Chapter 1
A History of Testing

THE IDEA OF testing is one that has evolved over many years in the development community. Developers used to have much less focus on testing up front and just wrote code and dealt with any problems that arose by quickly writing fixes after a testing period at the end of a project. That isn't to say that there weren't developers out there who were writing code that was trouble free when out in the wild, but on the whole, writing code without tests in general is going to lead to problems down the line. There were also cases where testing was a priority, such as code that could cause destruction or the possibility of a person dying. In such circumstances there would be rigorous testing, but this was very much the exception rather than the rule.

The first real change in ideology came with improvements in technology and the resulting development pressures that came with it. When computers were slower, code modification cycles took much longer. Even a simple program could take tens of minutes to build, and large projects could take hours. This resulted in a batch development process where people spent a great deal of time pouring over code, figuring out issues, and then making sets of changes. The amount of time spent verifying changes to the code was comparatively small compared with the cycle time.

As computers became faster, compilation times shrank and development cycle times correspondingly shrank. It became feasible to make small changes to code, quickly build the product, and then verify the results of those few changes. This meant that code was written and tests covered that code to ensure it behaved as expected. Also, as computer systems became more powerful, the complexity of software increased. Even a simple program these days often has both a client and server component running on different systems (such as a browser and web server). Operating systems offer a bewildering variety of services to a program. Choreographing these interactions requires managing complexity in a systematic way. Features of Python such as loose typing impose additional verification demands on developers, as errors in coding cannot be caught at a compilation stage. Similarly, because Python has no demands on the type of objects it is manipulating, you can end up with strange behavior if you have not handled all cases correctly.

Testing forces developers to think about the code that they are writing and consider all sorts of different scenarios and the outcomes rather than focusing on the happy path scenario that takes into account only how the code should be used. When combined with a test driven development approach (TDD; see Chapter 5), this ideology ensured that testing was baked in to the development process and not a tedious afterthought. One of the worst traps a developer can fall into is writing a bunch of code and then going back and testing it all at the end. Not only is this approach more time consuming and often rushed, but it also means revisiting code that isn't fresh in the mind like it was at the time of writing. When you revisit the method to write a test, the context and thought process at the time of writing is often lost to you.

Similarly, the change from the waterfall development processes to agile has brought a huge focus on testing while developing rather than treating testing as an afterthought, as I describe previously. Agile development advocates that teams include dedicated quality assurance (QA) personnel, whose sole focus is to write tests and maintain a solid test suite around the application. This allows someone who hasn't written the code to look at it from a fresh angle and perhaps spot weaknesses or bugs in the code before those glitches reach the customer.

Following on from TDD, agile development also spawned the concept of behavior driven development (BDD; see Chapter 6). This method takes unit testing one step further and looks to test the application's behavior in terms of functionality being delivered. BDD is also known as an acceptance test and generally comes in the form of a human readable feature file, which describes the functionality and then maps to step files, which execute the test code underneath. The huge benefit of this approach is that non-technical team members, such as a scrum master (person responsible for removing impediments that arise in a team and assists in organizational matters) or product owner (person wanting the deliverable and setting the requirements for the project), can write feature files, and then the developer or QA can implement the code underneath. With this setup in place, you basically have testable documentation for your system that anyone on the team can understand. This approach also allows you to create a failing acceptance test that you develop your code to pass, ensuring that you deliver the feature you have set out to create. Unit testing alone does not produce such reliable results. It is the combination of the two testing practices that ensures you can deliver quality software and be confident when it goes live.

Clearly, the mindset of developers has changed over the years from not just writing code but to ensuring that their code is tested from all angles. From unit testing to acceptance testing, Python developers have implemented libraries and tools to help Python developers follow these changes to the development process. This book covers their implementation and usage so that you too can get up to speed on the latest testing tools and techniques to ensure you are not left stuck in the past of testing history.

You Do Test, Don't You?

A huge shift has occurred in recent years of software development toward testing and ensuring that your application delivers absolute quality. With the advent of social networks and the ever-increasing pressure of media attention, defects in your code could be costly to both you and your reputation or that of any company you may represent. Whether it be security flaws exposing sensitive customer data, defects that allow hackers access to deface your website, or simply a payments page failing to execute orders, errors can cost your business huge sums of money.

