O'REILLY®
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
See this if you're having trouble printing code examples

Test-Driven Development in Python

by Jason Diamond
12/02/2004

Introduction

Test-driven development is *not* about testing. Test-driven development is about *development* (and design), specifically improving the quality and design of code. The resulting unit tests are just an extremely useful by-product.

That's all I'm going to tell you about test-driven development. The rest of this article will *show* you how it works. Come work on a project with me; we'll build a very simple tool together. I'll make mistakes, fix them, and change designs in response to what the tests tell me. Along the way, we'll throw in a few refactorings, design patterns, and object-oriented design principles.

To make this project fun, we'll do it in Python.

Python is an excellent language for test-driven development because it (usually) does exactly what you want it to without getting in your way. The standard library even comes with everything you need in order to start developing TDD-style.

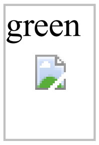
I assume that you're familiar with Python but not necessarily familiar with test-driven development or Python's `unittest` module. You need to know only a little in order to start testing.

Python's `unittest` Module

Since version 2.1, Python's standard library has included a `unittest` module, based on JUnit (by Kent Beck and Erich Gamma), the de facto standard unit test framework for Java developers. Formerly known as PyUnit, it also runs on Python versions prior to 2.1 with a separate download.

Let's jump right in. Here's a "unit" and its tests--all in one file:

```
import unittest

# Here's our "unit".
def IsOdd(n):
    return n % 2 == 1

# Here's our "unit tests".
class IsOddTests(unittest.TestCase):

    def testOne(self):
        self.failUnless(IsOdd(1))

    def testTwo(self):
        self.failIf(IsOdd(2))

def main():
    unittest.main()

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



Methods whose names start with the string `test` with one argument (`self`) in classes derived from `unittest.TestCase` are test cases. In the above example, `testOne` and `testTwo` are test cases.

Grouping related test cases together, test fixtures are classes that derive from `unittest.TestCase`. In the above example, `IsOddTests` is a test fixture. This is true even though `IsOddTests` derives from a class called `TestCase`, not `TestFixture`. Trust me on this.

Test fixtures can contain `setUp` and `tearDown` methods, which the test runner will call before and after every test case, respectively. Having a `setUp` method is the real justification for fixtures, because it allows us to extract common setup code from multiple test cases into the one `setUp` method.

In Python we typically don't need a `tearDown` method, because we can usually rely on Python's garbage collection facilities to clean up our objects for us. When testing against a database, however, `tearDown` could be useful for closing connections, deleting tables, and so on.

Lights

Throughout this article, I'll use a traffic light to show the state of the tests. Green indicates that the tests pass, and red warns that they fail. A shining yellow light indicates a problem that prevents us from completing a test. TDD practitioners often talk about receiving a "green light" or "green bar" from the graphical test runner that comes with JUnit.

Looking back at our example, the `main` function defined in the `unittest` module makes it possible to execute the tests in the same manner as executing any other script. This function examines `sys.argv`, making it possible to supply command-line arguments to customize the test output or to run only specific fixtures or cases (use `--help` to see the arguments). The default behavior is to run all test cases in all test fixtures found in the file containing the call to `unittest.main`.

Executing the test script above should produce output that resembles:

```
..
```

```
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

If the second test had failed, the output would have looked something like this:

```
.F
======================================================================
FAIL: testTwo (__main__.IsOddTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\jason\projects\tdd-py\test.py", line 14, in testTwo
    self.failIf(IsOdd(2))
  File "C:\Python23\lib\unittest.py", line 274, in failIf
    if expr: raise self.failureException, msg
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

Typically, we wouldn't have the tests and the unit being tested in the same file, but it doesn't hurt to start out that way and then extract the code or the tests later.

Motivation

Guess what I have trouble remembering to do:

