doctest – Testing through documentation

Purpose:	Write automated tests as part of the documentation for a module.
Available In:	2.1

`doctest` lets you test your code by running examples embedded in the documentation and verifying that they produce the expected results. It works by parsing the help text to find examples, running them, then comparing the output text against the expected value. Many developers find `doctest` easier than `unittest` because in its simplest form, there is no API to learn before using it. However, as the examples become more complex the lack of fixture management can make writing `doctest` tests more cumbersome than using `unittest`.

Getting Started

The first step to setting up doctests is to use the interactive interpreter to create examples and then copy and paste them into the docstrings in your module. Here, **`my_function()`** has two examples given:

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

To run the tests, use `doctest` as the main program via the `-m` option to the interpreter. Usually no output is produced while the tests are running, so the example below includes the `-v` option to make the output more verbose.

```
$ python -m doctest -v doctest_simple.py

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple
1 items passed all tests:
   2 tests in doctest_simple.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Examples cannot usually stand on their own as explanations of a function, so `doctest` also lets you keep the surrounding text you would normally include in the documentation. It looks for lines beginning with the interpreter

prompt, `>>>`, to find the beginning of a test case. The case is ended by a blank line, or by the next interpreter prompt. Intervening text is ignored, and can have any format as long as it does not look like a test case.

```
def my_function(a, b):
    """Returns a * b.

    Works with numbers:

    >>> my_function(2, 3)
    6

    and strings:

    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

The surrounding text in the updated docstring makes it more useful to a human reader, and is ignored by **doctest**, and the results are the same.

```
$ python -m doctest -v doctest_simple_with_docs.py

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple_with_docs
1 items passed all tests:
   2 tests in doctest_simple_with_docs.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Handling Unpredictable Output

There are other cases where the exact output may not be predictable, but should still be testable. Local date and time values and object ids change on every test run. The default precision used in the representation of floating point values depend on compiler options. Object string representations may not be deterministic. Although these conditions are outside of your control, there are techniques for dealing with them.

For example, in CPython, object identifiers are based on the memory address of the data structure holding the object.

```
class MyClass(object):
    pass

def unpredictable(obj):
    """Returns a new list containing obj.

    >>> unpredictable(MyClass())
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
    """
    return [obj]
```

These id values change each time a program runs, because it is loaded into a different part of memory.

```
$ python -m doctest -v doctest_unpredictable.py

Trying:
    unpredictable(MyClass())
Expecting:
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
**********************************************************************
File "doctest_unpredictable.py", line 16, in doctest_unpredictable.unpredictable
Failed example:
    unpredictable(MyClass())
Expected:
    [<doctest_unpredictable.MyClass object at 0x10055a2d0>]
Got:
    [<doctest_unpredictable.MyClass object at 0x10051df90>]
2 items had no tests:
    doctest_unpredictable
    doctest_unpredictable.MyClass
**********************************************************************
1 items had failures:
   1 of   1 in doctest_unpredictable.unpredictable
1 tests in 3 items.
0 passed and 1 failed.
***Test Failed*** 1 failures.
```

When the tests include values that are likely to change in unpredictable ways, and where the actual value is not important to the test results, you can use the `ELLIPSIS` option to tell `doctest` to ignore portions of the verification value.

```
class MyClass(object):
    pass

def unpredictable(obj):
    """Returns a new list containing obj.

    >>> unpredictable(MyClass()) #doctest: +ELLIPSIS
    [<doctest_ellipsis.MyClass object at 0x...>]
    """
    return [obj]
```

The comment after the call to **`unpredictable()`** (`#doctest: +ELLIPSIS`) tells `doctest` to turn on the `ELLIPSIS` option for that test. The `...` replaces the memory address in the object id, so that portion of the expected value is ignored and the actual output matches and the test passes.

```
$ python -m doctest -v doctest_ellipsis.py

Trying:
    unpredictable(MyClass()) #doctest: +ELLIPSIS
Expecting:
    [<doctest_ellipsis.MyClass object at 0x...>]
ok
2 items had no tests:
    doctest_ellipsis
    doctest_ellipsis.MyClass
1 items passed all tests:
   1 tests in doctest_ellipsis.unpredictable
1 tests in 3 items.
1 passed and 0 failed.
Test passed.
```

There are cases where you cannot ignore the unpredictable value, because that would obviate the test. For example, simple tests quickly become more complex when dealing with data types whose string representations are inconsistent. The string form of a dictionary, for example, may change based on the order the keys are added.

