Test-Driven Development With Python

Software development is easier and more accessible now than it ever has been. Unfortunately, rapid development speeds offered by modern programming languages make it easy for us as programmers to overlook the possible error conditions in our code and move on to other parts of a project. Automated tests can provide us with a level of certainty that our code really does handle various situations the way we expect it to, and these tests can save hundreds upon thousands of man-hours over the course of a project's development lifecycle.

Automated testing is a broad topic - there are many different types of automated tests that one might write and use. In this article we'll be concentrating on unit testing and, to some degree, integration testing using Python 3 and a methodology known as "test-driven development" (referred to as "TDD" from this point forward). Using TDD, you will learn how to spend more time coding than you spend manually testing your code.

To get the most out of this article, you should have a fair understanding of common programming concepts. For starters, you should be familiar with variables, functions, classes, methods, and Python's import mechanism. We will be using some neat features in Python, such as context managers, decorators, and monkey-patching. You don't necessarily need to understand the intricacies of these features to use them for testing.

About the Author

Josh VanderLinden is a life-long technology enthusiast, who started programming at the age of ten. Josh has worked primarily in web development, but he also has experience with network monitoring and systems administration. He has recently gained a deep appreciation for automated testing and TDD.

The main idea behind TDD is, as the name implies, that your tests drive your development efforts. When presented with a new requirement or goal, TDD would have you run through a series of steps:

- add a new (failing) test,
- run your entire test suite and see the new test fail,
- write code to satisfy the new test,
- run your entire test suite again and see all tests pass,
- refactor your code,
- repeat.

There are several advantages to writing tests for your code before you write the actual code. One of the most valuable is that this process forces you to really consider *what* you want the program to do before you start deciding *how* it will do so. This can help prepare you for unforeseen difficulties integrating your code with existing code or systems. You could also unearth possible conflicts between requirements that are delivered to you to fulfill.

Another incredibly appealing advantage of TDD is that you gain a higher level of confidence in the code that you've written. You can quickly detect bugs that new development efforts might introduce when combined with older, historically stable code. This high level of confidence is great not only for you as a developer, but also for your supervisors and clients.

The best way to learn anything like this is to do it yourself. We're going to build a simple game of Pig, relying on TDD to gain a high level of confidence that our game will do what we want it to long before we actually play it. Some of the basic tasks our game should be able to handle include the following:

- allow players to join,
- roll a six-sided die,
- track points for each player,
- prompt players for input,
- end the game when a player reaches 100 points.

We'll tackle each one of those tasks, using the TDD process outlined above. Python's built-in `unittest` library makes it easy to describe our expectations using assertions. There are many different types of assertions available in the standard library, most of which are pretty self-explanatory given a mild understanding of Python. For the rest, we have Python's wonderful documentation [1]. We can assert that values are equal, one object is an instance of another object, a string matches a regular expression, a specific exception is raised under certain conditions, and much more.

With `unittest`, we can group a series of related tests into subclasses of `unittest.TestCase`. Within those subclasses, we can add a series of methods whose names begin with `test`. These test methods should be designed to work independently of the other test methods. Any dependency between one test method and another introduces the potential to cause a chain reaction of failed tests when running your test suite in its entirety.

So let's take a look at the structure for our project and get into the code to see all of this in action.

```
pig/
    game.py
    test_game.py
```

Both files are currently empty. To get started, let’s add an empty test case to *test_game.py* to prepare for our game of Pig:

Listing 1. An empty TestCase subclass

```python
from unittest import TestCase


class GameTest(TestCase):

    pass
```

The Game Of Pig

The rules of Pig are simple: a player rolls a single die. If they roll anything other than one, they add that value to their score for that turn. If they roll a one, any points they’ve accumulated for that turn are lost. A player’s turn is over when they roll a one or they decide to hold. When a player holds before rolling a one, they add their points for that turn to their total points. The first player to reach 100 points wins the game.

For example, if player A rolls a three, player A may choose to roll again or hold. If player A decides to roll again and they roll another three, their total score for the turn is six. If player A rolls again and rolls a one, their score for the turn is zero and it becomes player B’s turn.

Player B may roll a six and decide to roll again. If player B rolls another six on the second roll and decides to hold, player B will add 12 points to their total score. It then becomes the next player’s turn.

We’ll design our game of Pig as its own class, which should make it easier to reuse the game logic elsewhere in the future.

Joining The Game

Before anyone can play a game, they have to be able to join it, correct? We need a test to make sure that works:

Listing 2. Our first test

```python
from unittest import TestCase

import game
```

```
class GameTest(TestCase):

    def test_join(self):
        """Players may join a game of Pig"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(pig.get_players(), ('PlayerA', 'PlayerB', 'PlayerC'))
```

We simply instantiate a new Pig game with some player names. Next, we check to see if we're able to get an expected value out of the game. As mentioned earlier, we can describe our expectations using assertions - we assert that certain conditions are met. In this case, we're asserting equality with `TestCase.assertEqual`. We want the players who start a game of Pig to equal the same players returned by `Pig.get_players`. The TDD steps suggest that we should now run our test suite and see what happens.

To do that, run the following command from your project directory:

```
python -m unittest
```

It should detect that the *test_game.py* file has a `unittest.TestCase` subclass in it and automatically run any tests within the file. Your output should be similar to this:

Listing 3. Running our first test

```
E
======================================================================
ERROR: test_join (test_game.GameTest)
Players may join a game of Pig
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./test_game.py", line 11, in test_join
    pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
AttributeError: 'module' object has no attribute 'Pig'

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (errors=1)
```

We had an error! The `E` on the first line of output indicates that a test method had some sort of Python error. This is obviously a failed test, but there's a little more to it than just our assertion failing. Looking at the output a bit more closely, you'll notice that it's telling us that our `game` module has no attribute `Pig`. This means that our *game.py* file doesn't have the class that we tried to instantiate for the game of Pig.

It is very easy to get errors like this when you practice TDD. Not to worry; all we need to do at this point is stub out the class in *game.py* and run our test suite again. A stub is just a function, class, or method definition that does nothing other than create a name within the scope of the program.

