Mastering assert statements

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

Theoretical structure of an assertion

```
assert boolean_expression
```

2. Theoretical structure of an assertion
So far, we have used only a boolean expression as an argument of the assert statement.

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!
```

3. The optional message argument
But the assert statement can take an optional second argument, called the message. **The message is only printed when** the assert statement raises an AssertionError and it should contain information about why the AssertionError was raised. If the assert statement passes, nothing is printed.

```
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
```

4. Adding a message to a unit test
Let's look at the unit test called test_for_missing_area() for the row_to_list() function that we encountered in Chapter 1.

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
```

5. Adding a message to a unit test
We could enhance this test by adding a message. We first capture the return value of row_to_list() in a variable called actual. We define a variable called expected holding the expected return value. Then we define a message that contains the argument, the actual and expected return values - basically everything that we need to debug in case the test fails. We follow it up with the appropriate assert statement including the 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
```

6. Test result report with message
Here, on the top, we run the unit test without the message and we get pytest's automatic output on failed tests. On the bottom, we run the modified one with the message. Now the automatic output is gone, and we get the nice human readable message next to the AssertionError.

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
```

8. Beware of float return values!
Next, we will learn about an especially tricky situation when the function returns a float. In Python, comparisons between floats don't always work as expected, as we can see in this surprising example.

Beware of float return values!

```
0.1 + 0.1 + 0.1
```

```
0.30000000000000004
```

9. Beware of float return values!
Because of the way Python represents floats, the digits on the right might be different from what we expect, causing comparisons to fail. Why this happens is out of the scope of this course.

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!
```

10. Don't do this
The bottom line is: we should not use the usual way to compare floats in the assert statement.

Do this

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

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

```

```

11. Do this
Instead, we should use the pytest.approx() function to wrap the expected value, as we see in this example. This ensures that the digits far on the right are ignored and we can compare floats safely. We get an empty output since the assert statement passes.

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]))
```

12. NumPy arrays containing floats
pytest.approx() also works for NumPy arrays containing floats. For example, notice how we are passing a NumPy array to the pytest.approx() function in the assert statement shown here.

Multiple assertions in one unit test

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

```
2081
```

13. Multiple assertions in one unit test
So far, we have only seen one assert statement per unit test, but unit tests can have more than one assert statement.
Remember the convert_to_int() function, which converts an integer valued string with commas to an integer?

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.

14. Multiple assertions in one unit test
Let's look at a unit test which you wrote for convert_to_int() in one of the exercises. We will modify this unit test on the right. We first want to test if the function returns an integer at all. For this, we use the isinstance() function, which takes the return value as a first argument, and the expected type of the return value as a second argument, which is int in this case. We follow this up with another assert statement which checks if the return value matches the expected value. The modified test will pass if both assert statements pass. It will fail if any of them raises an AssertionError.

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

1. Testing for exceptions instead of return values
So far, we have used the assert statement to check if a function returns the expected value. However, some functions may not return anything, but rather raise an exception, when called on certain arguments.

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)
```

2. Example
Consider the split_into_training_and_testing_sets() function that you tested in the last exercise. This function returns a two-tuple containing the training and the testing array. It puts 75% of the rows of the argument NumPy array into the training array, and the rest of the rows into the testing array.

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

3. Example
This function expects the argument array to have rows and columns, that is, the argument array must be two dimensional. Otherwise, splitting by rows is undefined.

```
(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!
```

4. Example
So if we pass a one dimensional array to this function, it should not return anything, but rather raise a ValueError, which is a specific type of exception.

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):
```

5. Unit testing exceptions
We will now learn how to test whether this function raises ValueError when it gets a one dimensional array as argument. Let's call the unit test test_valueerror_on_one_dimensional_argument(). The example argument is a one dimensional array, as shown here. Then we use a with statement that we will explain in detail now.

Theoretical structure of a with statement

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

6. Theoretical structure of a with statement
First, let's understand the with statement. Any code that is inside the with statement is known as the context.

Theoretical structure of a with statement

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

7. Theoretical structure of a with statement
The with statement takes a single argument, which is known as a **context manager.**

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
```

8. Theoretical structure of a with statement
The context manager runs some code before entering and exiting the context, just like a security guard who checks or does something when we enter or leave a building.

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.
```

9. Theoretical structure of a with statement
In this case, we are using a context manager called pytest.raises(). It takes a single argument, which is the type of exception that we are checking for, in this case, a ValueError. This context manager does not run any code on entering the context, but it does something on exit. **If the code in the context raises a ValueError, the context manager silences the error. And if the code in the context does not raise a ValueError, the context manager raises an exception itself.**

