https://www.youtube.com/watch?v=bbp_849-RZ4

Why unit test?

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

How can we test an implementation?

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

2. How can we test an implementation?
Consider this question. Suppose we have just implemented a Python function. How can we test whether our implementation is correct? The easiest way is to open an interpreter, test the function on a few arguments and check whether the return value is correct. If correct, we can accept the implementation and move on. Right? While testing on the interpreter is easy, it is actually very inefficient. This will become clear if we think about the big picture of a function's life cycle in a data science project.

```python
my_function(argument_1)
```

```
return_value_1
```

```python
my_function(argument_2)
```

```
return_value_2
```

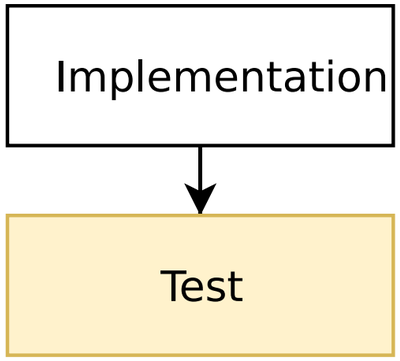
```python
my_function(argument_3)
```

```
return_value_3
```

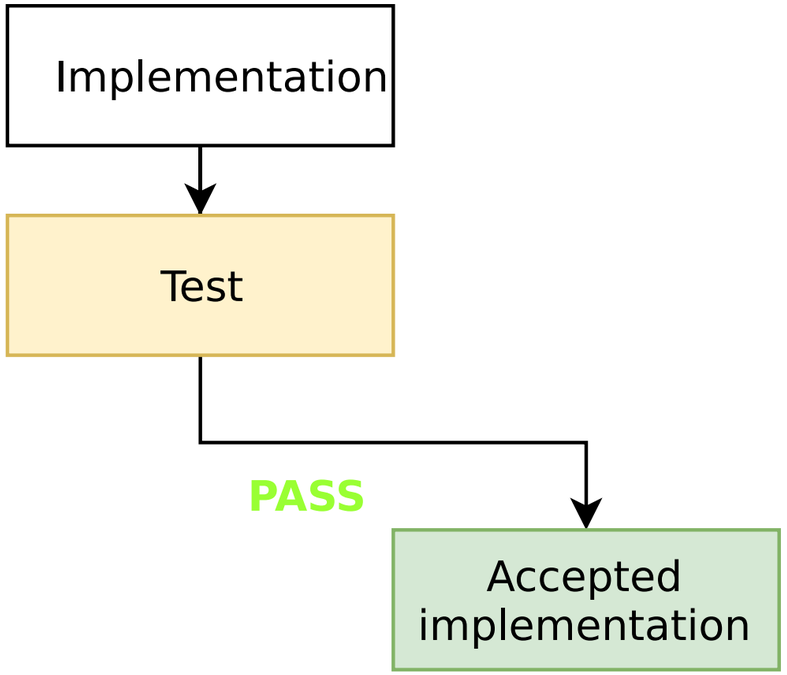
Life cycle of a function

Implementation

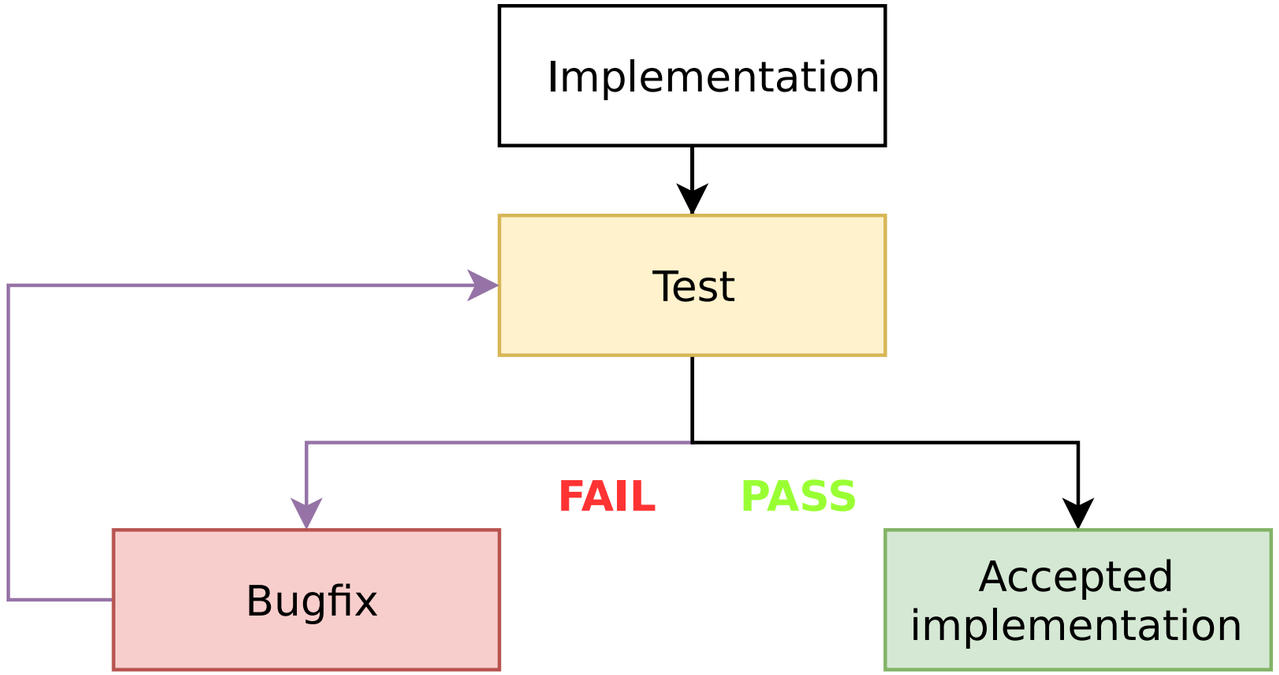
Life cycle of a function



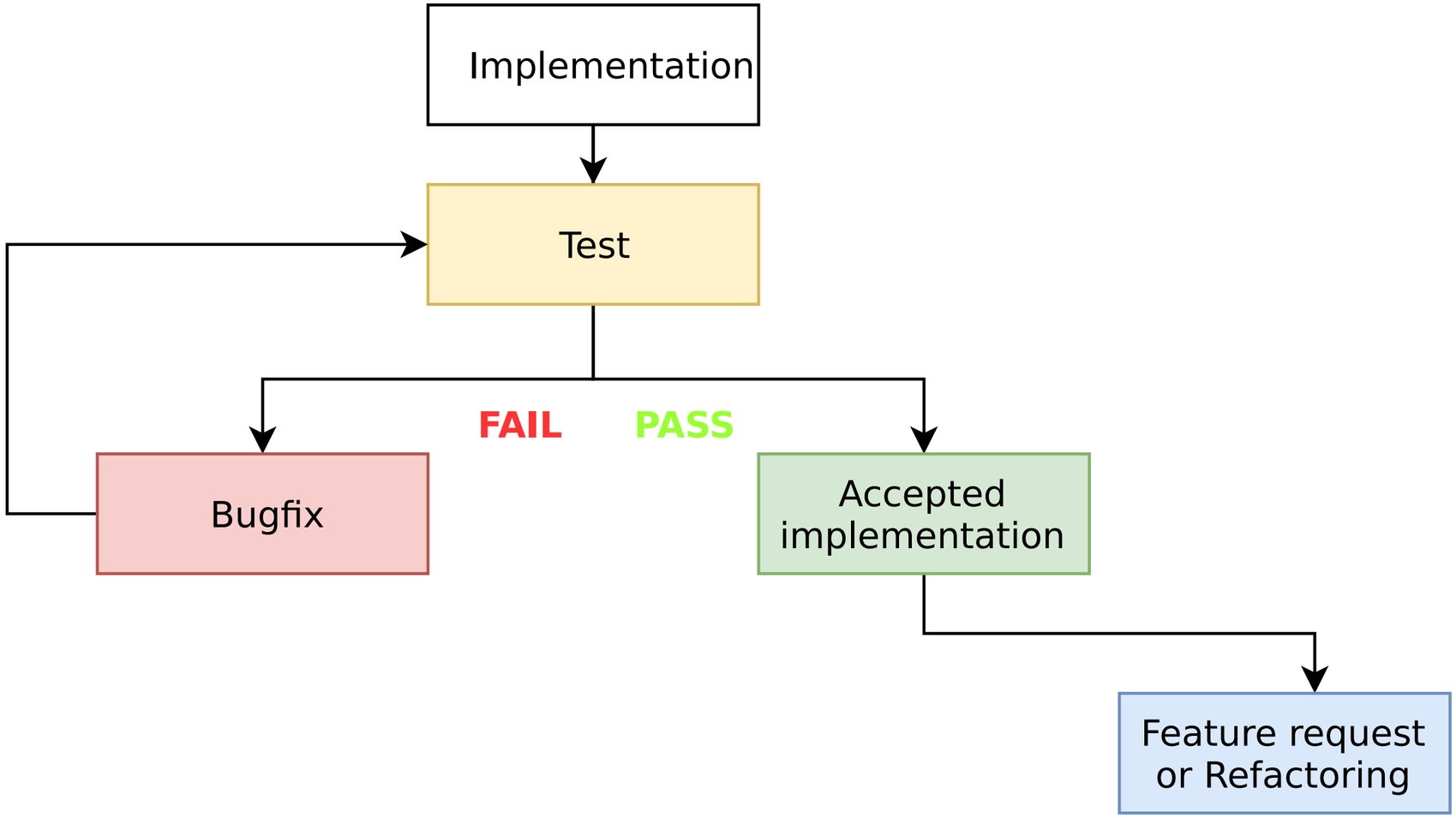
Life cycle of a function



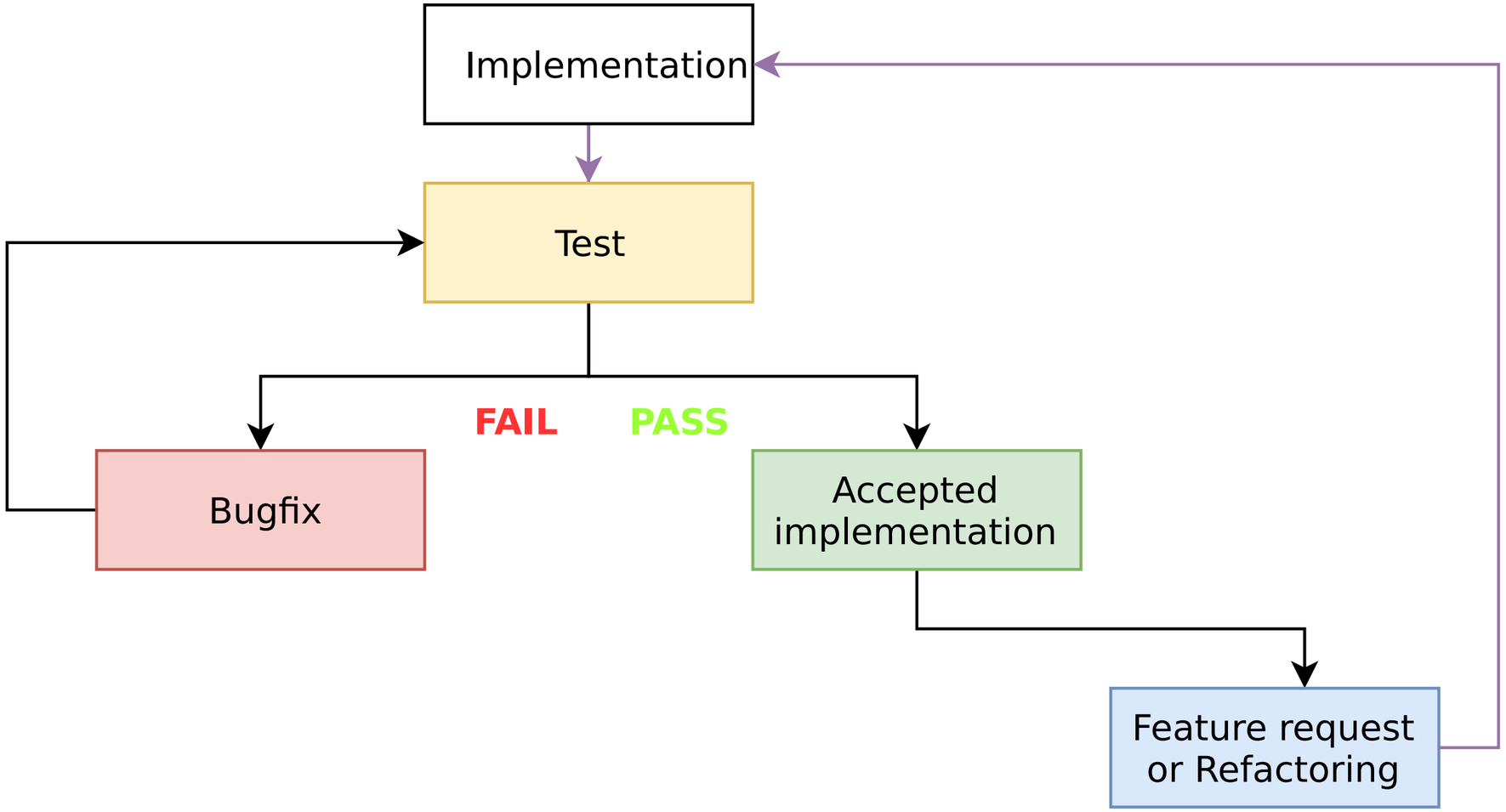
Life cycle of a function



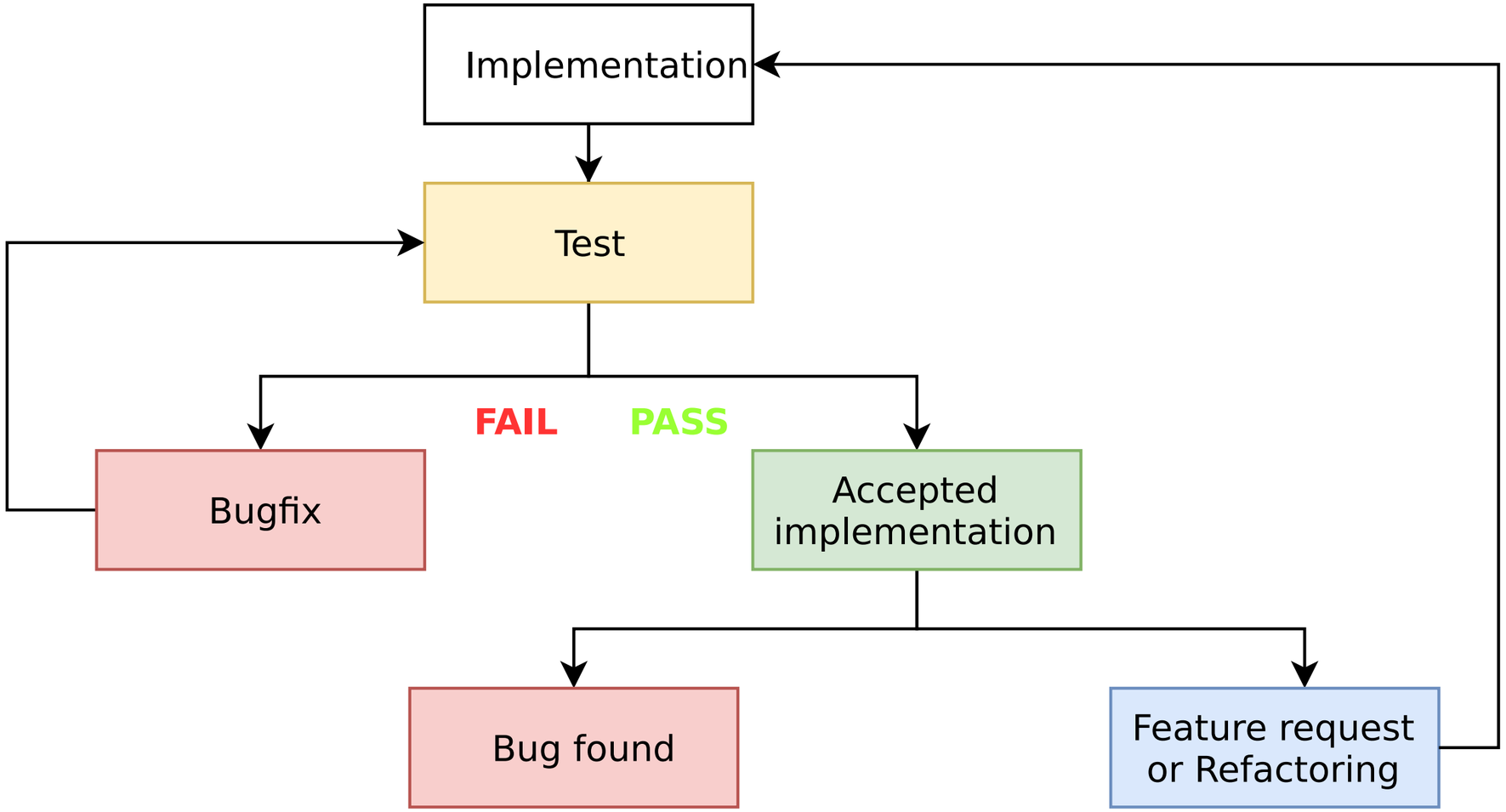
Life cycle of a function



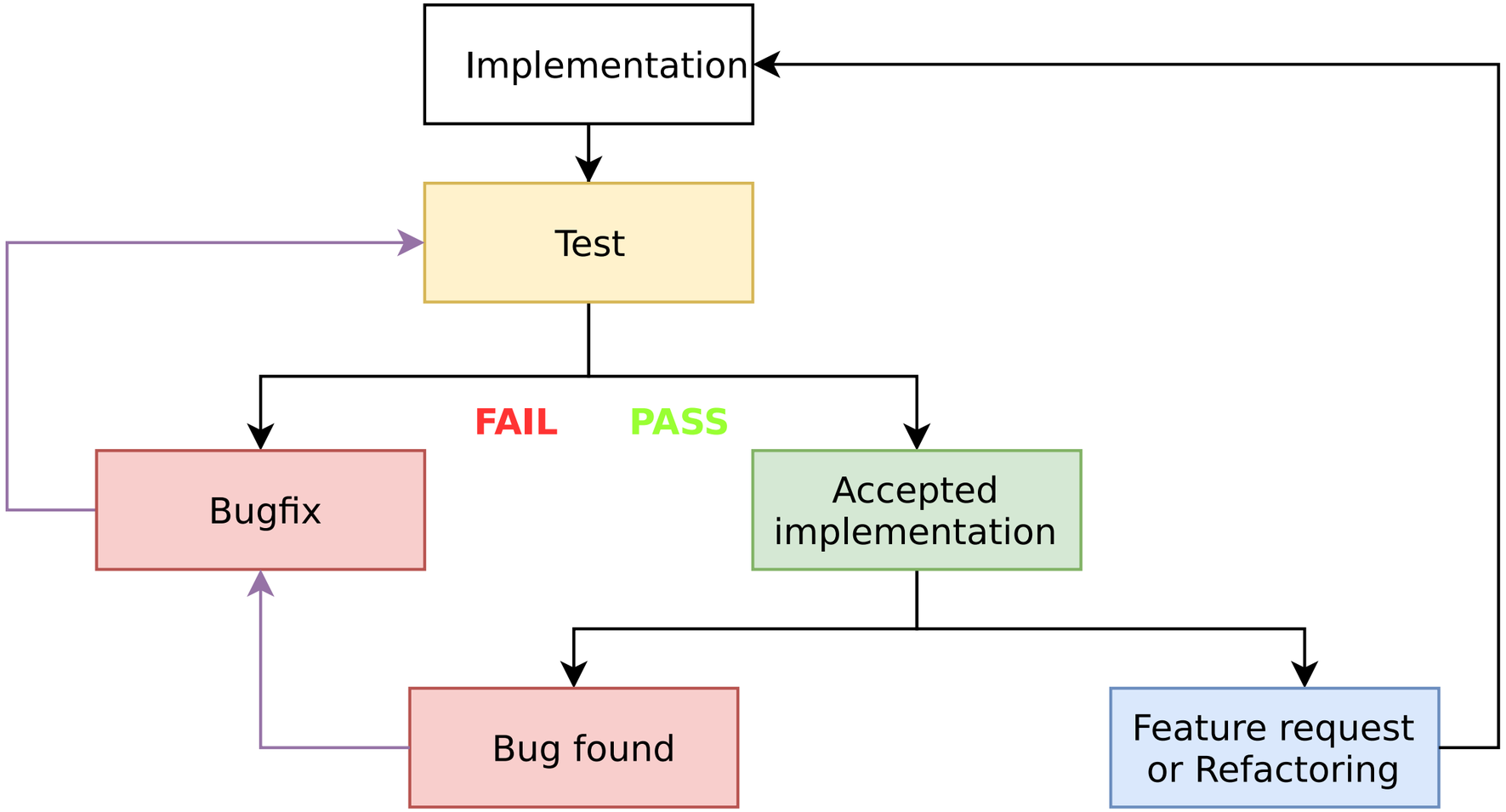
Life cycle of a function



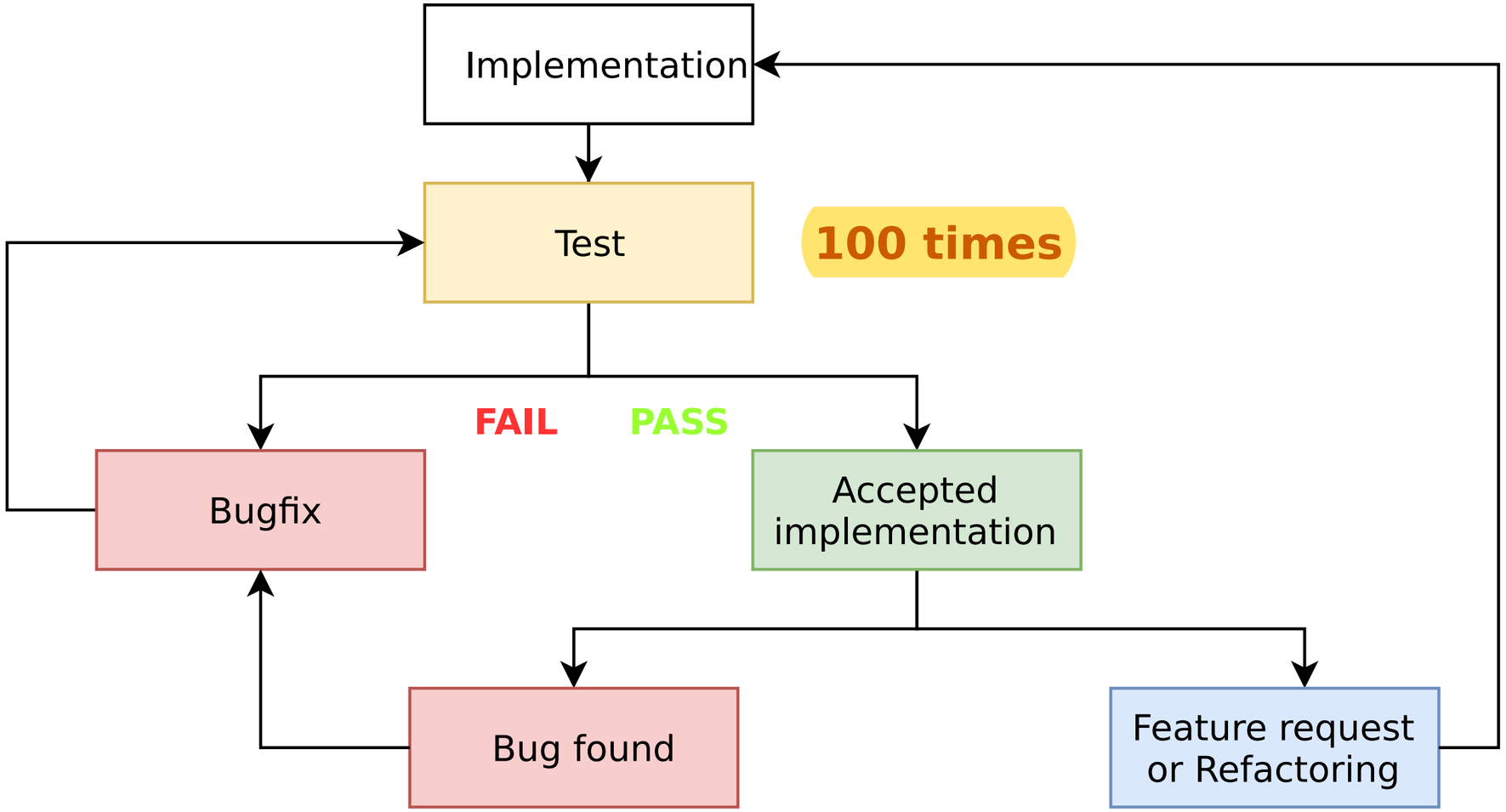
Life cycle of a function



Life cycle of a function



Life cycle of a function



Example

```python
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

```python
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

```python
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`

14. Data isn't clean
But this data file is not clean, and some rows in this data file do not follow this format. The third row has missing area,

