# Chapter 14
# Introduction to Testing


## 14.1 Introduction

This chapter considers the different types of tests that you might want to perform with the systems you develop in Python. It also introduces Test Driven Development.

## 14.2 Types of Testing

There are at least two ways of thinking about testing:

1. It is the process of executing a program with the intent of finding errors/bugs (see Glenford Myers, The Art of Software Testing).
2. It is a process used to establish that software components fulfil the requirements identified for them, that is that they do what they are supposed to do.

These two aspects of testing tend to have been emphasised at different points in the software lifecycle. Error Testing is an intrinsic part of the development process, and an increasing emphasis is being placed on making testing a central part of software development (see Test Driven Development).

It should be noted that it is extremely difficult—and in many cases impossible—to prove that software *works* and is completely *error free*. The fact that a set of tests finds no defects does not prove that the software is error-free. 'Absence of evidence is not evidence of absence!'. This was discussed in the late 1960s and early 1970s by Dijkstra and can be summarised as:

> Testing shows the presence, not the absence of bugs

Testing to establish that software components fulfil their contract involves checking operations against their requirements. Although this does happen at



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_14

development time, it forms a major part of Quality Assurance (QA) and User Acceptance testing. It should be noted that with the advent of Test-Driven Development, the emphasis on testing against requirements during development has become significantly higher.

There are of course many other aspects to testing, for example, Performance Testing which identifies how a system will perform as various factors that affect that system change. For example, as the number of concurrent requests increase, as the number of processors used by the underlying hardware changes, as the size of the database grows etc.

However you view testing, the more testing applied to a system the higher the level of confidence that the system will work as required.

## 14.3 What Should Be Tested?

An interesting question is 'What aspects of your software system should be subject to testing?'.

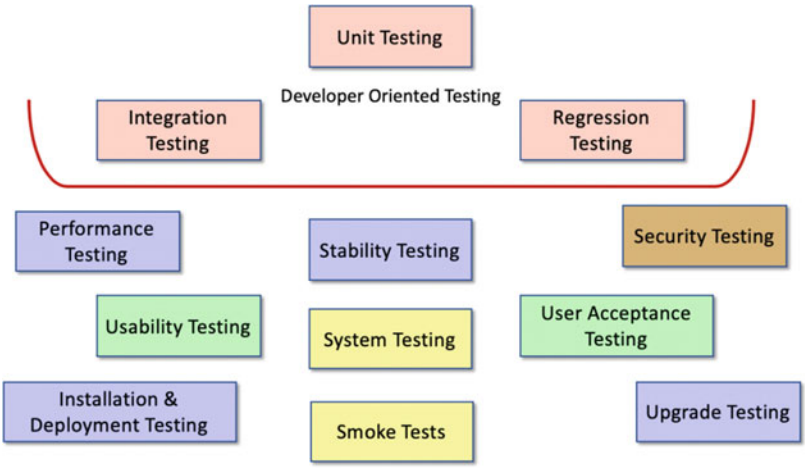
In general, anything that is repeatable should be subject to formal (and ideally automated) testing. This includes (but is not limited to):

- The build process for all technologies involved.
- The deployment process to all platforms under consideration.
- The installation process for all runtime environments.
- The upgrade process for all supported versions (if appropriate).
- The performance of the system/servers as loads increase.
- The stability for systems that must run for any period of time (e.g. $24 \times 7$ systems).
- The backup processes.
- The security of the system.
- The recovery ability of the system on failure.
- The functionality of the system.
- The integrity of the system.

Notice that only the last two of the above list might be what is commonly considered areas that would be subject to testing. However, to ensure the quality of the system under consideration, all of the above are relevant. In fact, testing should cover all aspects of the software development lifecycle and not just the QA phase. During requirements gathering testing is the process of looking for missing or ambiguous requirements. During this phase consideration should also be made with regard to how the overall requirements will be tested, in the final software system.

Test planning should also look at all aspects of the software under test for functionality, usability, legal compliance, conformance to regulatory constraints, security, performance, availability, resilience, etc. Testing should be driven by the need to identify and reduce risk.

