Quick Testing

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

A small confession: when writing code, I don't usually write tests first. There, I've said it. Hate me. I suspect I'm not alone among Python developers. Yes, yes, testing is important, and for my major projects, tests still get written. However, for a lot of small things like little scripts, utilities, and personal projects, I just don't bother because I don't want to think about all of the extra steps and tooling that's usually involved. However, a recent conference experience may have changed some of my views. In this installment, I discuss a more lightweight approach to testing along with a brief introduction to some third-party testing libraries, including pytest [1] and Hypothesis [2].

A Revelation

Early this summer, I attended a talk by Aur Saraf at PyCon Israel in which he live-coded a simple interpreter from scratch in about 25 minutes [3]. Live coding in front of an audience is always a dicey affair, but what struck me about this particular talk is the fact that it was done entirely in a test-driven development style with no connection to any sort of testing tools, third-party libraries, or even standard library modules. I was both stunned and amazed.

The gist of the idea is simple. If you're going to write a function, you might as well first write an assertion or two for it. For example, suppose you were writing a function to split a URL into parts. You might start by writing this:

```
def split_url(url):
    pass

assert split_url('http://www.python.org') == ('http', 'www.python.org')
```

The `assert` statement serves as a kind of expectation for what you want to happen. Naturally, the code is going to fail immediately as you haven't actually written the function. However, the assertion gives you a target to aim for. So your next step is to implement the function and make the assertion pass.

```
def split_url(url):
    parts = url.split('://')
    return (parts[0], parts[1])

assert split_url('http://www.python.org') == ('http', 'www.python.org')
```

It passes. Very good. At first glance, this might seem too minimal and maybe even a bit crazy. However, there's a certain genius to it. First, it doesn't require any special knowledge of libraries or tools (e.g., the `unittest` standard library module): `assert` is a built-in statement of the core Python language. There are also no separate files to maintain or extra functions to write—the `assert` is just inlined right there in the code. It executes right after the function is defined. This means that the code won't even run or import unless the test passes. Thus, if you're working on some new thing and changing your code a lot, it can be useful to just leave it in there for the time being. It's a minimal test that doesn't require too much thought and doesn't really interfere with what you're doing.

Getting back to Aur's talk for a moment, he proceeded to write his entire interpreter in this style. Assertions first and then functions. As I watched, I kept thinking, "I bet I could use something like this." I also recognized that it could be a useful stepping stone to other more advanced testing tools. So let's explore that further.

Putting It into Practice

In one of my current projects, I'm faced with the problem of implementing a priority queue. A standard technique for creating such a queue is to use a heap data structure. In fact, Python provides a `heapq` standard library module that can be used to do it. However, my specific problem has the extra requirement of supporting cancellation (i.e., the ability to remove/cancel items anywhere in the queue). Sadly, the standard `heapq` module has no support for that. In fact, efficiently removing items from a priority queue is a rather tricky algorithmic problem. Thus, it seems that I'm probably going to have to roll my own class for it.

Let's start by sketching out a class:

```
class PriQueue:
    def __init__(self):
        pass

    def push(self, item):
        pass

    def pop(self):
        pass

    def remove(self, item):
        pass
```

It does nothing, but let's write some assertions that encode our expectations of how it should work:

```
class PriQueue:
    ...

# Test code (put right after the class)
q = PriQueue()
q.push(4)
q.push(3)
q.push(7)
q.push(10)
q.remove(4)
# Popping all items produces them in order
assert [ q.pop() for _ in range(3) ] == [ 3, 7, 10 ]
```

Running this code, it will fail because we haven't implemented anything. However, we can now fill in some details of the implementation:

```
# pqueue.py

import heapq

class PriQueue:
    def __init__(self):
        self.heap = []

    def push(self, item):
        heapq.heappush(self.heap, item)

    def pop(self):
        return heapq.heappop(self.heap)

    def remove(self, item):
        self.heap.remove(item)

q = PriQueue()
q.push(4)
q.push(3)
q.push(7)
q.push(10)
q.remove(4)
assert [ q.pop() for _ in range(3) ] == [3, 7, 10]
```

If you run this code, it passes its simple test and we're on our way.

From Asserts to Functions

Having assertions placed in the code is really only a starting point. As the code evolves, you can move the test into a more proper function. For example, maybe you do this:

```
# pqueue.py
...

def test_priqueue():
    q = PriQueue()
    q.push(4)
    q.push(3)
    q.push(7)
    q.push(10)
    q.remove(4)
    assert [ q.pop() for _ in range(3) ] == [3, 7, 10]

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

Writing a function is an easy step—you don't even have to change your testing code (well, other than indenting it). However, if you do this, you'll open the doors to incorporating your tests with other testing tools.

For example, this code can be executed under a testing tool like pytest [1]. One of the nice things about pytest is that it works using standard Python `assert` statements. Assuming that you have it installed, drop into the shell and type this:

```
bash $ python3 -m pytest pqueue.py
================== test session starts ==================
platform darwin -- Python 3.6.1, pytest-3.0.2, py-1.4.31,
pluggy-0.3.1
rootdir: /Users/beazley/Desktop/UsenixLogin/beazley_fall_17,
inifile:
plugins: hypothesis-3.11.6
collected 1 items

pqueue.py .