Theoretical structure of a with statement

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

10. Theoretical structure of a with statement
Here is an example where a ValueError is raised in the context and silenced by the context manager. The second code block shows an example where no ValueError is raised and the context manager raises an exception called Failed.

```
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.

11. Unit testing exceptions
Getting back to the unit test, we call the function on the one dimensional array inside the context. If the function raises a ValueError as expected, it will be silenced and the test will pass. If the function is buggy and no ValueError is raised, the context manager raises a Failed exception, causing the test to fail.

Testing the error message

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

12. Testing the error message
We can unit test details of the raised exception as well. For example, we might want to check if the raised ValueError contains the correct error message which starts with "Argument data array must be two dimensional".

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.

13. Testing the error message
In order to do that, we extend the with statement with the as clause. If the ValueError is raised within the context, then exception_info will contain information about the silenced ValueError. After the context ends, we can check whether exception_info has the correct message. To do this, we use a simple assert statement with the match() method of exception_info. The match method takes a string as argument and checks if the string is present in the 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

1. The well tested function
In this lesson, we will explore the question: how many tests should one write for a function?

Dibya Chakravorty
Test Automation Engineer

datacamp

Example

2. Example
Consider the split_into_training_and_testing_sets() function that we saw in earlier lessons. This function takes a two dimensional NumPy array as an argument and randomly puts about 75% of its rows into a training array, and the remaining rows into a testing array. It returns the training and the testing arrays as a two tuple.

