Test-Driven Data

Overview of software testing

- Manual testing
  - Trying out your code in a shell or Notebook
  - Making sure your application 1) doesnt break and 2) works as expected
  - Sending your product off to a customer or partner for use/review
- Automated testing
  - Designed to be run frequently and provide instant feedback on your code
  - Integrated deeply into many business procedures/pipelines (DevOps)
  - Essential – not optional – for clean code and good practices
  - Just as much art as science

Benefits of testing

- Helps to keep your code **modular**
- Helps keep you focused and on the right path
- Reduces bugs by catching them sooner
- Documents what your code does and proves that it does it
- Makes your code extensible and generalizable

Considerations for Data Science

- Can you be confident that your code is doing what it says it does?
  - Can your advisors/supervisors?
  - Reproducibility
- Does your code properly handle all of the elements in your data set for all potential situations? What about future experiments or additional fields?
- Will others be executing/modifying your analysis scripts?
- Testing can sometimes be tricky in a data context, since we can have unknown outputs.
  - Can you be confident with a mock or should you let the data guide the way?

MYTH: automated testing is for app developers, not data scientists
TRUTH: all of the principles of testing apply to a data-centric context, however, tooling and methods are somewhat underdeveloped

When to use TDD

- Your data processing is complex
- Your analysis needs to be accurate with a high degree of certainty.
- You are participating in highly collaborative work
- You find yourself frequently playing whack-a-mole with your code.
- You want your code to be stable and flexible

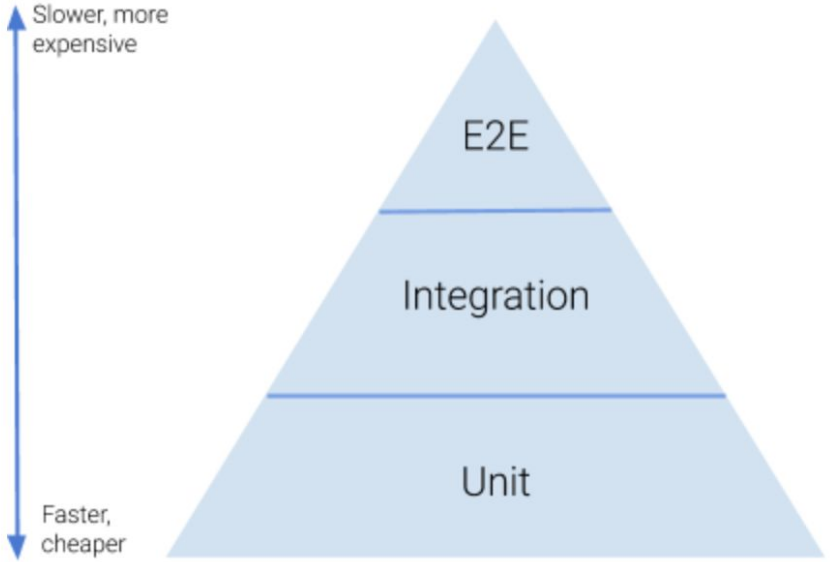
When not to use TDD*:

- You can perform your data analysis with common/built-in functions in Pandas/Plotly
- You are doing exploratory analysis/proof of concept
- You don't envision performing the analysis frequently (true one-off scripts)
- You are in a big rush, and a duct-tape approach is fine.

* - Rarely are there instances where a TDD approach will actually hinder you (other than the extra time). It makes you a better programmer, and working on the smaller problems can help you understand more complex testing cases and develop discipline. However, we all live in reality and even the most experienced/avid testers will skip some steps from time to time.

