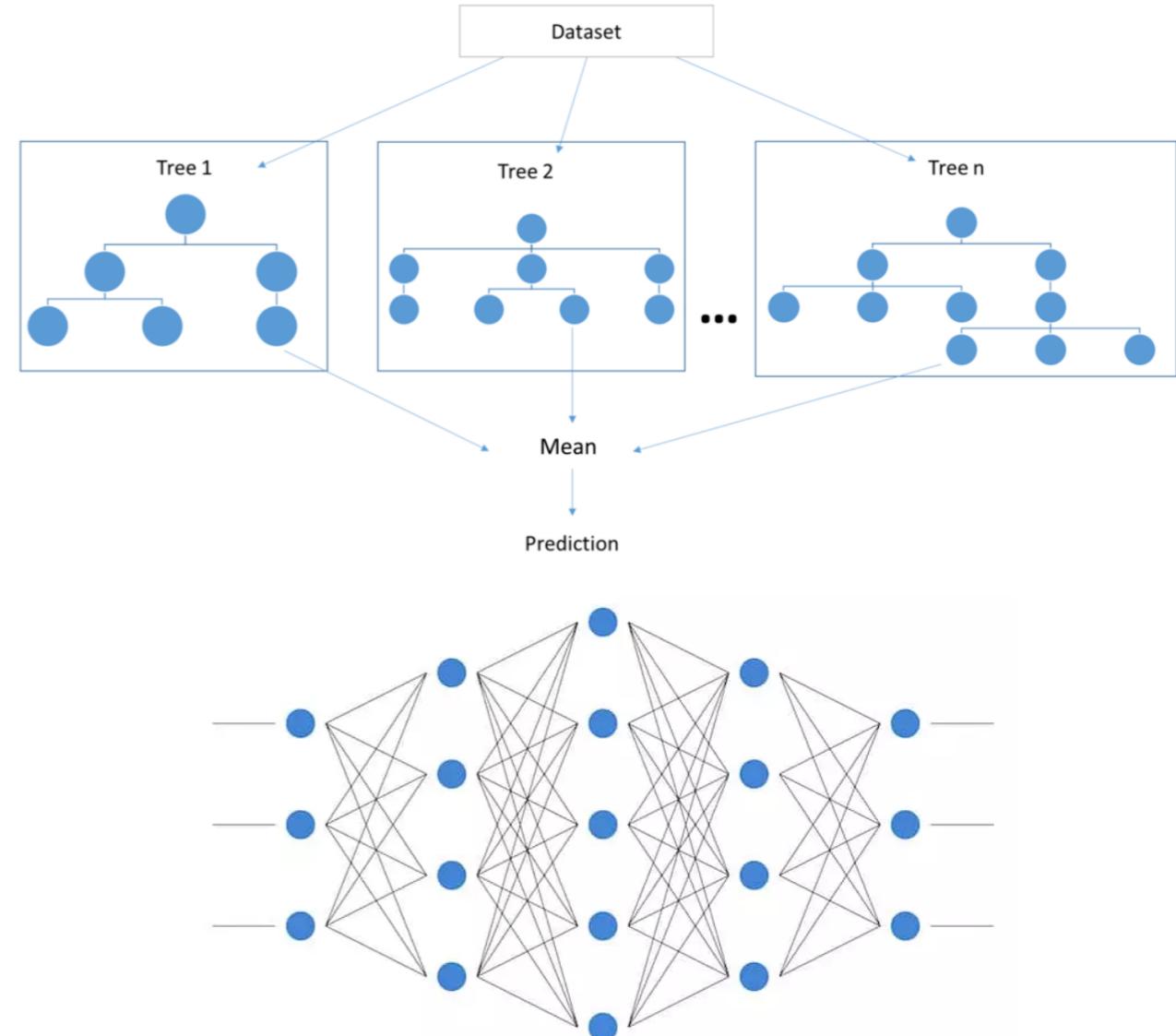
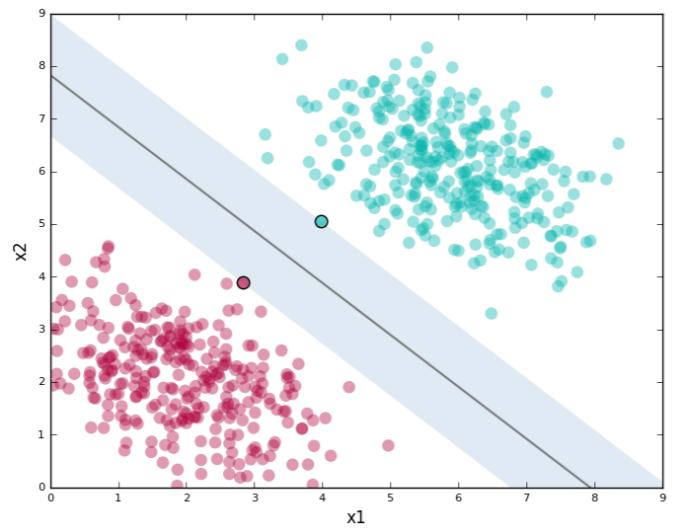
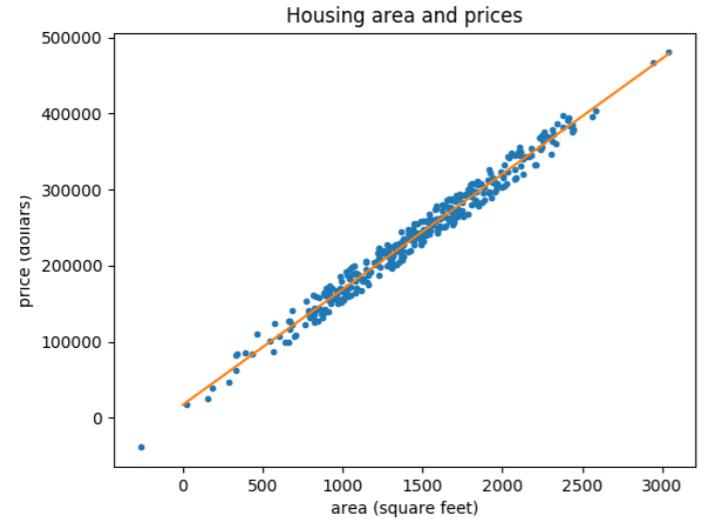