```
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

3. Test for length, not value
Because this function has randomness, we test for the lengths of the training and testing arrays rather than their actual values. The length of the training array is given by the integer part of 0.75 times the number of rows in the argument. The testing array gets the rest.

```
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	`int(0.75 * 8)` = 6	`8 - int(0.75 * 8)` = 2

4. Test arguments and expected return values
For example, if the argument has 8 rows, then the training array should have 6 rows and the testing array should have 2 rows.

5. Test arguments and expected return values
In general, the more arguments we check,

6. Test arguments and expected return values
the more confident we can be that the function is working correctly.

Test arguments and expected return values

Number of rows (argument)	Number of rows (training array)	Number of rows (testing array)
8	`int(0.75 * 8)` = 6	`8 - int(0.75 * 8)` = 2
10	`int(0.75 * 10)` = 7	`10 - 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	`int(0.75 * 8)` = 6	`8 - int(0.75 * 8)` = 2
10	`int(0.75 * 10)` = 7	`10 - int(0.75 * 10)` = 3
23	`int(0.75 * 23)` = 17	`23 - 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	`int(0.75 * 8)` = 6	`8 - int(0.75 * 8)` = 2
10	`int(0.75 * 10)` = 7	`10 - int(0.75 * 10)` = 3
23	`int(0.75 * 23)` = 17	`23 - int(0.75 * 23)` = 6
...	...	...
...	...	...
...	...	...

7. How many arguments to test?
But since we cannot write tests for hundreds of arguments because of time limitations, **how many tests can be considered enough?**

Test argument types

Test for these argument types

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

8. Test argument types
The best practice is to pick a few from each of the following categories of arguments, which are called bad arguments, special arguments and 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. ✓

9. Test argument types
If we have tested

10. Test argument types
for all

11. Test argument types
of these argument types,

12. The well tested function
then our function can be declared well tested.

The well tested function

12. The well tested function
then our function can be declared well tested.