=============== 1 passed in 0.00 seconds ===============
```

Excellent. Keep in mind it didn't take much to get here. No special imports or fooling around with the `unittest` module—just a function with an `assert` in it. Later on, you could move the testing function over to a more dedicated testing file. For now, it's fine where it is. After all, we're still working.

From a Function to Hypothesis

One of the problems with our code is that the test is fairly minimal. It tests just one case. How are we to know if our queue code actually works as intended across all inputs? We could generate more test cases by hand, but doing so is going to be rather painful and error-prone if it involves a bunch of cut-and-paste.

To better handle this, let's change our testing function so that it is parameterized with some inputs:

```
def test_priqueue(items, remove_item):
    q = PriQueue()
    for item in items:
        q.push(item)

    # Remove the given item
    q.remove(remove_item)
    items.remove(remove_item)

    # Verify that items come out in the proper order
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
```

This change allows us to feed different inputs into the function. For example, we can do this:

```
...
if __name__ == '__main__':
    test_priqueue([4,3,7,10], 4)
    test_priqueue([9,2,1,8,5], 2)
    test_priqueue([4,1,6], 1)
```

Running this, you'll find that the code still seems to pass for those three test cases. Our confidence is building. However, how do we really know that we've covered all of our bases? It's hard to say.

One of the more interesting tools on the Python testing front is Hypothesis [2]. In a nutshell, Hypothesis can randomly generate test cases for you as long as you are able to describe the parameters to the test. Take the above test function exactly as you've written it and decorate it as follows:

```
# pqueue.py

...

from hypothesis import given
from hypothesis.strategies import lists, integers

@given(lists(integers(min_value=0, max_value=9),
             unique=True, min_size=10, max_size=10),
       integers(min_value=0, max_value=9))
def test_priqueue(items, remove_item):
    q = PriQueue()
    for item in items:
        q.push(item)

    # Remove the given item
    q.remove(remove_item)
    items.remove(remove_item)

    # Verify that items come out in the proper order
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)

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

At first glance, this looks a bit scary, but the `@given` decorator is used to describe the arguments to the `test_priqueue()` function. In this case, the first argument (`items`) is going to be a 10-element list of unique integers with values in the range 0 to 9. The second argument (`remove_item`) is an integer with a value in the range 0 to 9.

Running the new code, you'll now find that it fails. Your output might vary from this, but it will look roughly like this:

```
$ python3 pqueue.py
Falsifying example: test_priqueue(items=[1, 2, 3, 4, 0, 5, 6, 7,
8, 9], remove_item=0)
Traceback (most recent call last):
  File "pqueue.py", line 35, in &lt;module&gt;
    test_priqueue()
  ...
  File "pqueue.py", line 32, in test_priqueue
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
AssertionError
```

What's happened here is that Hypothesis has automatically found a test-case that fails and is reporting it. To better see what happens, put a print statement in your test code:

```
# pqueue.py

...
@given(lists(integers(min_value=0, max_value=9),
             unique=True, min_size=10, max_size=10),
       integers(min_value=0, max_value=9))
def test_priqueue(items, remove_item):
    print('TRYING:', items, remove_item)
    q = PriQueue()
    for item in items:
        q.push(item)

    # Remove the given item
    q.remove(remove_item)
    items.remove(remove_item)

    # Verify that items come out in the proper order
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
```

Now, let's clear the environment and try running again:

```
bash $ rm -rf .hypothesis
bash $ python3 pqueue.py
TRYING: [3, 0, 1, 9, 8, 6, 4, 5, 2, 7] 5
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 7, 1, 0, 2, 5] 1
TRYING: [9, 6, 4, 8, 3, 7, 1, 0, 2, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 7, 0, 2, 1, 5] 1
TRYING: [9, 6, 4, 8, 3, 2, 0, 7, 1, 5] 1
...
TRYING: [0, 3, 4, 6, 1, 2, 8, 5, 7, 9] 0
Falsifying example: test_priqueue(items=[0, 3, 4, 6, 1, 8, 2, 5,
7, 9], remove_item=0)
TRYING: [0, 3, 4, 6, 1, 8, 2, 5, 7, 9] 0
Traceback (most recent call last):
  File "pqueue.py", line 36, in &lt;module&gt;
    test_priqueue()
  ...
  File "pqueue.py", line 33, in test_priqueue
    assert [ q.pop() for _ in range(len(items)) ] == sorted(items)
AssertionError
```

In this case, you'll see the test function invoked repeatedly with all sorts of inputs. Basically, Hypothesis is trying random inputs searching for a failure. Since our code is buggy, it will eventually find one although it might take some searching. That's pretty neat. It found a bad test case, and I really didn't have to do much work. Our testing code is still pretty small—just a single function.

Fixing the Bug

In the case of my example, there is a bug in item removal. When the item is removed, the underlying heap structure is not preserved properly. This can be fixed with a minor change.

```
# pqueue.py

import heapq

class PriQueue:
    def __init__(self):
        self.heap = []

    def push(self, item):
        heapq.heappush(self.heap, item)

    def pop(self):
        return heapq.heappop(self.heap)

    def remove(self, item):
        self.heap.remove(item)
        heapq.heapify(self.heap)    # <- Add this line
...
```

If you run the program again, you'll see Hypothesis fire 200 random inputs at the `test_priqueue()` function, but they'll all pass. In fact, each time you run the program, you'll get a different set of inputs as it searches for failing test cases. Should a failure be found, it will be recorded for inclusion in further tests. For now, we're safe though.

Final Thoughts

This whole approach to testing out new code and small libraries is interesting. When starting out, the inlined assertions provide a basic level of testing for implementing the initial code. Those tests can naturally evolve into a testing function that can be used with popular testing tools like pytest. Later, you can evolve that testing function into something for use with a package like Hypothesis, where hundreds of test cases can be generated for you automatically. The code is still small and it's allowing me to focus on the actual problem I'm trying to solve. For example, with just that one testing function, I can start experimenting with different queue implementations and have a reasonable expectation of finding bugs if I break anything. It's neat.

References

[1] pytest: http://pytest.org.

[2] Hypothesis: http://hypothesis.works.

[3] Aur Saraf, at PyCon, Israel: http://il.pycon.org/wwwpyconIL/agenda/174.