Manuel

Release 1

2022-06-24

Contents

Manuel lets you mix and match traditional doctests with custom test syntax.

Several plug-ins are included that provide new test syntax (see *Included Functionality*). You can also create your own plug-ins.

For example, if you've ever wanted to include a large chunk of Python in a doctest but were irritated by all the ">>>" and "..." prompts required, you'd like the `manuel.codeblock` module. It lets you execute code using Sphinx-style ".. code-block:: python" directives. The markup looks like this:

```
.. code-block:: python

    import foo

    def my_func(bar):
        return foo.baz(bar)
```

Incidentally, the implementation of `manuel.codeblock` is only 23 lines of code.

The plug-ins included in Manuel make good examples while being quite useful in their own right. The Manuel documentation makes extensive use of them as well. Follow the "Show Source" link to the left to see the reST source of this document.

For a large example of creating test syntax, take a look at the *FIT Table Example* or for all the details, *Theory of Operation*.

To see how to get Manuel wired up see *Getting Started*.

Contents

CHAPTER 1

Included Functionality

Manuel includes several plug-ins out of the box:

manuel.capture stores regions of a document in variables for later processing

manuel.codeblock executes code in ".. code-block:: python" blocks

manuel.doctest provides traditional doctest processing as a Manuel plug-in

manuel.footnote executes code in reST-style footnodes each time they're referenced (good for getting incidental code out of the main flow of a document)

manuel.ignore ignores parts of a document while running tests

manuel.isolation makes it easier to have test isolation in doctests

manuel.testcase identify parts of tests as individual test cases so they can be run independently

CHAPTER 2

Getting Started

The plug-ins used for a test are composed together using the "+" operator. Let's say you wanted a test that used doctest syntax as well as footnotes. You would create a Manuel instance to use like this:

```
import manuel.doctest
import manuel.footnote

m = manuel.doctest.Manuel()
m += manuel.footnote.Manuel()
```

You would then pass the Manuel instance to a `manuel.testing.TestSuite`, including the names of documents you want to process:

```
manuel.testing.TestSuite(m, 'test-one.txt', 'test-two.txt')
```

2.1 Using unittest

The simplest way to get started with Manuel is to use `unittest` to run your tests:

```
import manuel.codeblock
import manuel.doctest
import manuel.testing
import unittest

def test_suite():
    m = manuel.doctest.Manuel()
    m += manuel.codeblock.Manuel()
    return manuel.testing.TestSuite(m, 'test-one.txt', 'test-two.txt')

if __name__ == '__main__':
    unittest.TextTestRunner().run(test_suite())
```

2.2 Using zope.testing

If you want to use a more featureful test runner you can use zope.testing's test runner (usable stand-alone – it isn't dependent on the Zope application server). Create a file named `tests.py` with a `test_suite()` function that returns a test suite.

The suite can be either a `manuel.testing.TestSuite` object or a `unittest.TestSuite` as demonstrated below.

```
import manuel.codeblock
import manuel.doctest
import manuel.testing

def test_suite():
    suite = unittest.TestSuite()

    # here you add your other tests to the suite...

    # now you can add the Manuel tests
    m = manuel.doctest.Manuel()
    m += manuel.codeblock.Manuel()
    suite.addTest(manuel.testing.TestSuite(m,
        'test-one.txt', 'test-two.txt'))

    return suite
```

2.3 Others

To use another test runner, like nose or pytest:

```
import manuel.codeblock
import manuel.doctest
import manuel.testing

m = manuel.doctest.Manuel()
m += manuel.codeblock.Manuel()
manueltest = manuel.testing.TestFactory(m)

class MyTest(unittest.TestCase):

    def setUp(self):
        self.a = 1
        self.globs = dict(c=9)

    test1 = manueltest('doc1.ex')

    @manueltest('doc2.ex')
    def test2(self):
        self.x = 5

    test3 = manueltest('doc3.ex')
```

Here, we instantiated *TestFactory* with a *Manuel* instance to create *manueltest*, which is a factory for creating Manuel-based tests using on the given Manuel instance. We then used that to create 3 tests.

The first and third tests just execute tests in the named files, *doc1.ex* and *doc3.ex*. The class' *setUp* method is used to set up the test.

The second test also executes tests in a named file, *doc2.ex*, but it decorates a function that provides additional setup code that runs after the class setup code.

When tests are run this way:

- The test globals contain the test instance in the *test* variable.
- If a test case defines a *globs* attribute, it must be a dictionary and it's contents are added to the test globals.