```
keys = [ 'a', 'aa', 'aaa' ]

d1 = dict( (k,len(k)) for k in keys )
```

```
d2 = dict( (k,len(k)) for k in reversed(keys) )

print
print 'd1:', d1
print 'd2:', d2
print 'd1 == d2:', d1 == d2

s1 = set(keys)
s2 = set(reversed(keys))

print
print 's1:', s1
print 's2:', s2
print 's1 == s2:', s1 == s2
```

Because of cache collision, the internal key list order is different for the two dictionaries, even though they contain the same values and are considered to be equal. Sets use the same hashing algorithm, and exhibit the same behavior.

```
$ python doctest_hashed_values.py


d1: {'a': 1, 'aa': 2, 'aaa': 3}
d2: {'aa': 2, 'a': 1, 'aaa': 3}
d1 == d2: True

s1: set(['a', 'aa', 'aaa'])
s2: set(['aa', 'a', 'aaa'])
s1 == s2: True
```

The best way to deal with these potential discrepancies is to create tests that produce values that are not likely to change. In the case of dictionaries and sets, that might mean looking for specific keys individually, generating a sorted list of the contents of the data structure, or comparing against a literal value for equality instead of depending on the string representation.

```
def group_by_length(words):
    """Returns a dictionary grouping words into sets by length.

    >>> grouped = group_by_length([ 'python', 'module', 'of', 'the', 'week' ])
    >>> grouped == { 2:set(['of']),
    ...              3:set(['the']),
    ...              4:set(['week']),
    ...              6:set(['python', 'module']),
    ...              }
    True

    """
    d = {}
    for word in words:
        s = d.setdefault(len(word), set())
        s.add(word)
    return d
```

Notice that the single example is actually interpreted as two separate tests, with the first expecting no console output and the second expecting the boolean result of the comparison operation.

```
$ python -m doctest -v doctest_hashed_values_tests.py

Trying:
    grouped = group_by_length([ 'python', 'module', 'of', 'the', 'week' ])
Expecting nothing
ok
Trying:
    grouped == { 2:set(['of']),
                 3:set(['the']),
                 4:set(['week']),
                 6:set(['python', 'module']),
```

```
                }
Expecting:
    True
ok
1 items had no tests:
    doctest_hashed_values_tests
1 items passed all tests:
   2 tests in doctest_hashed_values_tests.group_by_length
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Tracebacks

Tracebacks are a special case of changing data. Since the paths in a traceback depend on the location where a module is installed on the filesystem on a given system, it would be impossible to write portable tests if they were treated the same as other output.

```
def this_raises():
    """This function always raises an exception.

    >>> this_raises()
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/no/such/path/doctest_tracebacks.py", line 14, in this_raises
        raise RuntimeError('here is the error')
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')
```

`doctest` makes a special effort to recognize tracebacks, and ignore the parts that might change from system to system.

```
$ python -m doctest -v doctest_tracebacks.py

Trying:
    this_raises()
Expecting:
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/no/such/path/doctest_tracebacks.py", line 14, in this_raises
        raise RuntimeError('here is the error')
    RuntimeError: here is the error
ok
1 items had no tests:
    doctest_tracebacks
1 items passed all tests:
   1 tests in doctest_tracebacks.this_raises
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

In fact, the entire body of the traceback is ignored and can be omitted.

```
def this_raises():
    """This function always raises an exception.

    >>> this_raises()
    Traceback (most recent call last):
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')
```

When `doctest` sees a traceback header line (either `Traceback (most recent call last):` or `Traceback (innermost last):`, depending on the version of Python you are running), it skips ahead to find the exception type and message, ignoring the intervening lines entirely.