Don't think of problems on only the large-scale, either. Without a proper testing suite in place, how do you know you have delivered the functionality you set out to deliver at the beginning of writing code? Take a simple data submission form. You have coded the fields to accept a name, address, and e-mail, without any testing. You quickly enter the data as expected and your submit works fine. But what if your customers enter something you didn't expect in the fields—for instance, a number in the name field? Does your code handle this? What if you make changes to the code? Are you sure that the program still functions as it should?

You can see some of the problems developers face when writing code of this nature and how testing can give you a repeatable process that ensures you are delivering working software every time. Luckily for you, this shift in mindset to place such importance in testing has spawned numerous, quality testing tools and frameworks to make the process as simple as possible.

You can certainly make great code without tests. In fact, it is highly likely that many software houses put out software without rigorous testing. The key advantage of writing tests, especially as part of the development process, is that testing gives you confidence in your code before it goes live. As a developer, you are often on call to support your applications in the middle of the night. Do you really want that phone call at 3 a.m. because you didn't write tests to cover that edge case? Testing won't stop this from ever happening again, but it will make it a very rare occurrence. You will have good knowledge of the different routes through your system, making it easier to debug the situations where the worst may happen.

Fundamentals and Best Practices

Before getting stuck in the process of writing tests, it is a good idea to take some time to get your machine in order and up to date with the tools you will need to proceed. First, ensure you have the correct version of Python installed. Then, getting set up with some of the basic tools Python developers use on a daily basis will mean you can easily follow the rest of this book and install libraries of code and keep your changes in check using source control. This section is essentially a prerequisite to the rest of the book, and it is recommended you follow the instructions carefully to get your machine in shape for the examples that will follow later.

Python Installation

Of course, this book assumes that you already have some background in Python programming, even at the most basic level. That said, for completeness it is worth mentioning how to get Python on your system and what version this book uses.

The book focuses on the Python 2.7 release, which is used quite widely in the Python community. It is the last version that was released prior to the backward-incompatible release of 3.0 and beyond. The vast majority of code will likely work with Python 3.0 and the official documentation will help with any problems that may arise.

Linux

Most Linux distributions come with some version of Python installed. Most notably, recent Ubuntu releases generally come with version 2.7.*x* preinstalled. If for some reason you find you don't have Python, or perhaps you have an older version and want to upgrade, you usually install using your distributions package manager. Should Python not be available in this form, then you can visit `www.python.org` to download the source and compile it yourself. Instructions should be included on the website.

Mac

Like Linux, Apple chose to ship a version of Python with every version of OS X. At the time of writing, Mavericks had just been released in October 2013; this version included Python 2.7.5 by default. Therefore, if you are following this book and working on a Mac, then you should be all set. If you find you need to get Python on your machine for some reason, then you could install a package manager for Mac. This not only will help with the install of Python itself, but will also come in handy for any other dependencies your system may need. Two popular package managers for Mac are available: MacPorts and Homebrew. I prefer the latter because its packages seem to be better maintained and more up to date than those for MacPorts. Homebrew is also a more lightweight installation, and the install scripts are written in Ruby, which means it's easy to write some brews yourself. You can find information on the two package managers here:

- MacPorts at `http://macports.com`
- Homebrew at `http://brew.sh`

Windows

Windows is considered out of scope in this book. Having had little to no experience working with Python on a Windows machine, I am not in the best place to offer advice. However, that does not mean the code and advice in this book are not of use to a Windows user.

Plenty of guides on the web can help a Windows user get set up with Python, at which point you can easily run the tests and code that this book offers. Some good Python Windows resources are

- Official Python website: `http://www.python.org/downloads/windows/`
- Python documentation: `http://docs.python.org/2/using/windows.html`

Pip

The new standard package manager for Python, Pip allows you to install any of numerous Python packages from the PyPi repository. For example, you may want to write a web application in which case a popular web framework such as Django or Flask could be installed. First find out if you have Python. If so, you also should have Pip. If you don't, you should at least have `easy_install`, the package manager that Pip has superseded. To get Pip in this scenario, simply try:

```
$ easy_install pip
```

You should then have Pip and be able to install packages, like so:

```
$ pip install flask
```

More on `easy_install` and Pip can be found at `http://www.pip-installer.org`.