# Return values difficult to compute manually

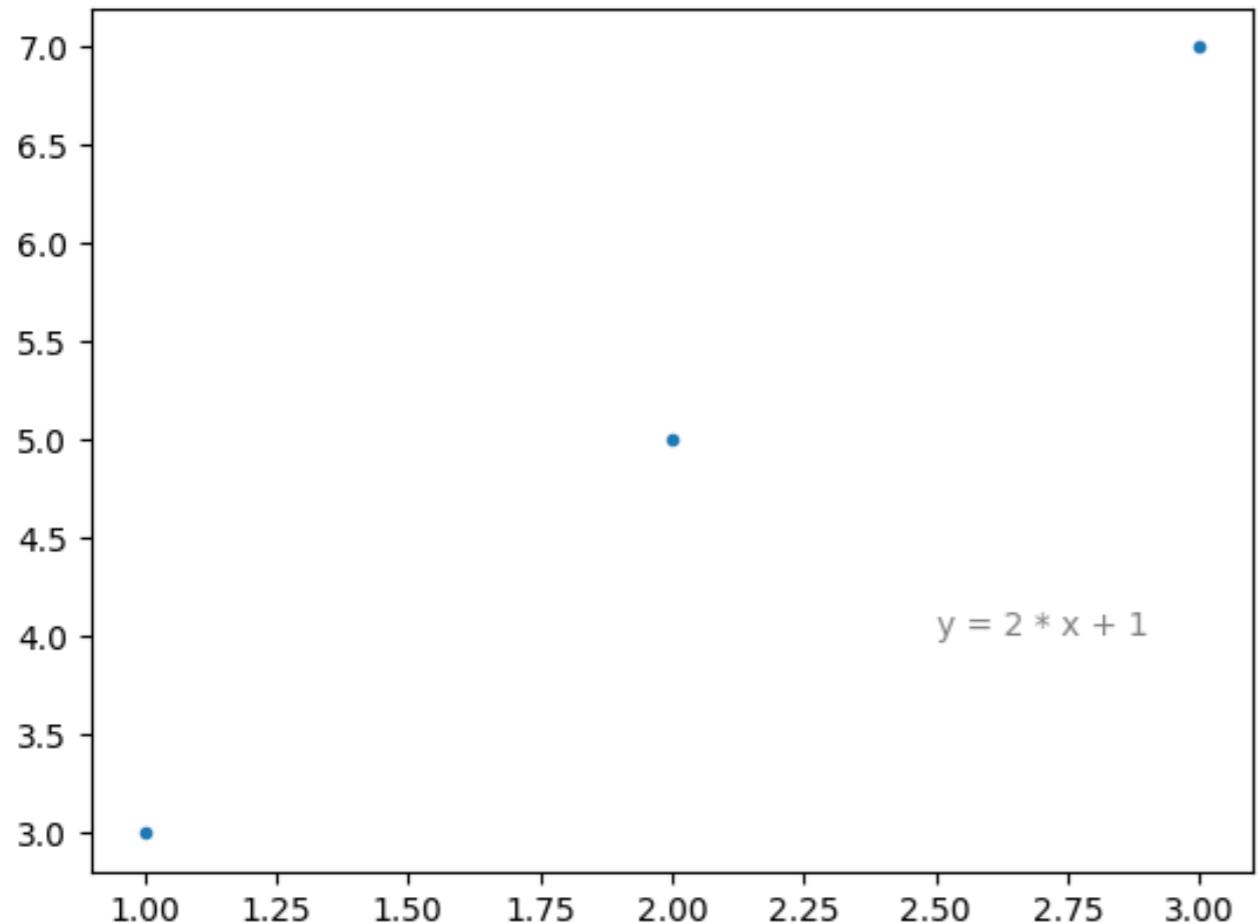


- Cannot test `train_model()` without knowing expected return values.

# True for all data science models



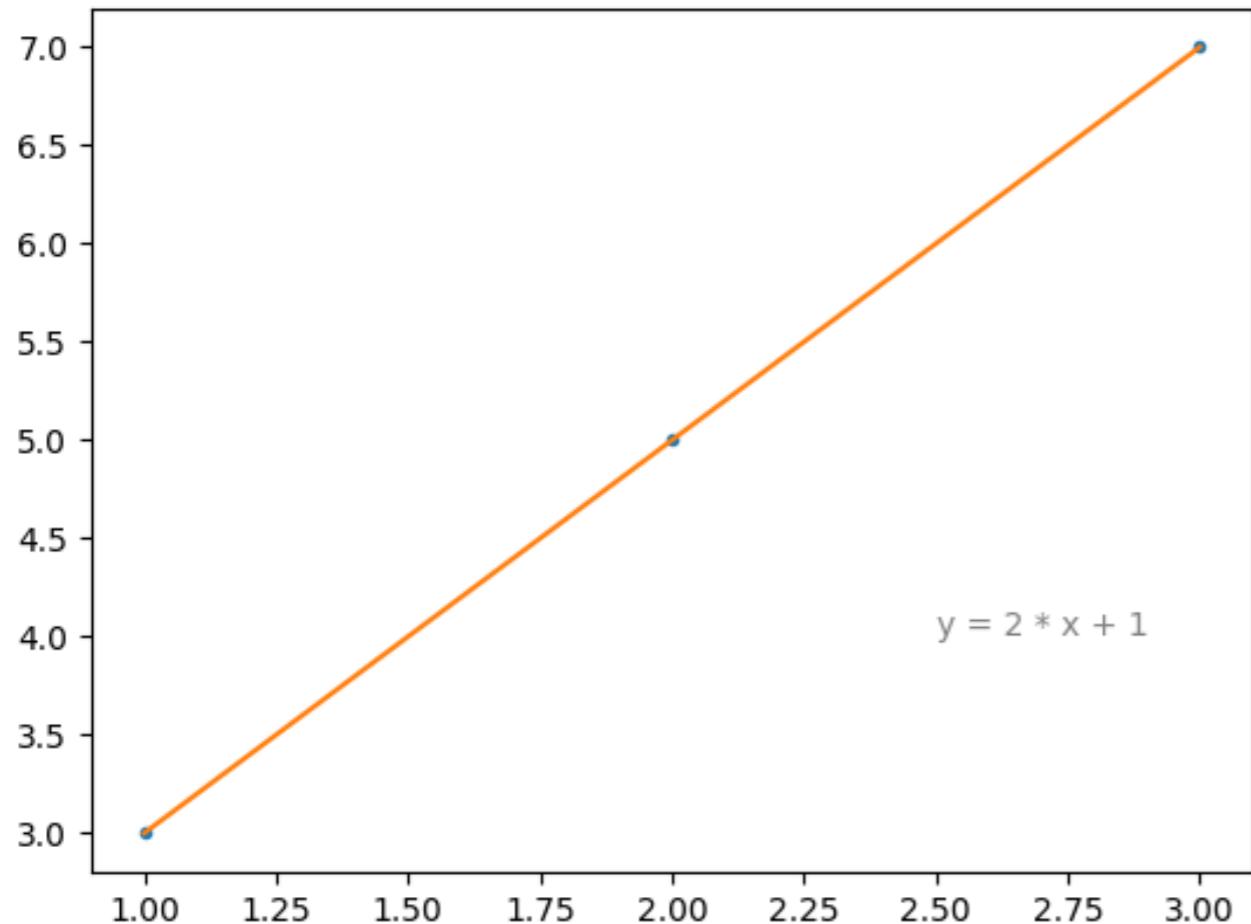
# Trick 1: Use dataset where return value is known



```
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]])
    )
```

