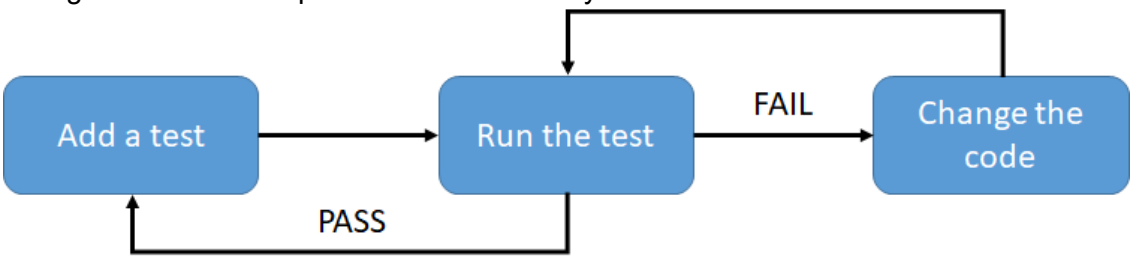
Unit Testing and Test-Driven Development
Software Engineering for Scientists

In broad terms, “testing” is a process by which you verify that your code is performing as intended/anticipated. There are essentially two types of tests you can perform: **unit tests** and **functional tests**.

Functional testing is a high-level testing process, done at the scale of the full piece of software. This can include the use of “test data sets,” that, when run, should produce a predefined output. The functional test passes if the predefined output is achieved. Generally, this testing is done for the sake of the user, to demonstrate that the software is performing as expected, without the user having to be directly concerned with the details of the code that comprise the software.

Unit testing is a low-level testing process, usually applied at the level of individual functions within the code. Unlike functional testing, unit testing is done during code development and is integrated into the developers “process.” Unit tests are small, specific, and performed frequently as the code base is created. Thus, unit testing is done for the sake of the developer and helps to guide the trajectory and overall architecture of the code itself.

Test-Driven Development is a formalized way of performing your unit tests. It consists of three interlocking parts: adding a test, running the test, and incrementally adding code until your test passes.The extremely non-intuitive aspect of TDD is that ***the test is written BEFORE the code is written***. The code is then changed by the smallest amount in order to pass the initial test (and no further!!). Once the test is passed, a new test is created (which fails) and then the code is again changed until that test passes as well. This cycle is carried out *ad infinitum*.



Though this may seem like a backwards way of writing your code it has been broadly demonstrated, empirically, to produce clearer and more robust code across all applications. Hypothetically, this is because the narrow scope of the tests prevent developers from over-engineering their code. And, by explicitly stating the pass condition, the developer concretizes the purpose of each code unit.

Within Python’s base libraries there is one that facilitates unit testing (aptly named `unittest`). It is this library we will use to practice our test-driven development process. But first we will familiarize ourselves with its functionality and syntax.

Say that your software has a module (`my_module.py`) in which you’ve created three functions:

`func1()`, `func2()`, and `func3()`. In order to unit test these functions, one must create a new script (lets call it `my_module_test.py`) in which we create a *testing class* that inherits (takes as input) the `unittest.TestCase` object. This object contains all the testing tools which we will apply to our functions (see the below code template).

For each of the functions in our module, we will create a companion test function within the testing class (let's call it `ModuleTest`). *The naming convention for these test functions should follow a consistent naming convention:* ***it's recommended they start with "`test_`" followed by the name of the function they are testing***. See the example code below.

File `my_module_test.py`:

```
import unittest
import my_module as mm

# A class in which we test the functions of our module
class ModuleTest(unittest.TestCase):

    # a function to test a function in our module (must start with 'test_')
    def test_func1(self):
        self.assertEqual(mm.func1(...), b)
        self.assertIsNotNone(mm.func1(...))
        …

    # This tests if the function output is of a certain type
    def test_func2(self):
        self.assertIsInstance(mm.func2(...), q)
        …

    # Checks if the value of the output is in x. Also, if it is less than y.
    def test_func3(self):
        self.assertIn(mm.func3(...), x)
        self.assertLess(mm.func3(...), y)
        …


if __name__ == "__main__":
    unittest.main()
```

Here I demonstrate a number of the test case methods that can be used to evaluate the output of your functions against the expected result. These methods include `assertEqual` (checks that the output is exactly equal to the expected), `assertNone` (checks that the output is a `None` type), `assertIsInstance` (checks that the data type of the output is of a particular type), `assertAlmostEqual` (checks that the output is equal to another numeric value to within a specified precision), `assertIn` (checks the output is contained in some set/list/tuple), etc.