Virtualenv

If you have been working on any Python projects without using Virtualenv, you are certainly missing out. Virtualenv helps to give you a clean Python environment for every project you work on. With all projects, you generally end up installing at least a few packages. If you use your system Python installation for every project, then you can end up installing many packages and possibly needing different versions of the same package for different projects. You could remove and install the package each time you worked on the project, but Virtualenv removes this headache and keeps your projects separate.

You need to install two packages to use Virtualenv effectively: the Virtualenv package itself, which provides the functionality already described, and Virtualenvwrapper, which is optional but highly recommended. The wrapper basically provides handy command line utilities for creating, deleting, and working with Virtualenvs. For instance, with the wrapper installed you can create a Virtualenv, like so:

```
$ mkvirtualenv myenv
(myenv) $
```

After you create the Virtualenv, it activates automatically. Virtualenv informs you which version you are using by including the Virtualenv name at the start of your command prompt. Another nicety of the wrapper is that you can write your own command hooks to perform actions after, say, activating a Virtualenv. For example, I have set mine up to change into the project directory of the Virtualenv I am activating. I won't go into any more detail on Virtualenv now, but I highly recommend you install it on your machine. You can find all the details here:

- **Virtualenv:** `http://www.virtualenv.org/en/latest/`
- **Virtualenvwrapper:** `https://pypi.python.org/pypi/virtualenvwrapper`

Source Control (SVN, Git)

Version control is vital when working on a project—whether alone or with many developers. Your software will evolve naturally as you work on it, but what if you want to go back to changes made a few days earlier? You may want to try an idea out but need an easy way to revert to the prior working state should your current idea not work. How do you manage multiple developers working on the same code base and but still keep code changes in check? Source control gives you the power to manage all these problems easily, with great tools and integration with things like your favorite IDE. Source control also serves as the integration point for many other processes, such as continuous integration, code review systems, code quality and coverage reporting and release and deployment among others. Therefore it is clear that having a solid, well understood and maintainable source control system in place is crucial and forms a clear backbone to your code base's organization.

The two most commonly used source control systems are Subversion (SVN) and Git. SVN is the older of the two and works based on a single repository model. This means you check out an SVN repository of the code, do your work, and commit the changes back to the repository on the server.

Git, however, flips this model, where you clone a Git repository from the server. You then do some work and commit to your local copy of the repository. You can do as many local commits as you like, before pulling changes from the repository on the server, fixing any conflicts of code, and then pushing their changes to the server to be pulled by other developers. Git is known as a distributed source control for this reason. The main benefits of Git are

- **Local commits:** You can check in without an Internet connection.
- **Branches:** Easily create and switch branches to work on ideas or features away from the main "master" branch.

- **Merge:** Move branches back into the master. Rebase option to replay commits one by one rather than whole changes in one go a la SVN.
- **Git SVN:** Easily convert SVN repository to Git with built-in tools.

The main disadvantage to using Git is the lack of a central source of truth for your project. Because Git allows so much flexibility and freedom with its distributed repository architecture, without careful management you can end up with a confusing project structure and merge issues. With SVN having only the one central repository it makes it simpler for developers to keep their code on their machine up to date and in sync with the rest of the checked in code.

Git is becoming the standard for version control, with the added benefits of the distributed model suiting the workflows of many developers and its handling of merging code much better than SVN.