```
0 0 * * * [ `date +\%m` -ne `date -d +4days +\%m` ] \
    && mail -s 'Pay the rent!' me-and-my-wife@example.org < /dev/null
```

That little puzzle is a line out of my *crontab* that emails me a reminder to pay the rent on the last four days of each month. Pathetic? Probably. It works, though. I haven't been late paying rent since I started using it.

As clever as I thought I was for coming up with this, it wasn't practical for everything--especially for events that occur only once. Also, there's no way I could teach my wife enough bash scripting techniques in order to add a reminder to our calendar.

Most people use a good old-fashioned wall calendar for this type of thing. That's not techno-geeky enough for me.

I could use Outlook or Evolution or some productivity application, but that would open up a whole new can of worms. We don't use just one computer. We both use multiple computers and operating systems at home and at work. How could we easily synchronize all of those machines?

It was after realizing that our email is available to us no matter where we were that I hit upon the motivation for my project. The email reminding me to pay the rent was with me no matter what machine I'm on because I *always* check my email via IMAP, so my email is accessible from everywhere.

Why not email the upcoming events in my calendar to me just like my reminder to pay the rent? Brilliant, I thought. I know just the tools that can do this, too: the BSD calendar application and the new kid on the block, pal.

My wife and I have a private wiki that we use for keeping track of notes. It's great. Despite the fact that my wife's an accountant and not a geek, she has no trouble using it. I figured we could use the wiki to edit our calendar file. I would write a little cron job to fetch the calendar file--probably using `wget`--from the wiki and pipe that into whatever tool best fit our needs.

Unfortunately, after looking at both `calendar` and `pal`, I discovered that neither was what I was looking for.

The calendar file format requires a <tab> character between dates and descriptions. Since I wanted to use our personal wiki to edit the calendar file, inserting <tab> characters would be an issue (upon hitting <tab>, focus jumps out of the text area to the next form control). `calendar` also doesn't support any of the fancy output options that `pal` does.

The `pal` format was much too geeky for even me to want to use, and it didn't support the one really important use case I had so far: setting a reminder for the last day of the month.

Sample Input

My wife and I sat down and came up with something both of us would want to use. Here are some examples:

```
30 Nov 2004: Dinner with the Darghams.

April 10: Happy Anniversary!

Wednesday: Piano lesson.

Last Thursday: Goody night at book study. Yum.

-1: Pay the rent!
```

Unlike the calendar format, a colon separates dates from descriptions. How Pythonic.

Like the calendar format, omitted certain fields are wildcards. The `April 10` event happens every year. The `Last Thursday` event happens on the final Thursday of every month of every year.

The `-1` event happens on the last day of every month of every year too. I took this idea from Python's array subscript syntax, where `foo[-1]` selects the last element in the foo array. I thought it was a little geeky, but my wife understood it right away.

My goal is to write a small application that can run from cron to read a file in this format and email my wife and me the events we have scheduled for the next seven days. That shouldn't be too hard, should it?

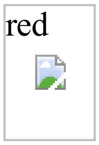
From this point on, I'm writing this article in real time, having contrived nothing. I didn't write the code first and then write the article--I'm writing the article as I write the code. Yes, I expect to make mistakes. In fact, I'm counting on it. Making mistakes is the *best* way to learn.

Getting Started

Being test infected means that I must write this tool by writing all of my unit tests *before* writing the code I expect the tests to exercise.

The first thing I do when starting a new project is to create an empty fixture that fails:

```
import unittest

class FooTests(unittest.TestCase):

    def testFoo(self):
        self.failUnless(False)

def main():
    unittest.main()

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



I do this out of habit, just to make sure I have everything typed in correctly and to test that the test runner can find the fixture.

Notice the class named `FooTests` and its `testFoo` method. At this point I have no idea what I'm going to test first. I just want to make sure that I have everything ready once things get going.

Let's start out easy and test the first example from above with the full day, month, and year specified for the event. In order to create this test, I need know *what* to test. Am I testing a class? A function?

This is where we put on our designer hats for a brief moment and try to use our experience and intuition to come up with some piece to the puzzle that will help us reach our goal. It's OK if we make a mistake here; the tests will reveal that right away, before we invest too much in this design. We certainly don't want to draft any documents filled with diagrams. Save those for later, after we have a clue about what will actually work.

For this project, I should probably create objects that can say whether they "match" a given date. These objects will act as a "pattern" for dates. (I'm using regular expressions as a metaphor here.)

Eventually, I'll have to write a parser that will read in a file and create these pattern objects, but I'll do that later. These pattern objects are probably an easier place to start.

There might be multiple types of patterns--but I won't think about that now, because I could be wrong. Instead, I'll start coding so I can let it tell me what it wants to become:

```
def testMatches(self):
    p = DatePattern(2004, 9, 28)
    d = datetime.date(2004, 9, 28)
    self.failUnless(p.matches(d))
```

red

Notice that I changed the name of the method from `testFoo` to something more appropriate, because I now have an idea about what to test. I've also invented a class name, `DatePattern`, and a method name, `matches`. (The `datetime` module is part of Python 2.3 and up--I had to import it at the top of my file in order to use it.)