```
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

```python
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`

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

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

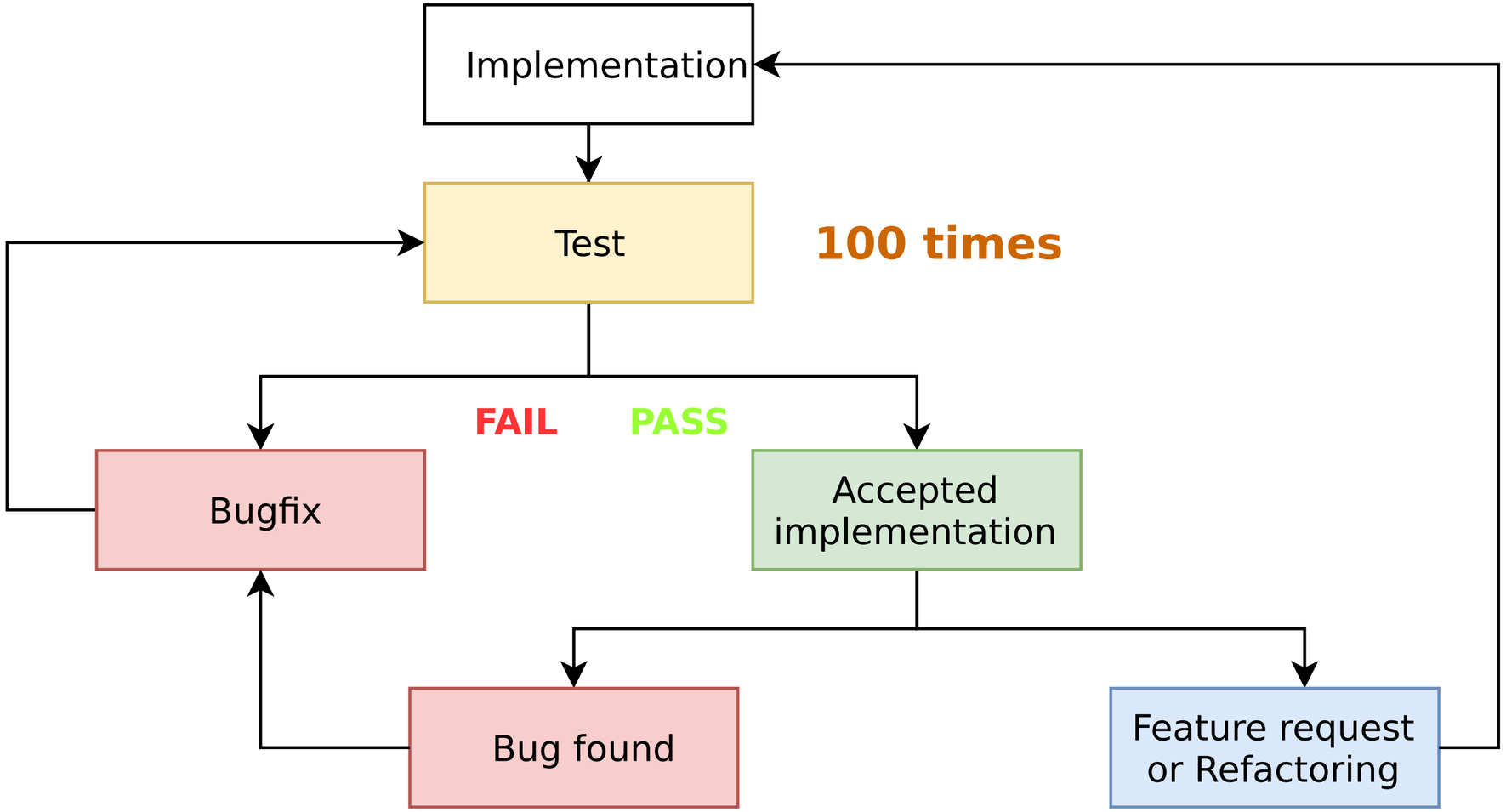
```python
row_to_list("\t293,410\n")
```

```
None
```

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

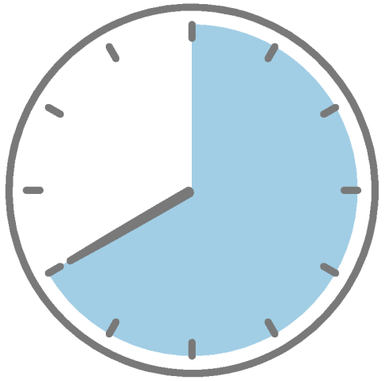
```
None
```

Time spent in testing this function



Time spent in testing this function

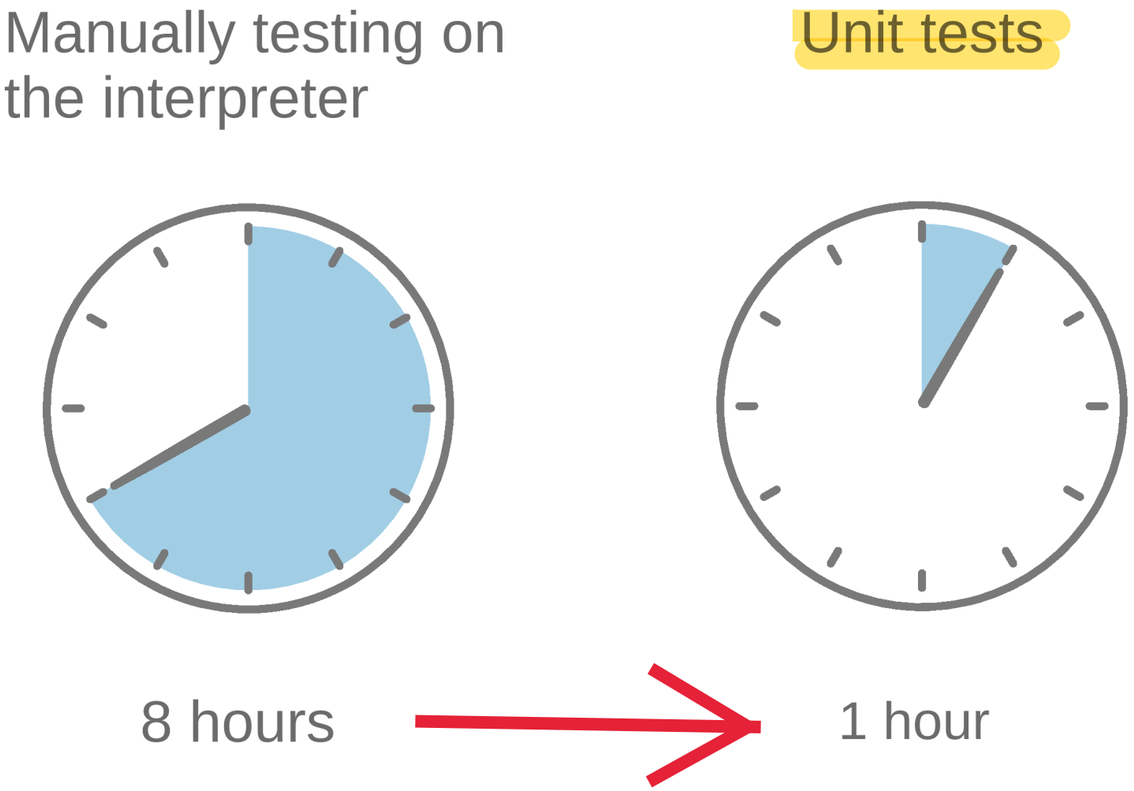
5 mins x 100 ~



8 hours

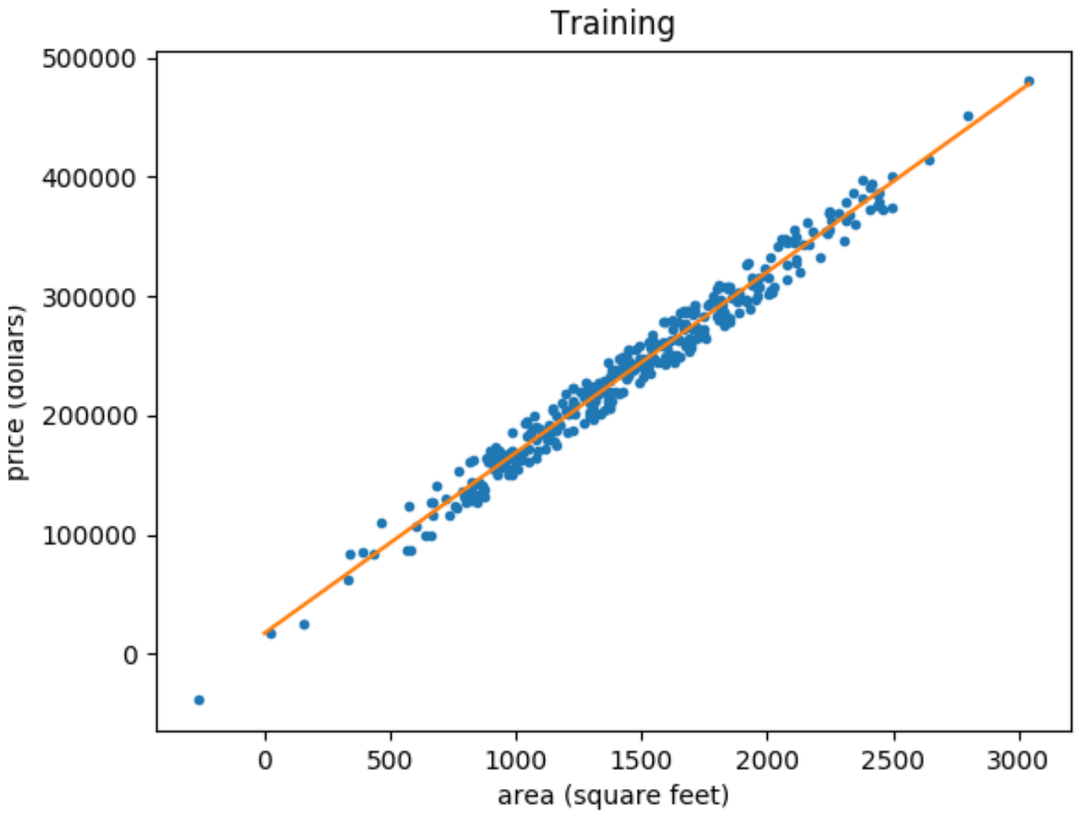
Manual testing vs. unit tests

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



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
1,463238,765
1,468    239,007
```



Linear regression of housing price against area

GitHub repository of the course

- Implementation of functions like `row_to_list()`.

21. GitHub repository of the course
The complete code for this project, including actual implementations for functions like row_to_list(), is available in a public GitHub repository. We will share the link to the repository at the end of the course.

