Writing Tests For Your Python Programs

> Much of the essence of building a program is in fact the debugging of the specification. – Fred Brooks

Let's start with a simple example of a `hello` program that should say "Hello, name!"

```
$ ./hello.py
Usage: hello.py NAME [NAME...]
$ ./hello.py Jan
Hello, Jan!
$ ./hello.py Bobby Peter Greg
Hello, Bobby!
Hello, Peter!
Hello, Greg!
```

Here is one way to write such a program *with an embedded test*:

```
 1  #!/usr/bin/env python3
 2
 3  import os
 4  import sys
 5
 6  def hello(name):
 7      return 'Hello, {}!'.format(name)
 8
 9  def test_hello():
10      assert hello('World') == 'Hello, World!'
11      assert hello('') == 'Hello, !'
12      assert hello('my name is Fred') == 'Hello, my name is Fred!'
13
14  def main():
15      args = sys.argv[1:]
16      if not args:
17          print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
18          sys.exit(1)
19
20      for arg in args:
21          print(hello(arg))
22
23  if __name__ == '__main__':
24      main()
```

Specifically I've written this to use the PyTest (https://docs.pytest.org/en/latest/) framework that will search for function names starting with `test_` and will run them.

```
$ pytest -v hello.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hello.py::test_hello PASSED                                              [100%]

=========================== 1 passed in 0.03 seconds ===========================
```

The `assert` function you see in the `test_hello` function is a built-in Python function that evaluates some predicate and will throw an error if the predicate is false. For instance, we `assert` that `hello("World")` should return the string `Hello, World!`. If this does not happen, the test will fail:

```
>>> def hello(name):
...     return 'Hello, {}!'.format(name)
...
>>> assert hello('World') == 'foo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

PyTest will find such errors and report them as failed tests:

```
$ cat -n hello_bad.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  def hello(name):
     7      return 'Hello, {}!'.format(name)
     8
     9  def test_hello():
    10      assert hello('World') == 'Hello, World.'
    11
    12  def main():
    13      args = sys.argv[1:]
    14      if not args:
    15          print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
    16          sys.exit(1)
    17
    18      for arg in args:
    19          print(hello(arg))
    20
```

```
    21  if __name__ == '__main__':
    22      main()
$ pytest -v hello_bad.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hello_bad.py::test_hello FAILED                                          [100%]

=================================== FAILURES ===================================
_________________________________ test_hello __________________________________

    def test_hello():
>       assert hello('World') == 'Hello, World.'
E       AssertionError: assert 'Hello, World!' == 'Hello, World.'
E         - Hello, World!
E         ?             ^
E         + Hello, World.
E         ?             ^

hello_bad.py:10: AssertionError
========================== 1 failed in 0.08 seconds ============================
```

The error output highlights the differences between what was expected (`Hello, World.` ending in a period) and what the `hello` function actually returned (`Hello, World!` ending in an exclamation point).

I would recommend writing your tests for every function directly below the function being tested and calling the test `test_function`. Try to make a function do just one thing, then write tests to ensure it does that thing. Try to write tests that probe the edge cases, e.g., passing an empty string or a very long string. Here's a version where the `hello` function will only greet if the argument is a `str`; otherwise it will return an admonishment. This is an extremely contrived example because everything coming in via `sys.argv` is by definition a string, so I will intentionally convert anything that looks like a digit to an `int` so that we can see the error:

```
$ cat -n hello_fail.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  def hello(name):
```

```
     7      if type(name) is str:
     8          return 'Hello, {}!'.format(name)
     9      else:
    10          return 'Can only say hello to a string'
    11
    12
    13  def test_hello():
    14      assert hello('World') == 'Hello, World!'
    15      assert hello('') == 'Hello, !'
    16      assert hello('my name is Fred') == 'Hello, my name is Fred!'
    17
    18      err = 'Can only say hello to a string'
    19      assert hello(4) == err
    20      assert hello(None) == err
    21      assert hello(float) == err
    22      assert hello(str) == err
    23
    24  def main():
    25      args = sys.argv[1:]
    26      if not args:
    27          print('Usage: {} NAME [NAME...]'.format(os.path.basename(sys.argv[0])))
    28          sys.exit(1)
    29
    30      for arg in args:
    31          if arg.isdigit(): arg = int(arg)
    32
    33          print(hello(arg))
    34
    35  if __name__ == '__main__':
    36      main()
$ ./hello_fail.py Bob 3 Sue
Hello, Bob!
Can only say hello to a string
Hello, Sue!
$ pytest -v hello_fail.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 1 item

hello_fail.py::test_hello PASSED                                         [100%]

=========================== 1 passed in 0.04 seconds ===========================
```

These types of tests that live *inside* each of your source files and test invididual functions are known as "unit tests". As your software grows, you may find yourself breaking your functions into logically grouped files or modules. We can also write tests that live *outside* our program files to ensure the proper integration of modules as well as the user interface we present. All the `test.py` programs that have been included in your assignments have these types of tests – ensuring, for instance, that your program will create a "usage" statement if passed no arguments or `-h|--help`, will print message to STDERR and `sys.exit()` with a non-zero value when there is an error, or will run to completion given good input and produce the expected STDOUT and/or output files.

Here is a `test.py` that tests for the usage statement and error code on no input and then tests for one, argument, more than one argument, and an argument that is more than one word:

```
$ cat -n test.py
     1  #!/usr/bin/env python3
     2
     3  from subprocess import getstatusoutput
     4
     5  prg = './hello.py'
     6
     7  def test_usage():
     8      rv, out = getstatusoutput('{}'.format(prg))
     9      assert rv != 0
    10      assert out.lower().startswith('usage')
    11
    12  def test_runs_ok():
    13      rv1, out1 = getstatusoutput('{} Carl'.format(prg))
    14      assert rv1 == 0
    15      assert out1 == 'Hello, Carl!'
    16
    17      rv2, out2 = getstatusoutput('{} Barbara McClintock'.format(prg))
    18      assert rv2 == 0
    19      assert out2 == 'Hello, Barbara!\nHello, McClintock!'
    20
    21      rv3, out3 = getstatusoutput('{} "Barbara McClintock"'.format(prg))
    22      assert rv3 == 0
    23      assert out3 == 'Hello, Barbara McClintock!'
```

I typically create a `Makefile` with a `test` target to show users how to run the tests:

```
$ cat -n Makefile
     1  .PHONY: test
     2
     3  test:
     4      pytest -v test.py
```

```
$ make test
pytest -v test.py
============================= test session starts ==============================
platform darwin -- Python 3.6.8, pytest-4.2.0, py-1.7.0, pluggy-0.8.1 -- /anaconda3/bin/pyth
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/biosys-analytics/lectures/18-writing-tests/examples, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.2.0, arraydiff-0.3
collected 2 items

test.py::test_usage PASSED                                               [ 50%]
test.py::test_runs_ok PASSED                                             [100%]

=========================== 2 passed in 0.19 seconds ===========================
```