Unit Testing

Reference: Chakravorty, D. *Unit testing for data science in Python* [online course]. DataCamp. https://campus.datacamp.com/courses/unit-testing-for-data-science-in-python

Introducing Unit Tests with a Hypothetical Example

As data science becomes more of a staple in large organizations, the need for proper **testing of code** slowly becomes more integrated into the skillset of a data scientist.

Imagine you have been creating a pipeline for predicting customer churn in your organization. A few months after deploying your solution, you learn that there are new variables that might improve its performance.

```
UnknownError: I have absolutely no idea what is going wrong! Help me!
```

Unfortunately, after adding those variables, the code suddenly stops working! You are not familiar with the error message, so you have trouble finding your mistake. Or, perhaps you leave the company and a former colleague of yours is using your functions/code and doesn't understand the error message. This is where **unit testing** comes in!

Writing tests for specific modules improves the stability of your code and makes mistakes easier to spot. Especially when working on large projects, having proper tests is essentially a basic need.

> *No data solution is complete without some form of testing*

Introduction

- **What** is unit testing? It is the systematization of the idiosyncratic testing that you do to see if the code you are writing works as intended.
  - A unit test can be either a function or a class.
  - Python unit testing libraries, include pytest, unittest, nosetests, and doctest.

- **Why** do we use unit tests?
  - Prevent unexpected output
  - Simplify updating code
  - Increase overall efficiency of developing code (errors can cause your program to crash, but unit tests allow you to test all parts of your code in one go, even if one test fails)
  - Help detect edge cases
  - Most importantly, **they prevent you from pushing broken code into production!**

- **When** do you need to write unit tests? Use your judgment! Probably not necessary for a 1–2-day analysis
  - Remember, you are a data scientist, not a developer or engineer!

Test Suite Organization

Every application code module should have a corresponding test module; for example:

src/	tests/
- - data_processing/	- - data_processing/
- - - - __init__.py	- - - - __init__.py
- - - - data_cleaning_classes.py	- - - - test_data_cleaning_classes.py
- - feature_generation/	- - feature_generation/
- - - - __init__.py	- - - - __init__.py
- - - - feature_generation_classes.py	- - - - test_feature_generation_classes.py

Test Suite Organization

- The file name for a test module should start with “test_”.
- A class name should be in CamelCase and should always start with “Test”.
- A unit test name should start with “test_”.

```
class TestCamelCase():
      def test_function_one(self):
            assert x == y
```

Assert Statements

- A unit test usually corresponds to exactly one entry in the argument and one return value. The unit test, using an assert statement, will check that the expected return value is generated when the argument being tested is entered.

- The first argument of an assert statement must be a Boolean expression. If the assertion is True, then there is no output, but if the assertion is False, then an AssertionError is raised.

```
assert x == y, “A message here about why the assertion error was raised.”
```

- A unit test can have more than one assert statement, but the test will only pass if both assertions are true. The unit test will fail if either assertion is not true.

Assert Statements

There are a few different kinds of assert statements, depending on your use case:

Checking	Example Code

Floats	`import pytest`
	
	`assert 0.1 + 0.2 == pytest.approx(0.3)`
	
	`assert numpy.array([0.1 + 0.2, 0.2 + 0.3]) == pytest.approx(numpy.array([0.3, 0.5]))`

Data Type	`assert isinstance(123, int)`
None	`assert return_value is None`

What to Test For

- Write tests that check the function with different types of arguments, specifically bad arguments, special arguments, and normal arguments.
    - **Bad arguments** = the argument passed into the tested function raises an exception
    - **Special arguments** = boundary values that are the minimum or maximum arguments that can be successfully passed into the tested function and/or values that trigger special logic in the function; values that neighbor values that trigger special logic are also considered boundary values, and thus, should also be tested
    - **Normal arguments** = the generic argument for which the function was built and that can be successfully passed into the tested function; test two or three normal arguments
- Not all functions have special or bad arguments, so it is not expected that unit tests will always have to cover all three types of arguments.
- Boundary values are also commonly called edge and corner cases. Refer to this resource for specific examples.

Running Tests

Once you have changed your working directory to the “tests” folder (i.e. `cd tests`), there are a number of different ways to run tests:

Purpose	Example Commands*
Run all tests in the “tests” folder	`!pytest`
Stop the running of tests after the first failed test	`!pytest -x`
Run the tests inside of a single test module	`!pytest folder/test_module.py`
Run a test class using its node ID	`!pytest folder/test_module.py::TestClassName`
Run a unit test using its node ID	`!pytest folder/test_module.py::TestClassName::unit_test_name`
Run tests using keyword expressions	`!pytest -k “string_to_match_here”`
Omit tests using a Python logical operator	`!pytest -k “StringToMatchHere and not this_one_here”`

* The exclamation mark before pytest is needed to run these commands in the IPython Console, but it is not always required.

Reading Test Results

Running tests generates reports with lots of content:

Content	Meaning
`collected # items`	The number of tests that pytest found to run; this number should equal the number of tests that you expected to run.
`.`	A dot next to a test module's name means that a unit test passed.
`F`	This letter next to a test module's name means a unit test failed.
`FAILURES`	This is the title of the section that contains information on failures.
`>`	This symbol points to the line in the code that raised the exception.
`E`	The lines following ">" with "E" contain details of the exception.
`E + where`	The part with "where" displays the calculated return value, so you can see how that return value differs from the expected return value.
The report's final line	The final line of the report shows the number of unit tests that failed and the number of unit tests that passed, as well as how long it took to run the tests.

What Makes Good or Bad Unit Tests? Pro Tips From Stack Overflow

- The **best unit tests** are **simple** to read and understand, are **quick to execute**, test **specific functionality**, are **well factored,** and are **well maintained**.

- **Readability is king** when it comes to unit tests. If it takes more than one minute to understand, then there's probably something wrong. Any test that's more than five lines long had better have a pretty good excuse.

- Tests that are brittle usually have **unacceptable maintenance overhead,** which will eventually lead to the tests not being updated and being left in a broken state. **Brittle unit tests usually have dependencies** on file systems, registry keys, databases etc... These dependencies are fine for integration and system tests, but sometimes there are unit tests with these dependencies and that's usually a problem.

- **Bad unit tests do not test enough or anything at all**. For example, just checking the return value from the method is not good enough if related output parameters or object data are not validated upon method execution. This leads to cases where tests are running fine and coverage may be high, but you are not validating anything really...

Conclusion

- Unit testing is the systematization of the idiosyncratic testing that you do to see if the code you are writing works as intended.
- Every application code module should have a corresponding test module.
- A unit test usually corresponds to exactly one entry in the argument and one return value. The unit test, using an assert statement, will check that the expected return value is generated when the argument being tested is entered.
- To have well tested functions, you should write tests that check the functions with different types of arguments, specifically bad arguments, special arguments, and normal arguments.
- You should write unit tests so that your code can be easily tested, and thus, you can securely believe that your code works as intended.