2.4 Customizing the TestCase class

Manuel has its own `manuel.testing.TestClass` class that `manuel.testing.TestSuite` uses. If you want to customize it, you can pass in your own class to *TestSuite*.

```
 import os.path
 import manuel.testing

 class StripDirsTestCase(manuel.testing.TestCase):
     def shortDescription(self):
         return os.path.basename(str(self))
 suite = manuel.testing.TestSuite(
     m, path_to_test, TestCase=StripDirsTestCase)

>>> list(suite)[0].shortDescription()
'bugs.txt'
```

CHAPTER 3

Doctests

Manuel is all about making testable documents and well-documented tests. Of course, Python's doctest module is a long-standing fixture in that space, so it only makes sense for Manuel to support doctest syntax.

Handling doctests is easy:

```
import manuel.doctest

m = manuel.doctest.Manuel()
suite = manuel.testing.TestSuite(m, 'my-doctest.txt')
```

Of course you can mix in other Manuel syntax plug-ins as well (including ones you write yourself).

```
import manuel.doctest
import manuel.codeblock

m = manuel.doctest.Manuel()
m += manuel.codeblock.Manuel()
suite = manuel.testing.TestSuite(m, 'my-doctest-with-code-blocks.txt')
```

The `manuel.doctest.Manuel` constructor also takes `optionflags` and `checker` arguments.

```
m = manuel.doctest.Manuel(optionflags=optionflags, checker=checker)
```

See the doctest documentation for more information about the available options and output checkers

Note: `zope.testing.renormalizing` provides an `OutputChecker` for smoothing out differences between actual and expected output for things that are hard to control (like memory addresses and time). See the module's doctests for more information on how it works. Here's a short example that smoothes over the differences between CPython's and PyPy's NameError messages:

```
import re
import zope.testing.renormalizing
```

(continues on next page)

(continued from previous page)

```
checker = zope.testing.renormalizing.RENormalizing([
    (re.compile(r"NameError: global name '([a-zA-Z0-9_]+)' is not defined"),
     r"NameError: name '\1' is not defined"),
])
```

CHAPTER 4

Capturing Blocks

When writing documentation the need often arises to describe the contents of files or other non-Python information. You may also want to put that information under test. `manuel.capture` helps with that.

For example, if you were writing the problems for a programming contest, you might want to describe the input and output files for each challenge, but you want to be sure that your examples are correct.

To do that you might write your document like this:

```
Challenge 1
===========

Write a program that sorts the numbers in a file.


Example
-------

Given this example input file::

    6
    1
    8
    20
    11
    65
    2

.. -> input

Your program should generate this output file::

    1
    2
    6
    8
```

(continues on next page)

(continued from previous page)

```
    11
    20
    65

.. -> output

    >>> input_lines = input.splitlines()
    >>> correct = '\n'.join(map(str, sorted(map(int, input_lines)))) + '\n'
    >>> output == correct
    True
```

This uses the syntax implemented in `manuel.capture` to capture a block of text into a variable (the one named after "->").

Whenever a line of the structure ".. -> VAR" is detected, the text of the *previous* block will be stored in the given variable.

Of course, lines that start with ".. " are reST comments, so when the document is rendered with docutils or Sphinx, the tests will dissapear and only the intended document contents will remain. Like so:

```
Challenge 1
===========

Write a program that sorts the numbers in a file.


Example
-------

Given this example input file::

    6
    1
    8
    20
    11
    65
    2

Your program should generate this output file::

    1
    2
    6
    8
    11
    20
    65
```

CHAPTER 5

Code Blocks

Sphinx and other docutils extensions provide a "code-block" directive, which allows inlined snippets of code in reST documents.

The `manuel.codeblock` module provides the ability to execute the contents of Python code-blocks. For example:

```
.. code-block:: python

    print('hello')
```

If the code-block generates some sort of error. . .

```
.. code-block:: python

    print(does_not_exist)
```

. . . that error will be reported:

```
>>> document.process_with(m, globs={})
Traceback (most recent call last):
    ...
NameError: name 'does_not_exist' is not defined
```

If you find that you want to include a code-block in a document but don't want Manuel to execute it, use *manuel.ignore* to ignore that particular block.

5.1 Docutils Code Blocks

Sphinx and docutils have different ideas of how code blocks should be spelled. Manuel supports the docutils-style code blocks too.

```
.. code:: python

    a = 1
```

Docutils options after the opening of the code block are also allowed:

```
.. code:: python
    :class: hidden

    a = 1
```

5.2 Invisible Code Blocks

At times you'll want to have a block of code that is executed but not displayed in the rendered document (like some setup for later examples).

