Test-Driven Development

The idea of TDD

The idea of **Test-Driven Development**.

1. Write the test.
2. Run it, and watch it fail.
3. THEN write code to make the test pass.

This has some surprising benefits:

- Generally more robust software
- Code clarity and testability
- State of Flow

And some downsides.

See The Test Fail, CORRECTLY

You MUST do this. Every time. If you don't, you will **suffer**.

Correctly failing test:

```
Traceback (most recent call last):
  File "/Users/amax/testdemo/test_splitting.py", line 6, in test_split_amount
    self.assertEqual([1], split_amount(1, 1))
AssertionError: [1] != None
```

Broken test:

```
ImportError: Failed to import test module: test_splitting
Traceback (most recent call last):
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/unittest/loa
der.py", line 154, in loadTestsFromName
    module = __import__(module_name)
  File "/Users/amax/testdemo/test_splitting.py", line 2, in <module>
    from splitting import split_amount
ImportError: cannot import name 'split_amount' from 'splitting'
(/Users/amax/testdemo/splitting.py)
```

Make It Pass

Only once you've verified the test fails correctly, then you write application code to make it pass.

```
Ran 1 test in 0.000s

OK
```

Once you start writing application code, you don't normally modify the test (unless you realize it has a bug or needs to be updated).

Benefits of TDD

Masterint TDD gives some massive, powerful benefits.

- Generally more robust software
- Code clarity
- Naturally architected for testability
- State of Flow

(Oddly, I’ve never heard anyone talk about the last one. But it’s strong enough that it’s a reason to do TDD all by itself.)

Learning TDD

The best way to learn TDD:

Strictly do it for at least a week.

No application code written AT ALL, unless you've first written a test, and you're writing code to make it pass.

This will be HARD for the first few days. Your productivity will temporarily plummet.

But it's the fastest, most solid way to master writing automated tests.

TDD and Version Control

A match made in heaven.

Work in a branch, so you can make as many commits as you want. And then:

- Commit #1: Correctly failing test. So you wrote the test, and watched it fail
- Commit #2: Write code to make that test pass
- Commit #3+: Refine the code, one or more times
- Repeat from the start, for the next feature.

Great way to leverage TDD to produce high quality code.

Test-Driven Downsides

TDD does have a few downsides:

- Hard to do without clarity on the program architecture or design
- Difficult with exploratory coding (e.g. data analysis)
- In certain coding situations, can lead to you repeatedly throwing out tests you just wrote

To TDD or Not?

People get religious about this. Be gentle with the zealots.

If you're new to writing tests, strictly following TDD for a while is a great way to get very good, very quickly. And remember, writing good tests is a critical skill.

Once you're fairly good at it: Consider following the 80-20 rule.

Lab: Textlib

In this self-directed lab, you implement a small library called `textlib`, and a test module named `test_textlib`.

Instructions: `lab-unittest.txt`

- In `labs` folder
- First follow the instructions to write `textlib.py` and `test_textlib.py`
- When you are done, study the solution - compare to what you wrote.
- … and then optionally follow the extra credit instructions