Test Driven Development
Best Practices

- There are a lot of things you can do to help with your practice of implementing unit tests using Test Driven Development. In this lecture I'll go over some best practices that I have found are essential to making TDD productive.

Always Do the Next Simplest Test Case

- Doing the next simplest test case allows you to gradually increase the complexity of your code.
- If you jump into the complex test cases to quickly you will find yourself stuck writing a lot of functionality all it once.
- Beyond just slowing you down, this can also lead to bad design decisions.

- First, You should always do the next simplest test case.
- This allows you to gradually increase the complexity of the code, refactoring as you go. This helps keep your code clean and understandable.
- If you jump to the complex cases to quickly you can find yourself stuck writing a lot of code for one test case which breaks the short feedback cycle we look for with TDD.
- Beyond just slowing you down this can also lead to bad design as you can miss some simple implementations that come from the incremental approach.

Use Descriptive Test Names

- Code is read 1000 times more than it's written. Make it clear and readable!
- Unit tests are the best documentation for how your code works. Make them easy to understand.
- Test suites should name the class or function under test and the test names should describe the functionality being tested.

- Always use descriptive test names.
- The code is read 1000's of times more than it's written as the years go by. Making the code clear and understandable should be the top priority.
- Unit tests are the best documentation for the developers that come after you for how you intended your code to work. If they can't understand what the unit test is testing that documentation value is lost.
- Test suites should name the class or function that is under test and the test name should describe the functionality that is being tested.

Keep Test Fast

- One of the biggest benefits of TDD is the fast feedback on how your changes have affected things.
- This goes away if your unit tests take more than a few seconds to build and run.
- To help your test stay fast try to:
  - Keep console output to a minimum. This slows things down and can clutter up the testing framework output.
  - Mock out any slow collaborators with test doubles that are fast.

- Keep your unit tests building and running fast.
- One of the biggest benefits of TDD is the fast feedback on how your changes have affected things.
- You lose this if the build and/or execution of your unit tests is taking a long time (i.e. more than a few seconds).
- To help your tests stay fast try to:
  - Keep console output to a minimum (or eliminate it all together). This output just slows down the test and clutters up the test results.
  - Mock out any slow collaborators that are being used with test doubles that are fast.

Use Code Coverage Tools

- Once you have all your test cases covered and you think you're done run your unit test through a code coverage tool
- This can help you identify any test cases you may have missed (i.e. negative test cases).
- You should have a goal of 100% code coverage in functions with real logic in them (i.e. not simple getters/setters).
- Pytest-cov is easy to install (pip install pytest-cov) and can generate an easy to use html output.

- Use Code Coverage Analysis Tools
- Once you've implemented all your test cases go back and run your unit tests through a code coverage tool.
- It can be surprising some of the areas of your code you'll miss (especially negative test cases).
- You should have a goal of 100% code coverage on functions with real logic. Don't waste your time on one line getter and setter functions.

Run Your Tests Multiple Time and In Random Order

- Running your tests many times will help ensure that you don’t have any flaky test that fail intermittently.
- Running your tests in random order ensures that your tests don’t have any dependencies between each other.
- Use the pytest-random-order plugin to randomize the order that the tests are executed pytest-repeat plugin to repeat one or more test a specific number of times.

- Make sure you run your unit tests multiple times and in a random order.
- Running your tests many times will help ensure that you don’t have any flaky tests that are failing intermittently.
- Running your tests in random order ensures that your tests don’t have dependencies between each other.
- You can use the pytest-random-order plugin to randomize the execution of the tests and pytest-repeat for repeating all or a sub-set of the unit tests as needed.

Use a Static Code Analysis Tool

- Pylint is an excellent open source static code analysis tool that will find errors in your code that you may have missed in your testing.
- Pylint can verify your python code meets your team's coding standard (or the PEP8 standard by default).
- Pylint can also detects duplicated code and can generate UML diagrams from it's analysis of the code.

- Using a static code analysis tool regularly on your code base is another core requirement for ensuring code quality.
- Pylint is an excellent open source static analysis tool for python that can be used for detecting bugs in your code.
- It can also verify the code is formatted to the team's standards
- And it can even generate UML diagrams based on it's analysis
- In the last lecture I'll review what was gone over in the course and where do go from here.