Test-Driven Python Development

This book starts with a look at the test-driven development process, and how it is different from the traditional way of writing code. All the concepts are presented in the context of a real application that is developed in a step-by-step manner over the course of the book. While exploring the common types of smelly code, we will go back into our example project and clean up the smells that we find.

Additionally, we will use mocking to implement the parts of our example project that depend on other systems. Towards the end of the book, we’ll take a look at the most common patterns and anti-patterns associated with test-driven development, including integration of test results into the development process.

Who this book is written for

This book is intended for Python developers who want to use the principles of test-driven development (TDD) to create efficient and robust applications. In order to get the best out of this book, you should have development experience with Python.

What you will learn from this book

- Implement the test-driven development process in Python applications
- Fully leverage Python’s powerful built-in unittest and doctest modules
- Effectively add features to existing code bases that do not have any tests
- Safely resolve problems with the code and design, without breaking any functionality
- Utilize Python’s powerful mock and patch functionality to test complex interactions
- Integrate unit testing into the overall software delivery process
- Use doctest to test code with examples
- Enhance TDD with the nose2 test runner

Develop high-quality and maintainable Python applications using the principles of test-driven development

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Test-Driven Development'
- A synopsis of the book’s content
- More information on Test-Driven Python Development

About the Author

**Siddharta Govindaraj** first encountered Python in 2002, and it has remained his favorite programming language ever since. He was an early participant in BangPypers, India’s first Python community, and a speaker at InPycon. He is the founder of Silver Stripe Software, an Indian product start-up that develops products based on Python. A vocal proponent of the language, Siddharta extols the virtues of Python to anyone who will listen—and also to those who won't. His blog is at http://www.siddharta.me.
Test-Driven Python Development

Today, organizations encounter an ever-increasing delivery frequency problem. The days of delivering a new version every few years are long gone. The test-driven development process enables individuals and teams to deliver code that is not only robust but maintainable as well. Combine this with the rapid development speed of the Python language and you have a combination that enables the delivery of new features at the pace demanded by the market.

*Test-Driven Python Development* covers the end-to-end unit testing process, from the first simple test to complex tests that involve multiple components and interactions.

**What This Book Covers**

*Chapter 1, Getting Started with Test-Driven Development*, is the introductory chapter. It eases you into the TDD process by setting the context of why we need TDD and quickly getting started with a test.

*Chapter 2, Red-Green-Refactor -- The TDD Cycle*, goes deeper into the TDD process, driving the implementation of our example project as we write more tests.

*Chapter 3, Code Smells and Refactoring*, explores common types of smelly code, and we go back to our example project and clean up the smells we find. A key benefit of TDD is that it provides a safety net so that we can go in and clean up messy code.

*Chapter 4, Using Mock Objects to Test Interactions*, shows the use of mocking to implement the parts of our example project that depend on other systems. How do you test code that depends on external subsystems? We answer that question here by introducing mock objects.

*Chapter 5, Working with Legacy Code*, is about the fact that we often need to clean up or add features to old code that doesn't have existing tests. This chapter looks at strategies for doing this by working through one of the modules in our sample application.

*Chapter 6, Maintaining Your Test Suite*, proves that unit tests are also code and good coding practices apply to test code as well. This means keeping them well organized and easy to maintain. This chapter covers techniques to do just that.

*Chapter 7, Executable Documentation with doctest*, goes through the usage of doctest by working through one of the modules of our application.

*Chapter 8, Extending unittest with nose2*, gives us a look at nose2, a powerful test runner and plugin suite that extends the unittest framework.
Chapter 9, *Unit Testing Patterns*, covers some other patterns for unit testing. We see how to speed up tests and how we can run specific subsets of tests. We also take a look at data-driven tests and mocking patterns.

Chapter 10, *Tools to Improve Test-Driven Development*, explains some popular thirdparty tools to help us improve our TDD practice. Some of these tools, such as py.test and trial, are test runners with some unique features.

Appendix A, *Answers to Exercises*, contains the answers to the exercises presented throughout this book. There are many possible solutions, each with their own advantages and disadvantages.

