- 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.

* 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.
* 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 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 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.
* 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.
* 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.