When using doctest's native format (">>>") that's easy to do, you just put the code in a reST comment, like so:

```
.. this is some setup, it is hidden in a reST comment

    >>> a = 5
    >>> b = a + 3
```

However, if you want to include a relatively large chunk of Python, you'd rather use a code-block, but that means that it will be included in the rendered document. Instead, `manuel.codeblock` also understands a variant of the code-block directive that is actually a reST comment: ".. invisible-code-block:: python":

```
.. invisible-code-block:: python

    a = 5
    b = a + 3
```

Note: The "invisible-code-block" directive will work with either one or two colons. The reason is that reST processers (like docutils and Sphinx) will generate an error for unrecognized directives (like invisible-code-block). Therefore you can use a single colon and the line will be interpreted as a comment instead.

CHAPTER 6

Footnotes

The `manuel.footnote` module provides an implementation of reST footnote handling, but instead of just plain text, the footnotes can contain any syntax Manuel can interpret including doctests.

```
>>> import manuel.footnote
>>> m = manuel.footnote.Manuel()
```

Here's an example of combining footnotes with doctests:

```
Here we reference a footnote. [1]_

    >>> x
    42

Here we reference another. [2]_

    >>> x
    100

.. [1] This is a test footnote definition.

    >>> x = 42

.. [2] This is another test footnote definition.

    >>> x = 100

.. [3] This is a footnote that will never be executed.

    >>> raise RuntimeError('nooooo!')
```

It is also possible to reference more than one footnote on a single line.

```
This line has several footnotes on it. [1]_ [2]_ [3]_

```

(continues on next page)

(continued from previous page)

```
    >>> z
    105

A little prose to separate the examples.

.. [1] Do something

    >>> w = 3

.. [2] Do something

    >>> x = 5

.. [3] Do something

    >>> y = 7

    >>> z = w * x * y
```

CHAPTER 7

Ignoring Blocks

Occasionally the need arises to ignore a block of markup that would otherwise be parsed by a Manuel plug-in.

For example, this document has a code-block that will generate a syntax error:

```
The following is invalid Python.

.. code-block:: python

    def foo:
        pass
```

We can see that when executed, the SyntaxError escapes.

```
>>> import manuel.codeblock
>>> m = manuel.codeblock.Manuel()
>>> document.process_with(m, globs={})
  File "<memory>:4", line 2
     def foo:
            ^
SyntaxError: invalid syntax
```

The `manuel.ignore` module provides a way to ignore parts of a document using a directive ".. ignore-next-block".

Because Manuel plug-ins are executed in the order they are accumulated, we want `manuel.ignore` to be the base Manuel object, with any additional plug-ins added to it.

```
import manuel.ignore
import manuel.doctest
m = manuel.ignore.Manuel()
m += manuel.codeblock.Manuel()
m += manuel.doctest.Manuel()
```

If we add an ignore marker to the block we don't want processed…

```
The following is invalid Python.

.. ignore-next-block
.. code-block:: python

    def foo:
        pass
```

. . . the error goes away.

```
>>> document.process_with(m, globs={})
>>> print(document.formatted())
```

7.1 Ignoring Literal Blocks

Ignoring literal blocks is a little more involved:

```
Here is some invalid Python:

.. ignore-next-block

::

   >>> lambda: x=1
```

CHAPTER 8

Test Isolation

One of the advantages of unittest over doctest is that the individual tests are isolated from one-another.

In large doctests (like this one) you may want to keep later tests from depending on incidental details of earlier tests, preventing the tests from becoming brittle and harder to change.

Test isolation is one approach to reducing this intra-doctest coupling. The `manuel.isolation` module provides a plug-in to help.

The ".. reset-globs" directive resets the globals in the test:

```
We define a variable.

    >>> x = 'hello'

It is still defined.

    >>> print(x)
    hello

Now we can reset the globals...

.. reset-globs

...and the name binding will be gone:

    >>> print(x)
    Traceback (most recent call last):
        ...
    NameError: name 'x' is not defined
```

We can see that after the globals have been reset, the second "print(x)" line raises an error.

Of course, resetting to an empty set of global variables isn't always what's wanted. In that case there is a ".. capture-globs" directive that saves a baseline set of globals that will be restored at each reset.

```
We define a variable.

    >>> x = 'hello'

It is still defined.

    >>> print(x)
    hello

We can capture the currently defined globals:

.. capture-globs

Of course capturing the globals doesn't disturb them.

    >>> print(x)
    hello

Now if we define a new global...

    >>> y = 'goodbye'
    >>> print(y)
    goodbye

.. reset-globs

...it will disappear after a reset.

    >>> print(y)
    Traceback (most recent call last):
        ...
    NameError: name 'y' is not defined

But the captured globals will still be defined.

    >>> print(x)
    hello
```

CHAPTER 9

Identifying Test Cases

If you want parts of a document to be individually accessible as test cases (to be able to run just a particular subset of them, for example), a parser can create a region that marks the beginning of a new test case.