Appendix B, *Working with Older Python Versions*, describes the changes needed in order to apply the techniques in this book for older versions of Python because this book has been written for Python 3.4.
My first encounter with Test-Driven Development (TDD) was back in 2002. At that time, it wasn't as mainstream as it is today, and I remember watching two developers writing some tests first and then implementing the functionality later. I thought it to be quite a strange way to write a code, and I promptly forgot about it. It was not until 2004, when I was involved with a challenging project, that I remembered TDD again. We were faced with a messy code that was difficult to test and every change seemed to create a series of new bugs. I thought, why not give TDD a shot and see how it worked? Suffice to say, TDD changed my outlook on software development. We stopped writing messy spaghetti code, and started writing better designed, more maintainable code. Regression failures dropped drastically. I was hooked.

Perhaps, like me, you face some challenges in a project and want to see how TDD can help you. Or, maybe you've heard a lot of people in the industry sing the praises of TDD and you're wondering what all the fuss is about. Maybe you've been reading about how TDD will be an essential skill in the near future, and want to get up to speed on it. No matter what your motivation, I hope this book will help you reach your goal.

TDD is a lot more than just a library or an API; it is a different way of developing software. In this book, we'll discuss how to apply this process to writing Python software. We're in luck, because Python has fantastic support for TDD right out of the box. In fact, unit testing has been an integral part of the Python standard library from the Python 2.1 release back in April 2001. Numerous improvements have been added since then, and the latest version that ships with Python 3.4 has a ton of exciting features that we'll explore over the course of this book.
Prerequisites

We will be using Python 3.4 in this book. Most of the techniques will work on Python 2.6+ as well, but some small changes may be required to the examples presented in this book in order to make them run. The Appendix B, Working with Older Python Versions lists these changes.

This book assumes that the reader has an intermediate level of Python understanding. In this book, we will be using Python language features such as lambdas, decorators, generators, and properties, and we assume that the reader is familiar with them. While we will give a brief description of these features as we encounter them, this book will not go into a lot of details about how they work, choosing instead to focus on how to test such code.

Note that if you have only Python 2.x installed on your system, then go to http://python.org and download the latest release in the Python 3.4 series. For Linux users, if Python 3.4 is not installed on your system, then check your distribution’s package repository to get the latest version. If no package exists, or you are using a non-standard or older version of a distribution, then you might have to compile it from source. The instructions to do so are available at https://docs.python.org/devguide/setup.html.

Since TDD is a hands-on coding activity, this book will use a lot of code snippets throughout. We recommend that you follow along by typing the code and running it yourself. It is much easier to understand the code and concepts when you can see it working (or not working) in front of you, rather than just reading through the code in this book.

Getting the code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

All the code in this book can be found online at https://github.com/siddhi/test_driven_python. You can select a specific branch of the repository to get the code for the start of this chapter, and work through this chapter from that starting point. You can also select a tag on the branch to get the code for the endpoint of this chapter, if you would prefer to jump to the end of the code.
Understanding test-driven development
After all the hype in the previous paragraphs, you might be wondering what exactly test-driven development is all about, and whether it is some complex procedure that requires a lot of skill to implement. Actually, test-driven development is very simple. The flowchart below shows the three steps in the process.

Let's walk through the preceding flowchart in a little more detail.

- **Red:** The first step is to write a small unit test case. As we have only written the test and haven't written the implementation yet, this test will naturally fail.

- **Green:** Next, we write the code that implements the desired functionality. At this point, we aren't looking to create the best design or the most readable code. We just want something simple that will pass the test.

- **Refactor:** Now that the test is passing, we go back and look at the code to see whether it can be improved. This may involve improving the design, or making it more readable or maintainable. We can use the tests written so far to ensure that we aren't breaking anything during the refactoring step.

- The cycle repeats as we proceed to the next test and implement the next bit of functionality.

Developers who are familiar with TDD usually go through this cycle many times an hour, implementing small steps of functionality each time.
TDD versus unit testing versus integration testing

Before we go further, let's take a short detour to define some terms and understand the differences between them. It is very easy to get confused between these terms, and they are often used with different meanings in different places.

