# Class 5: Code Testing A programmatic way to ensure your code does what you think

*Testing provides a way for us to programmatically ensure that our code does what we think it does, doesn’t have side effects, and fails in a way we expect, and doesn’t fail when we don’t expect it. Additionally, good tests provide assurance that code does not ‘regress’ when we make changes to it after initial implementation.*

## Testing in Python

### Lecture

**Unit Testing:**

A more modular approach to dynamic testing than end-to-end testing. Think of it as E2E testing, but at the function level: For a class, test the creation of the class and setting of initial variables. Test the public and private functions separately, etc.

**Pro’s:**

* Much easier to identify *where* a regression has occurred, since the specific test covering that will fail
* Tests scale linearly with the amount of code you write (since unit tests generally map to a small number linear function of the number of functions that you write
* Allows you to ‘mock’ other functions output, so that you don’t have to worry about writing something that will specifically produce a desired interaction across function boundaries

**Con’s:**

* Only as useful as your ability to unit test the large majority of your code, which means your test code base will likely be larger than your code base.
* Can be an enormous pain in the ass to mock out external dependencies, depending on how tightly coupled your code is.
* Mocking out external dependencies means making an assumption on the correct shape of their output-- if this is wrong, you will run into problems (potentially) at run-time that were never found in testing

**Mocking**:

creating a dummy version of a class, that provides rote outputs for declared inputs. This is useful for making tightly encapsulated tests, that do not end up having to test everything in order to test anything

The *Unittest* Library: a reflection-based library that has robust support for mocking external dependencies

The ‘@patch’ function decorator for use in the *unittest* paradigm

A lighter weight alternative: ***Doctests***

[*https://docs.python.org/3.5/library/doctest.html*](https://docs.python.org/3.5/library/doctest.html)

*Set Up:*

if \_\_name\_\_ == "\_\_main\_\_":

import doctest

doctest.testmod()

>: python your\_file.py

*Pros:*

Simple, low set-up, Intuitive, part of the docstring for a function (which is a good habit to include)

*Cons:*

Difficult to test certain things that are made easier via *reflection.* This includes File I/O, any other time you may call an outside program as part of a function, or when you are testing failure states that are difficult to manually cause to happen.

Doctest does not cover “reflection” or “mocking”