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/
```

23. Develop a complete unit test suite
During this course, we are going to write a complete unit test suite for this example project. It's going to be fun and exciting. This will prepare you to write unit tests for your own projects.

- 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

datacamp

Testing on the console

2. Testing on the console
At every step, you tested it by calling row_to_list() on different arguments and checked if the return values are correct. This was repetitive, tedious and time consuming. In this lesson, we will learn to write unit tests and improve this process.

```
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.**

Step 1: Create a file

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

4. Step 1: Create a file
To start unit testing with pytest, we will first create a file called test_row_to_list.py. When pytest sees a filename starting with "test_", it understands that this is not an usual Python file, but a special one containing unit tests. We must make sure to follow this naming convention. Files holding unit tests are also called test modules, and we just created our first test module.

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():
```

6. Step 3: Unit tests are Python functions
A unit test is written as a Python function, whose name starts with a "test_", just like the test module. This way, pytest can tell that it is a unit test and not an ordinary function.

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"]`

7. Step 3: Unit tests are Python functions
The unit test usually corresponds to exactly one entry in the argument and return value table for row_to_list(). The unit test checks whether row_to_list() has the expected return value when called on this particular argument. This particular argument is a clean row, so we call the unit test test_for_clean_row().

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"]`

8. Step 4: Assertion
The actual check is done via an assert statement, and every test must contain one.

Theoretical structure of an assertion

```
assert boolean_expression
```

```
assert True
```

9. Theoretical structure of an assertion
The assert statement has a required first argument, which can be any boolean expression. If the expression is True, the assert statement passes, giving us a blank output. If the expression is False, it raises an AssertionError.

```
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"]`

10. Step 4: Assertion
In this case, we want to check if row_to_list() returns the correct list when called on the clean row. The expression we use is row_to_list() called on the argument equal equals the correct list. If the function works, this will evaluate to True and the assert statement will pass. This will make the test pass. If the function has a bug, it will evaluate to False, the assert statement will raise an AssertionError, and the test will fail.

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`

11. A second unit test
For the second row in the table, we create a unit test called test_on_missing_area() because the argument has missing area data. Then we assert that the return value for this argument is None.

Checking for None values

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

12. Checking for None values
Note that the correct way to check if a variable's value is None is to use the boolean expression var is None and not var equal equals 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`

13. A third unit test
For the third row in the table, we create a unit test called test_for_missing_tab(), because the argument is missing the tab separating area and price. The assert statement is similar to the second test.

Step 5: Running unit tests

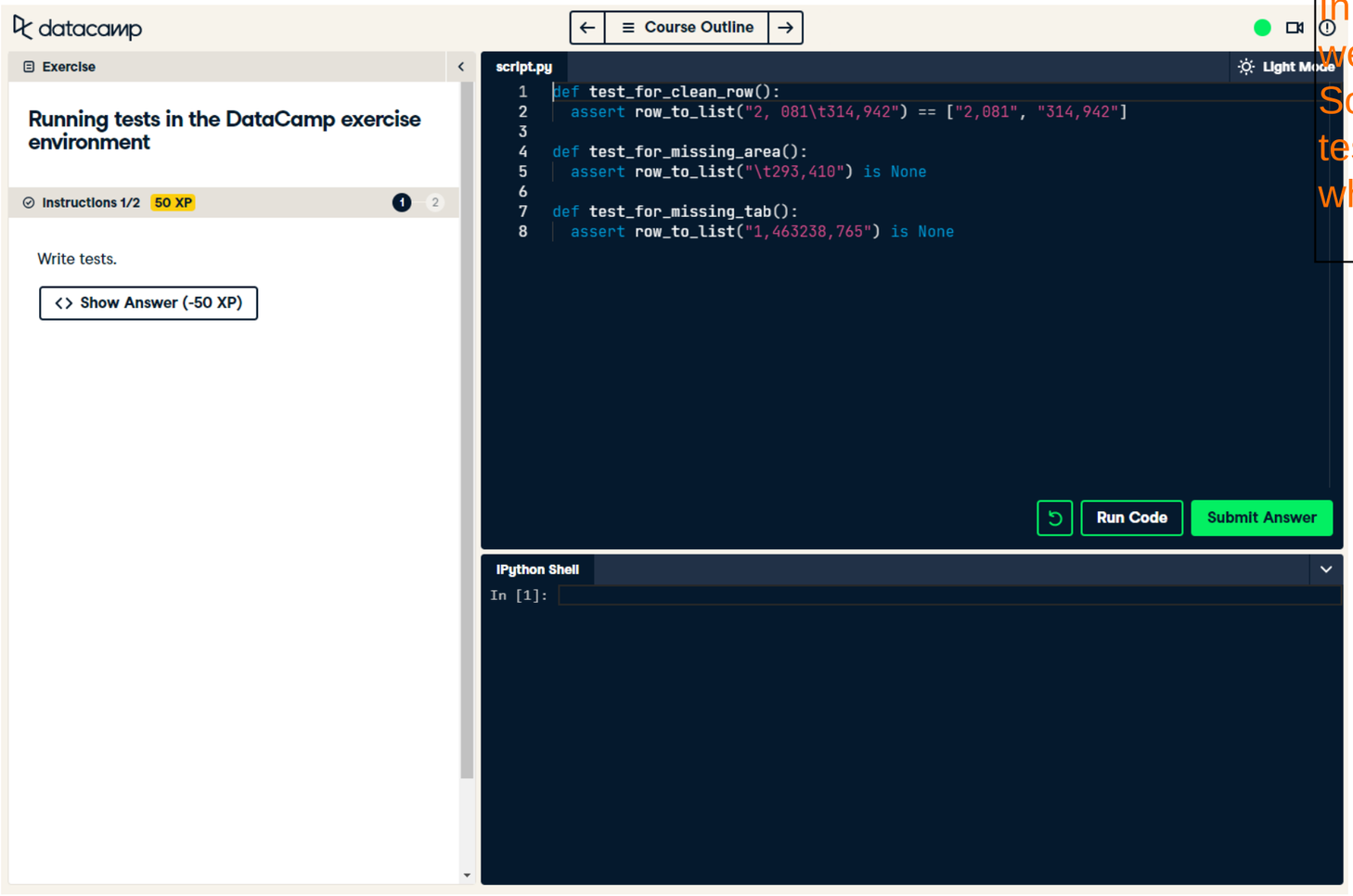
- Do this in the command line.

```
pytest test_row_to_list.py
```

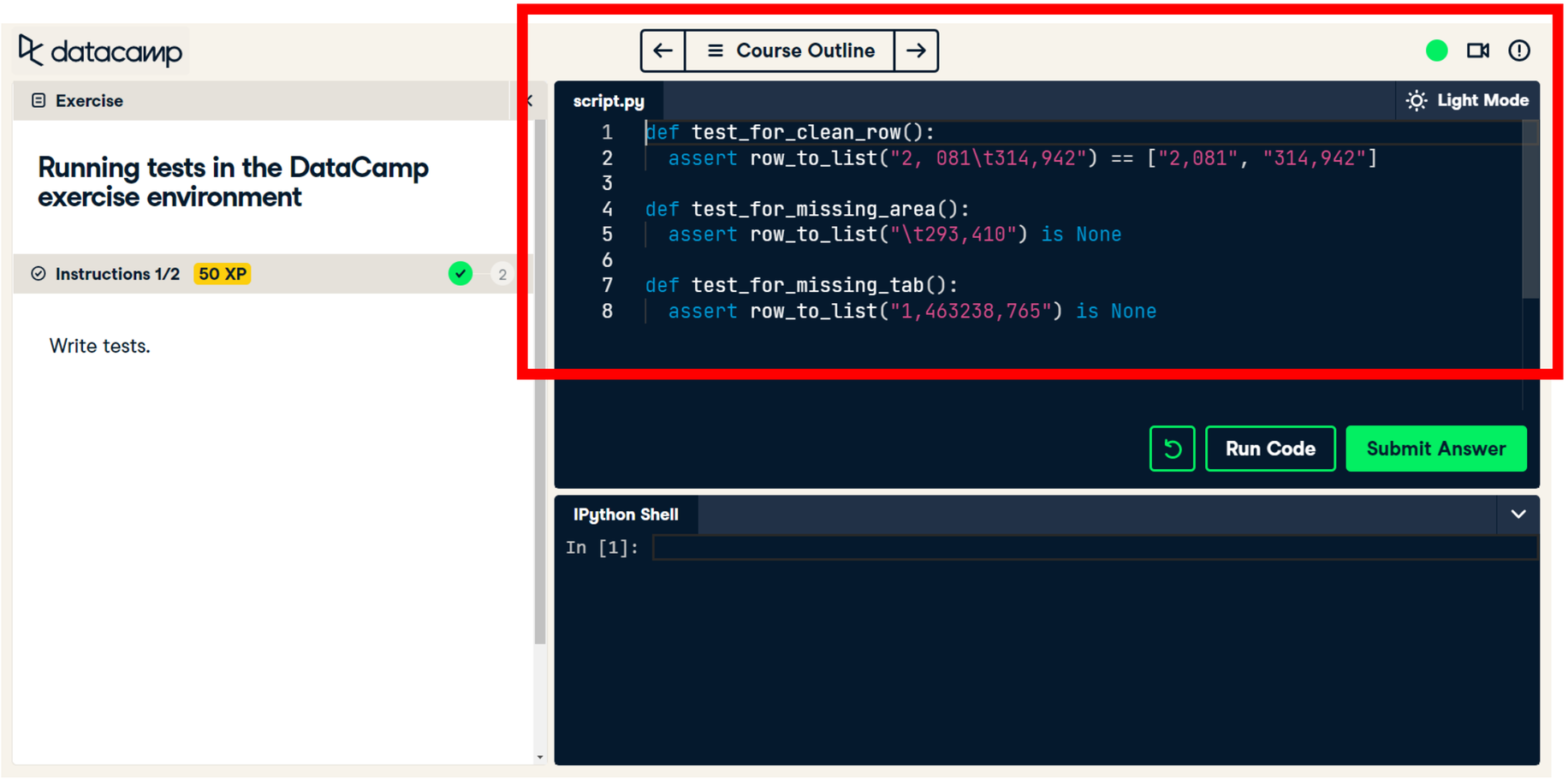
14. Step 5: Running unit tests
To test whether row_to_list() is working at any time in its life cycle, we simply run the test module. The standard way to run tests is to open a command line and type pytest followed by the test module name.

Running unit tests in DataCamp exercises

15. Running unit tests in DataCamp exercises
In the DataCamp exercises, we can't create files directly. So you will define the unit tests in the script.py area, which is highlighted here.



