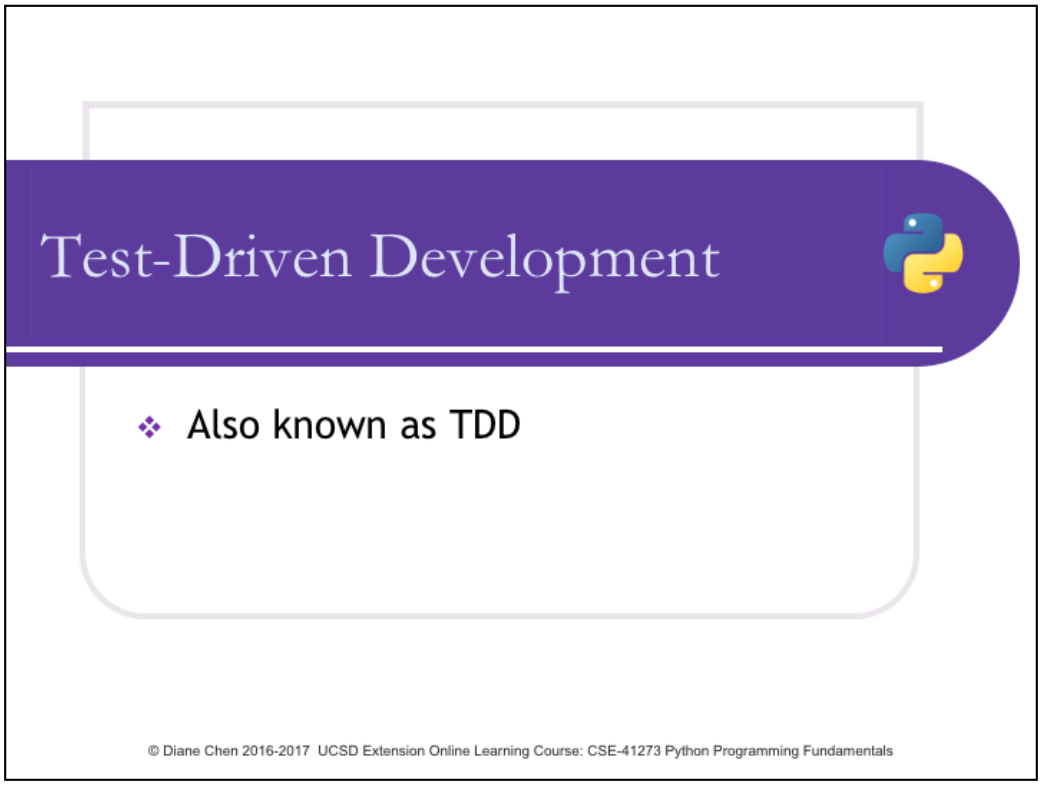
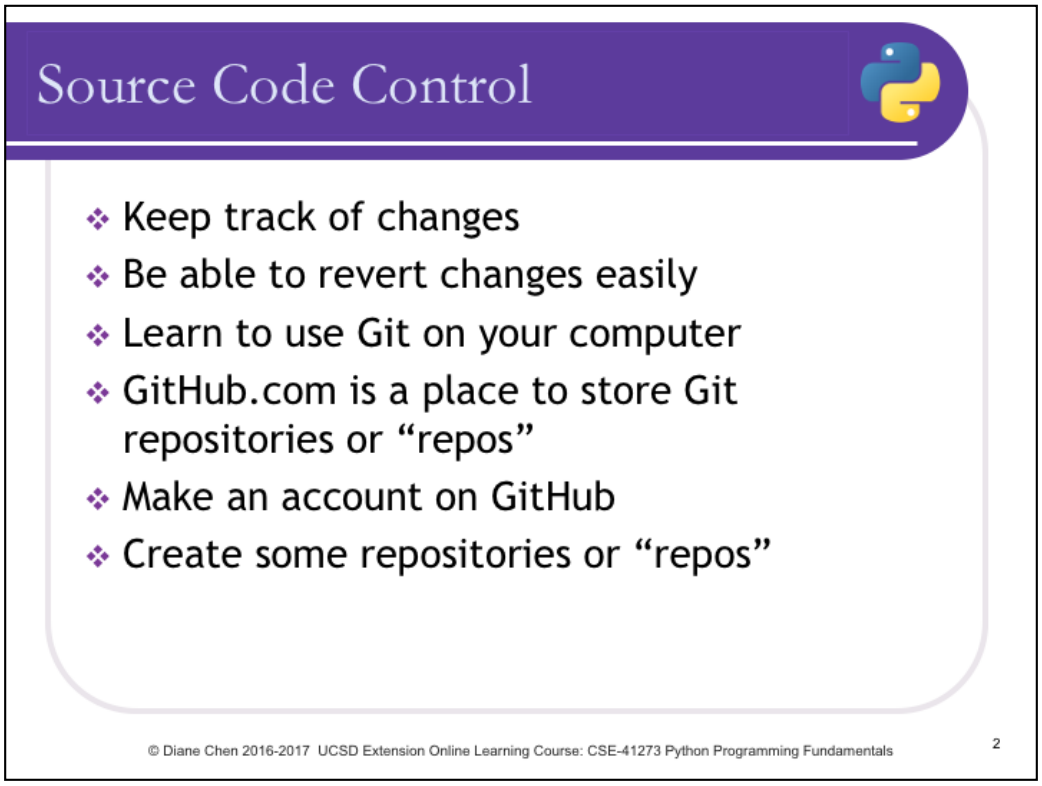