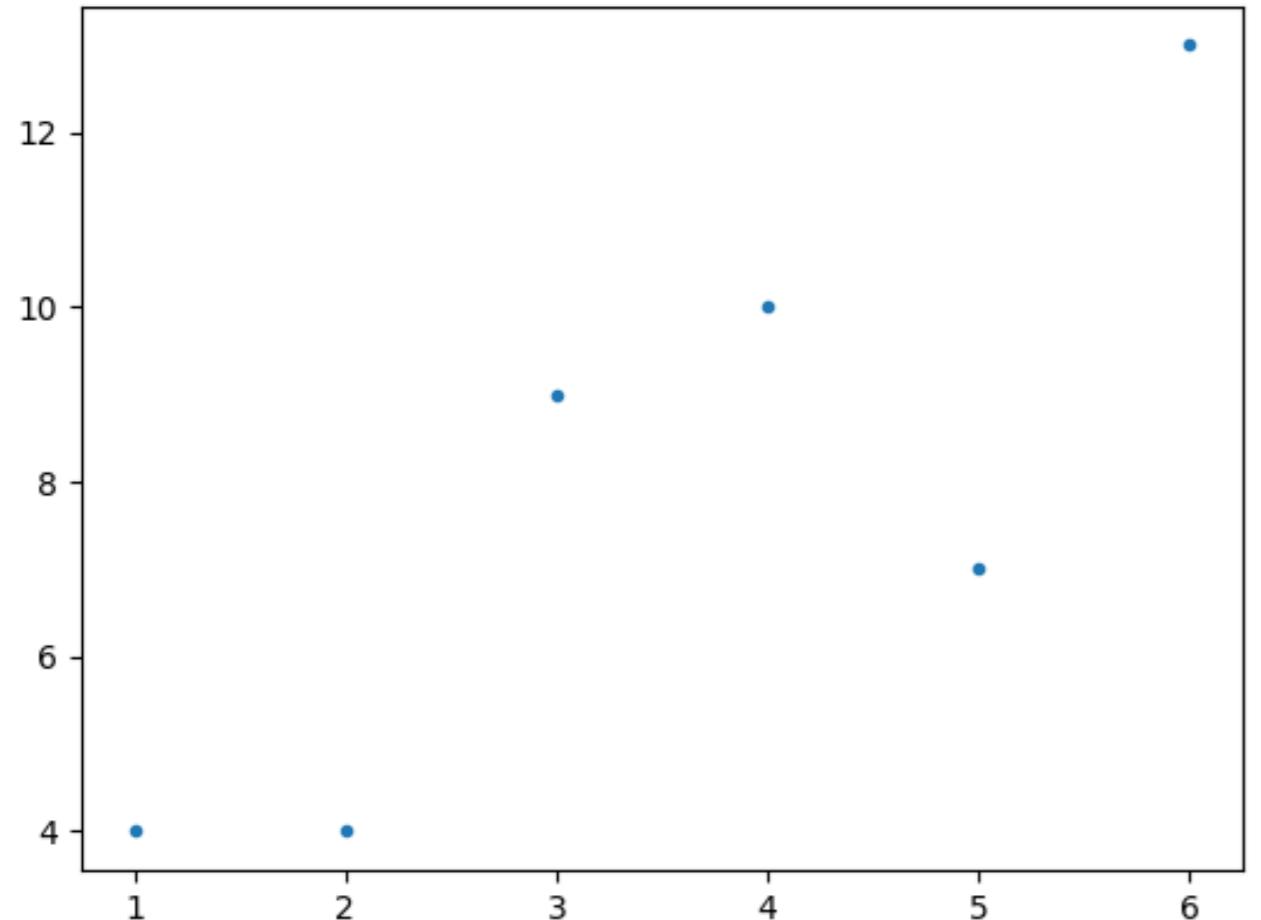
# Trick 1: Use dataset where return value is known



```
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

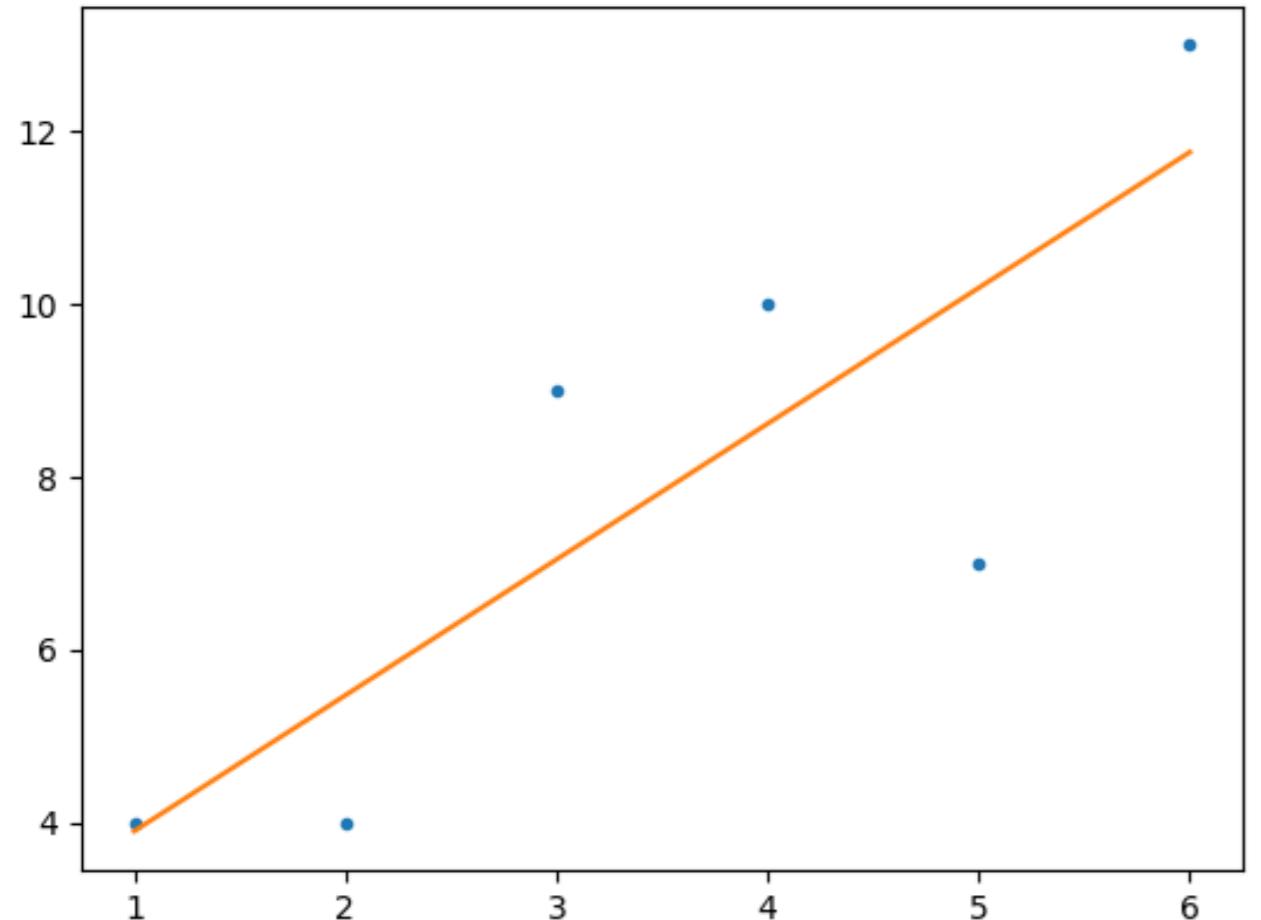