## 14.4 Testing Software Systems




As indicated above there are a number of different types of testing that are commonly used within industry. These types are:

- **Unit Testing**, which is used to verify the behaviour of individual components.
- **Integration Testing** that tests that when individual components are combined together to provide higher-level functional units, that the combination of the units operates appropriately.
- **Regression Testing**. When new components are added to a system, or existing components are changed, it is necessary to verify that the new functionality does not break any existing functionality. Such testing is known as Regression Testing.
- **Performance Testing** is used to ensure that the systems' performance is as required and, within the design parameters, and is able to scale as utilisation increases.
- **Stability Testing** represents a style of testing which attempts to simulate system operation over an extended period of time. For example, for a online shopping application that is expected to be up and running $24 \times 7$ a stability test might ensure that with an average load that the system can indeed run 24 hours a day for 7 days a week.

- **Security Testing** ensures that access to the system is controlled appropriately given the requirements. For example, for an online shopping system there may be different security requirements depending upon whether you are browsing the store, purchasing some products or maintaining the product catalogue.
- **Usability Testing** which may be performed by a specialist usability group and may involved filming users while they use the system.
- **System Testing** validates that the system as a whole actually meets the user requirements and conforms to required application integrity.
- **User Acceptance Testing** is a form of user oriented testing where users confirm that the system does and behaves in the way they expect.
- **Installation, Deployment and Upgrade Testing**. These three types of testing validate that a system can be installed and deployed appropriate including any upgrade processes that may be required.
- **Smoke Tests** used to check that the core elements of a large system operate correctly. They can typically be run quickly and in a faction of the time taken to run the full system tests.

Key testing approaches are discussed in the remainder of this section.

### *14.4.1 Unit Testing*

A unit can be as small as a single function or as large as a subsystem but typically is a class, object, self-contained library (API) or web page.

By looking at a small self-contained component an extensive set of tests can be developed to exercise the defined requirements and functionality of the unit.

Unit testing typically follows a *white box* approach, (also called *Glass Box* or *Structural* testing), where the testing utilizes knowledge and understanding of the code and its structure, rather than just its interface (which is known as the *black box* approach).

In *white box* testing, test coverage is measured by the number of code paths that have been tested. The goal in unit testing is to provide 100% coverage: to exercise every instruction, all sides of each logical branch, all called objects, handling of all data structures, normal and abnormal termination of all loops etc. Of course this may not always be possible but it is a goal that should be aimed for. Many automated test tools will include a code coverage measure so that you are aware of how much of your code has been exercised by any given set of tests.

Unit Testing is almost always automated—there are many tools to help with this, perhaps the best-known being the xUnit family of test frameworks such as JUnit for Java and PyUnit for Python. The framework allows developers to:

- focus on testing the unit,
- simulate data or results from calling another unit (representative good and bad results),

- create data driven tests for maximum flexibility and repeatability,
- rely on *mock* objects that represent elements outside the unit that it must interact with.

Having the tests automated means that they can be run frequently, at the very least after initial development and after each change that affects the unit.

Once confidence is established in the correct functioning of one unit, developers can then use it to help test other units with which it interfaces, forming larger units that can also be unit tested or, as the scale gets larger, put through Integration Testing.

### *14.4.2 Integration Testing*

Integration testing is where several units (or modules) are brought together to be tested as an entity in their own right. Typically, integration testing aims to ensure that modules interact correctly and the individual unit developers have interpreted the requirements in a consistent manner.

An integrated set of modules can be treated as a unit and unit tested in much the same way as the constituent modules, but usually working at a "higher" level of functionality. Integration testing is the intermediate stage between unit testing and full system testing.

Therefore, integration testing focuses on the interaction between two or more units to make sure that those units work together successfully and appropriately. Such testing is typically conducted from the bottom up but may also be conducted top down using mocks or stubs to represented called or calling functions. An important point to note is that you should not aim to test everything together at once (so called *Big Bang* testing) as it is more difficult to isolate bugs in order that they can be rectified. This is why it is more common to find that integration testing has been performed in a bottom up style.