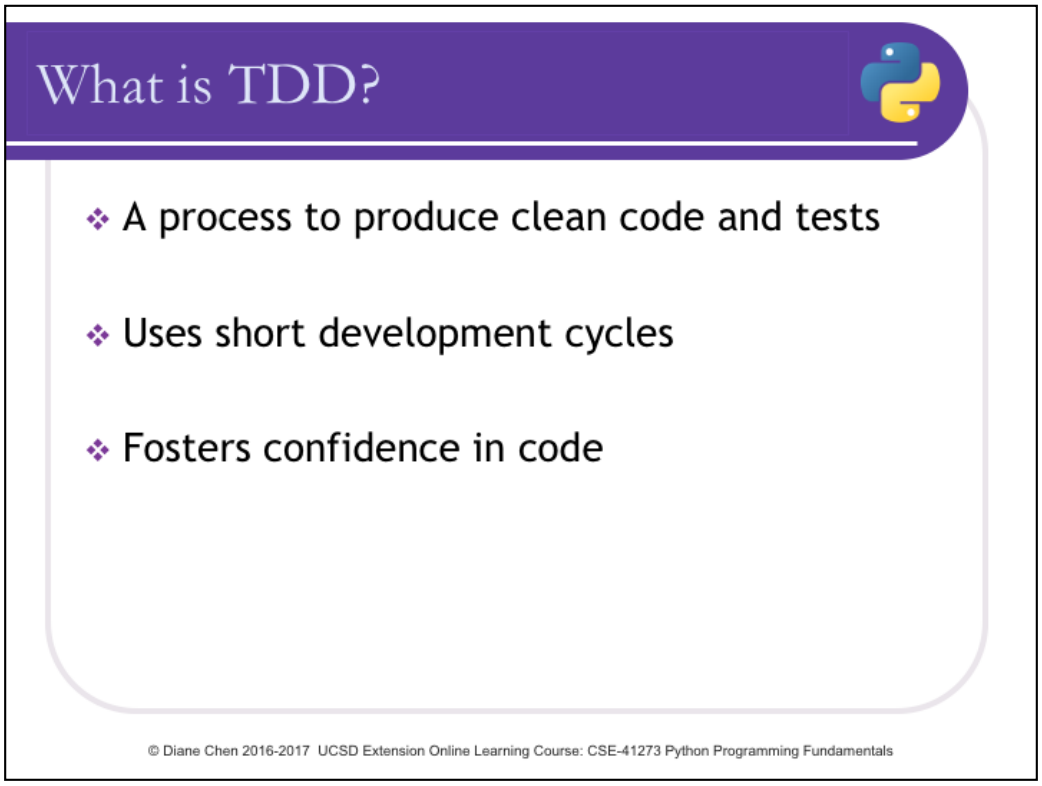


