Mastering assert statements

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

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
```

```
0.30000000000000004
```

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

datacamp

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],
        ]
       ),
 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)
```

```
(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

datacamp

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

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.

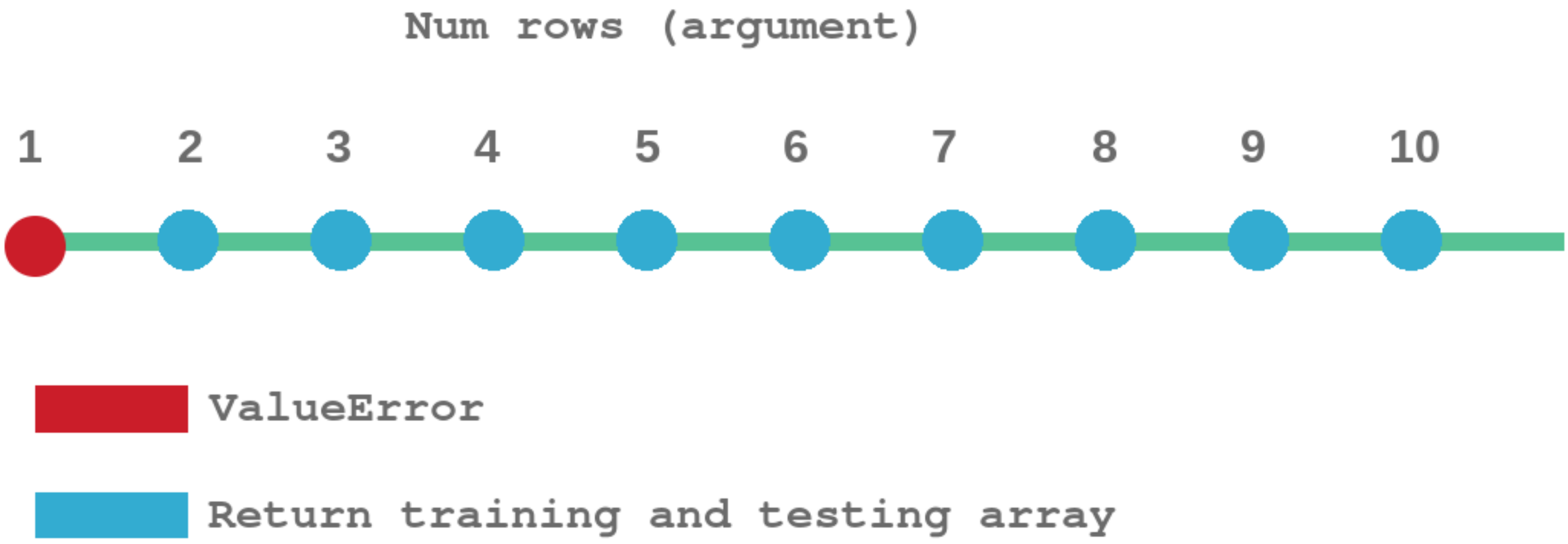
Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	`ValueError`
Contains 1 row	Bad	-	-	`ValueError`

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

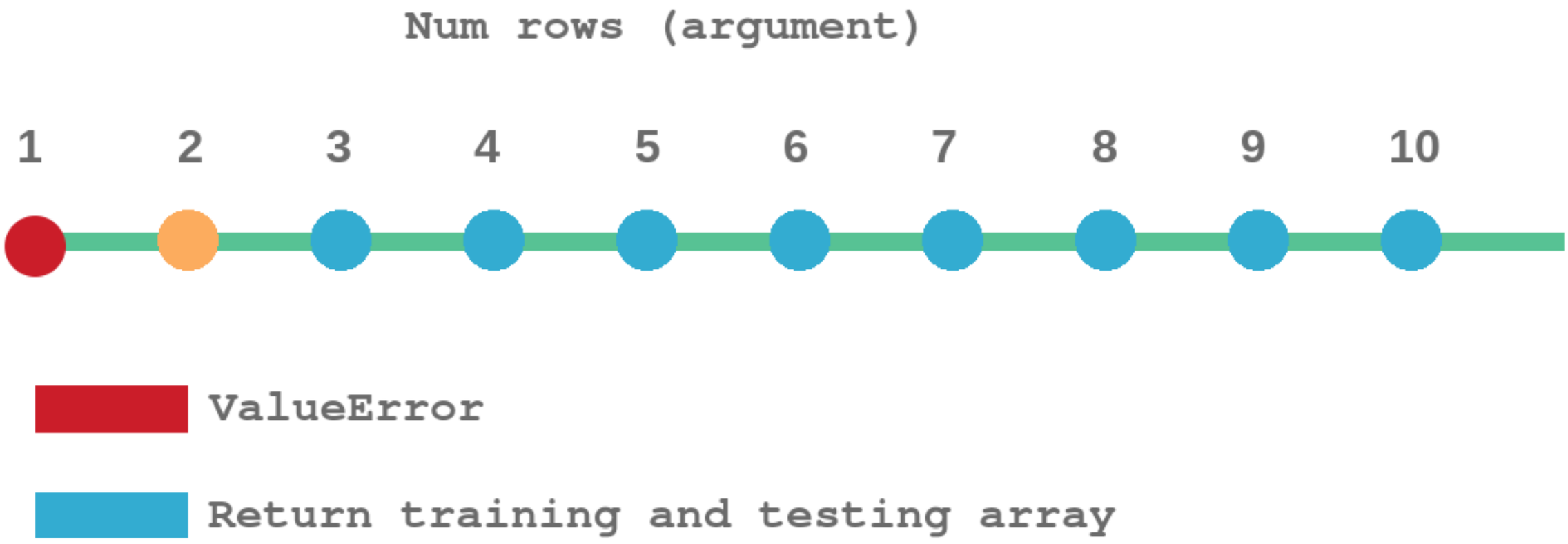
Type II: Special arguments

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

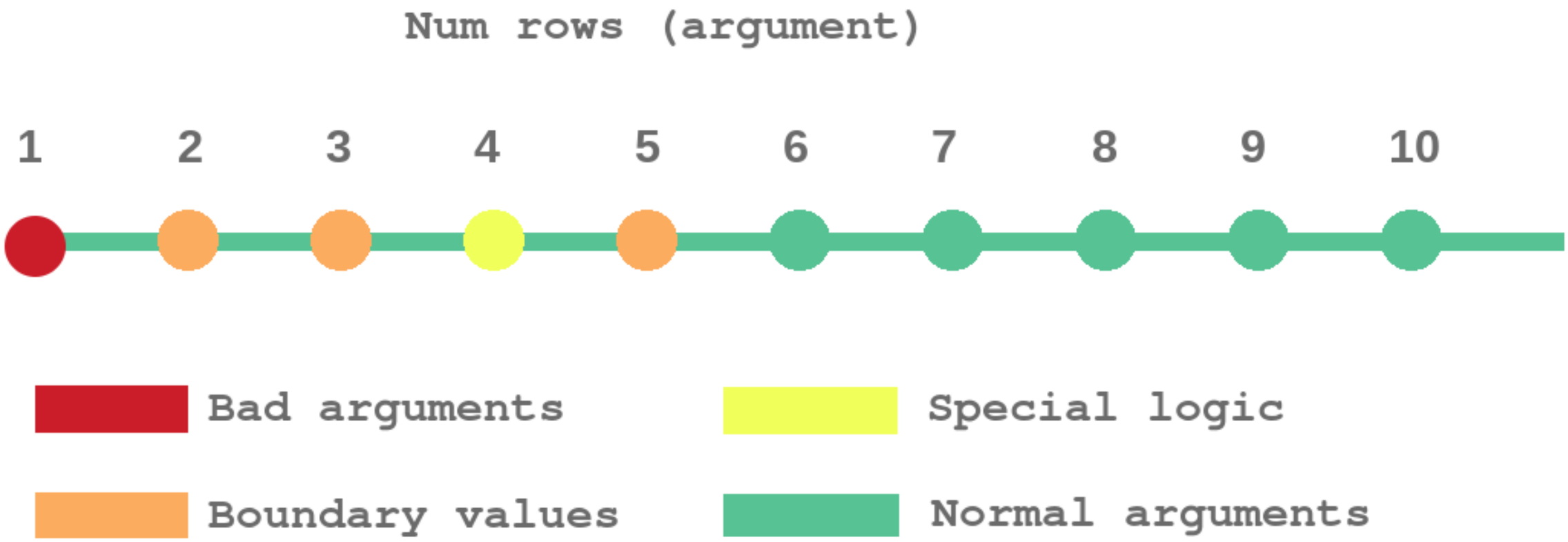
Arguments triggering special logic

Argument	Type	Num rows (training)	Num rows (testing)	exceptions
One dimensional	Bad	-	-	`ValueError`
Contains 1 row	Bad	-	-	`ValueError`
Contains 2 rows	Special	`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	-

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	-