This test, of course, fails miserably--the `DatePattern` class doesn't even exist yet! But I at least know now the name of the class I need to implement. I also know the name and signature of one of its methods and the signature for its `__init__` method. Here's what I can do with this knowledge:

```
class DatePattern:

    def __init__(self, year, month, day):
        pass

    def matches(self, date):
        return True
```

green

Now the test passes! It's time to move on to the next test.

You probably think I'm joking, don't you? I'm not.

Baby Steps

Test-driven development is best when you move in the smallest possible increments. You should only be writing code that makes the current failing test case(s) pass. Once the tests pass, you're done writing code. Stop!

The above code is worthless, right? It basically says that *every* pattern matches *every* date. How can I justify spending the time to come up with a "real" implementation? By adding another test:

```
def testMatchesFalse(self):
    p = DatePattern(2004, 9, 28)
    d = datetime.date(2004, 9, 29)
    self.failIf(p.matches(d))
```

red

We now have one passing test and one failing test.

I could change the `matches` method to return `False` in order to make this new test case pass, but that would break the old one! I now have no choice but to implement `DatePattern` correctly so that both tests can pass. Here's what I came up with:

```
class DatePattern:

    def __init__(self, year, month, day):
        self.date = datetime.date(year, month, day)

    def matches(self, date):
        return self.date == date
```

green

Both tests now pass. Woo-hoo! I'm not happy with the `DatePattern` class, though. So far, it's nothing more than a simple wrapper around Python's `date` class. Why am I not just using `date` instances for my "patterns"?

It might turn out that the `DatePattern` class is unnecessary, but I'm not going to make that decision on my own. Instead, I'm going to write another test--one that I *think* will confirm the necessity of the `DatePattern` class:

```
def testMatchesYearAsWildCard(self):
    p = DatePattern(0, 4, 10)
    d = datetime.date(2005, 4, 10)
    self.failUnless(p.matches(d))
```

red

Voilà! This test fails!

Why am I so happy about a failing test? My reasoning is simple: this *proves* that the current implementation of `DatePattern` is insufficient. It *can't* be just a simple wrapper around `date` and therefore can't be just a `date`.

While typing this test, I had to make a decision about how to represent wildcards. What occurred to me first was to use `0`. After all, there's no year 0 (contrary to popular belief), month 0, or day 0. This may not have been the best choice, but I'm going to roll with it for now.

It's time to make the new test pass (while making sure not to break the old ones):

```
class DatePattern:

    def __init__(self, year, month, day):
        self.year  = year
        self.month = month
        self.day   = day

    def matches(self, date):
        return ((self.year and self.year == date.year or True) and
                self.month == date.month and
                self.day   == date.day)
```

green

To be honest, I'm already starting to feel like I'll need to do some refactoring as I add more wildcard functionality to the class, but I want to write a few more tests first.

Let's add a test where the month is a wildcard:

```
def testMatchesYearAndMonthAsWildCards(self):
    p = DatePattern(0, 0, 1)
    d = datetime.date(2004, 10, 1)
    self.failUnless(p.matches(d))
```

red

Fixing `matches` so that the test passes results in this:

```
def matches(self, date):
    return ((self.year  and self.year  == date.year  or True) and
            (self.month and self.month == date.month or True) and
             self.day == date.day)
```

green

This method is getting uglier every time we touch it--I'm now positive that it will be my first refactoring victim.

I now have a test for using wildcards for both years and months. Will I need one for days? A pattern containing nothing but wildcards would match every day. When would that be useful?

At this point I can't think of a reason to support wildcard days, so I won't bother writing a test for it. Because of that, I also won't bother implementing any code to support it in the `DatePattern` class. Remember, code gets written only when there's a failing test that needs the new code in order to pass. This prevents us from writing code that should not exist in our application, which should help keep it from becoming unnecessarily complex.

Let's move on. We need to support events that occur on a specified day of every week:

```
def testMatchesWeekday(self):
    p = DatePattern(
```

Uh, what now?

At this point, I realized that the `DatePattern` class might not be what I want to use for this test. Its `__init__` method doesn't accept a weekday. Should I use a different class, or modify the existing one?

I decided to modify the existing one for now, as that will require the least amount of work. If this turns out to be a bad idea, I can always refactor later.

```
def testMatchesWeekday(self):
    p = DatePattern(0, 0, 0, 2) # 2 is Wednesday
    d = datetime.date(2004, 9, 29)
    self.failUnless(p.matches(d))
```

red

This doesn't pass because `DatePattern.__init__` doesn't accept five arguments (counting `self`). I modified `__init__` to look like this:

```
def __init__(self, year, month, day, weekday=0):
    self.year    = year
    self.month   = month
    self.day     = day
    self.weekday = weekday
```

red

I gave `weekday` a default value so that I wouldn't need to update the other test cases. Everything compiles and runs, but the new test case doesn't pass.