I'm going to talk briefly about the principles of Test-Driven Development, also known as TDD. It's an important concept to understand, as it is used more and more frequently for development.

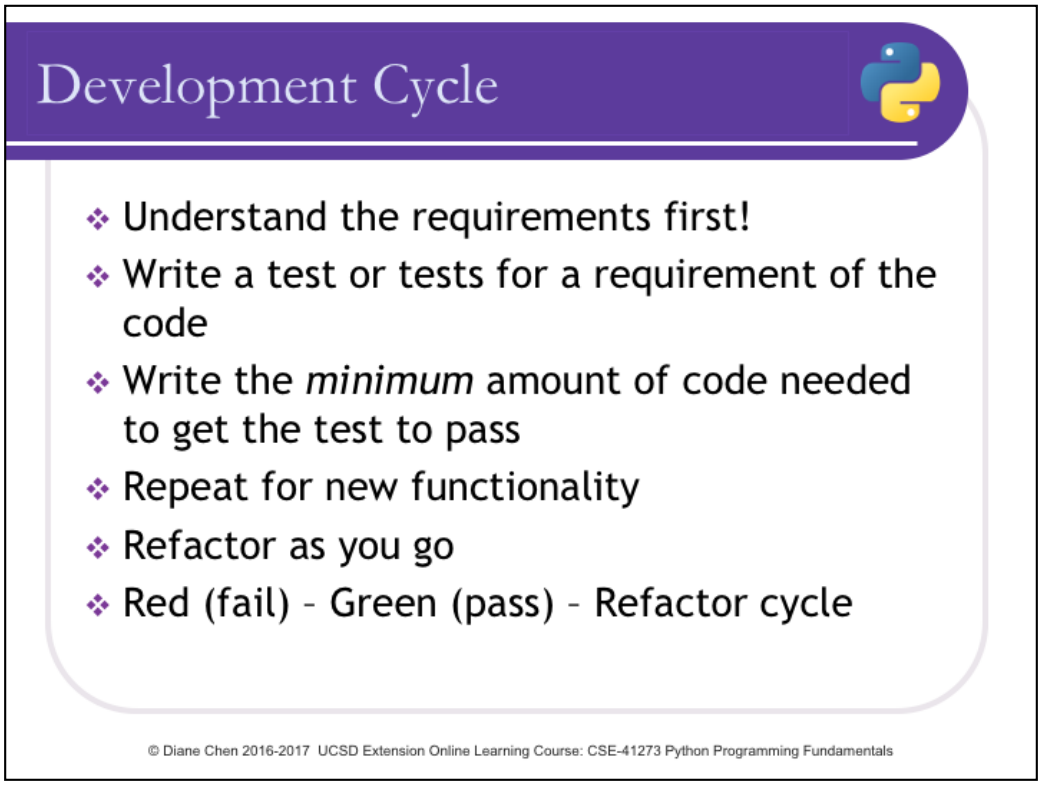


Before we get into TDD, I want to talk about source code control. No discussion of software quality would be complete without talking about Source Code Control. Source code control is a way of keeping track of changes to the software. It allows changes to be tracked and reverted easily if the change causes a problem. The most common source code control mechanism is Git. While we will not be using Git for this class, I recommend you learn to use it on your computer if you don't already know how. Start with the tutorials on the Git homepage. I have included links on this week's resources page. If you search the web for ”git tutorial for beginners” you should find a number of others. I recommend trying out several if you are still confused after using the tutorials from Git. Sometimes it just takes a different explanation for the light to come on! Don't feel bad if you don't understand it - it can be very confusing. Just keep at it and eventually it will be natural.