```
$ python -m doctest -v doctest_tracebacks_no_body.py

Trying:
    this_raises()
Expecting:
    Traceback (most recent call last):
    RuntimeError: here is the error
ok
1 items had no tests:
    doctest_tracebacks_no_body
1 items passed all tests:
   1 tests in doctest_tracebacks_no_body.this_raises
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Working Around Whitespace

In real world applications, output usually includes whitespace such as blank lines, tabs, and extra spacing to make it more readable. Blank lines, in particular, cause issues with **doctest** because they are used to delimit tests.

```
def double_space(lines):
    """Prints a list of lines double-spaced.

    >>> double_space(['Line one.', 'Line two.'])
    Line one.

    Line two.

    """
    for l in lines:
        print l
        print
    return
```

double_space() takes a list of input lines, and prints them double-spaced with blank lines between.

```
$ python -m doctest doctest_blankline_fail.py

**********************************************************************
File "doctest_blankline_fail.py", line 13, in doctest_blankline_fail.double_space
Failed example:
    double_space(['Line one.', 'Line two.'])
Expected:
    Line one.
Got:
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
**********************************************************************
1 items had failures:
   1 of   1 in doctest_blankline_fail.double_space
***Test Failed*** 1 failures.
```

The test fails, because it interprets the blank line after `Line one.` in the docstring as the end of the sample output. To match the blank lines, replace them in the sample input with the string `<BLANKLINE>`.

```
def double_space(lines):
    """Prints a list of lines double-spaced.

    >>> double_space(['Line one.', 'Line two.'])
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
    """
```

```
    for l in lines:
        print l
        print
    return
```

`doctest` replaces actual blank lines with the same literal before performing the comparison, so now the actual and expected values match and the test passes.

```
$ python -m doctest -v doctest_blankline.py

Trying:
    double_space(['Line one.', 'Line two.'])
Expecting:
    Line one.
    <BLANKLINE>
    Line two.
    <BLANKLINE>
ok
1 items had no tests:
    doctest_blankline
1 items passed all tests:
   1 tests in doctest_blankline.double_space
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Another pitfall of using text comparisons for tests is that embedded whitespace can also cause tricky problems with tests. This example has a single extra space after the `6`.

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Extra spaces can find their way into your code via copy-and-paste errors, but since they come at the end of the line, they can go unnoticed in the source file and be invisible in the test failure report as well.

```
$ python -m doctest -v doctest_extra_space.py

Trying:
    my_function(2, 3)
Expecting:
    6
**********************************************************************
File "doctest_extra_space.py", line 12, in doctest_extra_space.my_function
Failed example:
    my_function(2, 3)
Expected:
    6
Got:
    6
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_extra_space
**********************************************************************
1 items had failures:
   1 of   2 in doctest_extra_space.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Using one of the diff-based reporting options, such as `REPORT_NDIFF`, shows the difference between the actual and expected values with more detail, and the extra space becomes visible.

```
def my_function(a, b):
    """
    >>> my_function(2, 3) #doctest: +REPORT_NDIFF
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Unified (`REPORT_UDIFF`) and context (`REPORT_CDIFF`) diffs are also available, for output where those formats are more readable.

```
$ python -m doctest -v doctest_ndiff.py

Trying:
    my_function(2, 3) #doctest: +REPORT_NDIFF
Expecting:
    6
**********************************************************************
File "doctest_ndiff.py", line 12, in doctest_ndiff.my_function
Failed example:
    my_function(2, 3) #doctest: +REPORT_NDIFF
Differences (ndiff with -expected +actual):
    - 6
    ?  -
    + 6
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_ndiff
**********************************************************************
1 items had failures:
   1 of   2 in doctest_ndiff.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

There are cases where it is beneficial to add extra whitespace in the sample output for the test, and have **doctest** ignore it. For example, data structures can be easier to read when spread across several lines, even if their representation would fit on a single line.

```
def my_function(a, b):
    """Returns a * b.

    >>> my_function(['A', 'B', 'C'], 3) #doctest: +NORMALIZE_WHITESPACE
    ['A', 'B', 'C',
     'A', 'B', 'C',
     'A', 'B', 'C']

    This does not match because of the extra space after the [ in the list

    >>> my_function(['A', 'B', 'C'], 2) #doctest: +NORMALIZE_WHITESPACE
    [ 'A', 'B', 'C',
      'A', 'B', 'C' ]
    """
    return a * b
```

