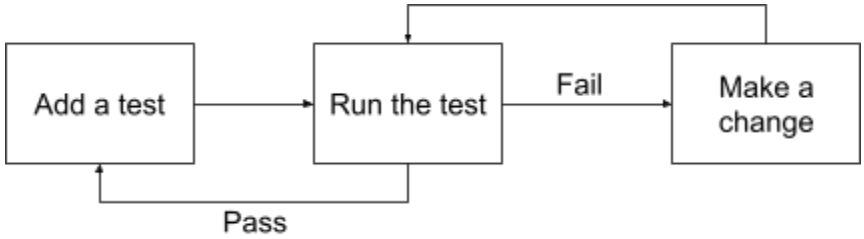
Test-Driven Development
Software Engineering for Scientists

https://github.com/swe4s/lectures/tree/master/src/tdd/math

In **Test-driven development** (TDD) tests are written first, then functions are written to pass the tests. In practice, tests and features co-evolve. You will start with a test that checks the most basic functionality, then add more sophisticated tests that address different features and corner cases. TDD ensures correctness from the beginning and good test coverage, encourages modularity, and forces you to think through the design and of your code before implementation.

The basic workflow of TDD is:



In this workflow, you do not write new code until you have a test that fails.

Math library example

Suppose we want to find the mean of a column in a file. Determining the first test requires us to decompose the high-level behavior into individual, and ideally independent, tasks. Here the independent tasks are to get a list of numbers from a file and to find the mean of a list of numbers. Starting with the latter, we will use the TDD workflow to develop the `list_mean` method that takes a list of numbers and returns the mean.

`list_mean`

The idea of TDD is to start with the most basic possible test. One option for `list_mean` is to pass `None` and expect `None` to be returned. Since `math_lib.py` is empty and this test fails, we can add the code to satisfy the test.

`test_math_lib.py`

```
import unittest
import math_lib

class TestMathLib(unittest.TestCase):
    def test_list_mean_empty_list(self):
        r = math_lib.list_mean(None)
        self.assertEqual(r, None)

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

`math_lib.py`

```
def list_mean(L):
    if L is None:
        return None
```

Next, we can write a test that passes an empty list and again expects `None` to be returned.

test_math_lib.py

```
def test_list_mean_empty_list(self):
    r = math_lib.list_mean([])
    self.assertEqual(r, None)
```

math_lib.py

```
def list_mean(L):
    if L is None:
        return None
    if len(L) == 0:
        return None
```

NOTE: Up to this point `list_mean` could just return None and it would satisfy both tests, but I would encourage you to write code that addresses the next and not write code that just so happens to pass the test because Python didn't complain.

Now that empty lists are taken care of we can test the basic behavior with a constant. The test fails and so we add code to find the mean.

test_math_lib.py

```
def test_list_mean_constants(self):
    r = math_lib.list_mean([1,1,1,1])
    self.assertEqual(r, 1)
```

math_lib.py

```
def list_mean(L):
    if L is None:
        return None
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        s += l

    return s/len(L)
```

With that test passing, we can some randomness to our test. Don't forget to `import random` and `import statistics`. To get the full benefit of adding randomness to your tests (the inner for-loop), it is important to run many random trials (the outer for-loop). This test passes, which means we do not need to write any more code.

test_math_lib.py

```
def test_list_mean_rand_ints(self):
    L = []
    for i in range(100):
        for j in range(10):
            L.append(
                random.randint(0,100))
        r = math_lib.list_mean(L)
        e = statistics.mean(L)
        self.assertEqual(r, e)
```

math_lib.py

```
def list_mean(L):
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        s += l

    return s/len(L)
```

Now we want to make sure that `list_mean` works for integers and floats, so we write a similar test for floats.

test_math_lib.py

```
def test_list_mean_rand_floats(self):
    L = []
    for i in range(100):
        for j in range(10):
            L.append(
                random.uniform(0,100))
        r = math_lib.list_mean(L)
        e = statistics.mean(L)
        self.assertEqual(r, e)
```

math_lib.py

```
def list_mean(L):
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        s += l

    return s/len(L)
```

This test is very likely to fail with the following error (or something similar).

```
$ python test_math_lib.py
.F.
======================================================================
FAIL: test_list_mean_rand_floats (__main__.TestMathLib)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "x.py", line 40, in test_list_mean_rand_floats
    self.assertEqual(r, e)
AssertionError: 42.71697048999344 != 42.716970489993436

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

The issue is in our test, not in our code, and it is a common issue when comparing floating-point numbers. Our observed value from `math_lib.list_mean` was `42.71697048999344`, and the expected value (from `statistics.mean`) was `42.716970489993436`. While these numbers are close (within 0.000000000000004), a small rounding difference caused them to not be identical. When comparing floats strict equality (`==` or `assertEqual`) should be replaced with a more lenient comparison. In Python, we can use `math.isclose` (don't forget to `import math`) or `assertAlmostEqual()` from `unittest`.

test_math_lib.py

```
def test_list_mean_rand_floats(self):
    L = []
    for i in range(100):
        for j in range(10):
            L.append(
                random.uniform(0,100))
        r = math_lib.list_mean(L)
        e = statistics.mean(L)
        self.assertTrue(math.isclose(r, e))
```

math_lib.py

```
def list_mean(L):
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        s += l

    return s/len(L)
```

Using `math.isclose` is also important when testing the implementation of a model where you have a set of equations that give expected values. In the case of finding the mean, we could use the formula for the mean of numbers from a uniform distribution (a.k.a random) instead of using `statistics.mean`.

test_math_lib.py

```
def test_list_mean_rand_floats_model(self):
    L = []
    u = (0 + 100)/2.0
    for i in range(100):
        for j in range(10):
            L.append(
                random.uniform(0,100))
        r = math_lib.list_mean(L)
        e = statistics.mean(L)
        self.assertTrue(math.isclose(
            r, u, rel_tol=10.0, abs_tol=0.0))
```

math_lib.py

```
def list_mean(L):
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        s += l

    return s/len(L)
```

Our tests should also make sure that our method is robust to bad parameters. There are many types of bad parameters, one is a list that contains values where the mean is undefined (i.e., something other than integers and floating-point numbers). In the case of bad parameters, we expect `list_mean` to raise a `ValueError` exception.

test_math_lib.py

```
Def test_list_mean_non_int_in_list(self):
    L = []
    for i in range(10):
        L.append(random.randint(0,100))
    L.append('X')

    with self.assertRaises(ValueError) as ex:
            math_lib.list_mean(L)
```

math_lib.py

```
def list_mean(L):
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        s += l

    return s/len(L)
```

While this test will pass, it is not a good test because it allows code that does not fit within our best practices the exception is raised in the method and not caught.

test_math_lib.py

```
Def test_list_mean_non_int_in_list(self):
    L = []
    for i in range(10):
        L.append(random.randint(0,100))
    L.append('X')

    with self.assertRaises(ValueError) as ex:
            math_lib.list_mean(L)

    self.assertEqual(
        'Unsupported value in list.')
```

math_lib.py

```
def list_mean(L):
    if len(L) == 0:
        return None

    s = 0

    for l in L:
        try:
            s += l
        except ValueError:
            raise ValueError(
                'Unsupported ' +
                'value in list.')

    return s/len(L)
```