Github.com is a place in the “cloud” to store git repositories or “repos”. A repository is a place for related files. For example, a repository might contain all the files for a Python package, a software system, a blog, or even recipes and writings – it is not just for code! Github.com has free accounts and tutorials on how to use it. I recommend trying it out, even if you don’t store much of anything there. Knowing how to use Git and Github are important for software developers. Many companies have in-house repos of their software and use Git for source code control. The basic cycle of development using Git is that you have your own development environment and your own copy of the code as a Git repository. You make changes to your copy of the code to fix a bug or implement some new feature, and keep the changes in your own Git repository. When you are ready with the new code, you can ”push” the code to the main repository. Although, in reality, typically what you do is have a copy of the code and go to the main repository and issue a “pull request”. What that means is that you tell the maintainer of the code that you are ready for them to pull your changes into the main repository. They will look at your changes and if they like it, they will pull it in and you are done. Of course, there is a lot more to it then that – this is just a bit of explanation and justification for why you need to learn to use it.



TDD is a process used in development that aids in producing clean code with full test coverage. "Full test coverage" means that all the code in the project is tested by at least one test. The key to TDD is using short development cycles of creating automated tests first, which naturally fail, and then writing the minimal amount of code to pass the tests. When done properly, it produces code for which the developer (or development team) has strong confidence in the code being satisfactory and fulfilling the goals of the software project.

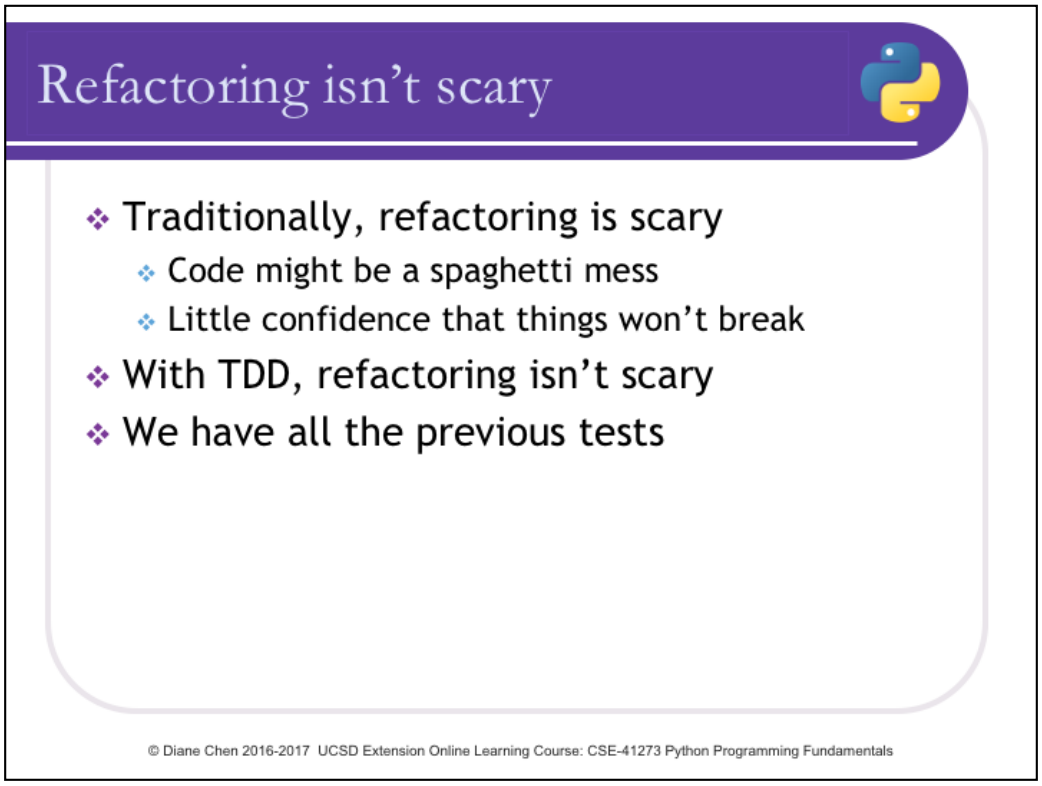


The number one factor to making TDD successful is to understand the requirements of the code needed for the project before starting. This might seem obvious, but so often project requirements start out vague or incomplete at the beginning of a project.