Listing 4. Stubbing code that we plan to test

```
class Pig:

    def __init__(self, *players):
        pass

    def get_players(self):
        """Return a tuple of all players"""

        pass
```

When we run our test suite again, the output should be a bit different:

Listing 5. The test fails for the right reason

```
F
======================================================================
FAIL: test_join (test_game.GameTest)
Players may join a game of Pig
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./test_game.py", line 12, in test_join
    self.assertEqual(pig.get_players(), ('PlayerA', 'PlayerB', 'PlayerC'))
AssertionError: None != ('PlayerA', 'PlayerB', 'PlayerC')

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Much better. Now we see `F` on the first line of output, which is what we want at this point. This indicates that we have a failing test method, or that one of the assertions within the test method did not pass. Inspecting the additional output, we see that we have an `AssertionError`. The return value of our `Pig.get_players` method is currently `None`, but we expect the return value to be a tuple with player names. Now, following with the TDD process, we need to satisfy this test. No more, no less.

Listing 6. Implementing code to satisfy the test

```
class Pig:

    def __init__(self, *players):
        self.players = players
    def get_players(self):
        """Returns a tuple of all players"""

        return self.players
```

And we need to verify that we've satisfied the test:

Listing 7. The test is satisfied

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

OK
```

Excellent! The dot (`.`) on the first line of output indicates that our test method passed. The return value of `Pig.get_players` is exactly what we want it to be. We now have a high level of confidence that players may join a game of Pig, and we will quickly know if that stops working at some point in the future. There's nothing more to do with this particular part of the game right now. We've satisfied our basic requirement.

As a side note, it's possible to get more verbose output from our tests if we pass in the `-v` flag to our command:

```
python -m unittest -v
```

And this is what it looks like:

```
test_join (test_game.GameTest)
Players may join a game of Pig ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Notice that the dot turned into `Players may join a game of Pig ... ok`. Python took the docstring from our test method and used it to describe which test is running. These docstrings can be a wonderful way to remind you and others about what the test is really testing. Let's move on to another part of the game.

Rolling The Die

The next critical piece of our game has to do with how players earn points. The game calls for a single six-sided die. We want to be confident that a player will *always* roll a value between one and six. Here's a possible test for that requirement.

Listing 8. Test for the roll of a six-sided die

```
    def test_roll(self):
        """A roll of the die results in an integer between 1 and 6"""

        pig = game.Pig('PlayerA', 'PlayerB')

        for i in range(500):
            r = pig.roll()
            self.assertIsInstance(r, int)
            self.assertTrue(1 <= r <= 6)
```

Since we're relying on "random" numbers, we test the result of the roll method repeatedly. Our assertions all happen within the loop because it's important that we always get an integer value from a roll and that the value is within our range of one to six. It's not bulletproof, but it should give us a fair level of confidence anyway. Don't forget to stub out the new `Pig.roll` method so our test fails instead of errors out.

Listing 9. Stub of our new Pig.roll method

```
    def roll(self):
        """Return a number between 1 and 6"""

        pass
```

Listing 10. Die rolling test fails

```
.F
======================================================================
FAIL: test_roll (test_game.GameTest)
A roll of the die results in an integer between 1 and 6
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./test_game.py", line 21, in test_roll
    self.assertIsInstance(r, int)
AssertionError: None is not an instance of <class 'int'>

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Let's check the output. There is a new `F` on the first line of output. For each test method in our test suite, we should expect to see some indication that the respective methods are executed. So far we've seen three common indicators:

- `E`, which indicates that a test method ran but had a Python error,
- `F`, which indicates that a test method ran but one of our assertions within that method failed,
- `.`, which indicates that a test method ran and that all assertions passed successfully.

There are other indicators, but these are the three we'll deal with for the time being.

The next TDD step is to satisfy the test we've just written. We can use Python's built-in `random` library to make short work of this new `Pig.roll` method.

Listing 11. Implementing the roll of a die

```
import random
```

```
    def roll(self):
        """Return a number between 1 and 6"""

        return random.randint(1, 6)
```

Listing 12. Implemention meets our expectations

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Checking Scores

Players might want to check their score mid-game, so let's add a test to make sure that's possible. Again, don't forget to stub out the new `Pig.get_scores` method before running the test.

Listing 13. Test that each player's score is available

```
    def test_scores(self):
        """Player scores can be retrieved"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(
            pig.get_score(),
            {
                'PlayerA': 0,
                'PlayerB': 0,
                'PlayerC': 0
            }
        )
```

Listing 14. Default score is not implemented

```
..F
======================================================================
FAIL: test_scores (test_game.GameTest)
Player scores can be retrieved
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./test_game.py", line 33, in test_scores
    'PlayerC': 0
AssertionError: None != {'PlayerA': 0, 'PlayerC': 0, 'PlayerB': 0}

----------------------------------------------------------------------
Ran 3 tests in 0.004s

FAILED (failures=1)
```

Note that ordering in dictionaries is not guaranteed, so your keys might not be printed out in the same order that you typed them in your code. And now to satisfy the test.

Listing 15. First implementation for default scores

```
    def __init__(self, *players):
        self.players = players

        self.scores = {}
        for player in self.players:
            self.scores[player] = 0
```

```
    def get_score(self):
        """Return the score for all players"""

        return self.scores
```

Listing 16. Checking our default scores implementation

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

The test has been satisfied. We can move on to another piece of code now if we'd like, but let's remember the fifth step from our TDD process. Let's try refactoring some code that we already know is working and make sure our assertions still pass.

Python's dictionary object has a neat little method called `fromkeys` that we can use to create a new dictionary with a list of keys. Additionally, we can use this method to set the default value for all of the keys that we specify. Since we've already got a tuple of player names, we can pass that directly into the `dict.fromkeys` method.

Listing 17. Another way to handle default scores

```
    def __init__(self, *players):
        self.players = players
        self.scores = dict.fromkeys(self.players, 0)
```