Two ways of identifying test cases are included in `manuel.testcase`:

1. by section headings
2. by explicit ".. test-case: NAME" markers.

9.1 Grouping Tests by Heading

```
First Section
=============

Some prose.

    >>> print('first test case')

Some more prose.

    >>> print('still in the first test case')

Second Section
==============

Even more prose.

    >>> print('second test case')
```

Given the above document, if you're using zope.testing's testrunner (located in bin/test), you could run just the tests in the second section with this command:

```
bin/test -t "file-name.txt:Second Section"
```

Or, exploiting the fact that -t does a regex search (as opposed to a match):

```
bin/test -t file-name.txt:Second
```

9.2 Grouping Tests Explicitly

If you would like to identify test cases separately from sections, you can identify them with a marker:

```
First Section
=============

The following test will be in a test case that is not individually
identifiable.

    >>> print('first test case (unidentified)')

Some more prose.

.. test-case: first-named-test-case

    >>> print('first identified test case')


Second Section
==============

The test case markers don't have to immediately proceed a test.

.. test-case: second-named-test-case

Even more prose.

    >>> print('second identified test case')
```

Again, given the above document and zope.testing, you could run just the second set of tests with this command:

```
bin/test -t file-name.txt:second-named-test-case
```

Or, exploiting the fact that -t does a regex search again:

```
bin/test -t file-name.txt:second
```

Even though the tests are individually accessable doesn't mean that they can't all be run at the same time:

```
bin/test -t file-name.txt
```

Also, if you create a hierarchy of names, you can run groups of tests at a time. For example, lets say that you append "-important" to all your really important tests, you could then run the important tests for a single document like so:

```
bin/test -t 'file-name.txt:.*-important$'
```

or all the "important" tests no matter what file they are in:

```
bin/test -t '-important$'
```

9.3 Both Methods

You can also combine more than one test case identification method if you want. Here's an example of building a Manuel stack that has doctests and both flavors of test case identification:

```
import manuel.doctest
import manuel.testcase

m = manuel.doctest.Manuel()
m += manuel.testcase.SectionManuel()
m += manuel.testcase.MarkerManuel()
```

CHAPTER 10

Further Reading

10.1 Theory of Operation

Manuel parses documents (tests), evaluates their contents, then formats the result of the evaluation. The functionality is accessed via the `manuel` package.

```
>>> import manuel
```

10.1.1 Parsing

Manuel operates on Documents. Each Document is created from a string containing one or more lines.

```
>>> source = """\
... This is our document, it has several lines.
... one: 1, 2, 3
... two: 4, 5, 7
... three: 3, 5, 1
... """
>>> document = manuel.Document(source)
```

For example purposes we will create a type of test that consists of a sequence of numbers. Lets create a NumbersTest object to represent the parsed list.

```
>>> class NumbersTest(object):
...     def __init__(self, description, numbers):
...         self.description = description
...         self.numbers = numbers
```

The Document is divided into one or more regions. Each region is a distinct “chunk” of the document and will be acted uppon in later (post-parsing) phases. Initially the Document is made up of a single element, the source string.

```
>>> [region.source for region in document]
['This is our document, it has several lines.\none: 1, 2, 3\ntwo: 4, 5, 7\nthree: 3,␣
↪5, 1\n']
```

The Document offers a “find_regions” method to assist in locating the portions of the document a particular parser is interested in. Given a regular expression (either as a string, or compiled), it will return “region” objects that contain the matched source text, the line number (1 based) the region begins at, as well as the associated re.Match object.

```
>>> import re
>>> numbers_test_finder = re.compile(
...     r'^(?P<description>.*?): (?P<numbers>(\d+,?[ ]?)+)$', re.MULTILINE)
>>> regions = document.find_regions(numbers_test_finder)
>>> regions
[<manuel.Region object at 0x...>,
 <manuel.Region object at 0x...>,
 <manuel.Region object at 0x...>]
>>> regions[0].lineno
2
>>> regions[0].source
'one: 1, 2, 3\n'
>>> regions[0].start_match.group('description')
'one'
>>> regions[0].start_match.group('numbers')
'1, 2, 3'
```

If given two regular expressions find_regions will use the first to identify the begining of a region and the second to identify the end.

```
>>> region = document.find_regions(
...     re.compile('^one:.*$', re.MULTILINE),
...     re.compile('^three:.*$', re.MULTILINE),
...     )[0]
>>> region.lineno
2
>>> six.print_(region.source)
one: 1, 2, 3
two: 4, 5, 7
three: 3, 5, 1
```