Test for these argument types

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

Type I: Bad arguments

- When passed bad arguments, function raises an exception.

13. Type I: Bad arguments
Let's define these argument types for the split_into_training_and_testing_sets() function. Bad arguments are arguments for which the function raises an exception instead of returning a value.

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])`

14. Type I: Bad arguments (one dimensional array)
For split_into_training_and_testing_sets(), a one dimensional array, like the one shown, is a bad argument. It doesn't have rows and columns, so splitting row wise doesn't make any sense. The expected outcome is a ValueError.

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]])`

14. Type I: Bad arguments (one dimensional array)
For split_into_training_and_testing_sets(), a one dimensional array, like the one shown, is a bad argument. It doesn't have rows and columns, so splitting row wise doesn't make any sense. The expected outcome is a ValueError.

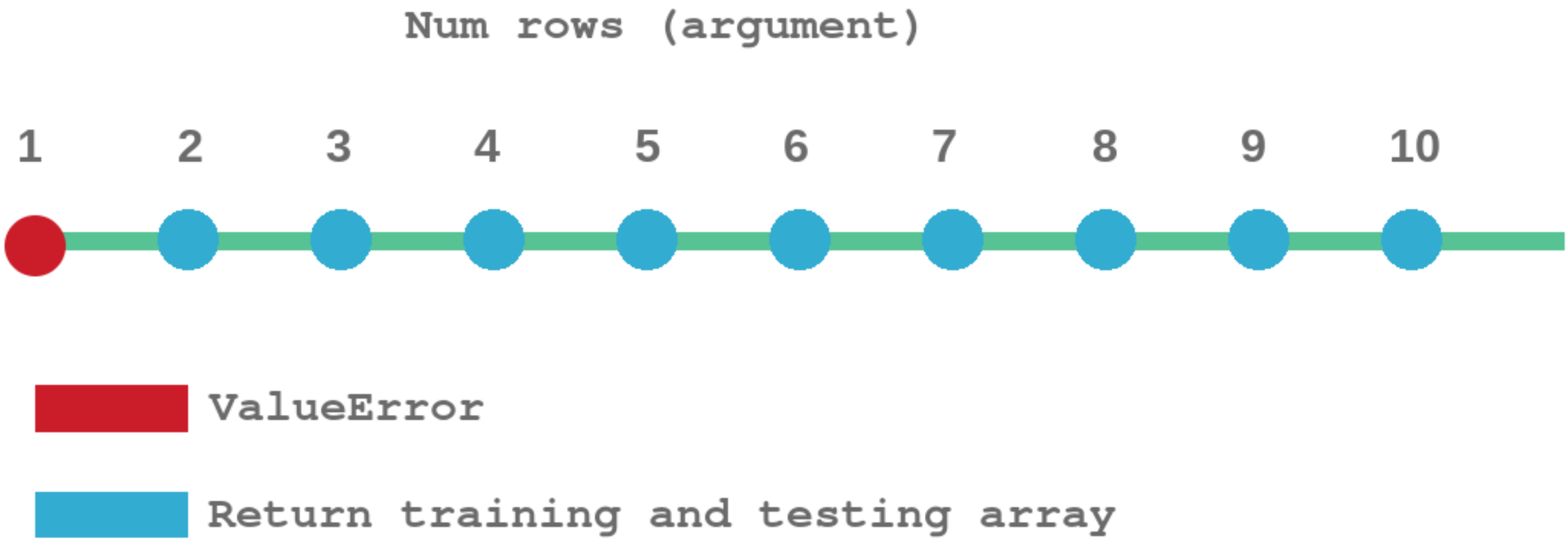
15. Type I: Bad arguments (array with only one row)
An array with a single row is also a bad argument, because this row can be put in the training array, but then the testing array will be empty. Or vice versa. Empty training or testing arrays are useless, so the function raises a ValueError.

Type II: Special arguments

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

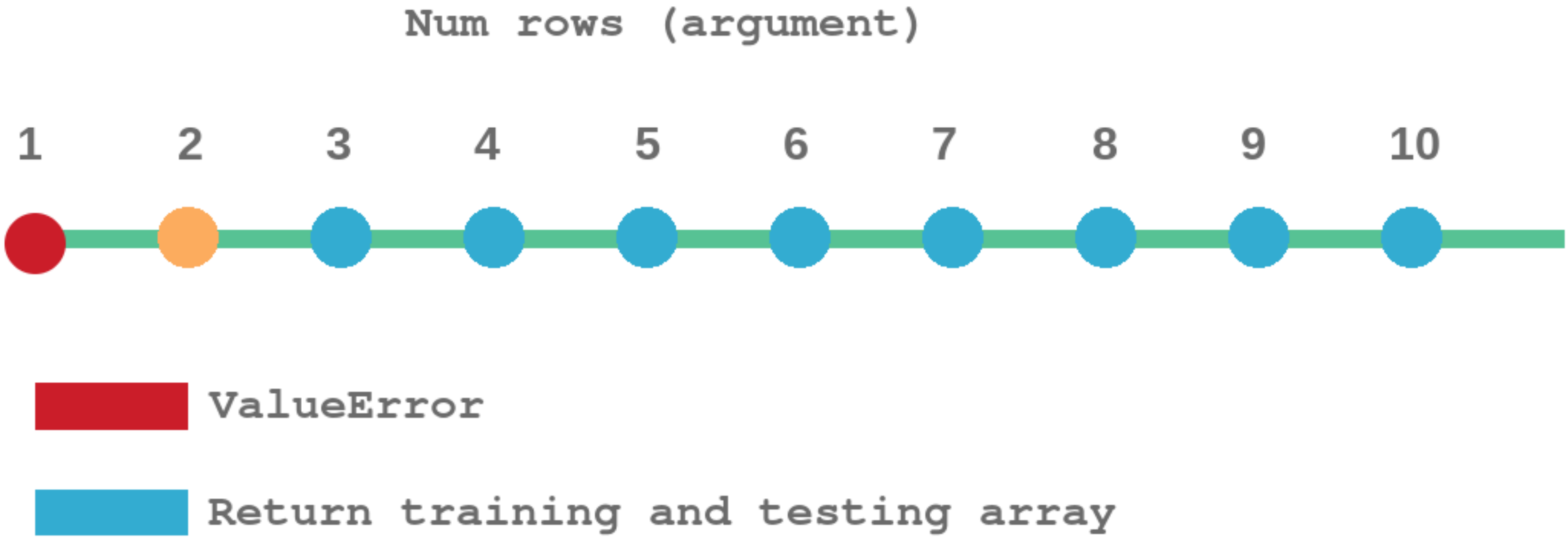
16. Type II: Special arguments
Next comes special arguments. These are of two types: boundary values and argument values for which the function uses a special logic to produce the return value.

Boundary values



Boundary values

17. Boundary values
So what are boundary values? If we look at the number of rows of the argument, we see that the function raises a ValueError for one row, but returns training and testing array for arguments having more than one row.



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	`int(0.75 * 2)` = 1	`2 - int(0.75 * 2)` = 1	-

18. Boundary values
The value two marks the boundary for this behavior change, and therefore, is a boundary value.

19. Test arguments table