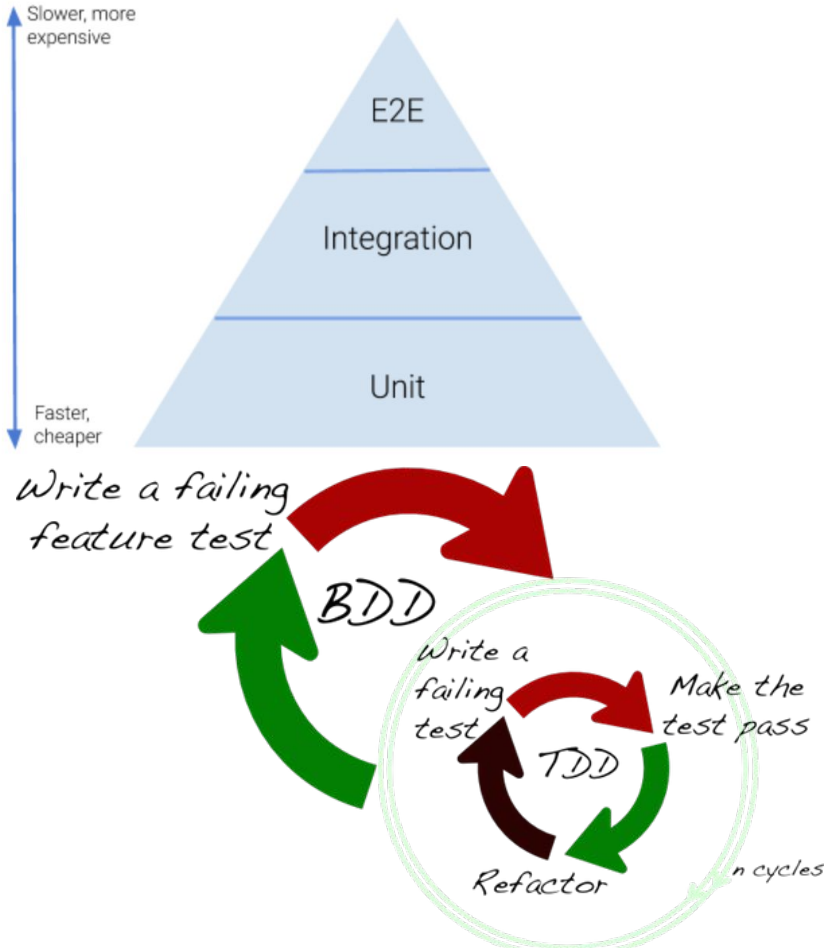
The Testing Pyramid

- Unit tests- test a single “unit” of code
  - Usually units are functions, but it ultimately comes does to behavior
- Integration tests - tests how your units interact with each other
- End-to-End tests - incorporates multiple aspects of your system and tests that everything is working from an end user standpoint



Two (main) approaches t integration testing

- “Top-down,” “outside-in,” or a “mockist” approach:
  - Start with E2E and write “mocks” for missing functionality
- “Bottom-up,” “inside-out,” or “classicist” approach -
  - Start with your unit tests and integrate them as you move up the pyramid, making ‘stubs’ for dependencies
- Various hybrid approaches



Test-Driven Development

- Write tests ***first***
  - What do I need to accomplish?
- **Red**-**Green**-Refactor
  - Only write code that makes your tests green
  - Only refactor when green, and only choose refactorings that keep it green.
- Make your test more strict or add new tests when you're satisfied.
- Requires discipline - Trust the Process

Anatomy of a test - Arrange/Act/Assert (AAA)

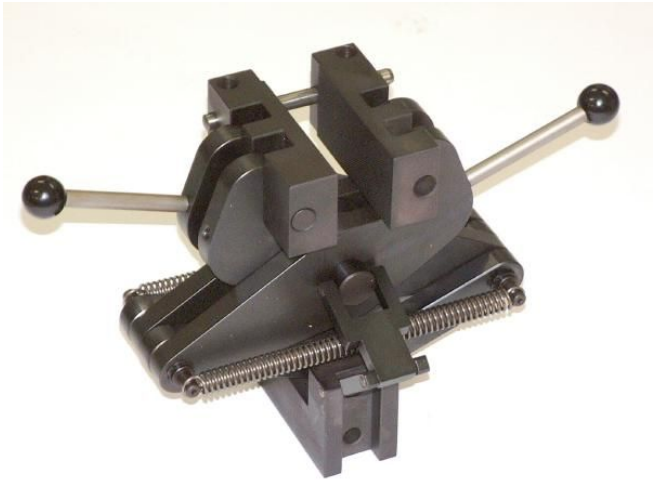
- **Arrange -** Set up the test. Import/mock data or external services.
- **Act** - Perform the function under test
- **Assert** - Ensure that the result of the action is as you expect.

```
def test_reverse():
    """
    list.reverse inverts the order of items in a list, in place
    """
    greek = ['alpha', 'beta', 'gamma', 'delta']

    result = greek.reverse()

    assert result is None
    assert greek == ['delta', 'gamma', 'beta', 'alpha']
```

Mocks and Fixtures

Your tests should be laser-focused on the problem at hand… but what do you do when your tests rely on complicated external dependencies? What about when you need to test multiple inputs?

- Pytest **fixtures** are fancy constructs that help us with the **arrange** step in our tests
  - Complicated or expensive setup steps
  - Cuts down on repetition
- **Mock objects** are substitutes for complicated (usually third-party) Python objects that you don't want to actually execute for whatever reason



A **fixture** is a work-holding or support device used in the manufacturing industry.[1][2] Fixtures are used to securely locate (position in a specific location or orientation) and support the work, ensuring that all parts produced using the fixture will maintain conformity and interchangeability. Using a fixture improves the economy of production by allowing smooth operation and quick transition from part to part, reducing the requirement for skilled labor by simplifying how workpieces are mounted, and increasing conformity across a production run.

The big “don’t”s of TDD

- Test the *behavior*, not the *implementation*
  - This can be difficult to conceptualize because you do have to implement your code. Think: if I solved this problem a different way in my application function, would my test still pass?
- Don’t generalize your code more than what you need to get your test to pass
  - YAGNI - You aren’t going to need it.
  - If you’re tempted to do this, improve your test
- Don’t skip your refactor step
  - Skipping refactoring explodes complexity instead of fighting it
- Don’t skip integration if you are going bottom-up
- Don’t skip unit tests if you are going top-down
- Don’t test on real data sets unless you really need to.