Listing 18. The new implementation is acceptable

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.006s

OK
```

The fact that our test still passes illustrates a few very important concepts to understand about valuable automated testing. The most useful unit tests will treat the production code as a "black box". We don't want to test *implementation*. Rather, we want to test the *output* or result of a unit of code given known input.

Testing the internal implementation of a function or method is asking for trouble. In our case, we found a way to leverage functionality built into Python to refactor our code. The end result is the same. Had we tested the specific low-level implementation of our `Pig.get_score` definition, the test could have easily broken after refactoring despite the code still ultimately doing what we want.

The idea of validating the output of a unit of code when given known input encourages another valuable practice. It stimulates the desire to design our code with more single-purpose functions and methods. It also discourages the inclusion of side effects.

In this context, side effects can mean that we're changing internal variables or state which could influence the behavior other units of code. If we only deal with input values and return values, it's very easy to reason about the behavior of our code. Side effects are not always bad, but they can introduce some interesting conditions at runtime that are difficult to reproduce for automated testing.

It's much easier to confidently test smaller, single-purpose units of code than it is to test massive blocks of code. We can achieve more complex behavior by chaining together the smaller units of code, and we can have a high level of confidence in these compositions because we know the underlying units meet our expectations.

Prompt Players For Input

Now we'll get into something more interesting by testing user input. This brings up a rather large stumbling block that many encounter when learning how to test their code: external systems. External systems may include databases, web services, local filesystems, and countless others.

During testing, we don't want to have to rely on our test computer, for example, being on a network, connected to the Internet, having routes to a database server, or making sure that a database server itself is online. Depending on all of those external systems being online is brittle and error-prone for automated testing. Why? We don't control all of those other systems, so we can't very well configure them all for each and every test.

In our case, user input can be considered an external system. We don't control values given to our program by the user, but we want to be able to deal with those values. Prompting the user for input each and every time we launch our test suite would adversely affect our tests in multiple ways. For example, the tests would suddenly take much longer, and the user would have to enter the same values each time they run the tests.

We can leverage a concept called "mocking" to remove all sorts of external systems from influencing our tests in bad ways. We can mock, or fake, the user input using known values, which will keep our tests running quickly and consistently. We'll use a fabulous library that is built into Python (as of version 3.3) called `mock` for this.

Let's begin testing user input by testing that we can prompt for player names. We'll implement this one as a standalone function that is separate from the `Pig` class. First of all, we need to modify our import line in *test_game.py* so we can use the `mock` library.

Listing 19. Importing the mock library

```
from unittest import TestCase, mock
```

If you're using Python 2, the `mock` library is still available. You can install it with `pip`:

```
pip install mock
```

Once installed, you would just add `import mock` to *test_game.py* instead of using the import line in Listing 19.

Listing 20. Introducing mocked objects

```
    def test_get_player_names(self):
        """Players can enter their names"""

        fake_input = mock.Mock(side_effect=['A', 'M', 'Z', ''])

        with mock.patch('builtins.input', fake_input):
            names = game.get_player_names()

        self.assertEqual(names, ['A', 'M', 'Z'])
```

Listing 21. Stub function that we will test

```
def get_player_names():
    """Prompt for player names"""

    pass
```

The `mock` library is extremely powerful, but it can take a while to get used to. Here we're using it to mock the return value of multiple calls to Python's built-in `input` function through `mock`'s `side_effect` feature. When you specify a list as the side effect of a mocked object, you're specifying the return value for each call to that mocked object. For each call to `input`, the first value will be removed from the `side_effect` list and used as the return value of the call.

In our code the first call to `input` will consume and return `'A'`, leaving `['M', 'Z', '']` as the remaining return values. The next call would consume and return `'M'`, leaving `['Z', '']`. We add an additional empty value as a side effect to signal when we're done entering player names. And we don't expect the empty value to appear as a player name.

Note that if you supply fewer return values in the `side_effect` list than you have calls to the mocked object, the code will raise a `StopIteration` exception. Say, for example, that you set the `side_effect` to `[1]` but that you called `input` twice in the code. The first time you call `input`, you'd get the `1` back. The second time you call `input`, it would raise the exception, indicating that our `side_effect` list has nothing more to return.

We're able to use this mocked `input` function through what's called a context manager. That is the block that begins with the keyword `with`. A context manager basically handles the setup and teardown for the block of code it contains. In this example, the `mock.patch` context manager will handle the temporary patching of the built-in `input` function while we run `game.get_player_names()`.

After the code in the `with` block as been executed, the context manager will roll back the `input` function to its original, built-in state. This is very important, particularly if the code in the `with` block raises some sort of exception. Even in conditions such as these, the changes to the `input` function will be reverted, allowing other code that may depend on `input`'s (or whatever object we have mocked) original functionality to proceed as expected.

Let's run the test suite to make sure our new test fails.

Listing 22. The new test fails

```
F...
======================================================================
FAIL: test_get_player_names (test_game.GameTest)
Players can enter their names
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./test_game.py", line 45, in test_get_player_names
    self.assertEqual(names, ['A', 'M', 'Z'])
AssertionError: None != ['A', 'M', 'Z']

----------------------------------------------------------------------
Ran 4 tests in 0.005s

FAILED (failures=1)
```

Well that was easy! Here's a possible way to satisfy this test:

Listing 23. Getting a list of player names from the user

```python
def get_player_names():
    """Prompt for player names"""

    names = []

    while True:
        value = input("Player {}'s name: ".format(len(names) + 1))
        if not value:
            break

        names.append(value)

    return names
```

The implementation is pretty straightforward. We create a variable to hold the list of names entered by the user. In an infinite loop, we prompt the user for a player name. If the value the enter is "falsy", we break out of the infinite loop and return the list of names. For every other value, we simply append it to the list of player names.

A “falsy” value is something that would evaluate to `False` in a boolean expression. Empty strings, lists, tuples, and dictionaries are all falsy, as is zero. When the user simply hits enter at the player name prompt, the value we receive is an empty string, which is falsy.

Listing 24. Our implementation meets expectations

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.004s

OK
```

Would you look at that?! We’re able to test user input without slowing down our tests much at all!

Notice, however, that we have passed a parameter to the input function. This is the prompt that appears on the screen when the program asks for player names. Let’s say we want to make sure that it’s actually printing out what we expect it to print out.

Listing 25. Test that the correct prompt appears on screen

```
    def test_get_player_names_stdout(self):
        """Check the prompts for player names"""

        with mock.patch('builtins.input', side_effect=['A', 'B', '']) as fake:
            game.get_player_names()

        fake.assert_has_calls([
            mock.call("Player 1's name: "),
            mock.call("Player 2's name: "),
            mock.call("Player 3's name: ")
        ])
```

This time we’re mocking the `input` function a bit differently. Instead of defining a new `mock.Mock` object explicitly, we’re letting the `mock.patch` context manager define one for us with certain side effects. When you use the context manager in this way, you’re able to obtain the implicitly-created `mock.Mock` object using the `as` keyword. We have assigned the mocked `input` function to a variable called `fake`, and we simply call the same code as in our previous test.

After running that code, we check to see if our fake `input` function was called with certain arguments using the `Mock.assert_has_calls` method on our mocked `input` function. This method takes a list of `mock.call` objects. Each `mock.call` object is used to describe the parameters that a function or method is called with. You may use both positional and keyword arguments for `mock.call` objects.

