# Getting Testy: The Basics of Software Testing


| | |
|---|---|
| **Author** | josh |
| **Date** | 2022-11-13 |


# Overview

All of the software we use day-to-day can have flaws. We software engineers do our best, but are known for making lots of mistakes. However, we want the most *stable* and bug free software possible. The best way to improve the quality of software is through *testing*. Every programmer should learn to test their code, find bugs, and fix them. Let's learn about the basic types of software testing and how to use them.

# Testing Types

## Manual Testing

The first, perhaps simplest and most intuitive form of software testing is *manual testing*. In other words, the practice of a human testing code by hand. This is the way many programmers test their small side projects without having to write additional automated test code. Whatever the type of project, manual testing involves direct interaction with the code and systems. For example, say you write a project that swaps out a Bitcoin address in the clipboard with an attacker address. The simplest way to test this code is to run the program, copy a Bitcoin address, then paste the clipboard into a URL bar or text file. You can verify with your own eyes that the address was changed as intended.

The downside to this type of testing is it is more time consuming, and easier to miss edge cases. For more complex programs, a human may not easily cover all possible inputs or interactions with the program and may miss bugs. Manual testing is simple, but requires more time and effort the more complex a project becomes.

## Unit Testing

*Unit testing* is a form of automated testing that looks at individual functions, modules, or "units" of a program, and tests them using repeatable scripts called *unit tests.* For a simple example, let's think about a library that takes a list or array of numerical data and computes statistics such as sum, average, min, max. Each individual function such as sum would be tested as a unit. The unit test would provide the list, compute a sum, and check that the sum is the intended answer. We can be sure that the sum of [ 1, 2, 3 ] is 6, so writing a unit test for that case is a simple sanity check that our sum function is working correctly.

Unit tests have the advantage of automation – so it's easy to write a large suite of tests that check many independent functions in a library or program. Again, in our example, we can easily check the results of sum, average, min, max functions without having to manually run each function. If we make a change that makes our max function more efficient, we can easily check for a "regression" – that is, a change that breaks our functionality. If the max of [ 1, 2, 3 ] is erroneously reported as 2, we know we have introduced a bug in the function.

Unit testing has some disadvantages, however. Unit testing, by definition, only tests individual functions or modules independently. So unit tests don't do a great job of testing how varying units interact. For example, unit tests might cover sum, average, min, and max functions but might not cover how the whole program interacts from end-to-end – reading from a file, computing statistics, and outputting the results in a correct format. That is where our next test type comes in.

### Integration Testing

The final type of test we will discuss is called *integration testing*. This type of testing tests "systems", rather than just individual units. In our unit tests, we focus on individual functions. In integration tests, we focus on how those units interact together to form our entire program. Let's say for example we have a program with a database full of fitness information, an API for searching that data, and a frontend webpage that displays the data in a nice table. An automated integration test might use a web driver to type a query into the web search box, click the button, and check that a nicely formed table is displayed on the page. It might test only two parts of the program, such as making REST calls to the API and sanity checking the data.

Integration tests have the advantage of testing *systems* through automation, rather than just individual functions. An integration test may ensure that a change to the database functionality doesn't bubble up and cause problems with the web user interface. However, integration tests may not catch obvious problems with individual functionality in the way unit tests would. If you're only checking for a well formed HTML table in your integration test, you may not realize that your query returns the wrong data.

## Test Like Aperture Science – Perpetually!

Testing software is important, so software developers should do so often! There's no right or wrong answer to how a software program should be tested, necessarily. Testing should fit the program at hand and can combine multiple forms. Large software projects, for example, may benefit from both unit and integration testing. Simple utilities may only need unit tests or manual tests. The important thing is that your software is tested – don't just write and run. The less bugs our software has, the more the users of our software can rely on it.