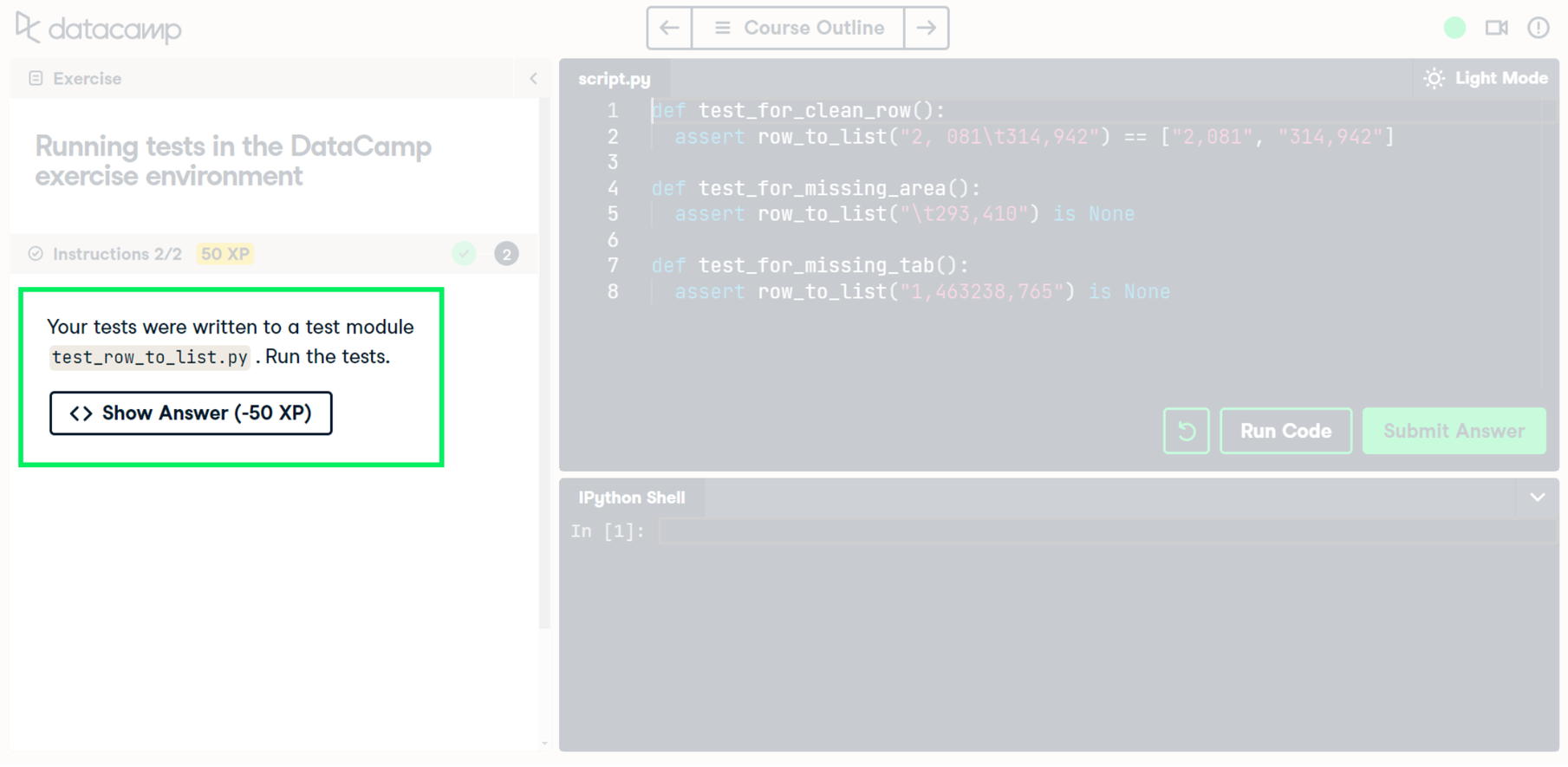
Running unit tests in DataCamp exercises



16. Running unit tests in DataCamp exercises

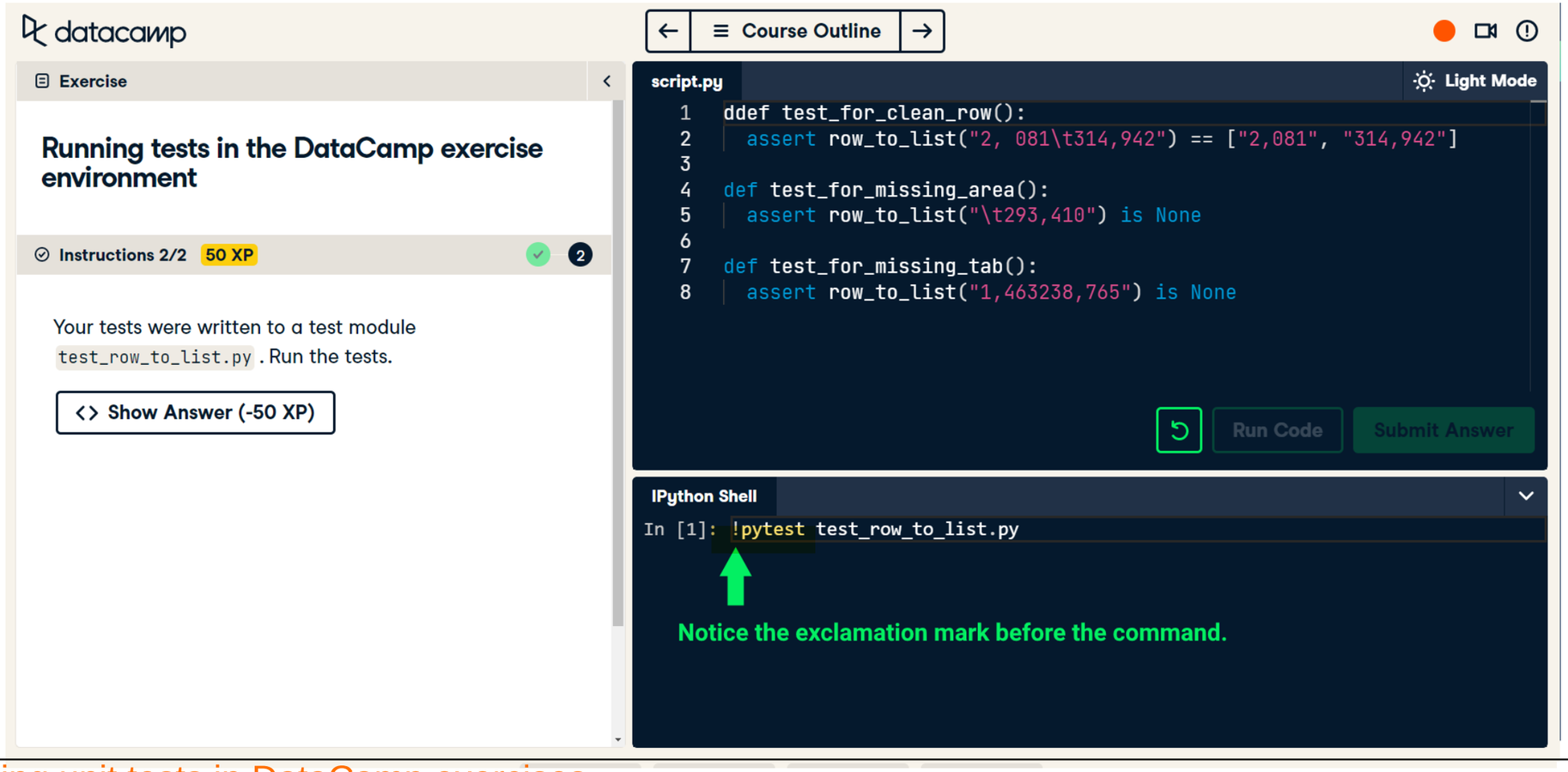
In the next step or next exercise, we will write the tests to a test module in the background and tell you its file path.

Running unit tests in DataCamp exercises



17. Running unit tests in DataCamp exercises
Once you know the test module's file path, you can run the tests in the IPython console at the bottom, which is highlighted here.

Running unit tests in DataCamp exercises



18. Running unit tests in DataCamp exercises

You can run any command line expression in the IPython console by adding an **exclamation mark** before the expression. For example, to run the pytest command, you have to use exclamation pytest, as shown in the picture.

Next lesson: test result report

datacamp

Course Outline

Exercise

Running tests in the DataCamp exercise environment

Instructions 2/2 50 XP

Your tests were written to a test module `test_row_to_list.py`. Run the tests.

Show Answer (-50 XP)

script.py

Light Mode

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

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

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

Run Code

Submit Answer

IPython Shell

```
In [1]: !pytest test_row_to_list.py
============================= test session starts
==============================
platform linux -- Python 3.6.7, pytest-5.2.2, py-1.9.0, pluggy-0.13.1
Matplotlib: 3.1.1
Freetype: 2.6.1
rootdir: /tmp/tmpmvodahy1
```

19. Next lesson: test result report

Running this command will output the test result report, which contains information about bugs in the function, if any.

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

datacamp

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`

2. Unit tests for row_to_list()
As an example, we will use the test module test_row_to_list.py, which we saw in the previous video lesson. It contains three unit tests for the row_to_list() function. The unit tests check if row_to_list() returns the correct return values for clean rows, rows missing area data and rows missing the tab separator respectively.

Test result report

```
!pytest test_row_to_list.py
```

3. Test result report
Running the tests in the IPython console produces lot of output. So much output that we had to truncate it in this slide. This is called the test result report. We will break this down into smaller pieces and understand them individually.

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

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, inifile:
plugins: mock-1.10.0
```

4. Section 1: general information
The first section provides information about the operating system, Python version, pytest package versions, the working directory and pytest plugins. There is not much to say about this section, so let's move ahead.

Section 2: Test result

```
collecting ...
collected 3 items

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

5. Section 2: Test result
The next bit is important. The output says "collected 3 items", which means that pytest found three tests to run. This is accurate as the test module test_row_to_list.py contains three unit tests. The next line contains the test module name, which is test_row_to_list.py, followed by the characters dot, capital F and dot. Each character represents the result of a 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.

6. Section 2: Test result
The character capital F stands for failure. A unit test fails if an exception is raised when running the unit test code.

Section 2: Test result

7. Section 2: Test result
This happens most often when the assert statement raises an AssertionError.
This means that the function has a bug and we should fix it

```
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`

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

Section 2: Test result

8. Section 2: Test result
A unit test may also fail if a different exception is raised while running the unit test code. In this case, execution will stop before the assert statement is run. For example, if we wrote None with a small n, this would raise a NameError. Since the assert statement did not run, we cannot conclude anything about the function under test from the capital F result. In this case, we should fix the unit test so that it can actually run the assert statement.

```
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

```python
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%]
```

9. Section 2: Test result
Dot means that the unit test passed. This means no exceptions were raised by the assert statement or any other part of the unit test. For the test module test_row_to_list.py, the first test passed, the second failed and the third passed.

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
```

10. Section 3: Information on failed tests
The next section contains detailed information about failed tests. We can see that the unit test test_on_missing_area() failed. The line raising the exception is marked by a greater than sign on the left. In this case, it is the assert statement.

- 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
```

11. Section 3: Information on failed tests
The following lines marked with capital E on the left contains details of the exception. In this case, the assert statement raised an 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
```

12. Section 3: Information about failed tests
The lines which contain the word "where" displays any return values that were calculated when running the assert statement. In this case, it shows that the actual return value of row_to_list() is a list containing an empty string and the string "293,410". The expected return value is, of course, None. The mismatch between the expected and actual value will be our starting point for debugging. We would have to figure out why row_to_list() returns a list instead of None in this case.

- 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!

13. Section 4: Test result summary
The final line is a test result summary, saying that "1 test failed, and 2 passed". Additionally, we also find out that the test took 0.03 seconds to run. That's really fast compared to the time we would need to test using 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

datacamp

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
```

2. Unit tests serve as documentation
The unit tests that we wrote for row_to_list() also serve as documentation. If a collaborator didn't know this function's purpose,