```
import numpy as np
from models.train import train_model

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

# 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

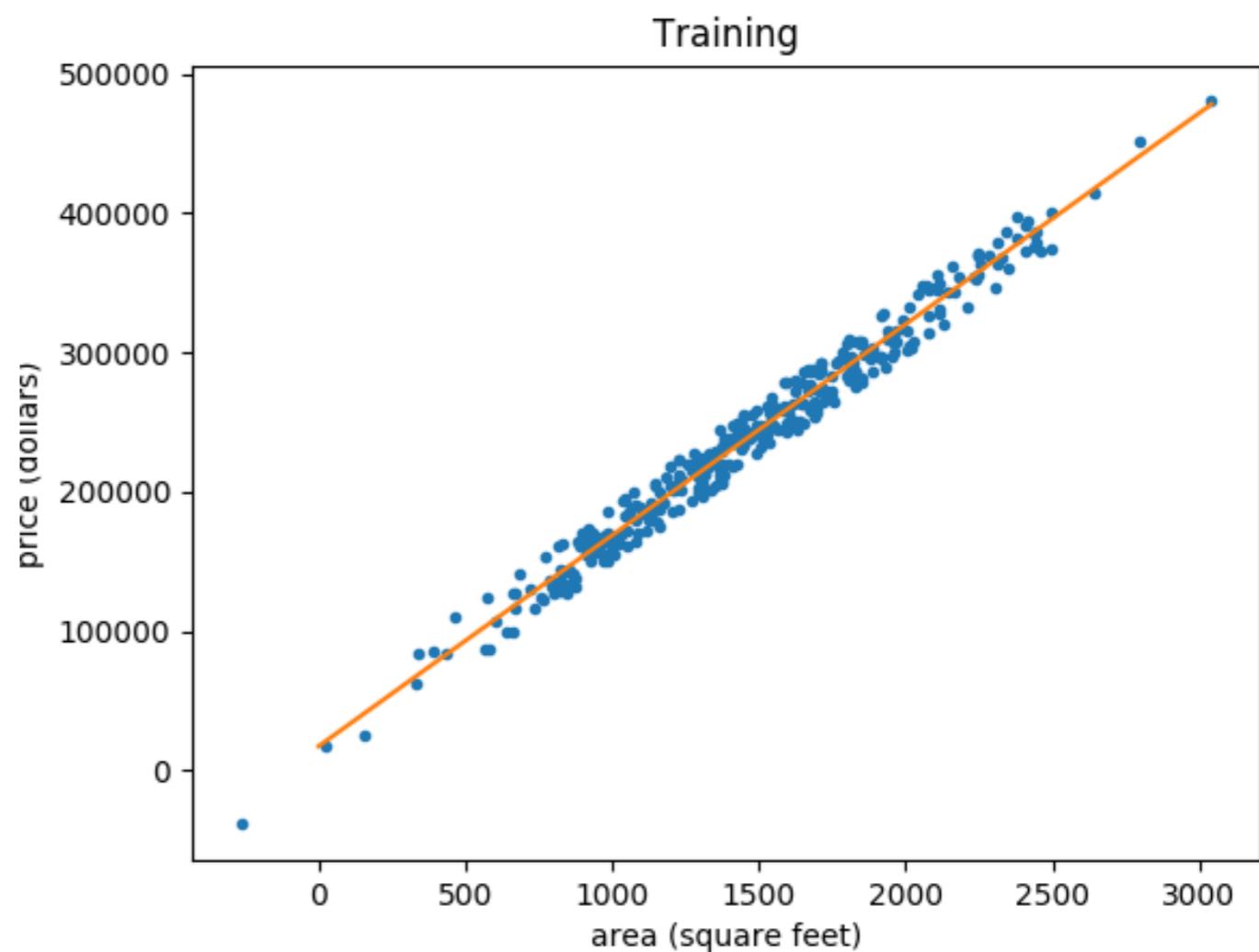
- plots.py

```
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/
```

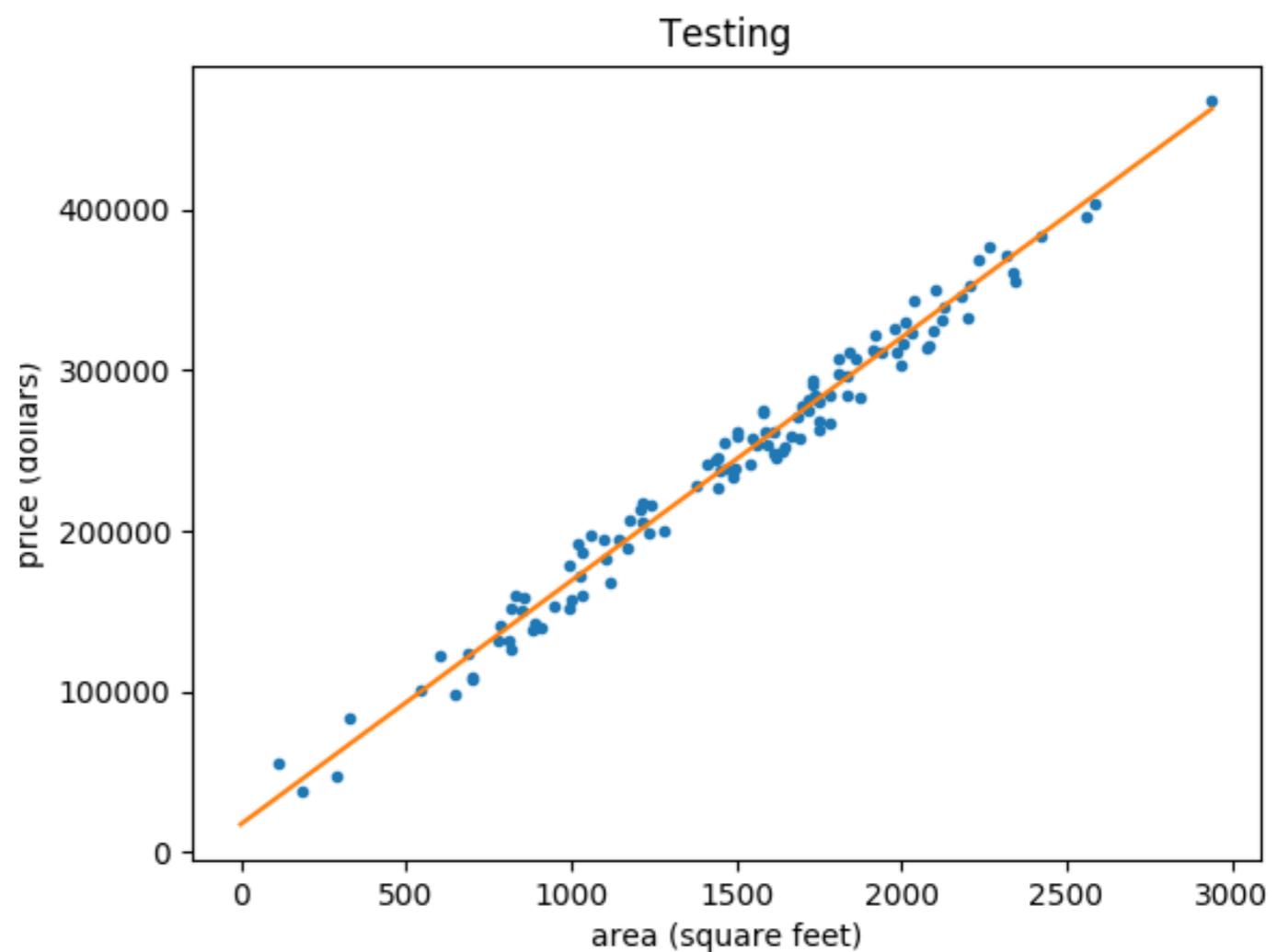
# Training plot

```
...  
from visualization import get_plot_for_best_fit:  
  
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

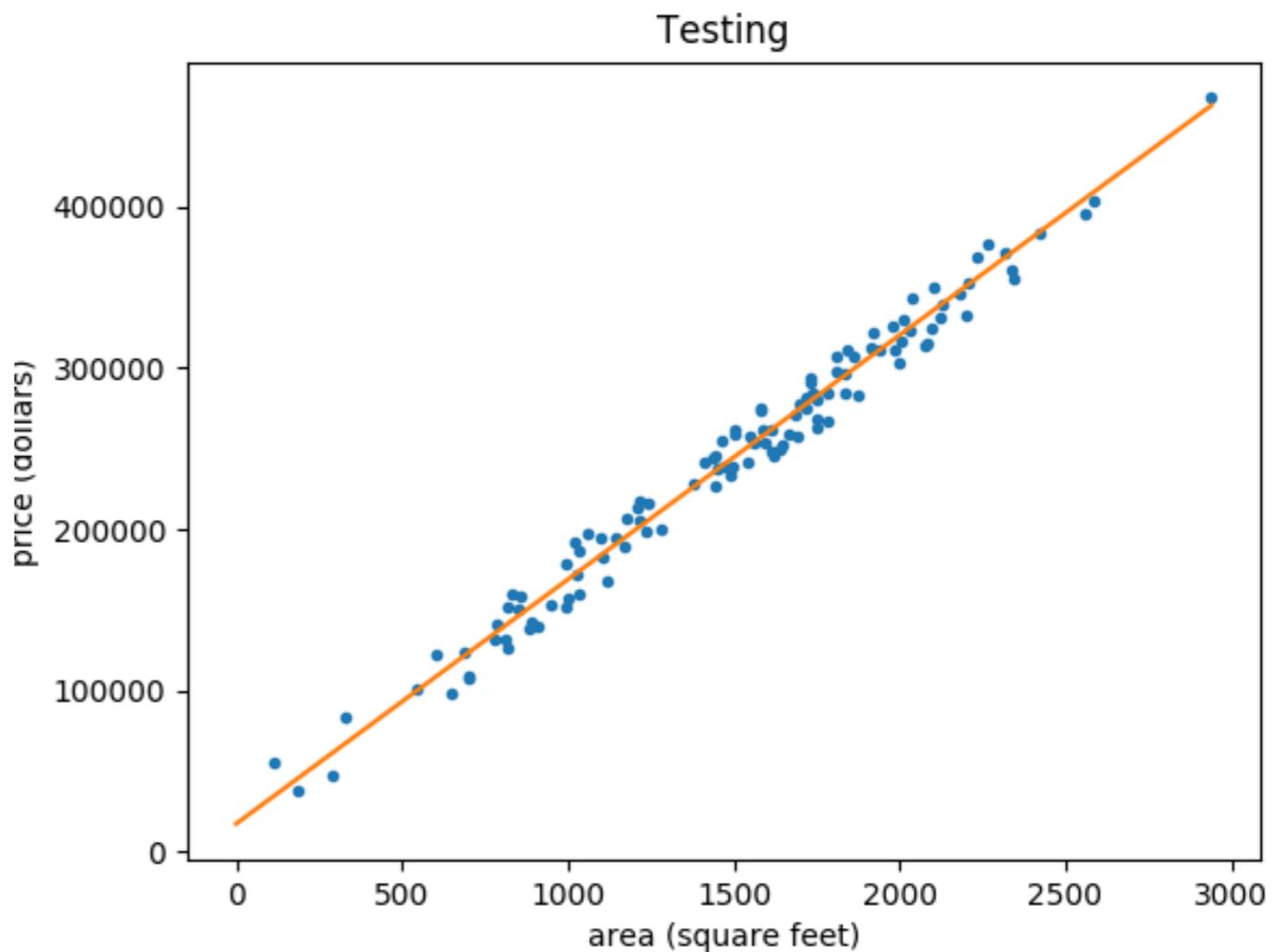
```
...  
from visualization import get_plot_for_best_fit:  
  