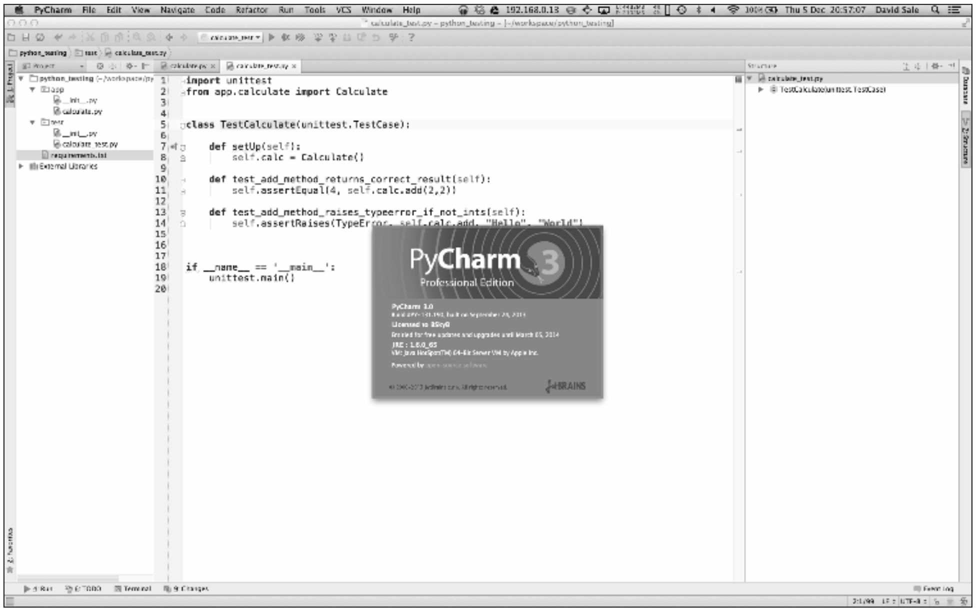
- **Git:** `http://git-scm.com`
- **Pro Git (Free eBook on Git):** `http://git-scm.com/book`
- **Try Git (Free online tutorial):** `http://try.github.com/`
- **SVN:** `http://subversion.apache.org`
- **Version Control with SVN (Free eBook):** `http://svnbook.red-bean.com/`

Interactive Development Environment (IDE)

Using a good IDE can be advantageous. Getting comfortable and familiar with an IDE is much like a car mechanic knowing every tool in his garage. An IDE doesn't make you a good programmer, but knowing how and when to use the right tools can make your life a lot easier. You can usually find a couple of stand-out IDEs for each language, and the development communities have their favorites. Two of my favorites for Python are PyCharm and the more all-encompassing IntelliJ, which handles many programming languages. IntelliJ and PyCharm are essentially the same product provided by Jetbrains, with PyCharm focused only on Python development and IntelliJ utilizing a Python plugin to provide the entire feature set of PyCharm alongside other language support. Both are regularly maintained, with new features releases and bug fixes every couple of months. The IDE is also well designed with support for many popular libraries and frameworks, such as Flask and Django web frameworks, and test support for running tests in the IDE itself instead of the command line and also Virtualenv's discussed earlier.

You also get the usual IDE features of code completion, syntax highlighting, and powerful searching, which is great when working with larger code bases. Both IDEs also offer expansion for many different types of tools you may want to interact with through the use of plug-ins. For example, if you write bash scripts to use with your code, you can install the bash plug-in and write the scripts in the IDE with full syntax highlight and support for adding code items like shebangs. A large repository of plug-ins is available, so should you need some extra functionality, you will likely find it there.

PyCharm, shown in Figure 1-1, is available on a trial, free (with some features unavailable), and full-feature set license basis. Pricing and further information is available at `http://www.jetbrains.com/pycharm/`.



```
import unittest
from app.calculate import Calculate


class TestCalculate(unittest.TestCase):

    def setUp(self):
        self.calc = Calculate()

    def test_add_method_returns_correct_result(self):
        self.assertEqual(4, self.calc.add(2,2))

    def test_add_method_raises_typeerror_if_not_ints(self):
        self.assertRaises(TypeError, self.calc.add, "Hello", "World")


if __name__ == '__main__':
    unittest.main()
```

FIGURE 1-1: PyCharm interface. This excellent Python IDE helps you develop great Python code.

Summary

This introductory chapter provided a brief description of whom this book is for. By now, you should have a feel for the concepts and topics that are conveyed in subsequent chapters. You were introduced to the subject of testing and given a brief background in the history of testing. You were shown how it has evolved from merely an afterthought of the product to a process, which in many cases is now baked in to the development.

"You do test, don't you?" poses a great question to many developers and hopefully serves as a reminder throughout your time reading this book and beyond. Testing is an important part of a developer's work and is shown to produce better results, especially in terms of aiming for zero defects in production. You looked into why testing is now such a pivotal process and how it can be beneficial both in a team and lone developer environment.

Finally, the chapter closes by ensuring you have some of the essential Python tooling on your machine, be it Linux, Mac, or Windows. By getting these fundamentals ready, you will be set to take on the examples and ideas in this book and it should also help you in your next projects and beyond.