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
```

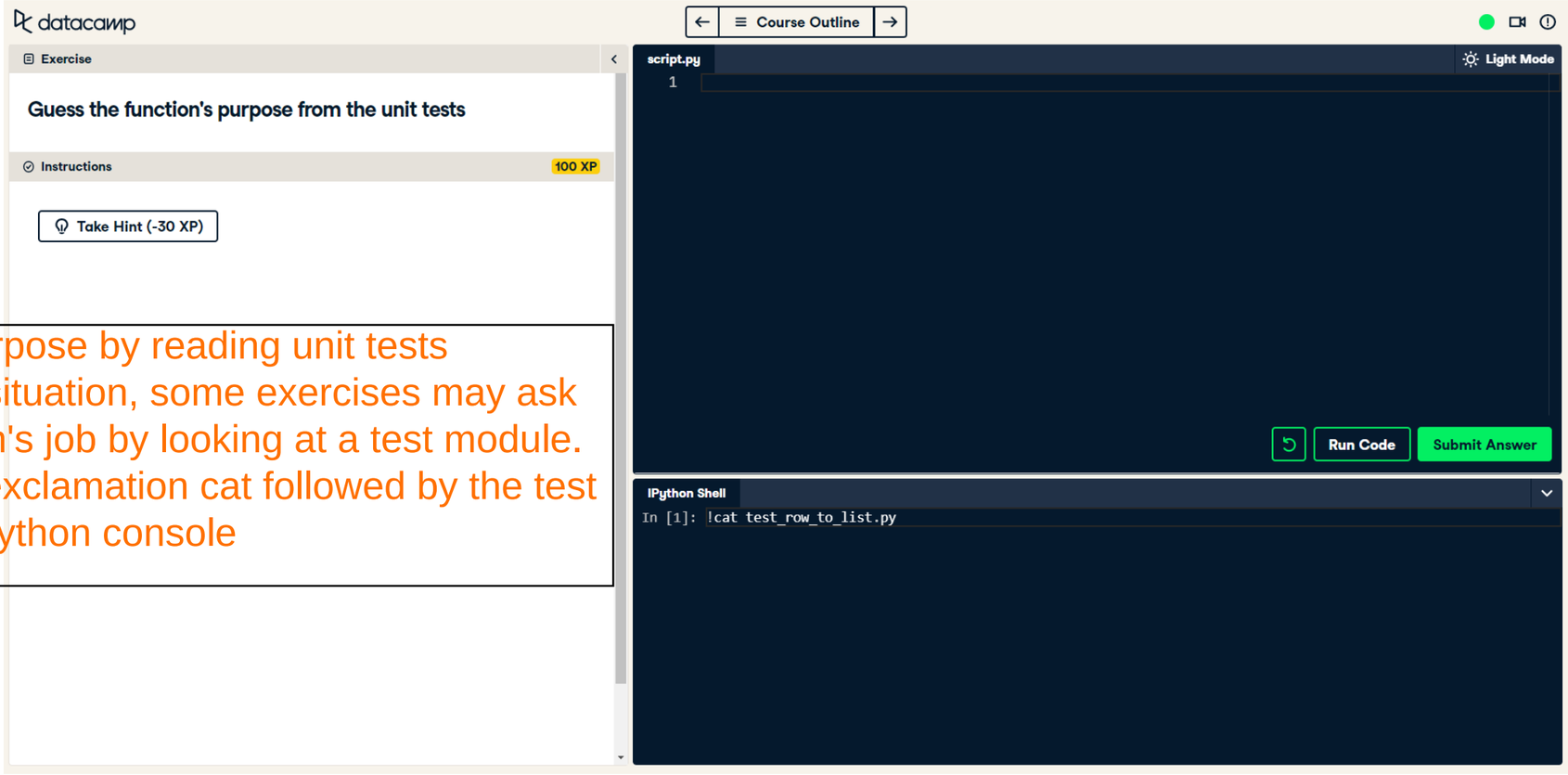
- Created from the test module

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

3. Unit tests serve as documentation
they could recreate the argument and return value table on the right by looking at the boolean expressions used in the assert statements. The table would give them a good hint about what row_to_list() does, helping them understand the function's code faster.

Guess function's purpose by reading unit tests

```
!cat test_row_to_list.py
```



4. Guess function's purpose by reading unit tests
To mimic this real life situation, some exercises may ask you to guess a function's job by looking at a test module. In this case, just type exclamation cat followed by the test module name in the IPython console

Guess function's purpose by reading unit tests

```
!cat test_row_to_list.py
```

datacamp

Course Outline

Exercise

Guess the function's purpose from the unit tests

Instructions 100 XP

Take Hint (-30 XP)

script.py

Light Mode

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    314,942") == ["2,081", "314,942"]

def test_for_missing_area():
  assert row_to_list("  293,410") == None

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

In [2]:
```

4. Guess function's purpose by reading unit tests
To mimic this real life situation, some exercises may ask you to guess a function's job by looking at a test module. In this case, just type exclamation cat followed by the test module name in the IPython console

More trust

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

NumPy

6. More trust
Unit tests also increase trust in a package, as users can run the unit tests and verify that the functions work. In the picture, we see NumPy's Github page.

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

More trust

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

7. More trust
This highlighted badge shows how much of the code base is unit tested

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

More trust

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

8. More trust
and this other badge shows whether the tests are passing. Users trust NumPy more because of these badges. We should implement these badges for our projects too, because user trust is important, and we will learn how to do that later in the course.

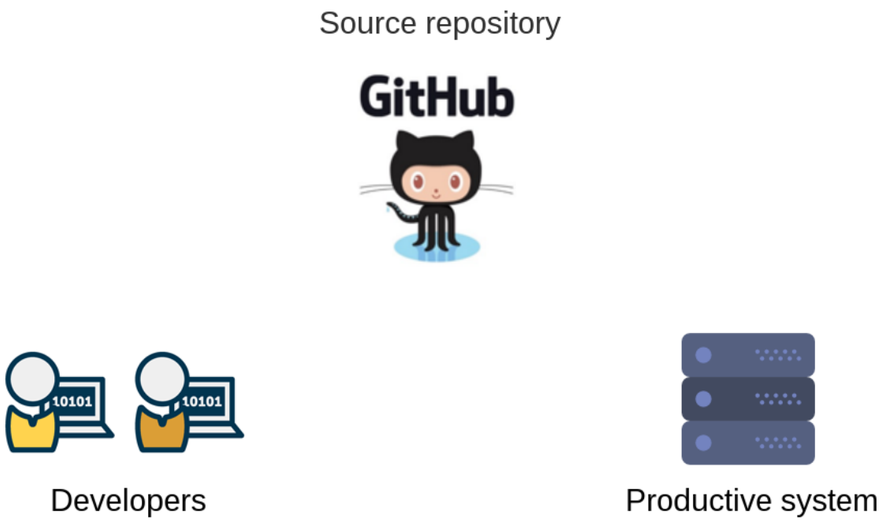
NumPy

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

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

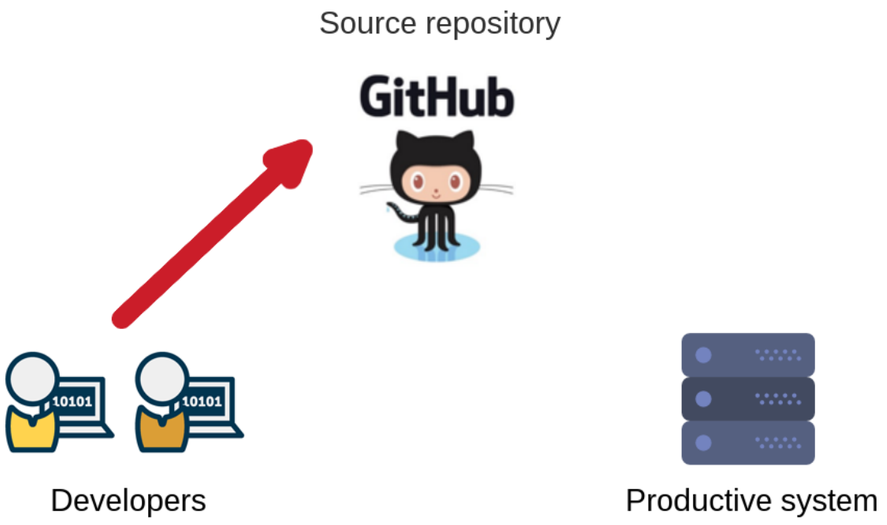
- **Website (including documentation):** https://www.numpy.org
- **Mailing list:** https://mail.python.org/mailman/listinfo/numpy-discussion

Reduced downtime



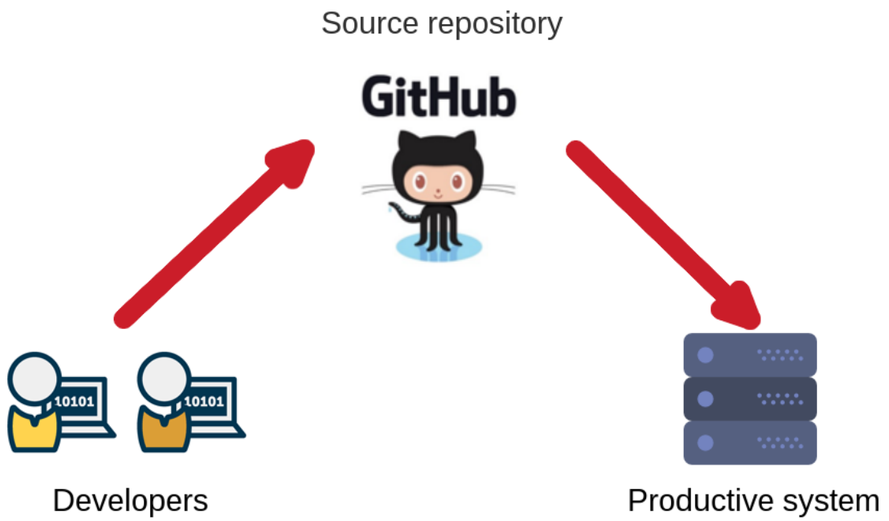
9. Reduced downtime
Unit tests can also reduce downtime for a productive system.

Reduced downtime



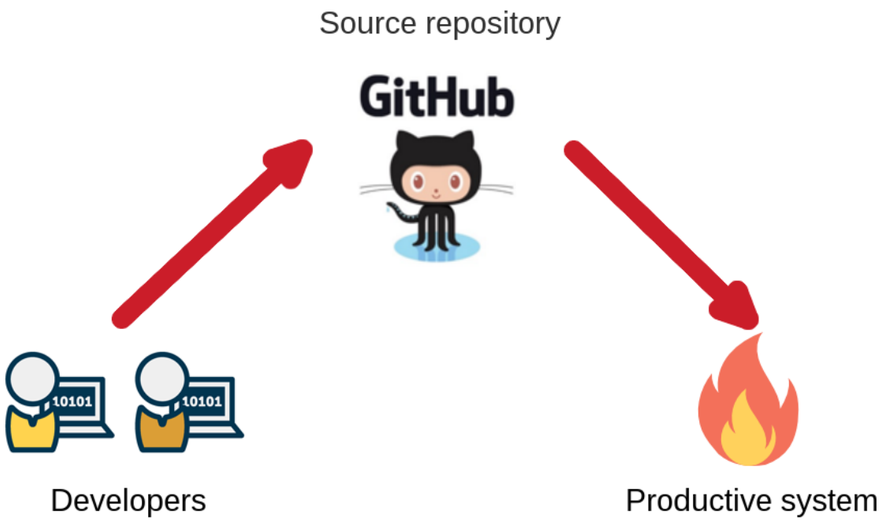
10. Reduced downtime
Suppose we make a mistake

Reduced downtime



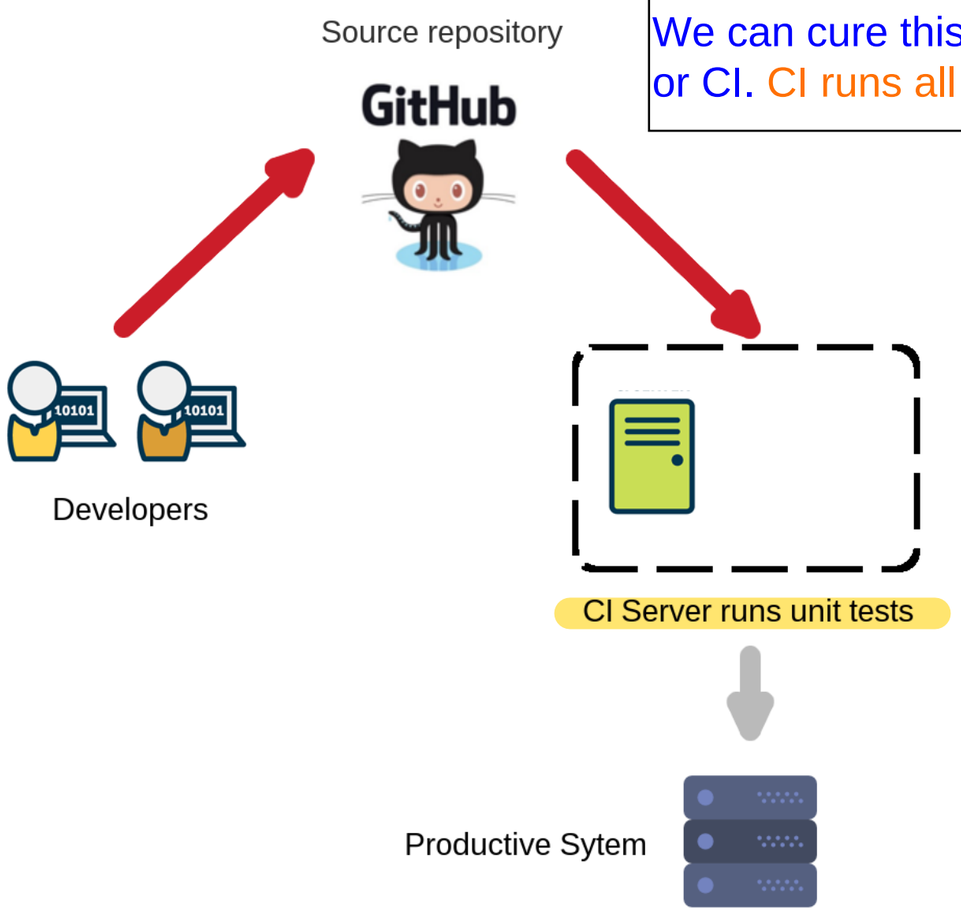
11. Reduced downtime
and push bad code to a productive system.
12. Reduced downtime
This will bring the system down and annoy users.