preprocess(...)  
data = get_data_as_numpy_array(...)  
training_set, testing_set = (  
    split_into_training_and_testing_sets(data)  
)  
slope, intercept = train_model(training_set)  
get_plot_for_best_fit_line(slope, intercept,  
 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



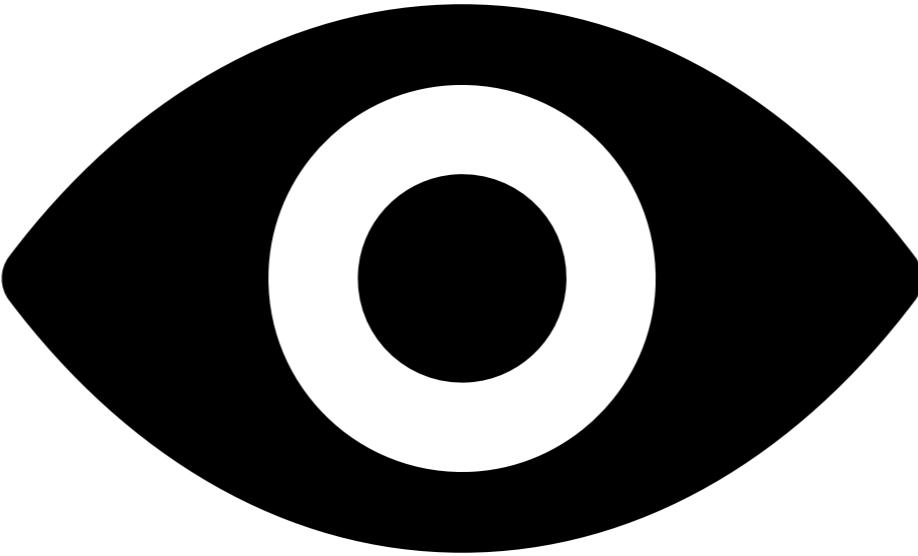
# Don't test properties individually

```
matplotlib.figure.Figure()
```