### *14.4.3 System Testing*

System Testing aims to validate that the combination of all the modules, units, data, installation, configuration etc. operates appropriately and meets the requirements specified for the whole system. Testing the system has a whole typically involves testing the top most functionality or behaviours of the system. Such Behaviour Based testing often involves end users and other stake holders who are less technical. To support such tests a range of technologies have evolved that allow a more *English* style for test descriptions. This style of testing can be used as part of the requirements gathering process and can lead to a Behaviour Driven Development (BDD) process. The Python module `pytest-bdd` provides a BDD style extension to the core pytest framework.

### 14.4.4 Installation/Upgrade Testing

Installation testing is the testing of full, partial or upgrade install processes. It also validates that the installation and transition software needed to move to the new release for the product is functioning properly. Typically, it

- verifies that the software may be completely uninstalled through its back-out process.
- determines what files are added, changed or deleted on the hardware on which the program was installed.
- determines whether any other programs on the hardware are affected by the new software that has been installed.
- determines whether the software installs and operates properly on all hardware platforms and operating systems that it is supposed to work on.

### 14.4.5 Smoke Tests

A smoke test is a test or suite of tests designed to verify that the fundamentals of the system work. Smoke tests may be run against a new deployment or a patched deployment in order to verify that the installation performs well enough to justify further testing. Failure to pass a smoke test would halt any further testing until the smoke tests pass. The name derives from the early days of electronics: If a device began to smoke after it was powered on, testers knew that there was no point in testing it further. For software technologies, the advantages of performing smoke tests include:

- Smoke tests are often automated and standardised from one build to another.
- Because smoke tests validate things that are expected to work, when they fail it is usually an indication that something fundamental has gone wrong (the wrong version of a library has been used) or that a new build has introduced a bug into core aspects of the system.
- If a system is built daily, it should be smoke tested daily.
- It will be necessary to periodically add to the smoke tests as new functionality is added to the system.

## 14.5 Automating Testing

The actual way in which tests are written and executed needs careful consideration. In general, we wish to automate as much of the testing process as is possible as this makes it easy to run the tests and also ensures not only that all tests are run but that

they are run in the same way each time. In addition, once an automated test is set up it will typically be quicker to re-run that automated test than to manually repeat a series of tests. However, not all of the features of a system can be easily tested via an automated test tool and in some cases the physical environment may make it hard to automate tests.

Typically, most unit testing is automated and most acceptance testing is manual. You will also need to decide which forms of testing must take place. Most software projects should have unit testing, integration testing, system testing and acceptance testing as a necessary requirement. Not all projects will implement performance or stability testing, but you should be careful about omitting any stage of testing and be sure it is not applicable.

## 14.6 Test Driven Development

Test Driven Development (or TDD) is a development technique whereby developers write test cases *before* they write any implementation code. The tests thus drive or dictate the code that is developed. The implementation only provides as much functionality as is required to pass the test and thus the tests act as a specification of *what* the code does (and some argue that the tests are thus part of that specification and provide documentation of what the system is capable of).

TDD has the benefit that as tests must be written first, there are always a set of tests available to perform unit, integration, regression testing etc. This is good as developers can find that writing tests and maintaining tests is boring and of less interest than the actual code itself and thus put less emphasis into the testing regime than might be desirable. TDD encourages, and indeed requires, that developers maintain an exhaustive set of repeatable tests and that those tests are developed to the same quality and standards as the main body of code.

There are three rules of TDD as defined by Robert Martin, these are:

1. You are not allowed to write any production code unless it is to make a failing unit test pass
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

This leads to the TDD cycle described in the next section.