In the broadest sense of the term, **unit testing** simply means testing a single unit of code, isolated from other code that it might be integrated with. Traditionally, unit testing was an activity that was primarily performed by test engineers. These engineers would take code given by the developers and run them through a suite of tests to verify that the code worked. Since this code was tested before integration, the process fits into the definition of a unit test. Traditional unit testing was typically a manual affair, with test engineers walking through the tests cases by hand, although some teams would go a step further and automate the tests.

An **integration test** is a test that involves exercising more than one unit of the system. The goal is to check whether these units have been integrated correctly. A typical integration test might be to go to a web page, fill in a form, and check whether the right message is displayed on the screen. In order for this test to pass, the UI must show the form correctly, the input must be captured correctly, and that input must be passed on to any logic processing. The steps might involve reading and writing from a database before a message is generated and the UI has to display it correctly. Only if all these interactions succeed will the integration test pass. If any one step should fail, the integration test will fail.

At this point, a valid question would be to ask why we need unit testing at all. Why not write only integration tests, where a single test could check so many parts of the application at once? The reason is that integration tests do not pinpoint the location of failure. A failing integration test could have an error in the UI, or in the logic, or somewhere in the way data is read or written. It will take a lot of investigation to see where the error is and fix it. By contrast, with well-written unit tests, a failing unit test will pinpoint exactly what is failing. Developers can go right to the point and fix the error.

Along the way, teams started moving to a process where developers themselves wrote tests for the code that they had implemented. These tests would be written after the developer had finished the implementation, and helped verify that the code worked as expected. These tests were usually automated. Such a process is generally called **developer testing** or **developer unit testing**.
TDD takes developer tests one step further, by writing the test before starting the implementation.

- **Developer tests**: Any kind of automated unit tests written by the developer, either before or after functionality is implemented.

- **Unit testing**: Any kind of testing of a particular unit of an application, either by a developer or a tester. These tests might be automated, or run manually.

- **Integration testing**: Any kind of testing that involves two or more units working together. These tests are typically performed by a tester, but they could be done by a developer as well. These tests might be manual or automated.

As we can see, unit testing is a general term, whereas developer testing is a specific subset of unit testing, and TDD is a specific form of developer testing.

On the surface, traditional unit testing, developer testing and TDD look similar. They all appear to be about writing tests for a single unit of code, with only minor variations based on who writes the test and whether the tests are written before the code or after.

However, dig deeper and differences appear. First, the intent is vastly different. Traditional unit testing and developer testing are all about writing tests to verify that the code works as it is supposed to. On the other hand, the main focus of TDD is not really about testing. The simple act of writing a test before the implementation changes the way we think when we implement the corresponding functionality. The resulting code is more testable, usually has a simple and elegant design, and is more maintainable and readable. This is because making a class easy to test also encourages good design practices, such as decoupling dependencies and writing small, modular classes.

Thus, one can say that TDD is all about writing better code, and it is just a happy side effect that we end up with a fully automated test suite as an outcome.

This difference in intent manifests itself in the type of tests. Developer testing usually results in large test cases, with a hefty part of the test code involved in test setup. By contrast, tests written using TDD are very small and numerous. Some people like to call them micro tests to differentiate them from other developer tests or traditional unit tests. TDD-style unit tests also try to be very fast to run because they are executed every few minutes during the development process.
Finally, the tests that are written in TDD are those that drive the development forward, and not necessarily those that cover all imaginable scenarios. For example, a function that is supposed to process a file might have tests to handle cases when the file exists or it doesn't exist, but probably won't have tests to see what happens if the file is 1 terabyte in size. The latter is something that a tester might conceivably test for, but would be an unusual test in TDD unless the function is clearly expected to work with such a file.

This really highlights the difference between TDD and other forms of unit testing.

Using TDD to build a stock alert application

Over the course of this book, we are going to be using TDD to build a simple stock alert application. The application will listen to stock updates from a source. The source can be anything—a server on the Internet, or a file on the hard drive, or something else. We will be able to define rules, and when the rule is matched, the application sends us an email or text message.

For example, we could define a rule as "If AAPL crosses the $550 level then send me an email". Once defined, the application will monitor updates and send an e-mail when the rule is matched.