The order of our `mock.call` objects is important. When you call `Mock.assert_has_calls` with a list of `mock.call` objects, those calls must happen in the exact sequence you specify for the assertion to pass, and no other calls may happen in between the calls you've specified.

Notice that we aren't checking the result of the `get_player_names` function here - we've already done that in another test. There's often no point in testing the same functionality in multiple test methods.

Listing 26. All tests pass

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.008s

OK
```

Perfect. It works as we expect it to. One thing to take away from this example is that there does not need to be a one-to-one ratio of test methods to actual pieces of code. Right now we've got two test methods for the very same `get_player_names` function. It is often good to have multiple test methods for a single unit of code if that code may behave differently under various conditions.

Also note that we didn't exactly follow the TDD process for this last test. The code for which we wrote the test had already been implemented to satisfy an earlier test. It is acceptable to veer away from the TDD process, particularly if we want to validate assumptions that have been made along the way. When we implemented the original `get_player_names` function, we assumed that the prompt would look the way we wanted it to look. Our latest test simply proves that our assumptions were correct. And now we will be able to quickly detect if the prompt begins misbehaving at some point in the future.

To Hold or To Roll

Now it's time to write a test for different branches of code for when a player chooses to hold or roll again. We want to make sure that our `Pig.roll_or_hold` method will only return `roll` or `hold` and that it won't error out with invalid input.

Listing 27. Player can choose to roll or hold

```
    @mock.patch('builtins.input')
    def test_roll_or_hold(self, fake_input):
        """Player can choose to roll or hold"""

        fake_input.side_effect = ['R', 'H', 'h', 'z', '12345', 'r']

        pig = game.Pig('PlayerA', 'PlayerB')

        self.assertEqual(pig.roll_or_hold(), 'roll')
        self.assertEqual(pig.roll_or_hold(), 'hold')
        self.assertEqual(pig.roll_or_hold(), 'hold')
        self.assertEqual(pig.roll_or_hold(), 'roll')
```

This example shows yet another option that we have for mocking objects. We've "decorated" the `GameTest.test_roll_or_hold` method with `@mock.patch('builtins.input')`. When we use this option, we basically turn the entire contents of the method into the block within a context manager. The `builtins.input` function will be a mocked object throughout the entire method.

Also notice that the test method needs to accept an additional parameter, which we've called `fake_input`. When you mock objects with decorators in this way, your test methods must accept an additional parameter for each mocked object.

This time we're expecting to prompt the player to see whether they want to roll again or hold to end their turn. We set the `side_effect` of our `fake_input` mock to include our expected values of `r` (roll) and `h` (hold) in both lower and upper case, along with some input that we don't know how to use.

When we run the test suite with this new test (after stubbing out our `Pig.roll_or_hold` method), it should fail.

Listing 28. Test fails with stub

```
....F.
======================================================================
FAIL: test_roll_or_hold (test_game.GameTest)
Player can choose to roll or hold
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **keywargs)
  File "./test_game.py", line 67, in test_roll_or_hold
    self.assertEqual(pig.roll_or_hold(), 'roll')
AssertionError: None != 'roll'

----------------------------------------------------------------------
Ran 6 tests in 0.013s

FAILED (failures=1)
```

Fantastic! Notice how I get excited when I see a failing test? It means that the TDD process is working. Eventually you will enjoy seeing failed tests as well. Trust me.

And to satisfy our new test, we could use something like this:

Listing 29. Implementing the next action prompt

```
    def roll_or_hold(self):
        """Return 'roll' or 'hold' based on user input"""

        action = ''
        while True:
            value = input('(R)oll or (H)old? ')
            if value.lower() == 'r':
                action = 'roll'
                break
            elif value.lower() == 'h':
                action = 'hold'
                break

        return action
```

Similar to our `get_player_names` function, we're creating a variable to hold the action that the player chooses. Using another infinite loop, we prompt the player for an `R` or an `H`. For greater flexibility, we use the lower case version of the input to decide if the user has chosen to roll or to hold. If we get an `r` or `h` from the player, we set the action accordingly and break out of the infinite loop. For all other values, we prompt the user for the desired action again.

Let's see how this implementation holds up.

Listing 30. All tests pass

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.014s

OK
```

We know that our new code works. Even better than that, we know that we haven't broken any existing functionality.

Refactoring Tests

Since we're doing so much with user input, let's take a few minutes to refactor our tests to use a common mock for the built-in `input` function before proceeding with our testing.

Listing 31. Refactoring test code

```python
from unittest import TestCase, mock

import game

INPUT = mock.Mock()


@mock.patch('builtins.input', INPUT)
class GameTest(TestCase):

    def setUp(self):
        INPUT.reset_mock()

    def test_join(self):
        """Players may join a game of Pig"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(pig.get_players(), ('PlayerA', 'PlayerB', 'PlayerC'))

    def test_roll(self):
        """A roll of the die results in an integer between 1 and 6"""

        pig = game.Pig('PlayerA', 'PlayerB')

        for i in range(500):
            r = pig.roll()
            self.assertIsInstance(r, int)
            self.assertTrue(1 <= r <= 6)

    def test_scores(self):
        """Player scores can be retrieved"""

        pig = game.Pig('PlayerA', 'PlayerB', 'PlayerC')
        self.assertEqual(
            pig.get_score(),
            {
                'PlayerA': 0,
                'PlayerB': 0,
                'PlayerC': 0
            }
        )

    def test_get_player_names(self):
        """Players can enter their names"""

        INPUT.side_effect = ['A', 'M', 'Z', '']

        names = game.get_player_names()

        self.assertEqual(names, ['A', 'M', 'Z'])

    def test_get_player_names_stdout(self):
        """Check the prompts for player names"""

        INPUT.side_effect = ['A', 'B', '']

        game.get_player_names()

        INPUT.assert_has_calls([
            mock.call("Player 1's name: "),
            mock.call("Player 2's name: "),
            mock.call("Player 3's name: ")
        ])

    def test_roll_or_hold(self):
        """Player can choose to roll or hold"""

        INPUT.side_effect = ['R', 'H', 'h', 'z', '12345', 'r']

        pig = game.Pig('PlayerA', 'PlayerB')

        self.assertEqual(pig.roll_or_hold(), 'roll')
        self.assertEqual(pig.roll_or_hold(), 'hold')
        self.assertEqual(pig.roll_or_hold(), 'hold')
        self.assertEqual(pig.roll_or_hold(), 'roll')
```

A lot has changed in our tests code-wise, but the behavior should be exactly the same as before. Let's review the changes.