When `NORMALIZE_WHITESPACE` is turned on, any whitespace in the actual and expected values is considered a match. You cannot add whitespace to the expected value where none exists in the output, but the length of the whitespace sequence and actual whitespace characters do not need to match. The first test example gets this rule correct, and passes, even though there are extra spaces and newlines. The second has extra whitespace after `[` and before `]`, so it fails.

```
$ python -m doctest -v doctest_normalize_whitespace.py

Trying:
    my_function(['A', 'B', 'C'], 3) #doctest: +NORMALIZE_WHITESPACE
Expecting:
    ['A', 'B', 'C',
     'A', 'B', 'C',
     'A', 'B', 'C']
ok
Trying:
    my_function(['A', 'B', 'C'], 2) #doctest: +NORMALIZE_WHITESPACE
Expecting:
    [ 'A', 'B', 'C',
      'A', 'B', 'C' ]
**********************************************************************
File "doctest_normalize_whitespace.py", line 20, in doctest_normalize_whitespace.my_funct
Failed example:
    my_function(['A', 'B', 'C'], 2) #doctest: +NORMALIZE_WHITESPACE
Expected:
    [ 'A', 'B', 'C',
      'A', 'B', 'C' ]
Got:
    ['A', 'B', 'C', 'A', 'B', 'C']
1 items had no tests:
    doctest_normalize_whitespace
**********************************************************************
1 items had failures:
   1 of   2 in doctest_normalize_whitespace.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Test Locations

All of the tests in the examples so far have been written in the docstrings of the functions they are testing. That is convenient for users who examine the docstrings for help using the funcion (especially with **`pydoc`**), but **`doctest`** looks for tests in other places, too. The obvious location for additional tests is in the docstrings elsewhere in the module.

```
#!/usr/bin/env python
# encoding: utf-8

"""Tests can appear in any docstring within the module.

Module-level tests cross class and function boundaries.

>>> A('a') == B('b')
False
"""

class A(object):
    """Simple class.

    >>> A('instance_name').name
    'instance_name'
    """
    def __init__(self, name):
        self.name = name
    def method(self):
        """Returns an unusual value.
```

```
        >>> A('name').method()
        'eman'
        """
        return ''.join(reversed(list(self.name)))

class B(A):
    """Another simple class.

    >>> B('different_name').name
    'different_name'
    """
```

Every docstring can contain tests at the module, class and function level.

```
$ python -m doctest -v doctest_docstrings.py

Trying:
    A('a') == B('b')
Expecting:
    False
ok
Trying:
    A('instance_name').name
Expecting:
    'instance_name'
ok
Trying:
    A('name').method()
Expecting:
    'eman'
ok
Trying:
    B('different_name').name
Expecting:
    'different_name'
ok
1 items had no tests:
    doctest_docstrings.A.__init__
4 items passed all tests:
   1 tests in doctest_docstrings
   1 tests in doctest_docstrings.A
   1 tests in doctest_docstrings.A.method
   1 tests in doctest_docstrings.B
4 tests in 5 items.
4 passed and 0 failed.
Test passed.
```

In cases where you have tests that you want to include with your source code, but do not want to have appear in the help for your module, you need to put them somewhere other than the docstrings. **`doctest`** also looks for a module-level variable called `__test__` and uses it to locate other tests. `__test__` should be a dictionary mapping test set names (as strings) to strings, modules, classes, or functions.

```
import doctest_private_tests_external

__test__ = {
    'numbers':"""
>>> my_function(2, 3)
6

>>> my_function(2.0, 3)
6.0
""",

    'strings':"""
>>> my_function('a', 3)
'aaa'

>>> my_function(3, 'a')
```

```
'aaa'
""",

    'external':doctest_private_tests_external,

    }

def my_function(a, b):
    """Returns a * b
    """
    return a * b
```

If the value associated with a key is a string, it is treated as a docstring and scanned for tests. If the value is a class or function, **doctest** searchs them recursively for docstrings, which are then scanned for tests. In this example, the module **doctest_private_tests_external** has a single test in its docstring.

```
#!/usr/bin/env python
# encoding: utf-8
#
# Copyright (c) 2010 Doug Hellmann.  All rights reserved.
#
"""External tests associated with doctest_private_tests.py.