The astute reader has probably already realized that I'm now passing in `0` for the `day` argument. There's the wildcard I didn't think I would need--now I need it!

Here's my new `matches` method:

```
def matches(self, date):
    return ((self.year and self.year == date.year or True) and
            (self.month and self.month == date.month or True) and
            (self.day and self.day == date.day or True) and
            (self.weekday and self.weekday == date.weekday() or True))
```

red

Now *all* of the components of a pattern allow for wildcards. How very interesting.

With this new method, `testMatchesWeekday` passes but `testMatchesFalse` now fails! What gives?

Refactoring

I honestly can't tell why `testMatchesFalse` fails by looking at the code. This is going to call for some simple debugging. Unfortunately, I tried to cram all of the logic for the `matches` method into one expression (spanning four lines!), so there's no place for me to insert any print statements to help me see which part is failing. It's finally time to do that refactoring I've been wanting to do.

The refactoring I want to apply is the Compose Method from Joshua Kerievsky's excellent book, Refactoring to Patterns. By extracting smaller methods from the current `matches` method, I can not only make `matches` clearer but also make it possible to debug whichever part is currently causing me grief.

This is the result:

```
def matches(self, date):
    return (self.yearMatches(date) and
            self.monthMatches(date) and
            self.dayMatches(date) and
            self.weekdayMatches(date))

def yearMatches(self, date):
    if not self.year: return True
    return self.year == date.year

def monthMatches(self, date):
    if not self.month: return True
    return self.month == date.month

def dayMatches(self, date):
    if not self.day: return True
    return self.day == date.day

def weekdayMatches(self, date):
    if not self.weekday: return True
    return self.weekday == date.weekday()
```

green

The `matches` method is now much clearer, don't you agree? It might seem like a ridiculous thing to do, but writing intention-revealing code is much more important than being clever. I was trying to be too clever before and it caused a bug--one that I wouldn't have come across if I had done this from the beginning.

After applying this refactoring and rerunning the tests, I expected to see the `testMatchesFalse` test still failing, but it's now passing. Somewhere in my original logic I made an error, and I have no idea where it was--I'll leave finding it as an exercise for the reader. In the meantime, not only do I have simpler code now but it also actually works the way I expect it to. Take that!

Would I have noticed this bug without tests? I have no doubt that I would, but how long would it have been before I realized that this was a problem? With my unit tests, I noticed it immediately, so I knew exactly what to fix.

Code Pickiness

I recently read a weblog post by Ian Bicking about what he considers to be code smells in Python code. *(Editors note: The link to the weblog post by Ian Bicking was not available at the time of publishing.)* I thought one, using "`bool and true_value or false_value` to simulate `bool ? true_value : false_value`", was odd because I was rather fond of that particular idiom. Upon seeing that in my code, Ian would have thought I was lazy. I now realize he was right. It's too bad that I had to learn that with all of you watching.

Wildcards essentially work for all of the components I'm testing so far. This is good, but I think the next test will cause trouble. It starts out innocently enough:

```
def testMatchesLastWeekday(self):
    p = DatePattern(0, 0, 0, 3
```

Er, I'm stuck again.

In case it's not obvious (and it's not--why didn't Python's `datetime` module define constants for weekdays?), the `3` represents Thursday.

How do I indicate that I only want to match the *last* Thursday in a month? Do I need to add yet another argument to `DatePattern.__init__`?

This is where that sneaking suspicion in the back of my head is finally starting to warrant some closer attention. I might be trying to cram too much functionality into one class.

Conclusion

I haven't written much code yet, but that's a good thing, since it seems that the code I have written might not have been sufficient for what I want to do with it. Without the tests, I might not have discovered what a mess I was writing until it was too late. At this point, I haven't invested too much time into the `DatePattern` class, so I won't feel bad about throwing it away if that's what I'll need to do.

I have some ideas about how to restructure the code so that it's as simple and yet as functional as I want it to be, but we're going to have to save those for Part 2 of this article, which will be published shortly.

Code and tests are available for download and inspection.

Jason Diamond is a consultant specializing in C++, C#, and XML, and is located in sunny Southern California.

Return to the Python DevCenter.