For the full list of available test case methods see the unittest documentation here.

You can use the above example as a template for all your unittesting. Though we haven't covered Python classes so far, you can leave the general structure of the class as is. The only things you'll need to change to implement it for yourself is importing your specific module, changing the

function names to reflect your own functions, and deciding which test case methods you want to use to check your output (NOTE: the `self`object is a reference to the class and is required syntax; just make sure all assert methods are referenced from `self` and that `self` is an input variable to all your test functions).

Test-Driven Development Example

Let's consider an example to demonstrate how to perform TDD. **Say we want to write a function to calculate the average of a list of values** (like we did in class for the example demonstrating modularity). Now let's recreate that function using the TDD approach. We will call the `function list_avg()` and it will be located in the python file `my_lib.py`.

We will start by creating a template for our testing script (`test_my_lib.py`), which will be testing the (yet to be created) function in the library file `my_lib.py`.

`test_my_lib.py`

```
import unittest
import my_lib

class TestLib(unittest.TestCase):
    pass

if __name__ == "__main__":
    unittest.main()
```

Now we create our first test function to start the development process. The idea is to start with the most basic possible test. For example, if we pass no list to the function (i.e. `None`) then we would expect the function to return `None`.

NOTE: The following side-by-side code shows the test functions we add to in order to get a failing test (left panels) and then the subsequent changes we make to the function in `my_lib.py` to pass the test (right panels).

`test_my_lib.py` (in `TestLib`)

```
def test_list_avg_empty(self):
    res = my_lib.list_avg(None)
    self.assertEqual(res, None)
```

`my_lib.py`

```
def list_avg(vals):
    if vals is None:
        return None
    return 0
```

For the above code, *before* we add functionality to `my_lib.py`, we get our failing test:

```
$ python test_my_lib.py
E
======================================================================
ERROR: test_avg_null (__main__.TestMyLib)
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
  File "/home/jovyan/python_testing/test_my_lib.py", line 6, in
test_list_avg_null
    res = my_lib.list_avg(None)
AttributeError: module 'my_lib' has no attribute 'list_avg'

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
$
```

Note the first line after the command indicates how many tests ran and whether or not they errored/failed or passed. This is indicated by a number of “F” (or “E”) characters in the event of a failed/errored test or “.” characters in the event of a pass, so in this case we see one “E” which means we had one test and it errored (this is due to the fact that we didn’t have a function.

After adding the functionality to `my_lib.py` we get our passing test:

```
$ python test_my_lib.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
$
```

Here we see a single “.” character, which indicates we have one test and it passed.

Next, we write a failing test in which we provide an empty list as input. Again, our expectation is that `None` should be returned.

`test_my_lib.py` (in `TestLib`)

```
def test_list_avg_empty(self):
    res = my_lib.list_avg(None)
    self.assertEqual(res, None)

    res = my_lib.list_avg([])
    self.assertEqual(res, None)
```

`my_lib.py`

```
def list_avg(vals):
    if vals is None:
        return None
    if len(vals) == 0:
        return None
    return 0
```

Prior to including the additional code in my_lib.py, we get the following failing test:

```
$ python test_my_lib.py
F.
======================================================================
FAIL: test_list_avg_empty (__main__.TestMyLib)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/jovyan/python_testing/test_my_lib.py", line 11, in
test_list_avg_empty
    self.assertEqual(res, None)
AssertionError: 0 != None
```

```
----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
$
```

Here we see the first line (“F.”) indicates one passing test and one failing test (this time it is a “fail” not an “error”). The failure occurred because we didn’t have a specific return for an empty list and so the default “0” was returned instead.

Now that we’ve addressed null inputs we can test the basic behavior with constant inputs. We should test multiple different lengths and values contained in the input list.