Reduced downtime



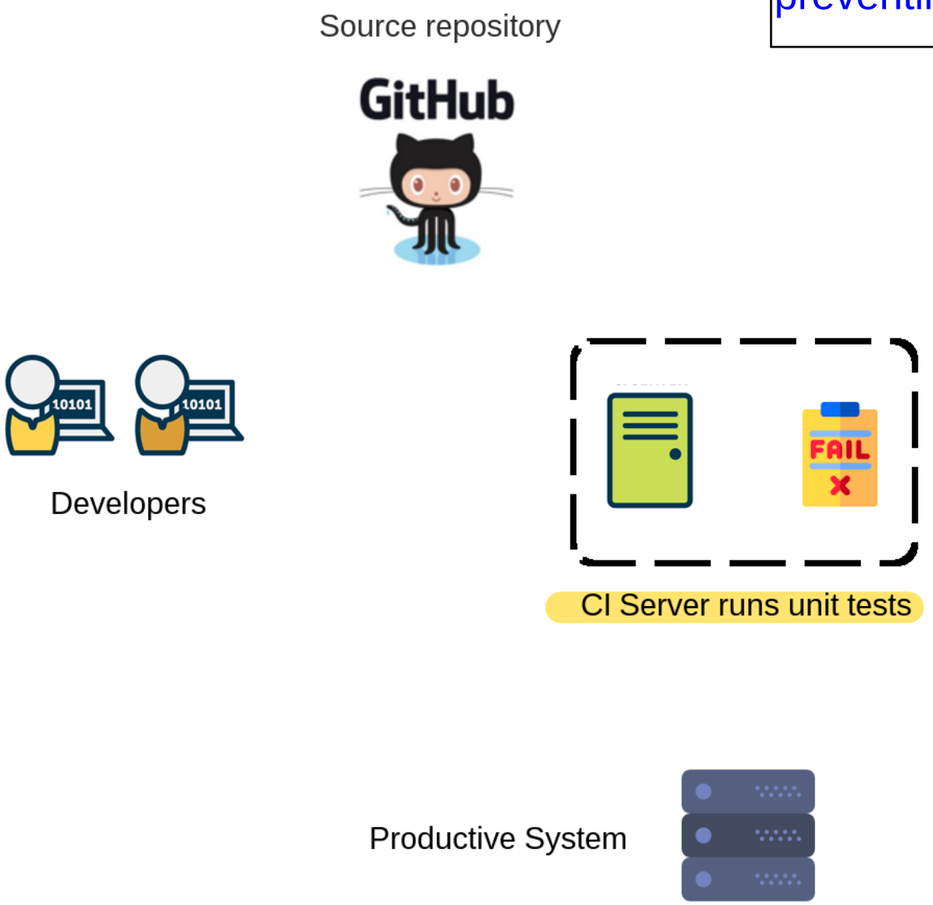
Reduced downtime

13. Reduced downtime
We can cure this by setting up Continuous Integration or CI. CI runs all unit tests when any code is pushed,



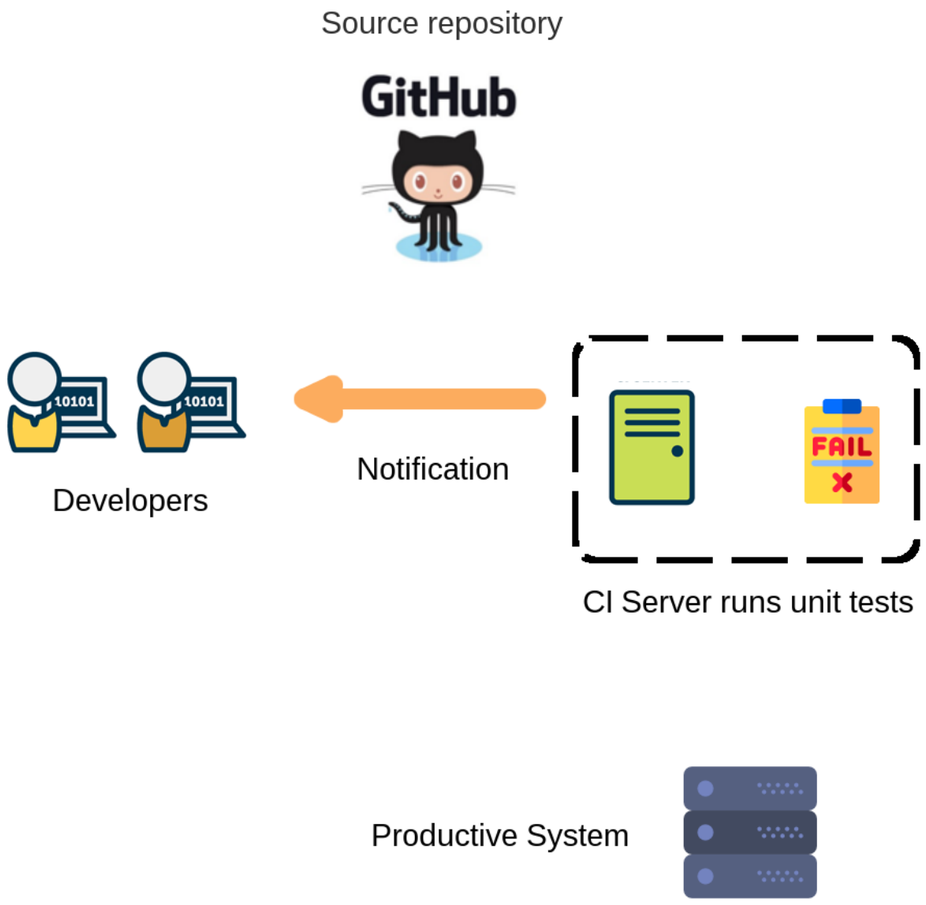
Reduced downtime

14. Reduced downtime
and if any unit test fails, it rejects the change, preventing downtime.



Reduced downtime

15. Reduced downtime
It also informs us that the code needs to be fixed. If we run productive systems that many people depend upon, we must write unit tests and setup CI.



All benefits

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

16. All benefits
All of these benefits make the case stronger for writing unit tests! In this course, we will write unit tests for all functions in the example linear regression project.

Tests we already wrote

row_to_list()

17. Tests we already wrote
We already wrote tests for row_to_list()

Tests we already wrote

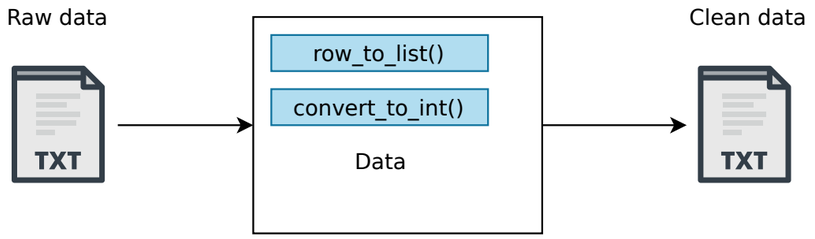
row_to_list()

convert_to_int()

18. Tests we already wrote
and convert_to_int().

19. Data module
They are part of the data module, which creates a clean data file from raw data on housing area and price.

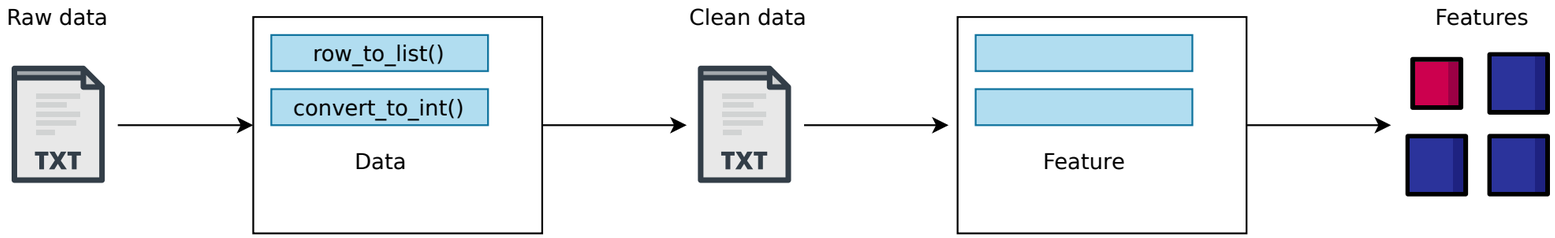
Data module



18. Tests we already wrote
and convert_to_int().

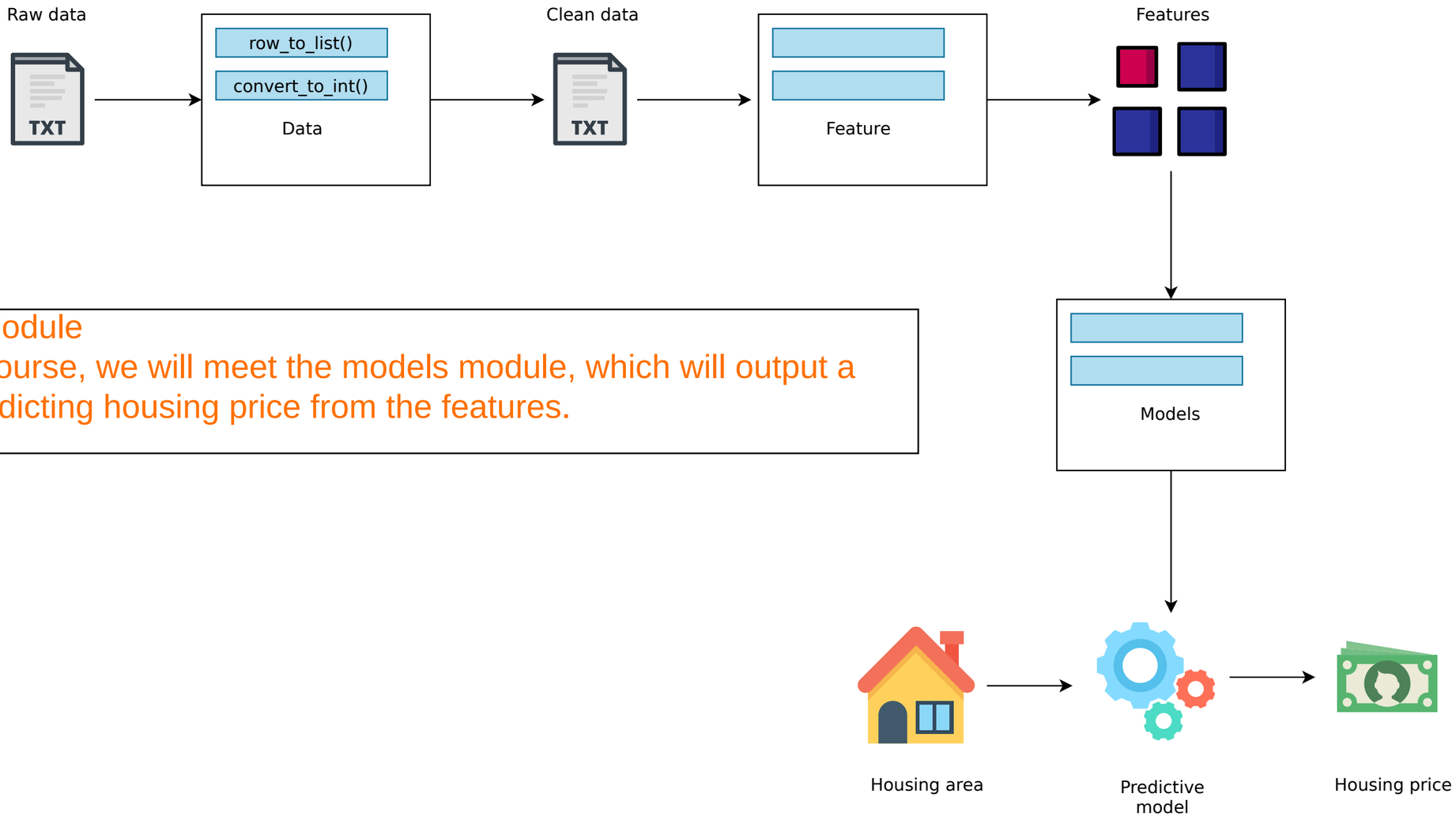
19. Data module
They are part of the data module, which creates a clean data file from raw data on housing area and price.