Writing our first test

Enough talk. Let's get started with our application. What is a good place to start? From examining the application description mentioned earlier, it looks like we will need the following modules:

- Some way to read stock price updates, either from the Internet or from a file
- A way to manage the stock information so that we can process it
- A way to define rules and match them against the current stock information
- A way to send an email or text message when a rule is matched
Based on these requirements, we will be using the following design:

Each term is discussed as follows:

- **Alert**: This is the core of the application. An alert will take a **Rule** and map it to an **Action**. When the rule is matched, the action is executed.

- **Rule**: A **Rule** contains the condition we want to check for. We should get alerted when the rule is matched.

- **Action**: This is the action to be performed when the rule is matched. This could be as simple as printing a message on the screen, or, in more real-work scenarios, we might send an e-mail or a text message.

- **Stock**: The **Stock** class keeps track of the current price and possibly a history of the prices for a stock. It sends an **Event** to the **Alert** when there is an update. The alert then checks if it's rule matched and whether any action needs to be executed.

- **Event**: This class is used to send events to the **Alert** when a **Stock** is updated.

- **Processor**: The processor takes stock updates from the **Reader** and updates the **Stock** with the latest data. Updating the stock causes the event to be fired, which, in turn, causes the alert to check for a rule match.

- **Reader**: The **Reader** gets the stock alerts from some source. In this book, we are going to get updates from a simple list or a file, but you can build other readers to get updates from the Internet or elsewhere.
Among all these classes, the way to manage stock information seems to be the simplest, so let's start there. What we are going to do is to create a Stock class. This class will hold information about the current stock. It will store the current price and possibly some recent price history. We can then use this class when we want to match rules later on.

To get started, create a directory called src. This directory is going to hold all our source code. In the rest of this book, we will refer to this directory as the project root. Inside the src directory, create a subdirectory called stock_alerter. This is the directory in which we are going to implement our stock alert module.

Okay, let's get started with implementing the class.

NO! Wait! Remember the TDD process that was described earlier? The first step is to write a test, before we code the implementation. By writing the test first, we now have the opportunity to think about what we want this class to do.

So what exactly do we want this class to do? Let's start with something simple:

- A Stock class should be instantiated with the ticker symbol
- Once instantiated, and before any updates, the price should be None

Of course, there are many more things we will want this class to do, but we'll think about them later. Rather than coming up with a very comprehensive list of functionality, we're going to focus on tiny bits of functionality, one at a time. For now, the preceding expectation is good enough.

To convert the preceding expectation into code, create a file called stock.py in the project root, and put the following code in it:

```python
import unittest

class StockTest(unittest.TestCase):
    def test_price_of_a_new_stock_class_should_be_None(self):
        stock = Stock("GOOG")
        self.assertIsNone(stock.price)

if __name__ == "__main__":
    unittest.main()
```
What does this code do?

1. First, we import `unittest`. This is the library that has the test framework that we are going to use. Luckily for us, it is bundled into the Python standard library by default and is always available, so we don’t need to install anything, we can just import the module directly.

2. Second, we create a class `StockTest`. This class will hold all the test cases for the `Stock` class. This is just a convenient way of grouping related tests together. There is no rule that every class should have a corresponding test class. Sometimes, if we have a lot of tests for a class, then we may want to create separate test classes for each individual behavior, or group the tests some other way. However, in most cases, creating one test class for an actual class is the best way to go about it.

3. Our `StockTest` class inherits from the `TestCase` class in the `unittest` module. All tests need to inherit from this class in order to be identified as a test class.

4. Inside the class, we have one method. This method is a test case. The `unittest` framework will pick up any method that starts with `test`. The method has a name that describes what the test is checking for. This is just so that when we come back after a few months, we still remember what the test does.

5. The test creates a `Stock` object and then checks if the price is `None`. `assertIsNone` is a method provided by the `TestCase` class that we are inheriting from. It checks that its parameter is `None`. If the parameter is not `None`, it raises an `AssertionError` and fails the test. Otherwise, execution continues to the next line. Since that is the last line of the method, the test completes and is marked as a pass.

6. The last segment checks if the module was executed directly from the command line. In such a case, the `__name__` variable will have the value `__main__`, and the code will execute the `unittest.main()` function. This function will scan the current file for all tests and execute them. The reason we need to wrap this function call inside the conditional is because this part does not get executed if the module is imported into another file.