`test_my_lib.py` (in `TestLib`)

```
def test_list_avg_const(self):
    ones = my_lib.list_avg([1, 1, 1])
    self.assertEqual(ones, 1)

    test_dat = [0, 0, 0, 0, 0]
    neg = my_lib.list_avg(test_dat)
    self.assertEqual(neg, 0)

    test_dat = [-5, -5, -5, -5, -5]
    neg = my_lib.list_avg(test_dat)
    self.assertEqual(neg, -5)

    test_dat = [12, 12, 12]
    neg = my_lib.list_avg(test_dat)
    self.assertEqual(neg, 12)

    test_dat = [23]
    neg = my_lib.list_avg(test_dat)
    self.assertEqual(neg, 23)
```

`my_lib.py`

```
def list_avg(vals):
    if vals is None:
        return None
    if len(vals) == 0:
        return None

    total = 0.0

    for val in vals:
        total += val

    avg = total/len(vals)
    return avg
```

Changing `list_avg()` to the code at right results in all tests passing.

Now, for more realistic data, let’s generate lists of random floats as inputs to test the function. We import the `random` and `statistics` libraries to carry out our test. In our test we generate a list of random length (between 1 and 100 elements) containing random floats between 0 and 100. And then we confirm the calculated average by comparing it to an independent function `statistics.mean()` using the `assertEqual` test case. We leave the function from above unchanged and run our test.

`test_my_lib.py` (in `TestLib`)

```
def test_list_avg_floats(self):
    for _ in range(10):
        vals = []
        size = random.randint(1, 100)

        for _ in range(size):
```

```
            vals.append(random.uniform(0, 100))

        res = my_lib.list_avg(vals)
        exp = statistics.mean(vals)

        self.assertEqual(res, exp)
```

We get the following failure when running the test:

```
$ python test_my_lib.py
..F.
======================================================================
FAIL: test_list_avg_floats (__main__.TestMyLib)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/jovyan/python_testing/test_my_lib.py", line 49, in
test_list_avg_floats
    self.assertEqual(res, exp)
AssertionError: 29.31280625663241 != 29.312806256632413

----------------------------------------------------------------------
Ran 4 tests in 0.002s

FAILED (failures=1)
$
```

We see that the `test_list_avg_floats` test fails, BUT the statement as to what caused the fail seems incorrect, namely:

"`AssertionError: 29.31280625663241 != 29.312806256632413`"

What's happening here is that the `assertEqual` test case requires that the two objects being compared must be *exactly* equal, down to the bit, but in the case of floats there is some uncertainty at a very high precision (in this case at 10e-15) which results in an ever so slightly different number. Therefore we must use a different test case `assertAlmostEqual` which is more suited to comparing floats and allows us to define a precision for our test. See the following change:

`test_my_lib.py` (in `TestLib`)

```
def test_list_avg_floats(self):
    for _ in range(10):
        vals = []
        size = random.randint(1, 100)

        for _ in range(size):
            vals.append(random.uniform(0, 100))

        res = my_lib.list_avg(vals)
        exp = statistics.mean(vals)

        self.assertAlmostEqual(res, exp, places=10)
```

Lastly, let's demonstrate how TDD can be applied to exception handling. Say we want our function `list_avg()` to only be applied to inputs of type `list`. So, in the event that any other data type is input to the function, we will want to raise a `TypeError` exception.

`test_my_lib.py` (in `TestLib`)

```
def test_list_avg_nonlist(self):
    # integer input
    self.assertRaises(
        TypeError,
        my_lib.list_avg,
        2.0)

    # dictionary input
    self.assertRaises(
        TypeError,
        my_lib.list_avg,
        {'a': 1, 'b': 2, 'c': 3})

    # set input
    self.assertRaises(
        TypeError,
        my_lib.list_avg,
        {4.4, 55, -0.4, 18})
```

`my_lib.py`

```
def list_avg(vals):
    if vals is None:
        return None
    if len(vals) == 0:
        return None
    if not isinstance(vals, list):
        raise TypeError(
            "Not correct data type")

    total = 0.0

    for val in vals:
        total += val

    avg = total/len(vals)
    return avg
```

Here, to pass the new test, we include an `if/raise` clause which checks that the data type of the input with the `isinstance()` function.