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



# Testing strategy for plots



# Testing strategy for plots

One-time baseline  
generation

Testing

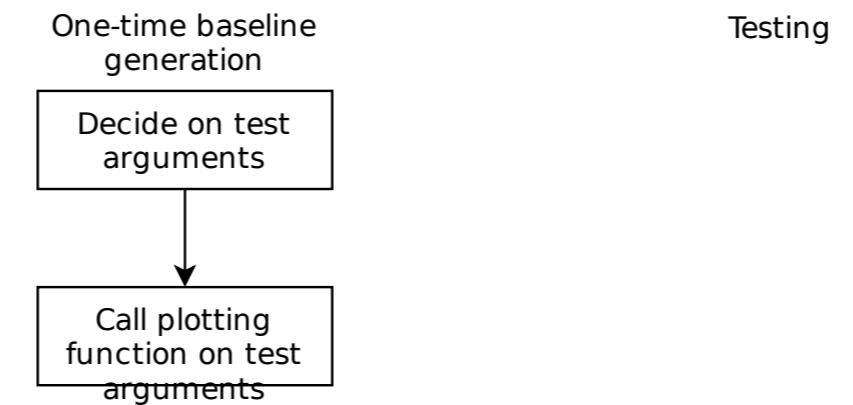
# One-time baseline generation

One-time baseline  
generation

Decide on test  
arguments

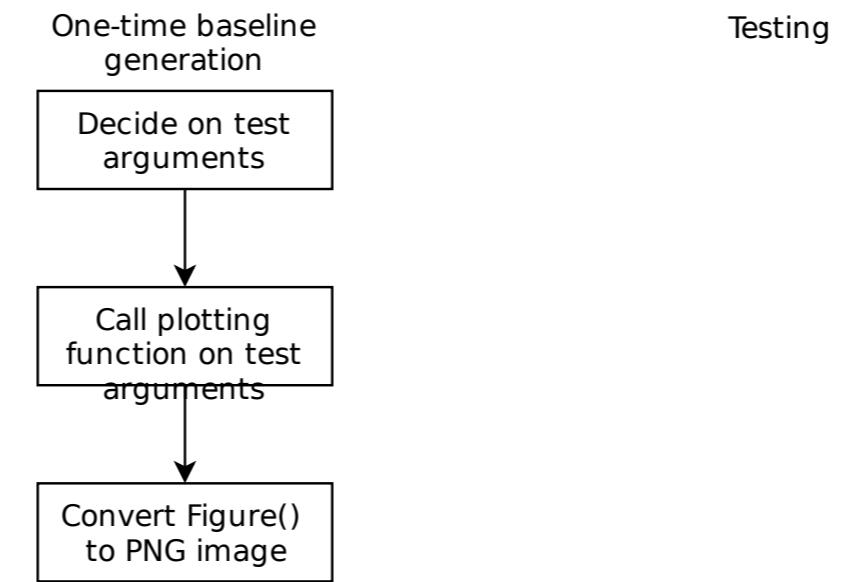
Testing

# One-time baseline generation

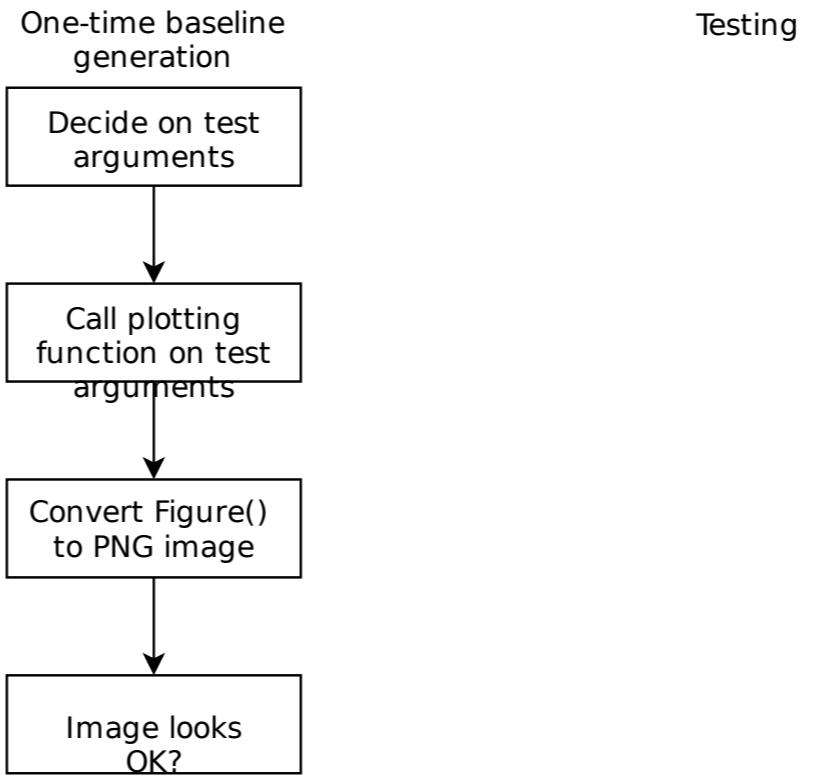


Testing

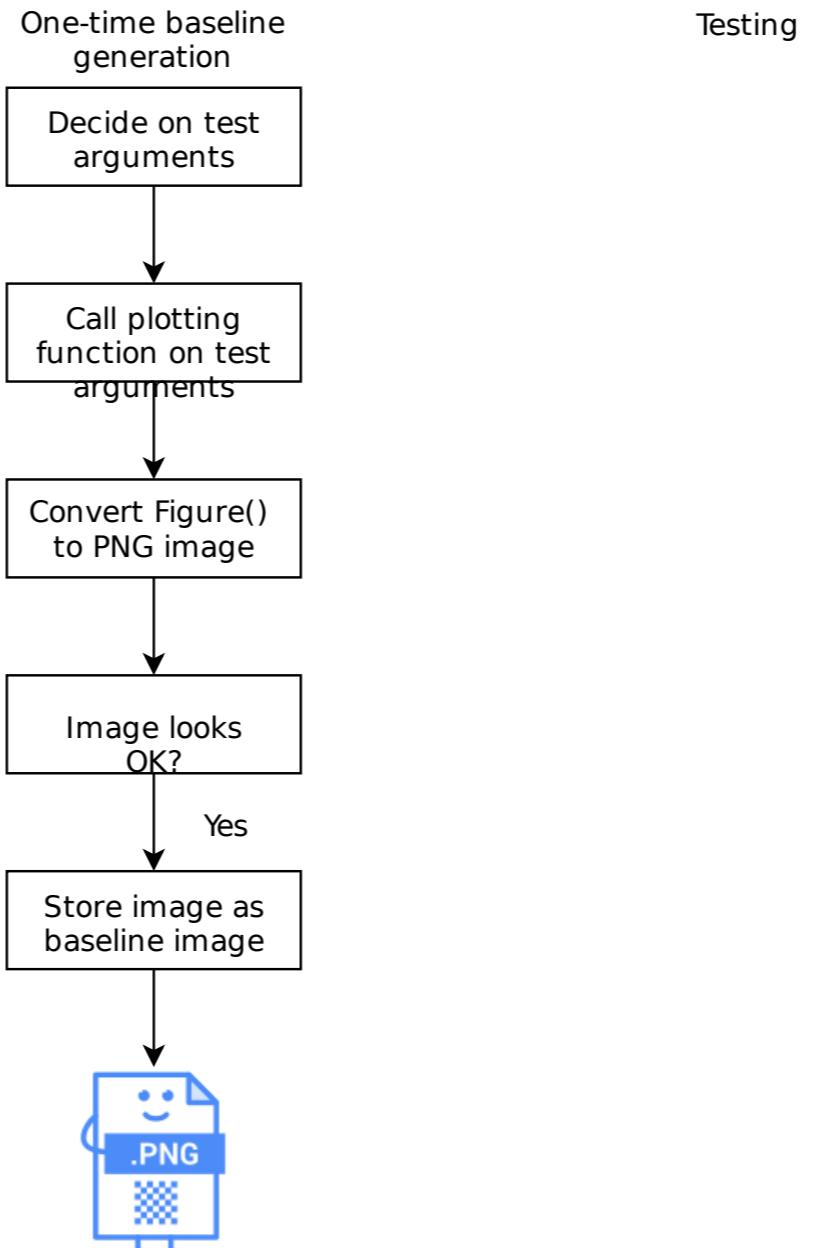
# One-time baseline generation



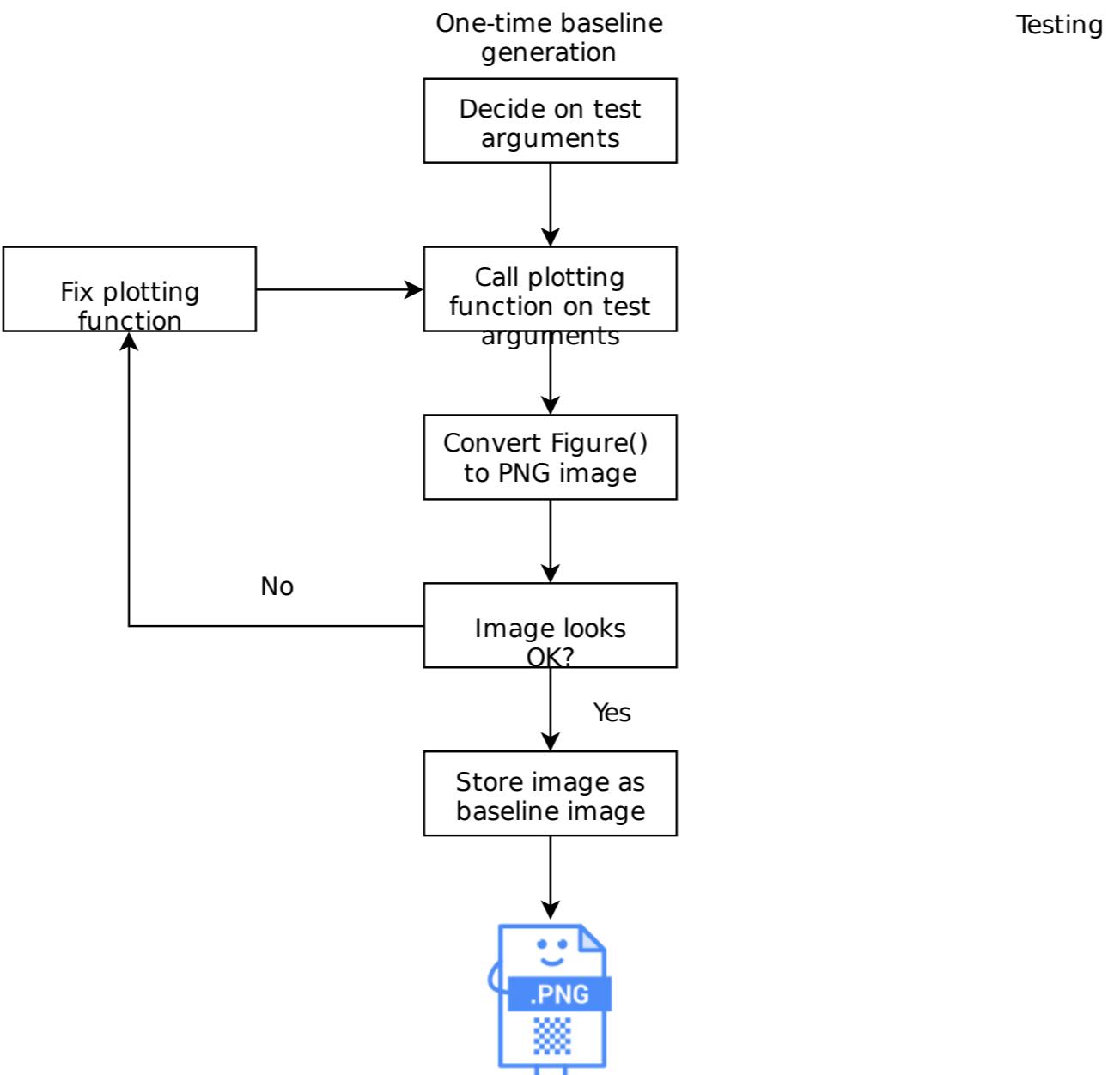