```
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

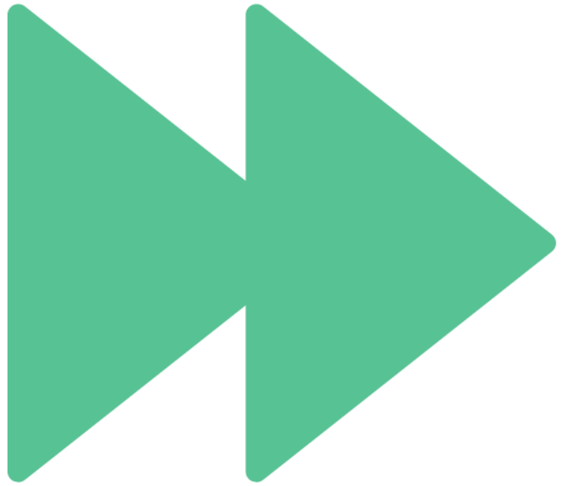
Test Driven Development (TDD)

UNIT TESTING FOR DATA SCIENCE IN PYTHON

Dibya Chakravorty
Test Automation Engineer

datacamp

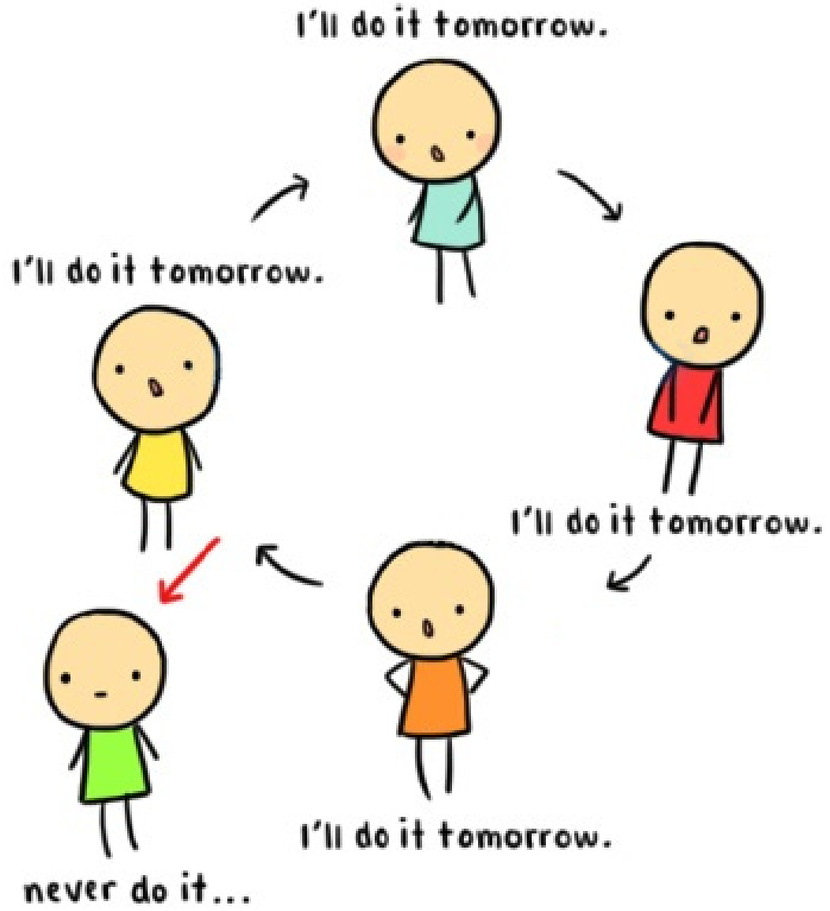
Writing unit tests is often skipped



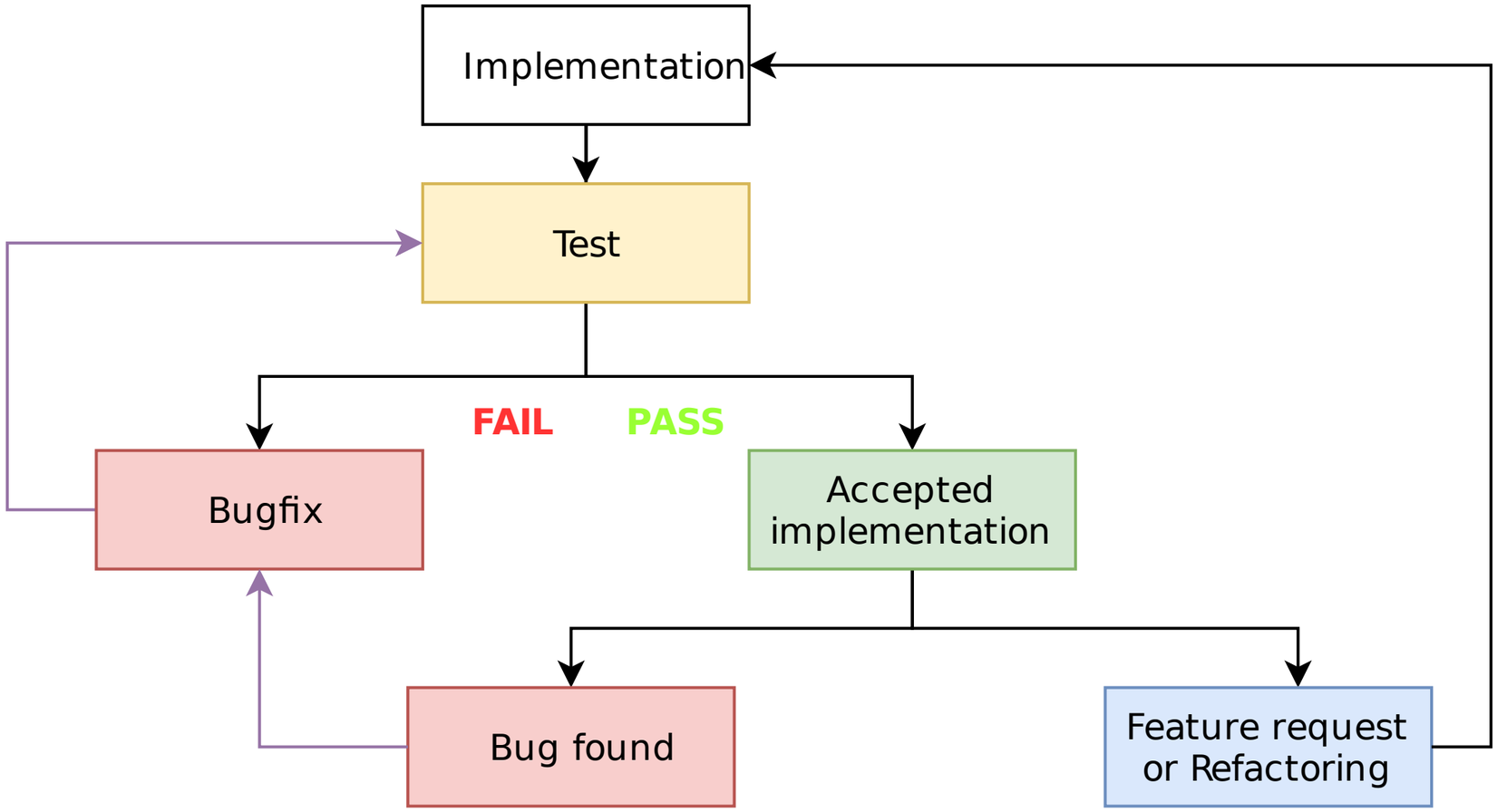
Usual priorities in the industry

1. Feature development.
2. Unit testing.