>>> my_function(['A', 'B', 'C'], 2)
['A', 'B', 'C', 'A', 'B', 'C']
"""
```

doctest finds a total of five tests to run.

```
$ python -m doctest -v doctest_private_tests.py

Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(3, 'a')
Expecting:
    'aaa'
ok
2 items had no tests:
    doctest_private_tests
    doctest_private_tests.my_function
3 items passed all tests:
   1 tests in doctest_private_tests.__test__.external
   2 tests in doctest_private_tests.__test__.numbers
   2 tests in doctest_private_tests.__test__.strings
5 tests in 5 items.
5 passed and 0 failed.
Test passed.
```

External Documentation

Mixing tests in with your code isn't the only way to use **doctest**. Examples embedded in external project documentation files, such as reStructuredText files, can be used as well.

```
def my_function(a, b):
    """Returns a*b
    """
    return a * b
```

The help for **doctest_in_help** is saved to a separate file, `doctest_in_help.rst`. The examples illustrating how to use the module are included with the help text, and **doctest** can be used to find and run them.

```
==============================
 How to Use doctest_in_help.py
==============================

This library is very simple, since it only has one function called
``my_function()``.

Numbers
=======

``my_function()`` returns the product of its arguments.  For numbers,
that value is equivalent to using the ``*`` operator.

::

    >>> from doctest_in_help import my_function
    >>> my_function(2, 3)
    6

It also works with floating point values.

::

    >>> my_function(2.0, 3)
    6.0

Non-Numbers
===========

Because ``*`` is also defined on data types other than numbers,
``my_function()`` works just as well if one of the arguments is a
string, list, or tuple.

::

    >>> my_function('a', 3)
    'aaa'

    >>> my_function(['A', 'B', 'C'], 2)
    ['A', 'B', 'C', 'A', 'B', 'C']
```

The tests in the text file can be run from the command line, just as with the Python source modules.

```
$ python -m doctest -v doctest_in_help.rst

Trying:
    from doctest_in_help import my_function
Expecting nothing
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
```

```
Expecting:
    6.0
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
1 items passed all tests:
   5 tests in doctest_in_help.rst
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

Normally `doctest` sets up the test execution environment to include the members of the module being tested, so your tests don't need to import the module explicitly. In this case, however, the tests aren't defined in a Python module, `doctest` does not know how to set up the global namespace, so the examples need to do the import work themselves. All of the tests in a given file share the same execution context, so importing the module once at the top of the file is enough.

Running Tests

The previous examples all use the command line test runner built into `doctest`. It is easy and convenient for a single module, but will quickly become tedious as your package spreads out into multiple files. There are several alternative approaches.

By Module

You can include instructions to run `doctest` against your source at the bottom of your modules. Use **`testmod()`** without any arguments to test the current module.

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Ensure the tests are only run when the module is called as a main program by invoking **`testmod()`** only if the current module name is `__main__`.

```
$ python doctest_testmod.py -v

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
```

```
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

The first argument to **testmod()** is a module containing code to be scanned for tests. This feature lets you create a separate test script that imports your real code and runs the tests in each module one after another.

```
import doctest_simple

if __name__ == '__main__':
    import doctest
    doctest.testmod(doctest_simple)
```

You can build a test suite for your project by importing each module and running its tests.

```
$ python doctest_testmod_other_module.py -v

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    doctest_simple
1 items passed all tests:
   2 tests in doctest_simple.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

By File

testfile() works in a way similar to **testmod()**, allowing you to explicitly invoke the tests in an external file from within your test program.

```
import doctest

if __name__ == '__main__':
    doctest.testfile('doctest_in_help.rst')
```

```
$ python doctest_testfile.py -v

Trying:
    from doctest_in_help import my_function
Expecting nothing
ok
Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function(2.0, 3)
Expecting:
    6.0
ok
```

```
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
Trying:
    my_function(['A', 'B', 'C'], 2)
Expecting:
    ['A', 'B', 'C', 'A', 'B', 'C']
ok
1 items passed all tests:
   5 tests in doctest_in_help.rst
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

Both **testmod()** and **testfile()** include optional parameters to let you control the behavior of the tests through the doctest options, global namespace for the tests, etc. Refer to the standard library documentation for more details if you need those features – most of the time you won't need them.

Unittest Suite

If you use both unittest and doctest for testing the same code in different situations, you may find the unittest integration in doctest useful for running the tests together. Two classes, **DocTestSuite** and **DocFileSuite** create test suites compatible with the test-runner API of unittest.