Once the requirements are fully understood, the first thing to do is to write a test or set of tests for one single piece of the requirement. Obviously, since no code has been written yet, the tests will fail.

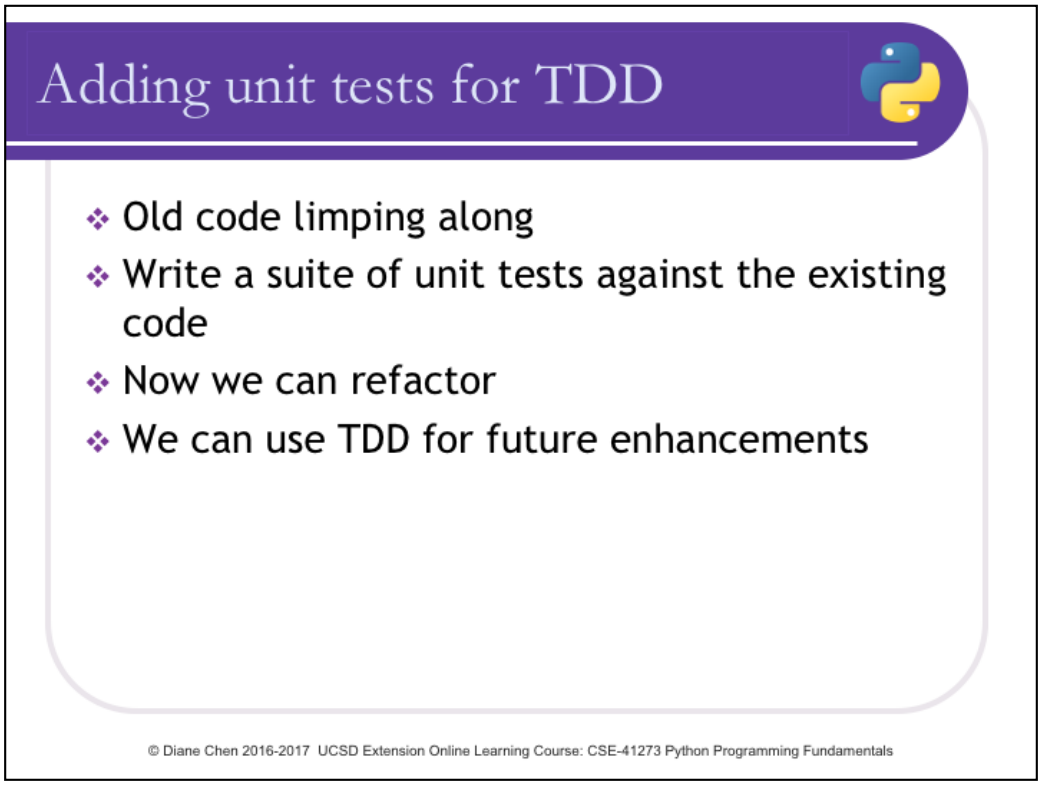
Then the developer writes the minimum amount of code to pass the tests. Then more tests are added for new functionality relating to this requirement or other requirements of the development project. And again, the developer writes the minimum amount of code to implement the requirement and pass the tests. This cycle is repeated continuously until done. Obviously, refactoring is often needed during the process, depending upon the increasing requirements and quantity of code.

This process is often referred to as Red-Green-Refactor cycle. Red is when tests are failing, and green is when the tests all pass. After getting to green, it's always a good idea to look over the code with a critical eye for things that need refactoring: is there some code duplication? Have some functions or classes become bloated and need splitting up into more logical smaller chunks? Does the class hierarchy still make sense? Refactoring consideration needs to take place at every Green step.