Congratulations! You have your first failing test. Normally, a failing test would be a cause for worry, but in this case, a failing test means that we’re done with the first step of the process and can move on to the next step.
Analyzing the test output

Now that we've written our test, it is time to run it. To run the test, just execute the file. Assuming that the current directory is the src directory, the following is the command to execute the file:

- Windows:
  python.exe stock_alerter\stock.py
- Linux/Mac:
  python3 stock_alerter/stock.py

If the python executable is not on your path, then you will have to give the full path to the executable here. In some Linux distributions, the file may be called python34 or python3.4 instead of python3.

When we run the file, the output looks like the following:

```
E
=====================================================================  
ERROR: test_price_of_a_new_stock_class_should_be_None (__main__.StockTest)
=====================================================================  
Traceback (most recent call last):
  File "stock_alerter\stock.py", line 6, in test_price_of_a_new_stock_class_should_be_None
    stock = Stock("GOOG")
NameError: name 'Stock' is not defined
------------------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

As expected, the test fails, because we haven't created the Stock class yet.

Let's look at that output in a little more detail:

- E on the first line signifies that the test gave an error. If a test passed, then you would have a dot on that line. A failed test would be marked with F. Since we have only a single test, there is only one character there. When we have multiple tests, then the status of each test will be displayed on that line, one character per test.
• After all the test statuses are displayed, we get a more detailed explanation of any test errors and failures. It tells us whether there was a failure or an error (in this case denoted by `ERROR`) followed by the name of the test and which class it belongs to. This is followed by a traceback, so we know where the failure occurred.
• Finally, there is a summary that shows how many tests were executed, how many passed or failed, and how many gave errors.

**Test errors versus test failures**

There are two reasons why a test might not pass: It might have failed or it might have caused an error. There is a small difference between these two. A **failure** indicates that we expected some outcome (usually via an assert), but got something else. For example, in our test, we are asserting that `stock.price` is `None`. Suppose `stock.price` has some other value apart from `None`, then the test will fail.

An error indicates that something unexpected happened, usually an unexpected exception was raised. In our previous example, we got an error because the Stock class has not yet been defined.

In both the cases, the test does not pass, but for different reasons, and these are reported separately as test failures and test errors.

**Making the test pass**

Now that we have a failing test, let's make it pass. Add the following code to the `stock.py` file, after the `import unittest` line:

```python
class Stock:
    def __init__(self, symbol):
        self.symbol = symbol
        self.price = None
```

What we have done here is to implement just enough code to pass the test. We've created the Stock class so the test shouldn't complain about it being missing, and we've initialized the price attribute to `None`.

What about the rest of the implementation for this class? This can wait. Our main focus right now is to pass the current expectation for this class. As we write more tests, we will end up implementing more of the class as well.
Run the file again, and this time the output should be like the following:

```
.  
Ran 1 test in 0.000s
OK
```

We've got a dot in the first line, which signifies that the test is passing. The `OK` message at the end tells us that all tests have passed.

The final step is to refactor the code. With so little code, there is really nothing much to clean up. So, we can skip the refactoring step and start with the next test.

**Reorganizing the test code**

We've added the test cases in the same file as the code. This is a good, simple way to add test cases to standalone scripts and applications that are not too complex. However, for larger applications, it is a good idea to keep test code separate from production code.

There are two common patterns for organizing test code this way.

The first pattern is to keep test code in a separate root directory, as shown in the following:

```
root
 |  
+-- package
 |  
|  
| +-- file1
| +-- file2
| 
+-- test
 |  
| +-- test_file1
| +-- test_file2
```
The other pattern is to keep test code as a submodule of the main code, as shown in the following:

```
root
 |
 +-- package
    |
     +-- file1
     +-- file2
     +-- test
        |
        +-- test_file1
        +-- test_file2
```

The first pattern is commonly used for standalone modules as it allows us to distribute the code and tests together. Tests can generally be run without having to perform a lot of setup or configuration. The second pattern has an advantage when the application has to be packaged without the test code, for example when deploying to production servers, or distributing to customers (in the case of a commercial application). However, both the patterns are in popular use, and it is mainly a personal preference as to which method to use.

We are going to follow the first pattern in this book. To get started, create a directory called `tests` inside the `stock_alerter` directory. Next, create a file called `test_stock.py` in this directory. We will put all our test cases in one-to-one correspondence with the source file. This means, a file called `sample.py` will have its test cases in the `tests/test_sample.py` file. This is a simple naming convention that helps to quickly locate test cases.

Finally, we move our test cases into this file. We also need to import the `Stock` class to be able to use it in the test case. Our `test_stock.py` file now looks like the following:

```python
import unittest
from ..stock import Stock

class StockTest(unittest.TestCase):
    def test_price_of_a_new_stock_class_should_be_None(self):
        stock = Stock("GOOG")
        self.assertIsNone(stock.price)
```
Remember to remove the import unittest line from stock.py, now that it no longer contains the test code. Previously we had just one standalone script, but we now have a stock_alerter module and a stock_alerter.tests submodule. Since we are now working with modules, we should also add in an empty __init__.py file in both the stock_alerter and tests directories.

Our file layout should now be like the following:

```
src
 |  
 |  +-- stock_alerter
 |      |  
 |      |   +-- __init__.py
 |      |   +-- stock.py
 |      |   +-- tests
 |      |       +-- __init__.py
 |      |       +-- test_stock.py
```

**Running the tests after the reorganization**

If you have noticed, we no longer have a call to unittest.main() in the test code. Including a call to unittest.main() works well with individual scripts since it allows us to run the tests by simply executing the file. However, it is not a very scalable solution. If we have hundreds of files, we would like to run all the tests at once, and not have to execute each file individually.

To address this, Python 3 comes with a very nice test discovery and execution capability from the command line. Simply go into the src directory and run the following command:

- Windows:
  ```bash
  python.exe -m unittest
  ```
- Linux/Mac:
  ```bash
  python3 -m unittest
  ```

This command will go through the current directory and all subdirectories and run all the tests that are found. This is the default autodiscover mode of execution, where the command searches all the files and runs the tests. Autodiscovery can also be explicitly run with the following command:

```
python3 -m unittest discover
```
Autodiscover can be customized to check in specific directories or files with the following parameters:

- `-s start_directory`: Specify the start directory from where the discovery should start. This defaults to the current directory.
- `-t top_directory`: Specify the top-level directory. This is the directory from which imports are performed. This is important if the start directory is inside the package and you get errors due to incorrect imports. This defaults to the start directory.
- `-p file_pattern`: The file pattern that identifies test files. By default it checks for python files that start with test. If we name our test files something else (for example, stock_test.py), then we have to pass in this parameter so that the file is correctly identified as a test file.

To illustrate the difference between the start and top directory, run the following command from the `src` directory:

```bash
python3 -m unittest discover -s stock_alerter
```

The preceding command will fail with an import error. The reason is because when the start directory is set to `stock_alerter`, then the `tests` directory is imported as a top-level module, and the relative import fails. To get around this, we need to use the following command:

```bash
python3 -m unittest discover -s stock_alerter -t .
```

This command will import all modules relative to the top directory, and so `stock_alerter` correctly becomes the main module.

You can also disable autodiscovery and specify only certain tests to be run:

- Passing in a module name will only run the tests within that module. For example, `python3 -m unittest stock_alerter.tests.test_stock` will run the tests only in `test_stock.py`.
- You can further refine to a specific class or method, such as `python3 -m unittest stock_alerter.tests.test_stock.StockTest`. 
Summary

Congratulations! You’ve completed one cycle of TDD. As you can see, each cycle is very quick. Some cycles, like the one we’ve just gone through, can be completed in a few seconds. Other cycles might involve a fair amount of cleanup and can take quite a long time. Each cycle will implement a small test, a small bit of functionality to pass the test, and then some cleanup to make the code of high quality.

In this chapter, we looked at what TDD is, how it is different from other forms of unit and integration testing, and wrote our first test.

At this point, our implementation is still very small and very simple. You might be wondering if it is worth all this hype just to write and implement four lines of very simple code. In the next few chapters, we’ll progress further with the examples and go more in-depth into the process.
Where to buy this book

You can buy Test-Driven Python Development from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.