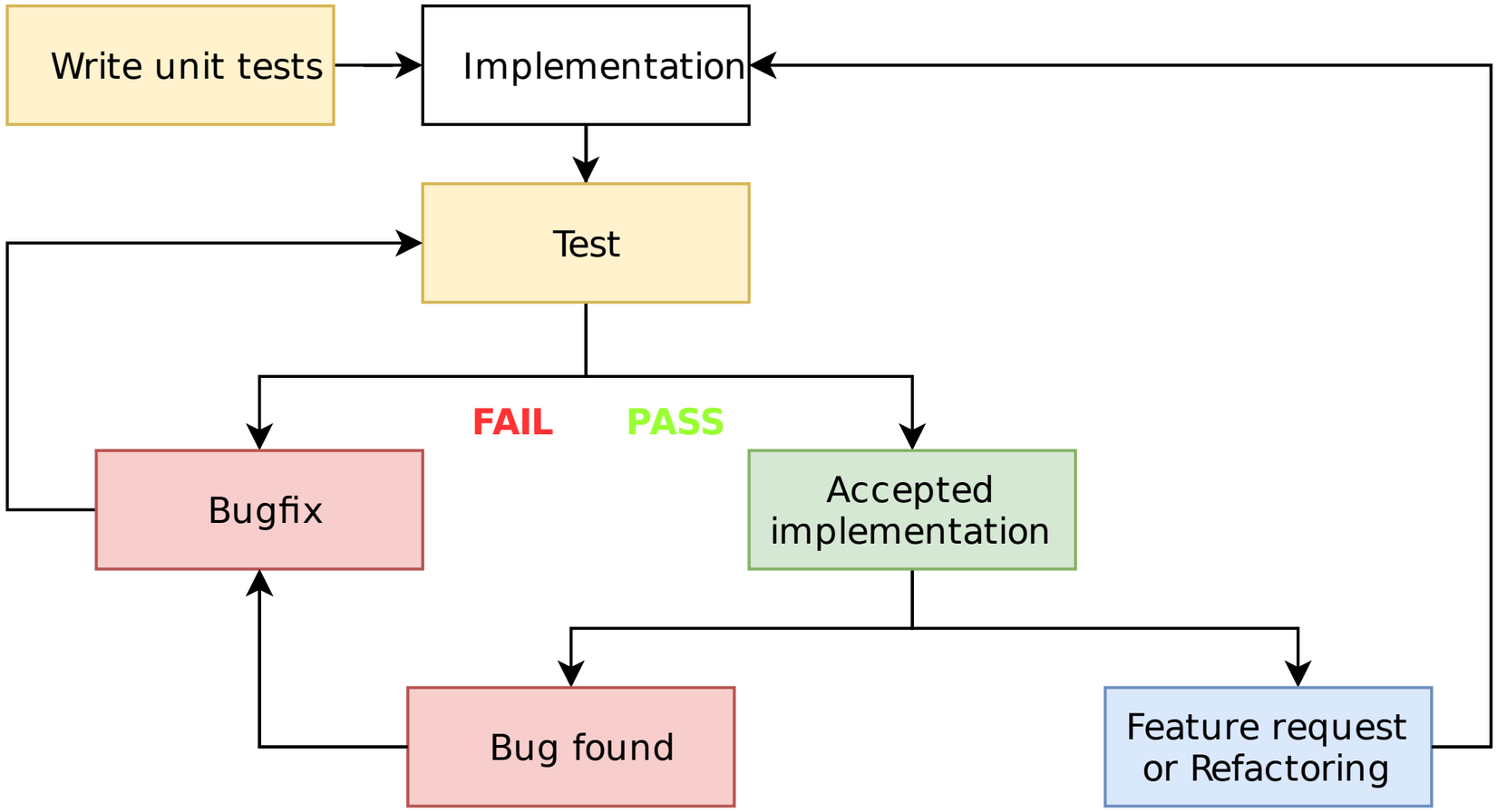
Unit tests never get written



Test Driven Development (TDD)

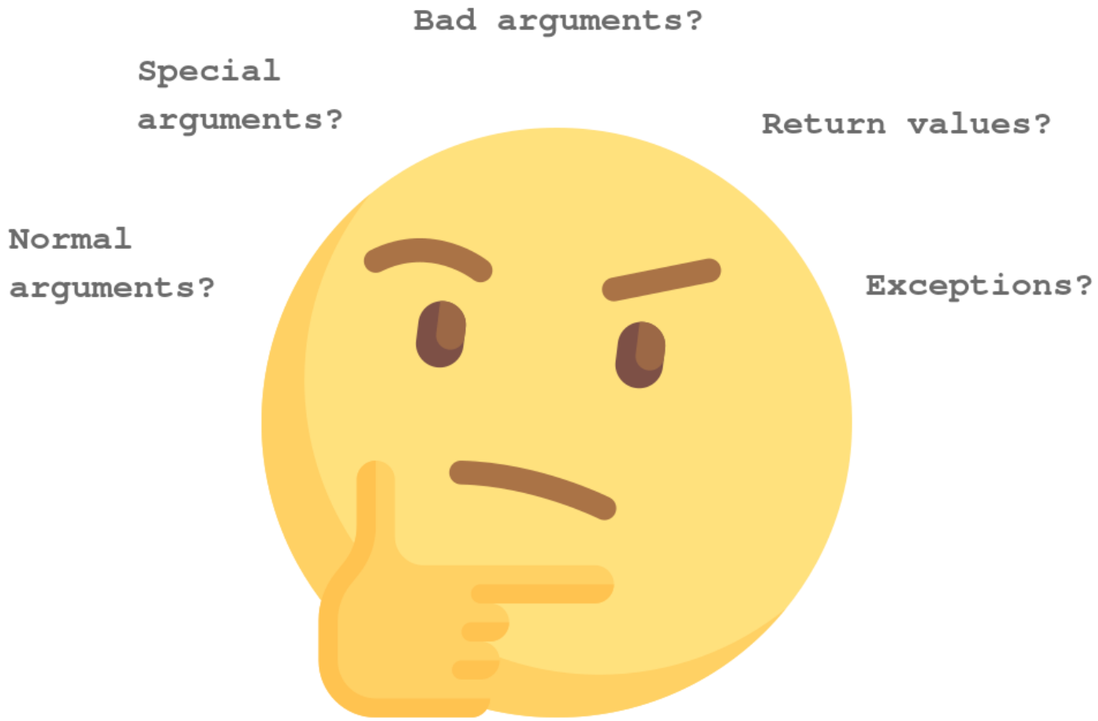


Test Driven Development (TDD)



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

test_convert_to_int.py FFFFFF                                            [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, 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