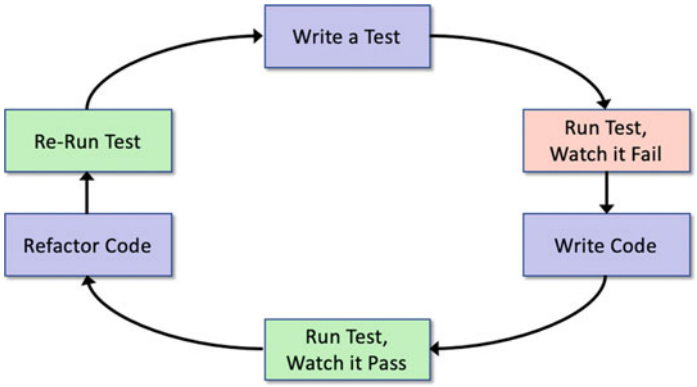
### 14.6.1 The TDD Cycle

There is a cycle to development when working in a TDD manner. The shortest form of this cycle is the TDD mantra:

Red / Green / Refactor

Which relates to the unit testing suite of tools where it is possible to write a unit test. Within tools such as PyCharm, when you run a pyunit or pytest test a Test View is shown with Red indicating that a test failed or Green indicating that the test passed. Hence Red/Green, in other words write the test and let it fail, then implement the code to ensure it passes. The last part of this mantra is *Refactor* which indicates once you have it working make the code cleaner, better, fitter by Refactoring it. Refactoring is the process by which the behaviour of the system is not changed but the implementation is altered to improve it.

The full TDD cycle is shown by the following diagram which highlights the test first approach of TDD:




The TDD mantra can be seen in the TDD cycle that is shown above and described in more detail below:

1. Write a single test.
2. Run the test and see it **fail**.
3. Implement *just enough* code to get the test to pass.
4. Run the test and see it **pass**.
5. **Refactor** for clarity and deal with any issue of reuse etc.
6. Repeat for next test.

### 14.6.2 Test Complexity

The aim is to strive for simplicity in all that you do within TDD. Thus, you write a test that fails, then do just enough to make that test pass (but no more). Then you refactor the implementation code (that is change the internals of the unit under test) to improve the code base. You continue to do this until all the functionality for a unit has been completed. In terms of each test, you should again strive for simplicity with each test only testing one thing with only a single assertion per test (although this is the subject of a lot of debate within the TDD world).

### 14.6.3 Refactoring

The emphasis on refactoring within TDD makes it more than just testing or Test First Development. This focus on refactoring is really a focus on (re)design and incremental improvement. The tests provide the specification of what is needed as well as the verification that existing behaviour is maintained, but refactoring leads to better design software. Thus, without refactoring TDD is not TDD!

## 14.7 Design for Testability

Testability has a number of facets

- Configurability. Set up the object under test to an appropriate configuration for the test
- Controllability. Control the input (and internal state)
- Observability. Observe its output
- Verifiability. That we can verify that output in an appropriate manner.

### 14.7.1 Testability Rules of Thumb

If you cannot test code then change it so that you can!
If your code is difficult to validate then change it so that it isn't!
Only one concrete class should be tested per Unit test and then *Mock the Rest*!
If you code is hard to reconfigure to work with Mocks then make it so that you code can use Mocks!
Design your code for testability!

## 14.8 Online Resources

See the following online resources for more information on testing and Test Driven Development (TDD).

- https://www.test-institute.org/Introduction_To_Software_Testing.php Introduction to Software Testing.
- https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing Introduction to software Testing wiki book.
- https://en.wikipedia.org/wiki/Test-driven_development Test Driven Development wikipedia page.
- http://agiledata.org/essays/tdd.html an introduction to Test Driven Development.
- https://medium.freecodecamp.org/learning-to-test-with-python-997ace2d8abe a simple introduction to TDD with Python.
- http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd Robert Martins three rules for TDD.
- http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata The Bowling Game Kata which presents a worked example of how TDD can be used to create a Ten Pin Bowls scoring keeping application.

## 14.9 Book Resources

- *The Art of Software Testing*, G. J. Myers, C. Sandler and T. Badgett, John Wiley & Sons, 3rd Edition (Dec 2011), 1118031962.