# One-time baseline generation



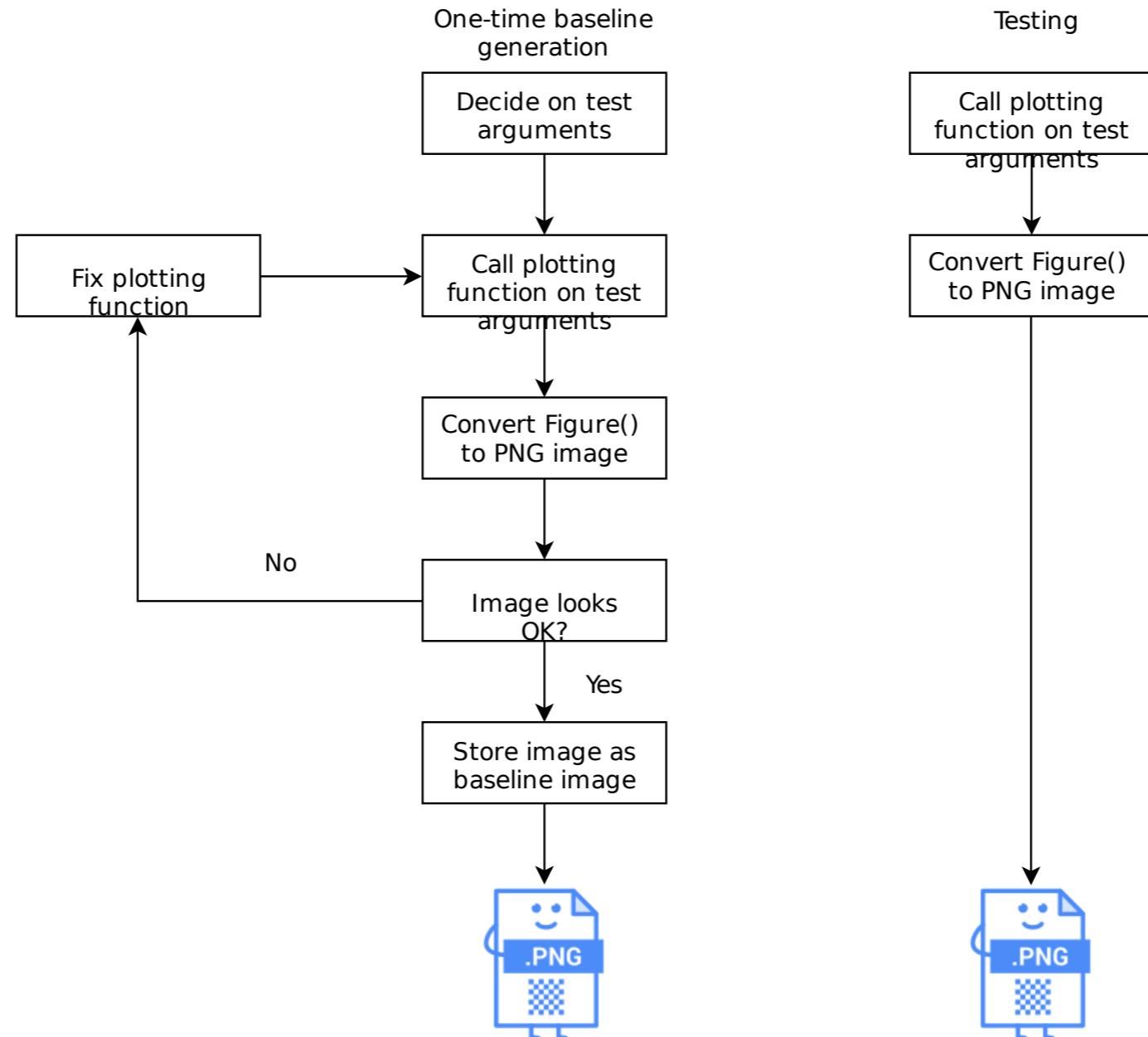
# One-time baseline generation



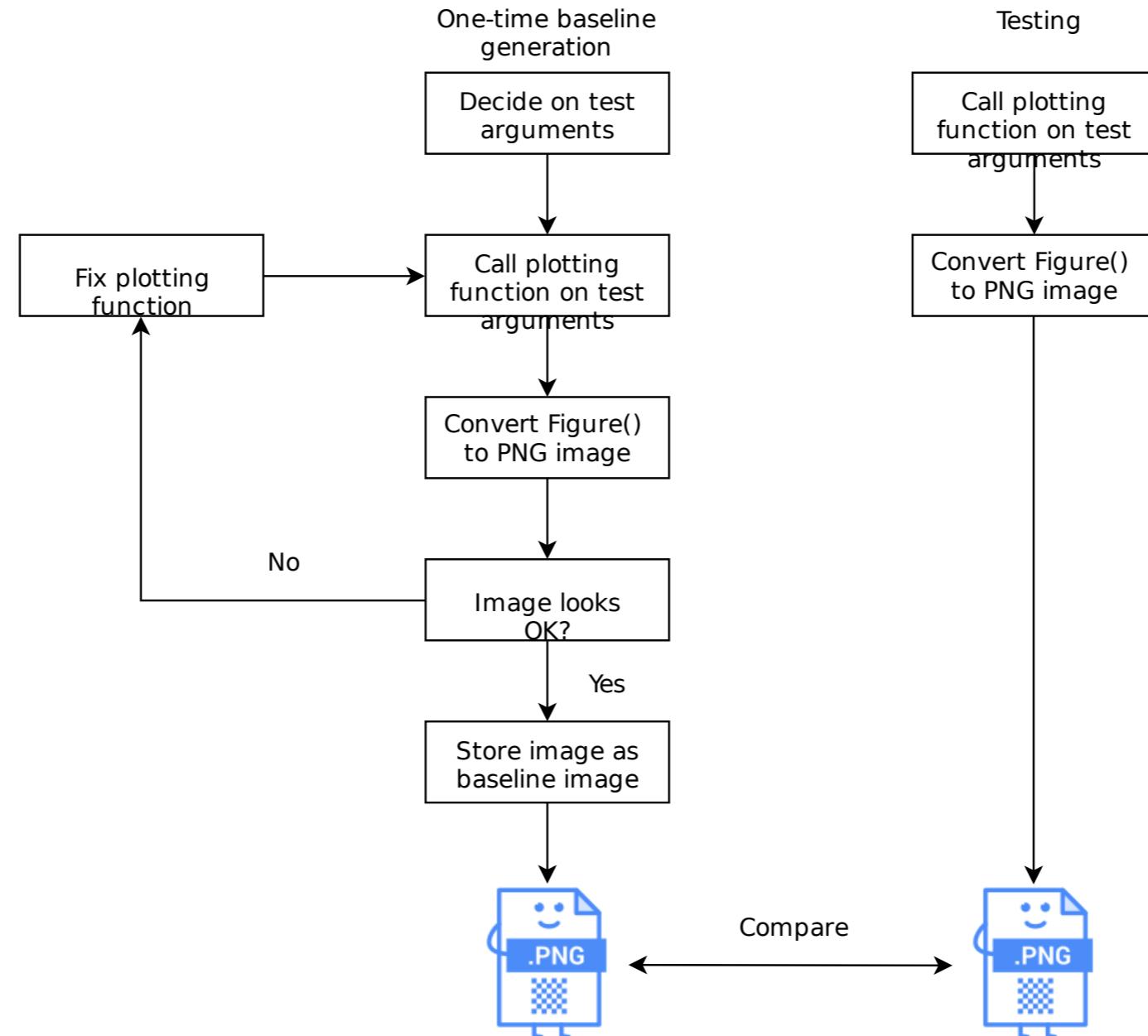
# One-time baseline generation



# Testing



# Testing



# 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)
```

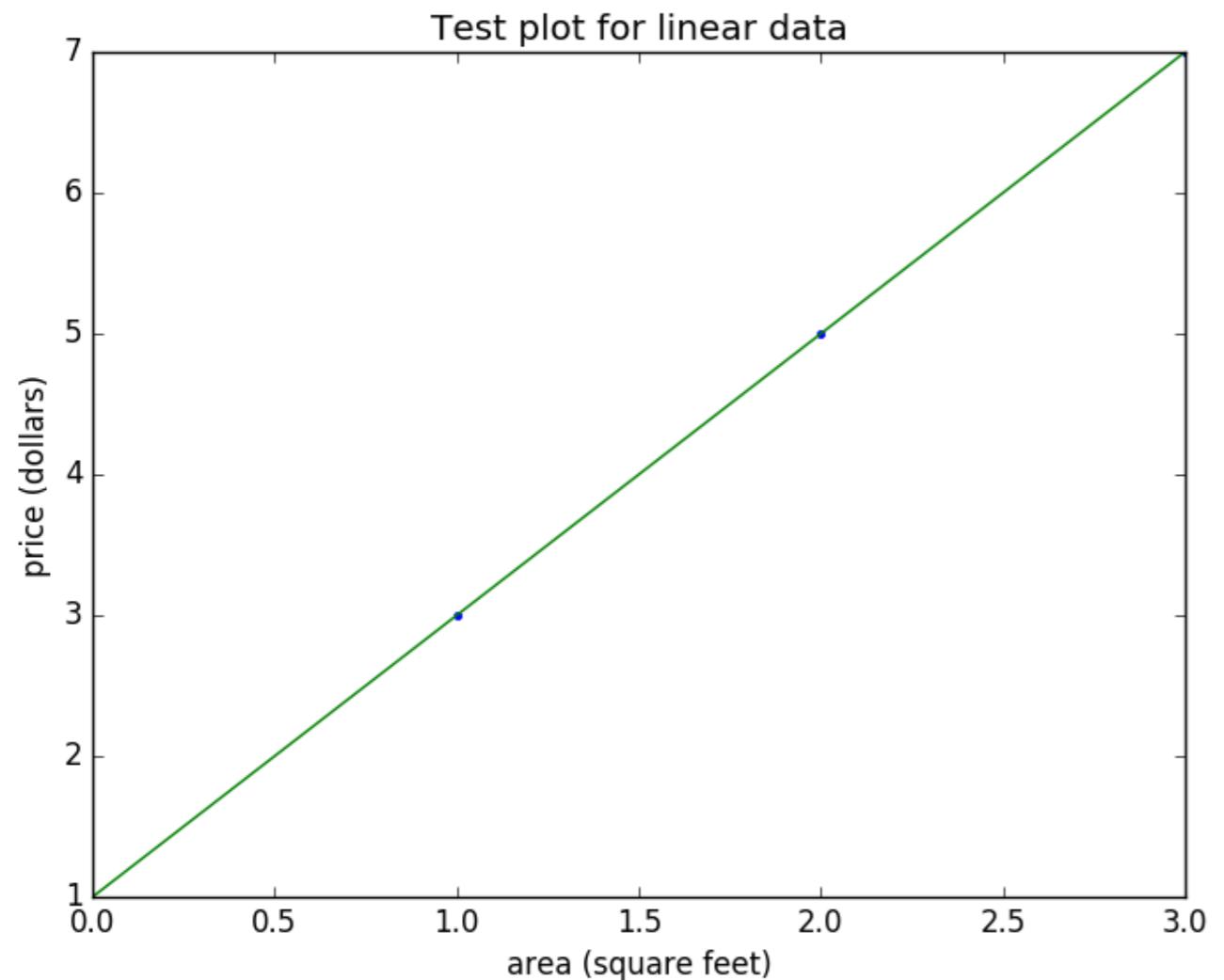
# Generating the baseline image

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/tmplcbs10/baseline-test_plot_for_linear_data.png
Actual:
/tmp/tmplcbs10/test_plot_for_linear_data.png
Difference:
/tmp/tmplcbs10/test_plot_for_linear_data-failed-diff.png
Tolerance:
2