So we append the test arguments table with the boundary value.

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	`int(0.75 * 2)` = 1	`2 - int(0.75 * 2)` = 1	-
Contains 4 rows		`int(0.75 * 4)` = 3	`4- int(0.75 * 4)` = 1	-

Arguments triggering special logic

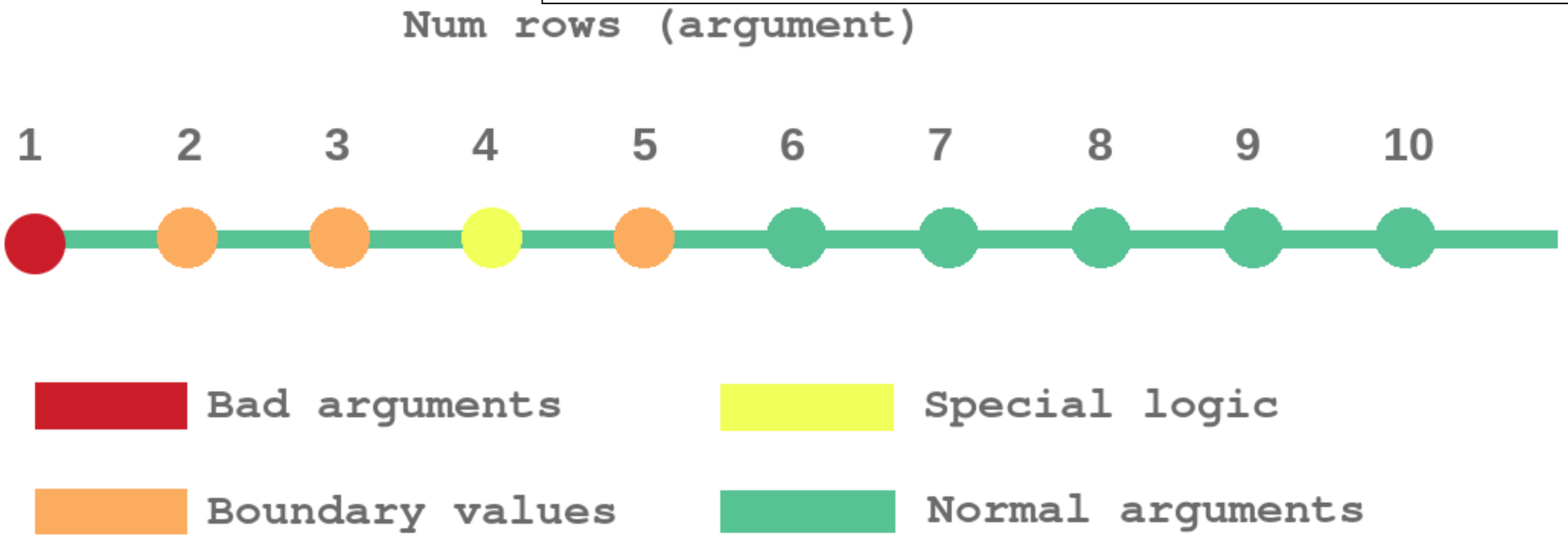
Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	`ValueError`
Contains 1 row	Bad	-	-	`ValueError`
Contains 2 rows	Special	`int(0.75 * 2)` = 1	`2 - int(0.75 * 2)` = 1	-
Contains 4 rows	Special	~~3~~ 2	~~1~~ 2	-

20. Arguments triggering special logic
The other type of special arguments are those that trigger special logic in the function. Look at the last row where the argument has 4 rows. The standard logic of a 75% and 25% split would produce a training array with 3 rows and a testing array with 1 row.

21. Arguments triggering special logic
But we might want the function to return a training array with 2 rows and testing array with 2 rows instead. Then 4 rows would be a special case, because the function isn't using the usual 75% and 25% logic.

Normal arguments

22. Normal arguments
Finally, anything that is not a bad or special argument is a normal argument. First notice that since the function uses special logic at 4, the behavior at 4 is different from the behavior on either side of 4. This turns 3 and 5, which flanks 4, into boundary values. The remaining arguments, with number of rows exceeding 5 are then normal arguments. It is recommended to test for two or three normal arguments.



Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	`ValueError`
Contains 1 row	Bad	-	-	`ValueError`
Contains 2 rows	Special	`int(0.75 * 2)` = 1	`2 - int(0.75 * 2)` = 1	-
Contains 3 rows	Special	`int(0.75 * 3)` = 2	`3 - int(0.75 * 3)` = 1	-
Contains 4 rows	Special	~~3~~ 2	~~1~~ 2	-
Contains 5 rows	Special	`int(0.75 * 5)` = 3	`5 - int(0.75 * 5)` = 2	-
Contains 6 rows	Normal	`int(0.75 * 6)` = 4	`6 - int(0.75 * 6)` = 2	-
Contains 8 rows	Normal	`int(0.75 * 8)` = 6	`8 - int(0.75 * 6)` = 2	-

23. Final test arguments table

The final rows of the arguments table correspond to two normal arguments, having 6 and 8 rows respectively. If we test the split_into_training_and_testing_sets() function with all these arguments,

