Roadmap For Mastering Test-Driven Development

Different Assertion Types

You already learned about `assertEqual()`.

There are many other distinctions you can make:

- Assert two quantities are NOT equal
- Assert an expression is true or false
- Assert an element is in a collection or sequence
- Assert relations (x is greater than y, etc.)
- Assert subclass and instance relationships
- Assert matching regular expressions

And quite a few more.

Asserting Errors

Often, correctly working code will trigger an ERROR under certain conditions...

And if it does NOT trigger that error, it's a bug.

It is critical you learn how to write tests that model these error conditions, and verify your code raises the correct exception.

Test Fixtures

Complex code needs complex data.

Your test will sometimes need to set up data structures, even external resources, in a known starting state. Doing it again for the next test run, and cleaning up after itself every time.

These artifacts are called *test fixtures*. And they are frequently needed for anything beyond simple test cases.

Parameterized Tests

Functions like `split_amount()` require many combinations of input values to thoroughly test their behavior.

For more complex functions, the sheer number of combinations requires a better approach than endless calls to `assertEqual()`.

Using *parameterized tests*, also called subtests, allows you to compactly express thorough combinations of test input ranges, with more pinpointed reporting and systematic, exhaustive test validation.

Unit vs Integration Tests

There are several different kinds of automated test.

The most important distinction to make:

Unit tests versus Integration tests.

Understand the key differences between these two... where each is useful... and where you need a different approach.

Test-First vs Test-Last

Do you write tests BEFORE you write the code being tested? Or after?

The first is called "Test-Driven Development". And it has pros and cons.

Instead of always doing one or the other, grok the rich trade-offs of each approach. So you know when (and how) to do both.

Mocks

Once you are writing longer programs, you will be writing components that depend on each other. And this inter-twining can make writing good tests extremely difficult.

A **mock** is a tool that lets you isolate interlocked components, so that you can write solid, self-contained unit tests.

Mocks are immensely valuable. But they are extremely complex tools. And tragically, they are almost never taught correctly.

Alternate Testing Frameworks

The built-in `unittest` module is only one option.

There are several other libraries and frameworks that are widely used and important to master as well.

There are also some that claim to be important... but are frankly not.