Listing 32. Global mock.Mock object and class decoration

```
INPUT = mock.Mock()


@mock.patch('builtins.input', INPUT)
class GameTest(TestCase):
```

We have defined a global `mock.Mock` instance called `INPUT`. This will be the variable that we use in place of the various uses of mocked input. We are also using `mock.patch` as a class decorator now, which will allow all test methods within the class to access the mocked `input` function through our `INPUT` global.

This decorator is a bit different from the one we used earlier. Instead of allowing a `mock.Mock` object to be implicitly created for us, we're specifying our own instance. The value in this solution is that you don't have to modify the method signatures for each test method to accept the mocked `input` function. Instead, any test method that needs to access the mock may use the `INPUT` global.

Listing 33. Reset global mocks before each test method

```
    def setUp(self):
        INPUT.reset_mock()
```

We've added a `setUp` method to our class. This method name has a special meaning when used with Python's `unittest` library. The `TestCase.setUp` method will be executed *before* each and every test method within the class. There's a similar special method called `TestCase.tearDown` that is executed *after* each and every test method within the class.

These methods are useful for getting things into a state such that our tests will run successfully or cleaning up after our tests. We're using the `GameTest.setUp` method to reset our mocked `input` function. This means that any calls or side effects from one test method are removed from the mock, leaving it in a pristine state at the start of each test.

Listing 34. Updating existing test methods to use the global mock

```
    def test_get_player_names(self):
        """Players can enter their names"""

        INPUT.side_effect = ['A', 'M', 'Z', '']

        names = game.get_player_names()

        self.assertEqual(names, ['A', 'M', 'Z'])
```

```
    def test_get_player_names_stdout(self):
        """Check the prompts for player names"""

        INPUT.side_effect = ['A', 'B', '']

        game.get_player_names()

        INPUT.assert_has_calls([
            mock.call("Player 1's name: "),
            mock.call("Player 2's name: "),
            mock.call("Player 3's name: ")
        ])
```

The `GameTest.test_get_player_names` test method no longer defines its own mock object. The context manager is also not necessary anymore, since the entire method is effectively executed within a context manager because we've decorated the entire class. All we need to do is specify the side effects, or list of return values, for our mocked `input` function. The `GameTest.test_get_player_names_stdout` test method has also been updated in a similar fashion.

Listing 35. Using the global mock

```
    def test_roll_or_hold(self):
        """Player can choose to roll or hold"""

        INPUT.side_effect = ['R', 'H', 'h', 'z', '12345', 'r']

        pig = game.Pig('PlayerA', 'PlayerB')

        self.assertEqual(pig.roll_or_hold(), 'roll')
        self.assertEqual(pig.roll_or_hold(), 'hold')
        self.assertEqual(pig.roll_or_hold(), 'hold')
        self.assertEqual(pig.roll_or_hold(), 'roll')
```

Finally, our `GameTest.test_roll_or_hold` test method no longer has its own decorator. Also note that the additional parameter to the method is no longer necessary.

When you find that you are mocking the same thing in many different test methods, as we were doing with the `input` function, a refactor like what we've just done can be a good idea. Your test code becomes much cleaner and more consistent. As your test suite continues to grow, just like with any code, you need to be able to maintain it. Abstracting out common code early on, both in your tests and in your production code, will help you and others to maintain and understand the code.

Now that we've reviewed the changes, let's verify that our tests haven't broken.

Listing 36. Refactoring has not broken our tests

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.010s

OK
```

Wonderful. All is well with our refactored tests.

Tying It All Together

We have successfully implemented the basic components of our Pig game. Now it's time to tie everything together into a game that people can play. What we're about to do could be considered a sort of integration test. We aren't integrating with any external systems, but we are going to combine all of our work up to this point together. We want to be sure that the previously tested units of code will operate nicely when meshed together.

Listing 37. Testing actual gameplay

```
    def test_gameplay(self):
        """Users may play a game of Pig"""

        INPUT.side_effect = [
            # player names
            'George',
            'Bob',
            '',

            # roll or hold
            'r', 'r',                # George
            'r', 'r', 'r', 'h',      # Bob
            'r', 'r', 'r', 'h',      # George
        ]
        pig = game.Pig(*game.get_player_names())
        pig.roll = mock.Mock(side_effect=[
            6, 6, 1,                # George
            6, 6, 6, 6,             # Bob
            5, 4, 3, 2,             # George
        ])

        self.assertRaises(StopIteration, pig.play)

        self.assertEqual(
            pig.get_score(),
            {
                'George': 14,
                'Bob': 24
            }
        )
```

This test method is different from our previous tests in a few ways. First, we're dealing with two mocked objects. We've got our usual mocked `input` function, but we're also monkey patching our game's `roll` method. We want this additional mock so that we're dealing with known values as opposed to randomly generated integers.

Instead of monkey patching the `Pig.roll` method, we could have mocked the `random.randint` function. However, doing so would be walking the fine and dangerous line of relying on the underlying implementation of our `Pig.roll` method. If we ever changed our algorithm for rolling a die and our tests mocked `random.randint`, our test would likely fail.

Our first course of action is to specify the values that we want to have returned from both of these mocked functions. For our input, we start with responses for player name prompts and also include some "roll or hold" responses. Next we instantiate a `Pig` game and define some not-so-random values that the players will roll.

All we are interested in checking for now is that players each take turns rolling and that their scores are adjusted according to the rules of the game. We don't need to worry just yet about a player winning when they earn 100 or more points.

We're using the `TestCase.assertRaises` method because we know that neither player will obtain at least 100 points given the side effect values for each mock. As discussed earlier, we know that the game will exhaust our list of return values and expect that the `mock` library itself (not our game!) will raise the `StopIteration` exception.

After defining our input values and "random" roll values, we run through the game long enough for the players to earn some points. Then we check that each player has the expected number of points. Our test is relying on the fact that all of our assertions up to this point are passing.

So let's take a look at our failing test (again, after stubbing the new `Pig.play` method):

Listing 38. Test fails with the stub

```
F......
======================================================================
FAIL: test_gameplay (test_game.GameTest)
Users may play a game of Pig
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **keywargs)
  File "./test_game.py", line 99, in test_gameplay
    self.assertRaises(StopIteration, pig.play)
AssertionError: StopIteration not raised by play