```
split_into_training_and_testing_sets()
```

24. Insert title here...
then we can declare it well tested.

Caveat

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

25. Caveat
When applying this logic to other functions, note that not all functions have bad or special arguments. In that case, we should just ignore those types of arguments.

Let's apply this to other functions!

UNIT TESTING FOR DATA SCIENCE IN PYTHON

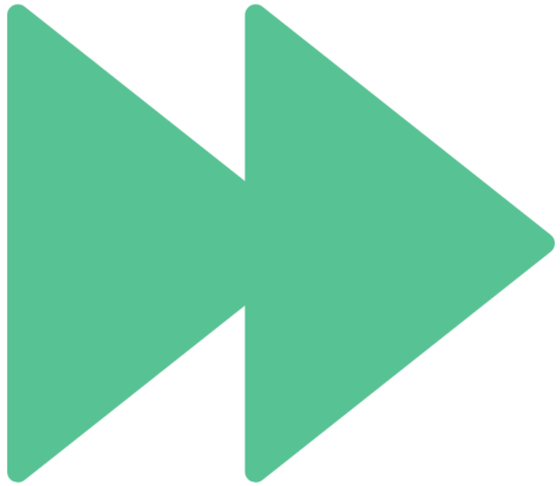
Test Driven Development (TDD)

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

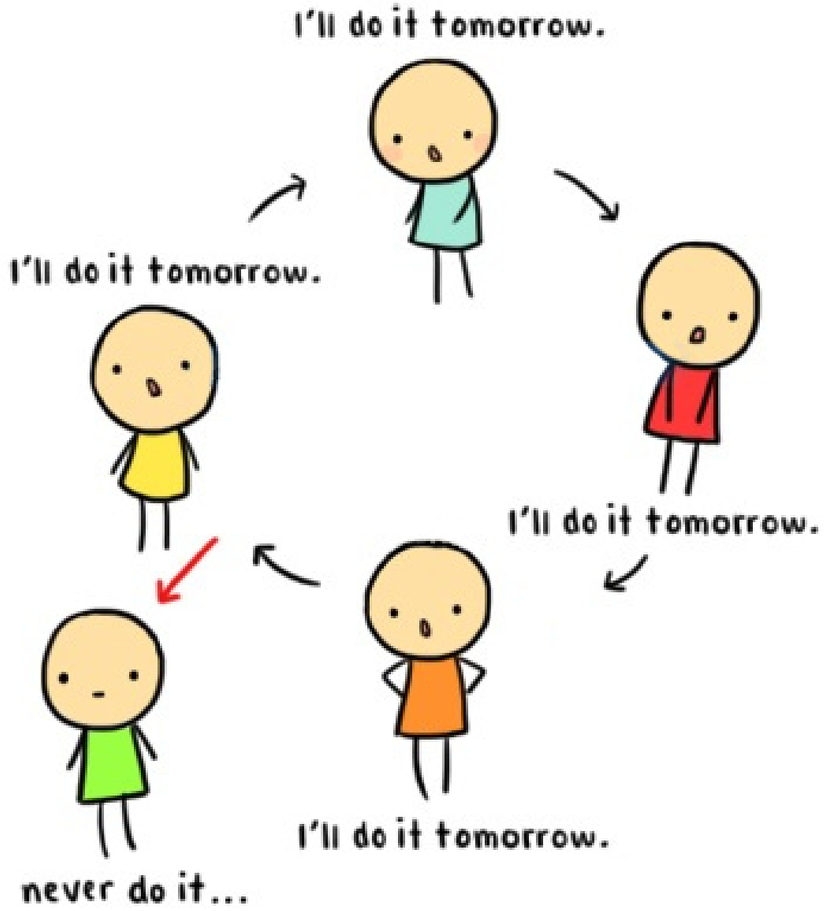
Writing unit tests is often skipped



Usual priorities in the industry

1. Feature development.
2. Unit testing.

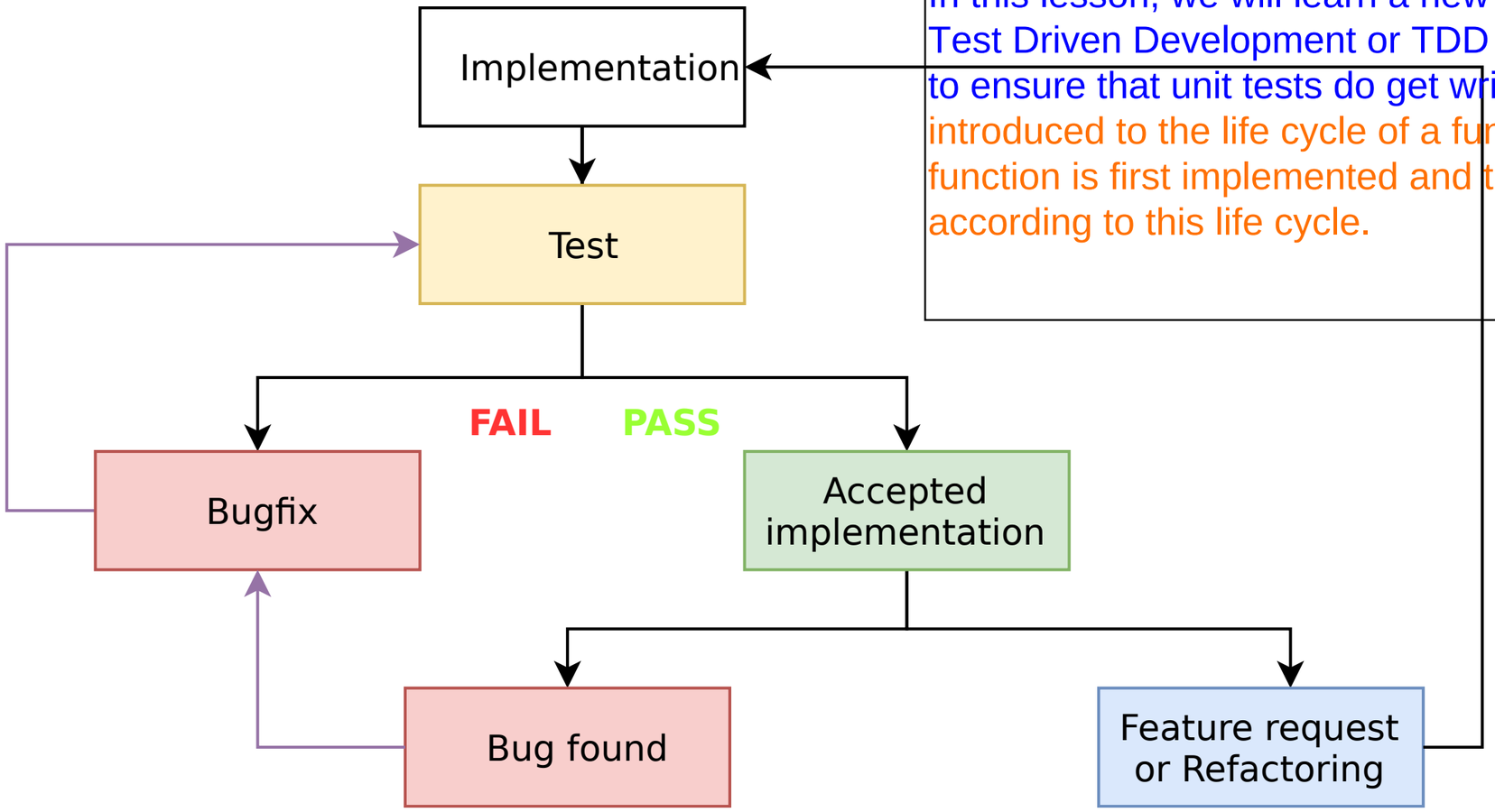
Unit tests never get written



1. Test Driven Development (TDD)
By now, we understand why it is so important to write unit tests.
2. Writing unit tests is often skipped
But in the real world, it is all too common to skip writing them.
3. Usual priorities in the industry
Bosses want to prioritize feature implementation, because they want fast results. Unit tests only get second priority.
4. Unit tests never get written
So we tell ourselves that we will write the not-so-urgent unit tests tomorrow. Eventually, the unit tests never get written. Of course, we pay for this mistake in the long term.

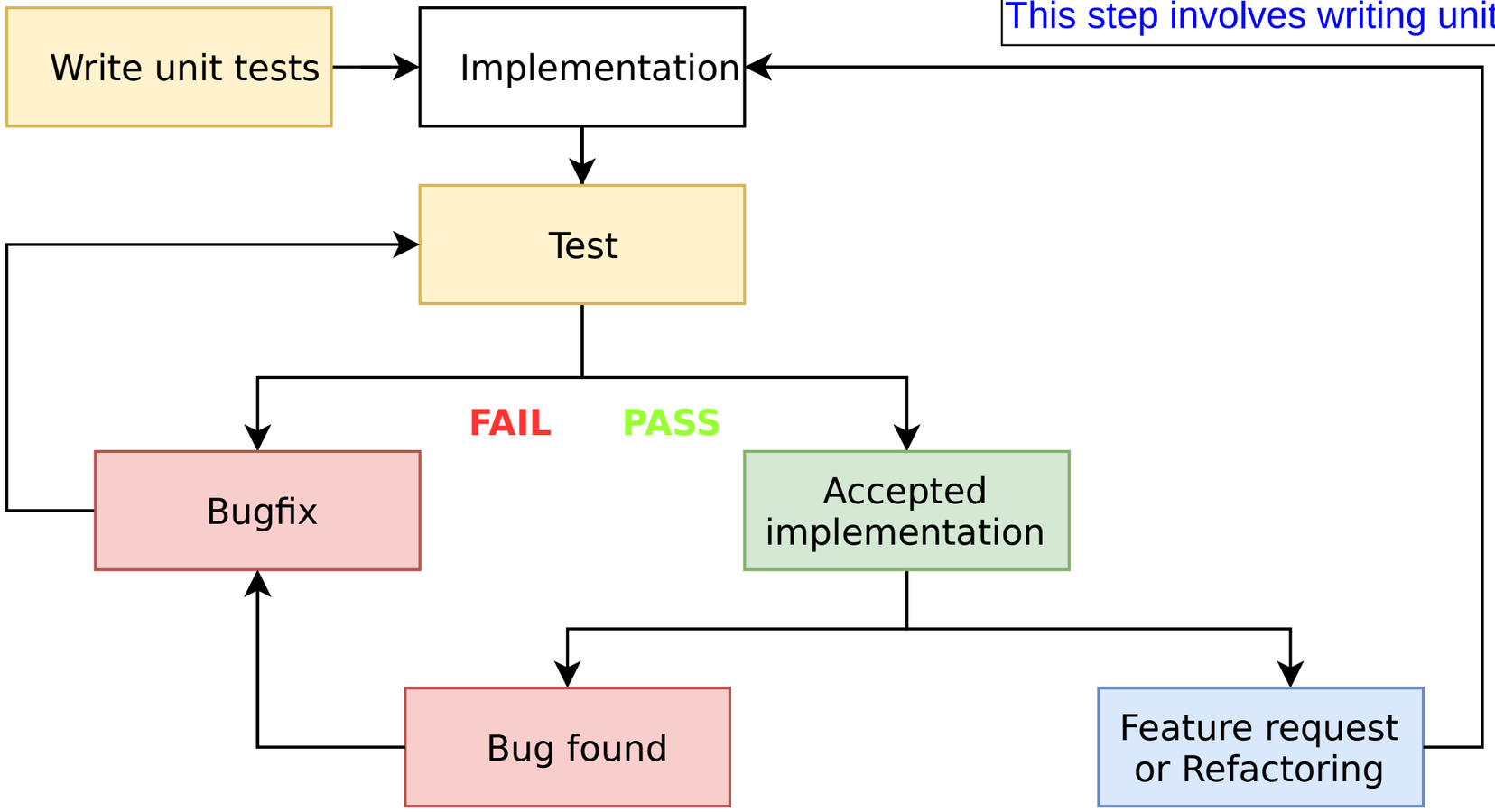
Test Driven Development (TDD)

5. Test Driven Development (TDD)