Feature module



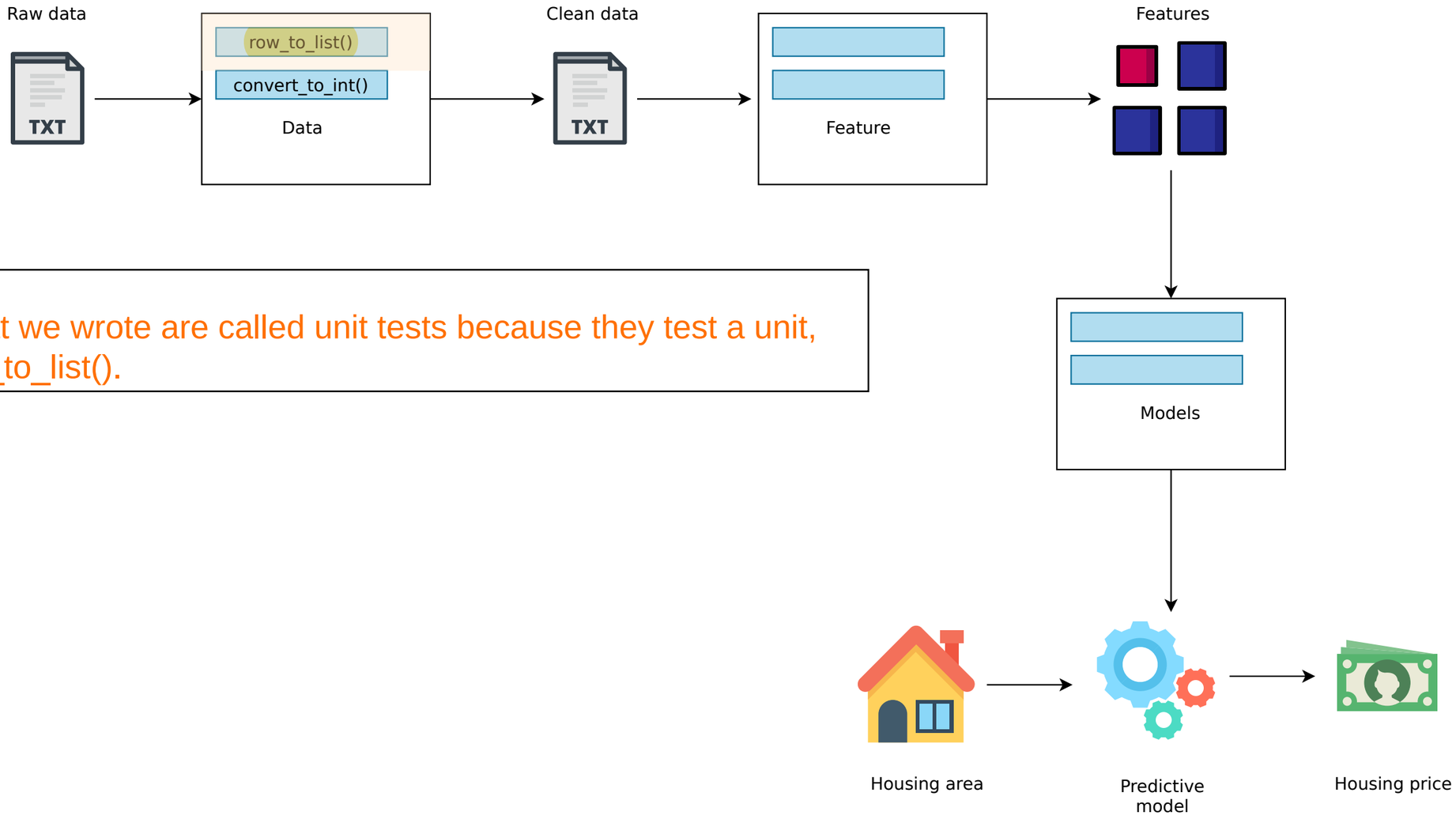
20. Feature module
Very soon, we will see functions from the feature module, which compute features from the clean data.

Models module



21. Models module
Later in the course, we will meet the models module, which will output a model for predicting housing price from the features.

Unit test



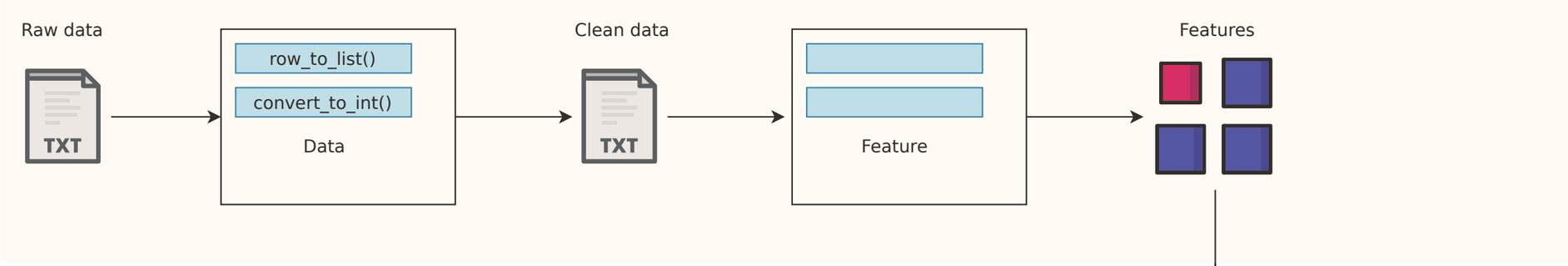
22. Unit test
The tests that we wrote are called unit tests because they test a unit, such as row_to_list().

What is a unit?

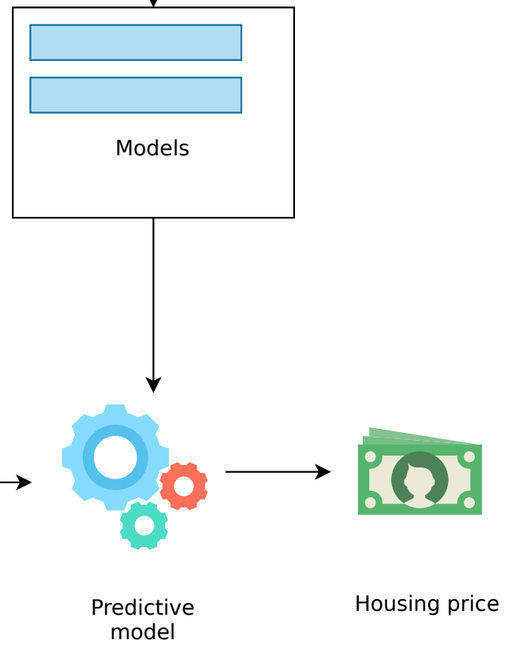
- Small, independent piece of code.
- Python function or class.

23. What is a unit?
A unit is any small independent piece of code, and could be a Python function or class.

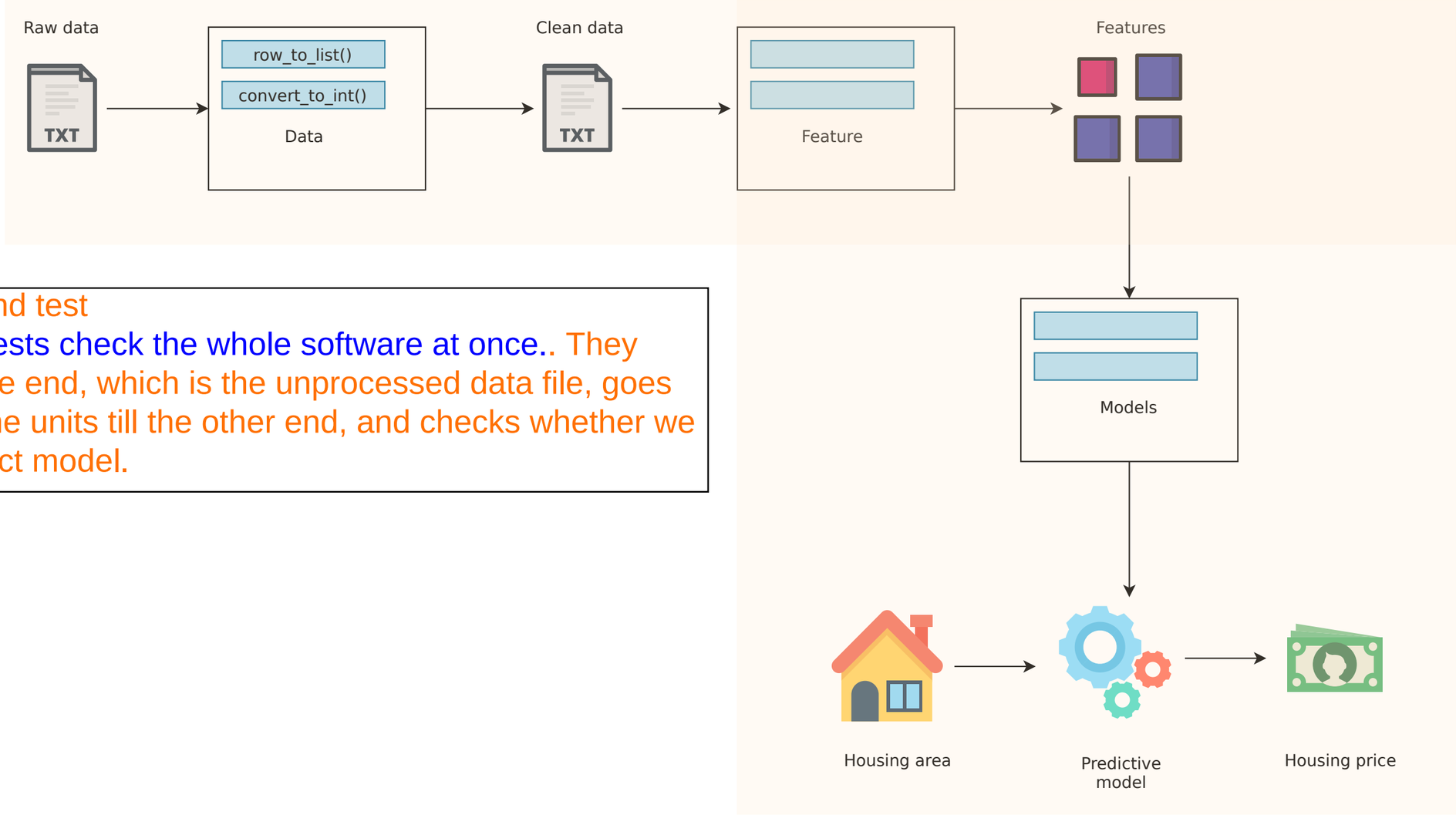
Integration test



24. Integration test
In contrast, integration tests check if multiple units work well together when they are connected, and not just independently. For example, we could check if the data and the feature module work well when connected. Here, the argument will be the raw data, and the return values to check would be the features.



End to end test



25. End to end test
End to end tests check the whole software at once.. They start from one end, which is the unprocessed data file, goes through all the units till the other end, and checks whether we get the correct model.

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