----------------------------------------------------------------------
Ran 7 tests in 0.012s

FAILED (failures=1)
```

Marvelous, the test fails, exactly as we want it to. Let’s fix that by implementing our game.

Listing 39. Gameplay implementation

```
from itertools import cycle
```

```
 def play(self):
     """Start a game of Pig"""

     for player in cycle(self.players):
         print('Now rolling: {}'.format(player))
         action = 'roll'
         turn_points = 0

         while action == 'roll':
             value = self.roll()
             if value == 1:
                 print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                 break

             turn_points += value
             print('{} rolled a {} and now has {} points for this turn'.format(
                 player, value, turn_points
             ))

             action = self.roll_or_hold()

         self.scores[player] += turn_points
```

So the core of any game is that all players take turns. We will use Python’s built-in `itertools` library to make that easy. This library has a `cycle` function, which will continue to return the same values over and over. All we need to do is pass our list of player names into `cycle()`. Obviously, there are other ways to achieve this same functionality, but this is probably the easiest option.

Next, we print the name of the player who is about to roll and set the number of points earned during the turn to zero. Since each player gets to choose to roll or hold most of the time, we roll the die within a `while` loop. That is to say, while the user chooses to roll, execute the code block within the `while` statement.

The first step to that loop is to roll the die. Because of the values that we specified in our test for the `Pig.roll` method, we know exactly what will come of each roll of the die. Per the rules of Pig, we need to check if the rolled value is a one. If so, the player loses all points earned for the turn and it becomes the next player's turn. The `break` statement allows us to break out of the `while` loop, but continue within the `for` loop.

If the rolled value is something other than one, we add the value to the player's points for the turn. Then we use our `roll_or_hold` method to see if the user would like to roll again or hold. When the user chooses to roll again, `action` is set to `'roll'`, which satisfies the condition for the `while` loop to iterate again. If the user chooses to hold, `action` is set to `'hold'`, which does not satisfy the `while` loop condition.

When a player's turn is over, either from rolling a one or choosing to hold, we add the points they earned during their turn to their overall score. The `for` loop and `itertools.cycle` function takes care of moving on to the next player and starting all over again.

Let's run our test to see if our code meets our expectations.

Listing 40. Broken implementation and print output in test results

```
F......Now rolling: George
George rolled a 6 and now has 6 points for this turn
George rolled a 6 and now has 12 points for this turn
George rolled a 1 and lost 12 points
Now rolling: Bob
Bob rolled a 6 and now has 6 points for this turn
Bob rolled a 6 and now has 12 points for this turn
Bob rolled a 6 and now has 18 points for this turn
Bob rolled a 6 and now has 24 points for this turn
Now rolling: George
George rolled a 5 and now has 5 points for this turn
George rolled a 4 and now has 9 points for this turn
George rolled a 3 and now has 12 points for this turn
George rolled a 2 and now has 14 points for this turn
Now rolling: Bob

======================================================================
FAIL: test_gameplay (test_game.GameTest)
Users may play a game of Pig
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
```

```
    return func(*args, **keywargs)
  File "./test_game.py", line 105, in test_gameplay
    'Bob': 24
AssertionError: {'George': 26, 'Bob': 24} != {'George': 14, 'Bob': 24}
- {'Bob': 24, 'George': 26}
?                        ^^

+ {'Bob': 24, 'George': 14}
?                        ^^


----------------------------------------------------------------------
Ran 7 tests in 0.014s

FAILED (failures=1)
```

Oh boy. This is not quite what we expected. First of all, we see the output of all of the `print` functions in our game, which makes it difficult to see the progression of our tests. Additionally, our player scores did not end up as we expected.

Let's fix the broken scores problem first. Notice that George has many more points than we expected- he ended up with 26 points instead of the 14 that he should have earned. This suggests that he still earned points for a turn when he shouldn't have. Let's inspect that block of code:

Listing 41. The culprit

```
              if value == 1:
                  print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                  break
```

Ah hah! We *display* that the player loses their turn points when they roll a one, but we don't actually have code to do that. Let's fix that:

Listing 42. The solution

```
              if value == 1:
                  print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                  turn_points = 0
                  break
```

Now to verify that this fixes the problem.

Listing 43. Acceptable implementation still with print output

```
.......Now rolling: George
George rolled a 6 and now has 6 points for this turn
George rolled a 6 and now has 12 points for this turn
George rolled a 1 and lost 12 points
Now rolling: Bob
Bob rolled a 6 and now has 6 points for this turn
Bob rolled a 6 and now has 12 points for this turn
Bob rolled a 6 and now has 18 points for this turn
Bob rolled a 6 and now has 24 points for this turn
Now rolling: George
George rolled a 5 and now has 5 points for this turn
George rolled a 4 and now has 9 points for this turn
George rolled a 3 and now has 12 points for this turn
George rolled a 2 and now has 14 points for this turn
Now rolling: Bob

----------------------------------------------------------------------
Ran 7 tests in 0.015s

OK
```

Perfect. The scores end up as we expect. The only problem now is that we still see all of the output of the `print` function, which clutters our test output. There a many ways to hide this output. Let’s use `mock` to hide it.

One option for hiding output with `mock` is to use a decorator. If we want to be able to assert that certain strings or patterns of strings will be printed to the screen, we could use a decorator similar to what we did previously with the `input` function:

```
@mock.patch('builtins.print')
def test_something(self, fake_print):
```

Alternatively, if we don’t care to make any assertions about what is printed to the screen, we can use a decorator such as:

```
@mock.patch('builtins.print', mock.Mock())
def test_something(self):
```

The first option requires an additional parameter to the decorated test method while the second option requires no change to the test method signature. Since we aren’t particularly interested in testing the `print` function right now, we’ll use the second option.

Listing 44. Suppressing print output

```
    @mock.patch('builtins.print', mock.Mock())
    def test_gameplay(self):
        """Users may play a game of Pig"""

        INPUT.side_effect = [
            # player names
            'George',
            'Bob',
            '',

            # roll or hold
            'r', 'r',                   # George
            'r', 'r', 'r', 'h',         # Bob
            'r', 'r', 'r', 'h',         # George
        ]

        pig = game.Pig(*game.get_player_names())
        pig.roll = mock.Mock(side_effect=[
            6, 6, 1,                    # George
            6, 6, 6, 6,                 # Bob
            5, 4, 3, 2,                 # George
        ])

        self.assertRaises(StopIteration, pig.play)

        self.assertEqual(
            pig.get_score(),
            {
                'George': 14,
                'Bob': 24
            }
        )
```

Let's see if the test output has been cleaned up at all with our updated test.

Listing 45. All tests pass with no print output

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.009s

OK
```

Isn't `mock` wonderful? It is so very powerful, and we're only scratching the surface of what it offers.