Also, instead of just a “start_match” attribute, the region will have start_match and end_match attributes.

```
>>> region.start_match
<_sre.SRE_Match object...>
>>> region.end_match
<_sre.SRE_Match object...>
```

Regions must always consist of whole lines.

```
>>> document.find_regions('1, 2, 3')
Traceback (most recent call last):
    ...
ValueError: Regions must start at the begining of a line.
```

Now we can register a parser that will identify the regions we’re interested in and create NumbersTest objects from the source text.

```
>>> def parse(document):
...     for region in document.find_regions(numbers_test_finder):
...         description = region.start_match.group('description')
...         numbers = list(map(
...             int, region.start_match.group('numbers').split(',')))
...         test = NumbersTest(description, numbers)
...         document.claim_region(region)
...         region.parsed = test
```

```
>>> parse(document)
>>> [region.source for region in document]
['This is our document, it has several lines.\n',
 'one: 1, 2, 3\n',
 'two: 4, 5, 7\n',
 'three: 3, 5, 1\n']
>>> [region.parsed for region in document]
[None,
 <NumbersTest object at 0x...>,
 <NumbersTest object at 0x...>,
 <NumbersTest object at 0x...>]
```

10.1.2 Evaluation

After a document has been parsed the resulting tests are evaluated. Unlike parsing and formatting, evaluation is done one region at a time, in the order that the regions appear in the document. Lets define a function to evaluate NumberTests. The function determines whether or not the numbers are in sorted order and records the result along with the description of the list of numbers.

```
class NumbersResult(object):
    def __init__(self, test, passed):
        self.test = test
        self.passed = passed

def evaluate(region, document, globs):
    if not isinstance(region.parsed, NumbersTest):
        return
    test = region.parsed
    passed = sorted(test.numbers) == test.numbers
    region.evaluated = NumbersResult(test, passed)
```

10.1.3 Formatting

Once the evaluation phase is completed the results are formatted. You guessed it: Manuel provides a method for formatting results. We'll build one to format a message about whether or not our lists of numbers are sorted properly. A formatting function returns None when it has no output, or a string otherwise.

```
def format(document):
    for region in document:
        if not isinstance(region.evaluated, NumbersResult):
            continue
        result = region.evaluated
        if not result.passed:
            region.formatted = (
```

(continues on next page)

(continued from previous page)

```
                "the numbers aren't in sorted order: %s\n"
                % ', '.join(map(str, result.test.numbers)))
```

Since one of the test cases failed we get an appropriate message out of the formatter.

```
>>> format(document)
>>> [region.formatted for region in document]
[None, None, None, "the numbers aren't in sorted order: 3, 5, 1\n"]
```

10.1.4 Manuel Objects

We'll want to use these parse, evaluate, and format functions later, so we bundle them together into a Manuel object.

```
>>> sorted_numbers_manuel = manuel.Manuel(
...     parsers=[parse], evaluaters=[evaluate], formatters=[format])
```

10.1.5 Doctests

We can use Manuel to run doctests. Let's create a simple doctest to demonstrate with.

```
>>> source = """This is my
... doctest.
...
...     >>> 1 + 1
...     2
... """
>>> document = manuel.Document(source)
```

The `manuel.doctest` module has handlers for the various phases. First we'll look at parsing.

```
>>> import manuel.doctest
>>> m = manuel.doctest.Manuel()
>>> document.parse_with(m)
>>> for region in document:
...     print((region.lineno, region.parsed or region.source))
(1, 'This is my\ndoctest.\n\n')
(4, <doctest.Example ...>)
```

Now we can evaluate the examples.

```
>>> document.evaluate_with(m, globs={})
>>> for region in document:
...     print((region.lineno, region.evaluated or region.source))
(1, 'This is my\ndoctest.\n\n')
(4, <manuel.doctest.DocTestResult ...>)
```

And format the results.

```
>>> document.format_with(m)
>>> document.formatted()
''
```

Oh, we didn't have any failing tests, so we got no output. Let's try again with a failing test. This time we'll use the "process_with" function to simplify things.

```
>>> document = manuel.Document("""This is my
... doctest.
...
...     >>> 1 + 1
...     42
... """)
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
File "<memory>", line 4, in <memory>
Failed example:
    1 + 1
Expected:
    42
Got:
    2
```