```
import doctest
import unittest

import doctest_simple

suite = unittest.TestSuite()
suite.addTest(doctest.DocTestSuite(doctest_simple))
suite.addTest(doctest.DocFileSuite('doctest_in_help.rst'))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

The tests from each source are collapsed into a single outcome, instead of being reported individually.

```
$ python doctest_unittest.py

my_function (doctest_simple)
Doctest: doctest_simple.my_function ... ok
doctest_in_help.rst
Doctest: doctest_in_help.rst ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Test Context

The execution context created by doctest as it runs tests contains a copy of the module-level globals for the module containing your code. This isolates the tests from each other somewhat, so they are less likely to interfere with one another. Each test source (function, class, module) has its own set of global values.

```
class TestGlobals(object):

    def one(self):
        """
        >>> var = 'value'
        >>> 'var' in globals()
        True
```

```
        """

    def two(self):
        """
        >>> 'var' in globals()
        False
        """
```

TestGlobals has two methods, **one()** and **two()**. The tests in the docstring for **one()** set a global variable, and the test for **two()** looks for it (expecting not to find it).

```
$ python -m doctest -v doctest_test_globals.py

Trying:
    var = 'value'
Expecting nothing
ok
Trying:
    'var' in globals()
Expecting:
    True
ok
Trying:
    'var' in globals()
Expecting:
    False
ok
2 items had no tests:
    doctest_test_globals
    doctest_test_globals.TestGlobals
2 items passed all tests:
   2 tests in doctest_test_globals.TestGlobals.one
   1 tests in doctest_test_globals.TestGlobals.two
3 tests in 4 items.
3 passed and 0 failed.
Test passed.
```

That does not mean the tests *cannot* interfere with each other, though, if they change the contents of mutable variables defined in the module.

```
_module_data = {}

class TestGlobals(object):

    def one(self):
        """
        >>> TestGlobals().one()
        >>> 'var' in _module_data
        True
        """
        _module_data['var'] = 'value'

    def two(self):
        """
        >>> 'var' in _module_data
        False
        """
```

The module varabile `_module_data` is changed by the tests for **one()**, causing the test for **two()** to fail.

```
$ python -m doctest -v doctest_mutable_globals.py

Trying:
    TestGlobals().one()
Expecting nothing
ok
```

```
Trying:
    'var' in _module_data
Expecting:
    True
ok
Trying:
    'var' in _module_data
Expecting:
    False
**********************************************************************
File "doctest_mutable_globals.py", line 24, in doctest_mutable_globals.TestGlobals.two
Failed example:
    'var' in _module_data
Expected:
    False
Got:
    True
2 items had no tests:
    doctest_mutable_globals
    doctest_mutable_globals.TestGlobals
1 items passed all tests:
   2 tests in doctest_mutable_globals.TestGlobals.one
**********************************************************************
1 items had failures:
   1 of   1 in doctest_mutable_globals.TestGlobals.two
3 tests in 4 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
```

If you need to set global values for the tests, to parameterize them for an environment for example, you can pass values to **`testmod()`** and **`testfile()`** and have the context set up using data you control.

See also:

doctest
: The standard library documentation for this module.

The Mighty Dictionary
: Presentation by Brandon Rhodes at PyCon 2010 about the internal operations of the **`dict`**.

`difflib`
: Python's sequence difference computation library, used to produce the ndiff output.

Sphinx
: As well as being the documentation processing tool for Python's standard library, Sphinx has been adopted by many third-party projects because it is easy to use and produces clean output in several digital and print formats. Sphinx includes an extension for running doctests as is processes your documentation, so you know your examples are always accurate.

nose
: Third-party test runner with **`doctest`** support.

py.test
: Third-party test runner with **`doctest`** support.

Manuel
: Third-party documentation-based test runner with more advanced test case extraction and integration with Sphinx.