Winning The Game

The final piece to our game is that one player must be able to win the game. As it stands, our game will continue indefinitely. There's nothing to check when a player's score reaches or exceeds 100 points. To make our lives easier, we'll assume that the players have already played a few rounds (so we don't need to specify a billion input values or "random" roll values).

Listing 46. Check that a player may indeed win the game

```
    @mock.patch('builtins.print')
    def test_winning(self, fake_print):
        """A player wins when they earn 100 points"""
        INPUT.side_effect = [
            # player names
            'George',
            'Bob',
            '',

            # roll or hold
            'r', 'r',               # George
        ]

        pig = game.Pig(*game.get_player_names())
        pig.roll = mock.Mock(side_effect=[2, 2])

        pig.scores['George'] = 97
        pig.scores['Bob'] = 96

        pig.play()

        self.assertEqual(
            pig.get_score(),
            {
                'George': 101,
                'Bob': 96
            }
        )
        fake_print.assert_called_with('George won the game with 101 points!')
```

The setup for this test is very similar to what we did for the previous test. The primary difference is that we set the scores for the players to be near 100. We also want to check some portion of the screen output, so we changed the method decorator a bit.

We've introduced a new call with our screen output check: `Mock.assert_called_with`. This handy method will check that the most recent call to our mocked object had certain parameters. Our assertion is checking that the last thing our `print` function is invoked with is the winning string.

What happens when we run the test as it is?

Listing 47. Players currently cannot win

```
.......E
======================================================================
ERROR: test_winning (test_game.GameTest)
A player wins when they earn 100 points
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **keywargs)
  File "./test_game.py", line 130, in test_winning
    pig.play()
  File "./game.py", line 50, in play
    value = self.roll()
  File "/usr/lib/python3.3/unittest/mock.py", line 846, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/lib/python3.3/unittest/mock.py", line 904, in _mock_call
```

```
    result = next(effect)
StopIteration

----------------------------------------------------------------------
Ran 8 tests in 0.015s

FAILED (errors=1)
```

Hey, there's the `StopIteration` exception that we discussed a couple of times before. We've only specified two roll values, which should be just enough to push George's score over 100. The problem is that the game continues, even when George's score exceeds the maximum, and our mocked `Pig.roll` method runs out of return values.

We don't want to use the `TestCase.assertRaises` method here. We expect the game to end after any player's score reaches 100 points, which means the `Pig.roll` method should not be called anymore.

Let's try to satisfy the test.

Listing 48. First attempt to allow winning

```python
def play(self):
    """Start a game of Pig"""

    for player in cycle(self.players):
        print('Now rolling: {}'.format(player))
        action = 'roll'
        turn_points = 0

        while action == 'roll':
            value = self.roll()
            if value == 1:
                print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                turn_points = 0
                break

            turn_points += value
            print('{} rolled a {} and now has {} points for this turn'.format(
                player, value, turn_points
            ))

            action = self.roll_or_hold()

        self.scores[player] += turn_points
        if self.scores[player] >= 100:
            print('{} won the game with {} points!'.format(
                player, self.scores[player]
            ))
            return
```

After each player's turn, we check to see if the player's score is 100 or more. Seems like it should work, right? Let's check.

Listing 49. Players still cannot win

```
.......E
======================================================================
ERROR: test_winning (test_game.GameTest)
A player wins when they earn 100 points
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **keywargs)
  File "./test_game.py", line 130, in test_winning
    pig.play()
  File "./game.py", line 50, in play
    value = self.roll()
  File "/usr/lib/python3.3/unittest/mock.py", line 846, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/lib/python3.3/unittest/mock.py", line 904, in _mock_call
    result = next(effect)
StopIteration

----------------------------------------------------------------------
Ran 8 tests in 0.011s

FAILED (errors=1)
```

Hmmm... We get the same `StopIteration` exception. Why do you suppose that is? We're just checking to see if a player's total score reaches 100, right? That's true, but we're only doing it at the *end* of a player's turn. We need to check to see if they reach 100 points *during* their turn, not when they lose their turn points or decide to hold. Let's try this again.

Listing 50. Winning check needs to happen elsewhere

```python
 def play(self):
     """Start a game of Pig"""

     for player in cycle(self.players):
         print('Now rolling: {}'.format(player))
         action = 'roll'
         turn_points = 0

         while action == 'roll':
             value = self.roll()
             if value == 1:
                 print('{} rolled a 1 and lost {} points'.format(player, turn_points))
                 turn_points = 0
                 break

             turn_points += value
             print('{} rolled a {} and now has {} points for this turn'.format(
                 player, value, turn_points
             ))

             if self.scores[player] + turn_points >= 100:
                 self.scores[player] += turn_points
                 print('{} won the game with {} points!'.format(
                     player, self.scores[player]
```

```
                    ))
                    return

              action = self.roll_or_hold()

          self.scores[player] += turn_points
```

We’ve moved the total score check into the `while` loop, after the check to see if the player rolled a one. How does our test look now?

Listing 51. Players may now win the game

```
........
----------------------------------------------------------------------
Ran 8 tests in 0.009s

OK
```

Playing From the Command Line

It would appear that our basic Pig game is now complete. We’ve tested and implemented all of the basics of the game. But how can we play it ourselves? We should probably make the game easy to run from the command line. But first, we need to describe our expectations in a test.