Alternate doctest parsers

You can pass an alternate doctest parser to manuel.doctest.Manuel to customize how examples are parsed. Here's an example that changes the example start string from “>>>” to “py>”:

```
>>> import doctest
>>> class DocTestPyParser(doctest.DocTestParser):
...    _EXAMPLE_RE = re.compile(r'''
...        (?P<source>
...             (?:^(?P<indent> [ ]*) py>    .*)    # PS1 line
...            (?:\n           [ ]*  \.\.\. .*)*)  # PS2 lines
...        \n?
...        (?P<want> (?:(?![ ]*$)    # Not a blank line
...                     (?![ ]*py>)  # Not a line starting with PS1
...                     .*$\n?       # But any other line
...                  )*)
...        ''', re.MULTILINE | re.VERBOSE)
```

```
>>> m = manuel.doctest.Manuel(parser=DocTestPyParser())
>>> document = manuel.Document("""This is my
... doctest.
...
...     py> 1 + 1
...     42
... """)
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
File "<memory>", line 4, in <memory>
Failed example:
    1 + 1
Expected:
    42
Got:
    2
```

Multiple doctest parsers

You may use several doctest parsers in the same session, for example, to support shell commands and Python code in the same document.

```
>>> m = (manuel.doctest.Manuel(parser=DocTestPyParser()) +
...       manuel.doctest.Manuel())
```

```
>>> document = manuel.Document("""
...
...     py> i = 0
...     py> i += 1
...     py> i
...     1
...
...     >>> j = 0
...     >>> j += 1
...     >>> j
...     1
...
... """)
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
```

10.1.6 Globals

Even though each region is parsed into its own object, state is still shared between them. Each region of the document is executed in order so state changes made by earlier evaluaters are available to the current evaluator.

```
>>> document = manuel.Document("""
...     >>> x = 1
...
... A little prose to separate the examples.
...
...     >>> x
...     1
... """)
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
```

Imported modules are added to the global namespace as well.

```
>>> document = manuel.Document("""
...     >>> import string
...
... A little prose to separate the examples.
...
...     >>> string.digits
...     '0123456789'
...
... """)
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
```

10.1.7 Combining Test Types

Now that we have both doctests and the silly "sorted numbers" tests, let's create a single document that has both.

```
>>> document = manuel.Document("""
... We can have a list of numbers...
...
...     a very nice list: 3, 6, 2
...
... ... and we can test Python.
...
...     >>> 1 + 1
...     42
...
... """)
```

Obviously both of those tests will fail, but first we have to configure Manuel to understand both test types. We'll start with a doctest configuration and add the number list testing on top.

```
>>> m = manuel.doctest.Manuel()
```

Since we already have a Manuel instance configured for our "sorted numbers" tests, we can extend the built-in doctest configuration with it.

```
>>> m += sorted_numbers_manuel
```

Now we can process our source that combines both types of tests and see what we get.

```
>>> document.process_with(m, globs={})
```

The document was parsed and has a mixture of prose and parsed doctests and number tests.

```
>>> for region in document:
...     print((region.lineno, region.parsed or region.source))
(1, '\nWe can have a list of numbers...\n\n')
(4, <NumbersTest object at 0x...>)
(5, '\n... and we can test Python.\n\n')
(8, <doctest.Example ...>)
(10, '\n')
```

We can look at the formatted output to see that each of the two tests failed.

```
>>> for region in document:
...     if region.formatted:
...         six.print_('-'*70)
...         six.print_(region.formatted, end='')
----------------------------------------------------------------------
the numbers aren't in sorted order: 3, 6, 2
----------------------------------------------------------------------
File "<memory>", line 8, in <memory>
Failed example:
    1 + 1
Expected:
    42
Got:
    2
```

10.1.8 Priorities

Some functionality requires that code be called early or late in a phase. The "timing" decorator allows either EARLY or LATE to be specified.

Early functions are run first (in arbitrary order), then functions with no specified timing, then the late functions are called (again in arbitrary order). This function also demonstrates the "copy" method of Region objects and the "insert_region_before" and "insert_region_after" methods of Documents.

```
>>> @manuel.timing(manuel.LATE)
... def cloning_parser(document):
...     to_be_cloned = None
...     # find the region to clone
...     document_iter = iter(document)
...     for region in document_iter:
...         if region.parsed:
...             continue
...         if region.source.strip().endswith('my clone:'):
...             to_be_cloned = six.advance_iterator(document_iter).copy()
...             break
...     # if we found the region to cloned, do so
...     if to_be_cloned:
...         # make a copy since we'll be mutating the document
...         for region in list(document):
...             if region.parsed:
...                 continue
...             if 'clone before *here*' in region.source:
...                 clone = to_be_cloned.copy()
...                 clone.provenance = 'cloned to go before'
...                 document.insert_region_before(region, clone)
...             if 'clone after *here*' in region.source:
...                 clone = to_be_cloned.copy()
...                 clone.provenance = 'cloned to go after'
...                 document.insert_region_after(region, clone)
```

```
>>> m.add_parser(cloning_parser)
```

```
>>> source = """\
... This is my clone:
...
... clone: 1, 2, 3
...
... I want some copies of my clone.
...
... For example, I'd like a clone before *here*.
...
... I'd also like a clone after *here*.
... """
>>> document = manuel.Document(source)
>>> document.process_with(m, globs={})
>>> [(r.source, r.provenance) for r in document]
[('This is my clone:\n\n', None),
 ('clone: 1, 2, 3\n', None),
 ('clone: 1, 2, 3\n', 'cloned to go before'),
 ("\nI want some copies of my clone.\n\nFor example, I'd like a clone before *here*.
↪\n\nI'd also like a clone after *here*.\n", None),
 ('clone: 1, 2, 3\n', 'cloned to go after')]
```