===== 1 failed, 36 deselected in 1.13 seconds =====
```

# Yummy!

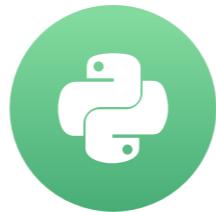


# Let's test plots!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

# Congratulations

UNIT TESTING FOR DATA SCIENCE IN PYTHON



**Dibya Chakravorty**  
Test Automation Engineer



# 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

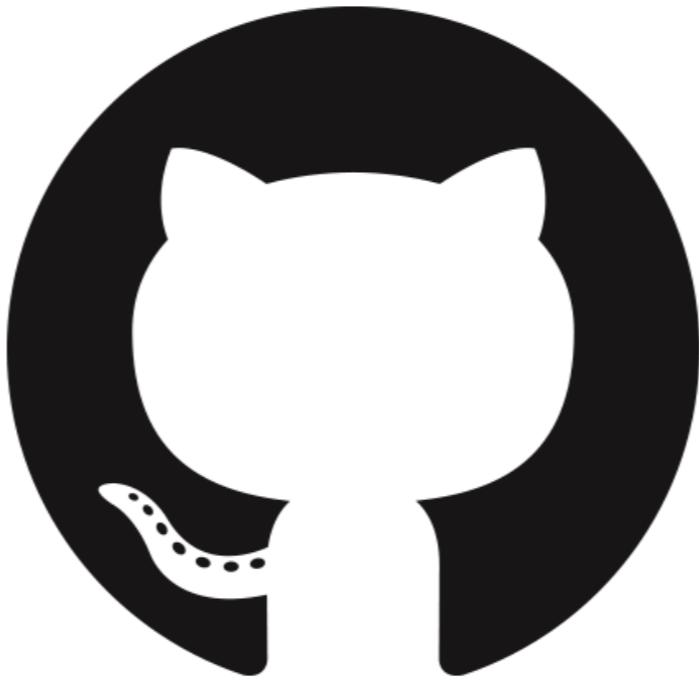
25

# 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>



# Icon sources

Icons made by the following authors from [flaticon.com](https://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