In this lesson, we will learn a new coding method called Test Driven Development or TDD in short, which tries to ensure that unit tests do get written. We got introduced to the life cycle of a function in Chapter 1. A function is first implemented and then it is tested, according to this life cycle.



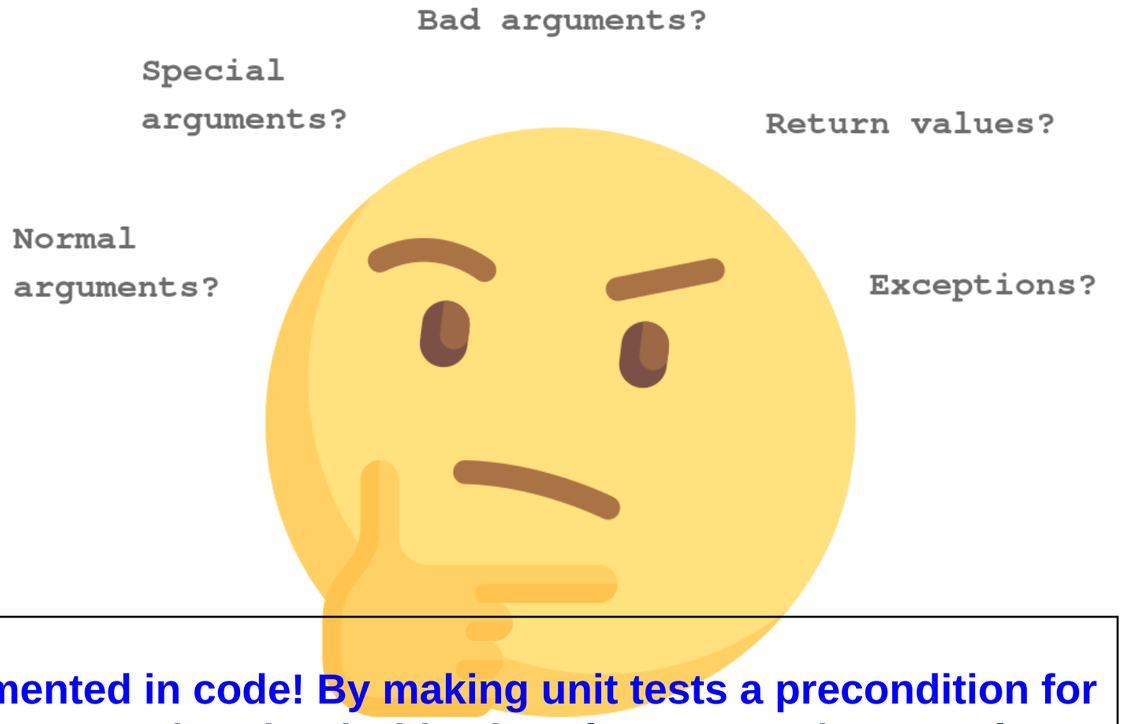
Test Driven Development (TDD)

6. Test Driven Development (TDD)
Test Driven Development alters the usual life cycle by adding a single step before implementation.
This step involves writing unit tests for the function.



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.



7. Write unit tests before implementation!
Yes, that right. We write tests even before the function is implemented in code! By making unit tests a precondition for implementation, this ensures that writing unit tests cannot be postponed or deprioritized. It also means that we, along with our bosses, should factor in the time for writing unit tests as a part of implementation time. Furthermore, when we write unit tests first, we have to think of possible arguments and return values - which includes normal, special and bad arguments. This type of thinking before implementation actually helps in finalizing the requirements for a function. When the requirements for a function is clear and precise, it makes the implementation much easier.

In the coding exercises...

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

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

```
2081
```

8. In the coding exercises...
That's all the theory we need. In the coding exercises following this video lesson, you will apply this coding method to the function convert_to_int(). We have seen this function before. It converts an integer valued string with comma as thousands separator to an integer.

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():
    ...
```

9. Step 1: Write unit tests and fix requirements
In the exercises, you will start with a blank slate, which means that the function is not yet implemented. Then you will go through a three step process. First, you will write the tests for this function in the test module test_convert_to_int.py. As you write the unit tests, you will think more about the requirements of this function.

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, inifile:
plugins: mock-1.10.0
collecting ...
collected 6 items

test_convert_to_int.py FFFFFF                                            [100%]

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

10. Step 2: Run tests and watch it fail
Then you will execute the test module. Of course, the tests will not pass because the function does not even exist yet!!

Step 3: Implement function and run tests again

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

11. Step 3: Implement function and run tests again
Finally you will implement the function and run the tests again. If you implemented the function correctly, then the tests should pass this time. Otherwise, you would have to fix bugs and repeat this step.

```
!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, inifile:
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