10.1.9 Enhancing Existing Manuels

Lets say that you'd like failed doctest examples to give more information about what went wrong.

First we'll create an evaluater that includes pertinant variable binding information on failures.

```
import doctest

def informative_evaluater(region, document, globs):
    if not isinstance(region.parsed, doctest.Example):
        return
    if region.evaluated.getvalue():
        info = ''
        for name in sorted(globs):
            if name in region.parsed.source:
                info += '\n    ' + name + ' = ' + repr(globs[name])

        if info:
            region.evaluated.write('Additional Information:')
            region.evaluated.write(info)
```

To do that we'll start with an instance of `manuel.doctest.Manuel` and add in our additional functionality.

```
>>> m = manuel.doctest.Manuel()
>>> m.add_evaluater(informative_evaluater)
```

Now we'll create a document that includes a failing test.

```
>>> document = manuel.Document("""
... Set up some variable bindings:
...
...     >>> a = 1
...     >>> b = 2
...     >>> c = 3
...
... Make an assertion:
...
...     >>> a + b
...     5
... """)
```

When we run the document through our Manuel instance, we see the additional information.

```
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
File "<memory>", line 10, in <memory>
Failed example:
    a + b
Expected:
    5
Got:
    3
Additional Information:
    a = 1
    b = 2
```

Note how only the referenced variable bindings are displayed (i.e., "c" is not listed). That's pretty nice, but the way interesting variables are identified is a bit of a hack. For example, if a variable's name just happens to appear in the source (in a comment for example), it will be included in the output:

```
>>> document = manuel.Document("""
... Set up some variable bindings:
...
...     >>> a = 1
...     >>> b = 2
...     >>> c = 3
...
... Make an assertion:
...
...     >>> a + b # doesn't mention "c"
...     5
... """)
```

```
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
File "<memory>", line 10, in <memory>
Failed example:
    a + b # doesn't mention "c"
Expected:
    5
Got:
    3
Additional Information:
    a = 1
    b = 2
    c = 3
```

Instead of a text-based apprach, let's use the built-in tokenize module to more robustly identify referenced variables.

```
>>> from six import StringIO
>>> import token
>>> import tokenize
```

```
>>> def informative_evaluater_2(region, document, globs):
...     if not isinstance(region.parsed, doctest.Example):
...         return
...
...     if region.evaluated.getvalue():
...         vars = set()
...         reader = StringIO(region.source).readline
...         for ttype, tval, _, _, _ in tokenize.generate_tokens(reader):
...             if ttype == token.NAME:
...                 vars.add(tval)
...
...         info = ''
...         for name in sorted(globs):
...             if name in vars:
...                 info += '\n    ' + name + ' = ' + repr(globs[name])
...
...         if info:
...             region.evaluated.write('Additional Information:')
...             region.evaluated.write(info)
```

```
>>> m = manuel.doctest.Manuel()
>>> m.add_evaluater(informative_evaluater_2)
```

Now when we have a failure, only the genuinely referenced variables will be included in the debugging information.

```
>>> document = manuel.Document(document.source)
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
File "<memory>", line 10, in <memory>
Failed example:
    a + b # doesn't mention "c"
Expected:
    5
Got:
    3
Additional Information:
    a = 1
    b = 2
```

10.1.10 Defining Test Cases

If you want parts of a document to be accessable individually as test cases (to be able to run just a particular part of a document, for example), a parser can create a region that marks the beginning of a new test case.

```
new_test_case_regex = re.compile(r'^.. new-test-case: \w+', re.MULTILINE)

def parse(document):
    for region in document.find_regions(new_test_case_regex):
        document.claim_region(region)
        id = region.start_match.group(1)
        region.parsed = manuel.testing.TestCaseMarker(id)
```

XXX finish this section

10.2 FIT Table Example

Here is an example of writing a relatively complex Manuel plug-in.