Listing 52. Command line invocation

```
    def test_command_line(self):
        """The game can be invoked from the command line"""

        INPUT.side_effect = [
            # player names
            'George',
            'Bob',
            '',

            # roll or hold
            'r', 'r', 'h',            # George
            # Bob immediately rolls a 1
            'r', 'h',                 # George
            'r', 'r', 'h'             # Bob
        ]

        with mock.patch('builtins.print') as fake_print, \
             mock.patch.object(game.Pig, 'roll') as die:

            die.side_effect = cycle([6, 2, 5, 1, 4, 3])
            self.assertRaises(StopIteration, game.main)

        # check output
        fake_print.assert_has_calls([
            mock.call('Now rolling: George'),
            mock.call('George rolled a 6 and now has 6 points for this turn'),
            mock.call('George rolled a 2 and now has 8 points for this turn'),
            mock.call('George rolled a 5 and now has 13 points for this turn'),
            mock.call('Now rolling: Bob'),
            mock.call('Bob rolled a 1 and lost 0 points'),
            mock.call('Now rolling: George'),
            mock.call('George rolled a 4 and now has 4 points for this turn'),
            mock.call('George rolled a 3 and now has 7 points for this turn'),
            mock.call('Now rolling: Bob'),
            mock.call('Bob rolled a 6 and now has 6 points for this turn'),
            mock.call('Bob rolled a 2 and now has 8 points for this turn'),
            mock.call('Bob rolled a 5 and now has 13 points for this turn')
        ])
```

This test starts out much like our recent gameplay tests by defining some return values for our mocked `input` function. After that, though, things are very much different. We see that multiple context managers can be used with one `with` statement. It's also possible to do multiple nested `with` statements, but that depends on your preference.

The first object we're mocking is the built-in `print` function. Again, this way of mocking objects is very similar to mocking with class or method decorators. Since we will be invoking the game from the command line, we won't be able to easily inspect the internal state of our `Pig` game instance for scores. As such, we're mocking `print` so that we can check screen output with our expectations.

We're also patching our `Pig.roll` method as before, only this time we're using a new `mock.patch.object` function. Notice that all of our uses of `mock.patch` thus far have been passed a simple string as the first parameter. This time we're passing an actual object as the first parameter and a string as the second parameter.

The `mock.patch.object` function allows us to mock members of another object. Again, since we won't have direct access to the `Pig` instance, we can't monkey patch the `Pig.roll` the way we did previously. The outcome of this method should be the same as the other method.

Being the lazy programmers that we are, we've chosen to use the `itertools.cycle` function again to continuously return some value back for each roll of the die. Since we don't want to specify roll-or-hold values for an entire game of Pig, we use `TestCase.assertRaises` to say we expect `mock` to raise a `StopIteration` exception when there are no additional return values for the `input` mock.

I should mention that testing screen output as we're doing here is not exactly the best idea. We might change the strings, or we might later add more `print` calls. Either case would require that we modify our test itself, and that's added overhead. Having to maintain production code is a chore by itself, and adding test case maintenance to that is not exactly appealing.

That said, we will push forward with our test this way for now. We should run our test suite now, but be sure to mock out the new `main` function in *game.py* first.

Listing 53. Expected failure

```
F........
======================================================================
FAIL: test_command_line (test_game.GameTest)
The game can be invoked from the command line
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/lib/python3.3/unittest/mock.py", line 1087, in patched
    return func(*args, **keywargs)
  File "./test_game.py", line 162, in test_command_line
    self.assertRaises(StopIteration, game.main)
AssertionError: StopIteration not raised by main

----------------------------------------------------------------------
Ran 9 tests in 0.012s

FAILED (failures=1)
```

We haven't implemented our `game.main` function yet, so none of the mocked input values are consumed, and no `StopIteration` exception is raised. Just as we expect for now. Let's write some code to launch the game from the command line now.

Listing 54. Basic command line entry point

```python
def main():
    """Launch a game of Pig"""

    game = Pig(*get_player_names())
    game.play()


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

Hey, that code looks pretty familiar, doesn't it? It's pretty much the same code we've used in previous gameplay test methods. Awesome!

There's one small bit of magic code that we've added at the bottom. That `if` statement is the way that you allow a Python script to be invoked from the command line. A Python module's `__name__` attribute will only be `__main__` when that module is invoked directly, as opposed to just being imported by another Python module.

Let's run the test again to make sure the `game.main` function does what we expect.

Listing 55. All tests pass

```
.........
----------------------------------------------------------------------
Ran 9 tests in 0.013s

OK
```

Beauty! At this point, you should be able to invoke your very own Pig game on the command line by running:

```
python game.py
```

Isn't that something? We waited to manually run the game until we had written and satisfied tests for all of the basic requirements for a game of Pig. The first time we play it ourselves, the game just works!

Reflecting On Our Pig

Now that we've gone through that exercise, we need to think about what all of this new-found TDD experience means for us. All tests passing absolutely does not mean the code is *bug-free*. It simply means that the code *meets the expectations* that we've described in our tests.

There are plenty of situations that we haven't covered in our tests or handled in our code. Can you think of anything that is wrong with our game right now? What will happen if you don't enter any player names? What if you only enter one player name? Will the game be able to handle a large number of players?

We can make assumptions and predictions about how the code will behave under such conditions, but wouldn't it be nice to have a high level of confidence that the code will handle each scenario as we expect?

Also, what happens when we find an unexpected bug in our game? The very first thing you should try to do is write a test case that reproduces the bug. Once you do that, just continue with the TDD process. Run your test suite to see that your new test truly does reproduce the bug. Then fix the bug and run your test suite to see all tests passing. You'll know that you haven't broken anything that was once working, and you'll be confident that the new bug has been resolved.

It's important to resist the urge to fix bugs in the production code without first writing a test to reproduce the unexpected behavior. Fixing the bug right away might give you a short-term reassurance that the code works the way you want, but as you continue to build onto your projects these bugs may reappear.

Additionally, you should get into the habit of running your test suite after you make any change to your production code. Many developers use hooks in their version control software to automatically run the test suite before allowing a commit. Other developers use continuous integration software to run a project's test suite after every commit, notifying developers of failed tests via email.

The automated testing ecosystem is vast and very diverse. There are an incredible number of tools at your disposal for needs ranging from code coverage to automated GUI testing. New tools appear all the time, so if you're interested in automated testing, it would be beneficial for you to research the other resources that are out there.

What Now?

Our game is far from perfect. Now that we have a functional game of Pig and a good foundation in TDD, here are some tasks that you might consider implementing to practice TDD.

- accept player names via the command line (without the prompt),
- bail out if only one player name is given,
- allow the maximum point value to be specified on the command line,
- allow players to see their total score when choosing to roll or hold,
- track player scores in a database,
- print the runner-up when there are three or more players,
- turn the game into an IRC bot.

The topics covered in this article should have sufficiently prepared you to write tests for each one of these additional tasks. Good luck in your TDD endeavors!

[1] `unittest` documentation: http://docs.python.org/dev/library/unittest