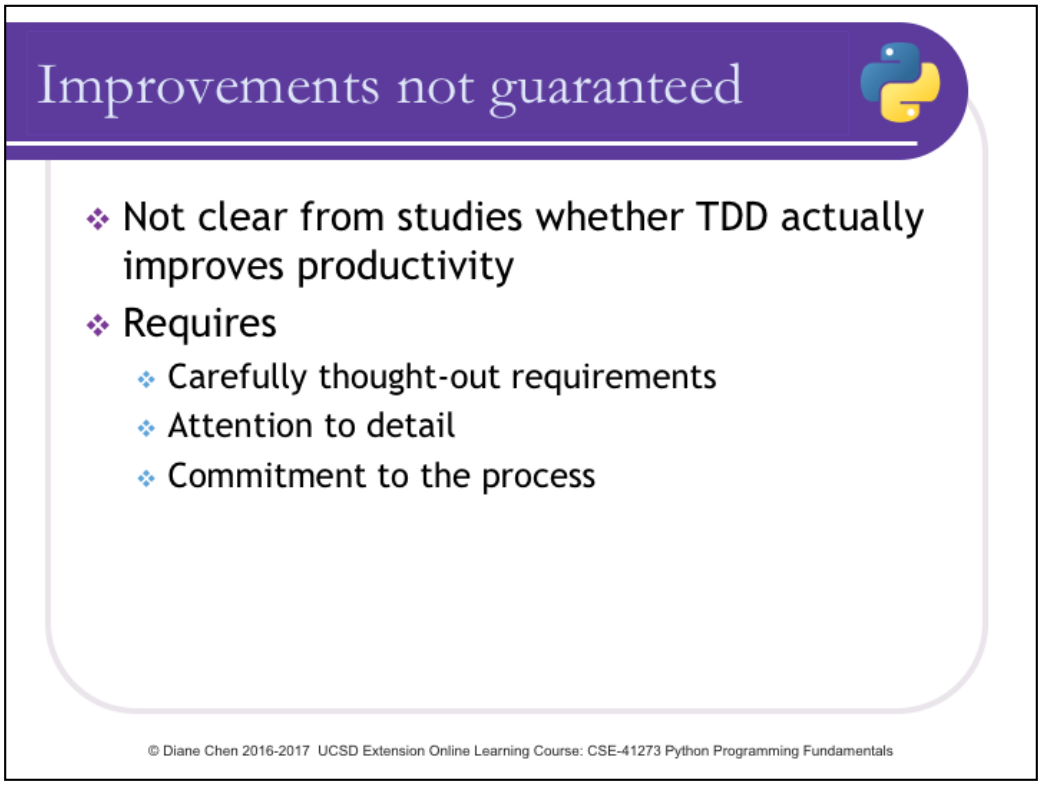
Traditionally, refactoring messy or partially broken code is scary, especially when there isn't any kind of automated testing. The code might be a big spaghetti mess, and we might not have confidence that we won't break things with our refactoring. There might be strange undocumented behavior that we will break if we refactor. Of course, if there is undocumented behavior in a code package, you can be sure that someone is depending on that behavior.

But if TDD has been used correctly, refactoring is not scary at all. Why? Because we have all the previous tests that were written as we developed the code. As we refactor, if we break something, we know about it immediately and can fix it immediately.



One way TDD is also used is for existing software that needs enhancements, performance improvements, etc. Think about the spaghetti code I just mentioned. What if we really needed to fix some bugs or add enhancements to it?

A full suite of automated tests can be written against the software and its requirements. Sometimes by doing this, we might find other bugs we didn't know about before. Once these tests are completed, the code can be refactored and repaired, and from then on, TDD is used for fixing bugs, adding enhancements, etc.



There have been some studies trying to prove or disprove whether TDD actually improves productivity. It is something that is extremely difficult to test, and so far it is not clear, at least from an academic perspective, whether there are superior advantages to TDD. Studies (not just for TDD) have shown that there is a definite correlation between the number of tests and fewer code defects, so the fact that TDD results in lots of tests is definitely in its favor.

I believe that TDD, if used properly, can enhance the quality of the code tremendously, and result in code that is more maintainable. However, it is not just a technical issue, but also one of management commitment and developer buy-in. Some developers just want to write gobs of code and don't want to have to worry about automated testing or short development cycles. This is more fun for the developer, but does not get the job done for the whole project.