Occasionally when writing a doctest, you want a better way to express a test than doctest by itself provides.

For example, you may want to succinctly express the result of an expression for several sets of inputs and outputs.

That's something FIT tables do a good job of.

We can use Manuel to write a parser that can read the tables, an evaluator that can check to see if the assertions made in the tables match reality, and a formatter to display the results if they don't.

We'll use reST tables as the table format. The table source will look like this:

```
=====  =====  ======
\      A or B
--------------------
  A      B    Result
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
```

When rendered to HTML, it will look like this:

A or B		
A	B	Result
False	False	False
True	False	True
False	True	True
True	True	True

10.2.1 Documents

Here is an example of a source document we want our plug-in to be able to understand:

```
The "or" operator
=================

Here is an example of the "or" operator in action:

=====  =====  ======
\      A or B
--------------------
  A      B    Result
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
```

Manuel plug-ins operate on instances of `manuel.Document`.

```
import manuel
document = manuel.Document(source, location='fake.txt')
```

10.2.2 Parsing

We need an object to represent the tables.

```
class Table(object):
    def __init__(self, expression, variables, examples):
        self.expression = expression
        self.variables = variables
        self.examples = examples
```

We'll also need a function to find the tables in the document, extract the pertinent details, and instantiate Table objects.

```
import re
import six

table_start = re.compile(r'(?<=\n\n)=[= ]+\n(?=[ \t]*?\S)', re.DOTALL)
table_end = re.compile(r'\n=[= ]+\n(?=\Z|\n)', re.DOTALL)

def parse_tables(document):
    for region in document.find_regions(table_start, table_end):
        lines = enumerate(iter(region.source.splitlines()))
```

(continues on next page)

(continued from previous page)

```
        six.advance_iterator(lines) # skip the first line

        # grab the expression to be evaluated
        expression = six.advance_iterator(lines)[1]
        if expression.startswith('\\'):
            expression = expression[1:]

        six.advance_iterator(lines) # skip the divider line
        variables = [v.strip() for v in six.advance_iterator(lines)[1].split()][:-1]

        six.advance_iterator(lines) # skip the divider line

        examples = []
        for lineno_offset, line in lines:
            if line.startswith('='):
                break # we ran into the final divider, so stop

            values = [eval(v.strip(), {}) for v in line.split()]
            inputs = values[:-1]
            output = values[-1]

            examples.append((inputs, output, lineno_offset))

        table = Table(expression, variables, examples)
        document.claim_region(region)
        region.parsed = table
```

If we parse the Document we can see that the table was recognized.

```
>>> parse_tables(document)
>>> region = list(document)[1]
>>> import six
>>> six.print_(region.source, end='')
=====  =====  ======
\      A or B
--------------------
  A      B    Result
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
>>> region.parsed
<Table object at ...>
```

10.2.3 Evaluating

Now that we can find and extract the tables from the source, we need to be able to check them for correctness.

The parse phase decomposed the `Document` into several `Region` instances. During the evaluation phase each evaluater is called once for each region.

The evaluate_table function iterates over each set of inputs given in a single table, evaluate the inputs with the expression and compare the result with what was expected. Each discrepancy will be stored as a `TableError` in a `TableErrors` object.

```
class TableErrors(list):
    pass


class TableError(object):
    def __init__(self, location, lineno, expected, got):
        self.location = location
        self.lineno = lineno
        self.expected = expected
        self.got = got

    def __str__(self):
        return '<%s %s:%s>' % (
            self.__class__.__name__, self.location, self.lineno)


def evaluate_table(region, document, globs):
    if not isinstance(region.parsed, Table):
        return

    table = region.parsed
    errors = TableErrors()
    for inputs, output, lineno_offset in table.examples:
        result = eval(table.expression, dict(zip(table.variables, inputs)))
        if result != output:
            lineno = region.lineno + lineno_offset
            errors.append(
                TableError(document.location, lineno, output, result))

    region.evaluated = errors
```

Now we can use the function to evaluate our table.

```
>>> evaluate_table(region, document, {})
```

Yay! There were no errors:

```
>>> region.evaluated
[]
```

What would happen if there were errors?

```
The "or" operator
=================

Here is an (erroneous) example of the "or" operator in action:

=====  =====  ======
\      A or B
--------------------
  A      B    Result
=====  =====  ======
False  False  True
True   False  True
False  True   False
True   True   True
=====  =====  ======
```

. . . the result of evaluaton would include them:

```
>>> region.evaluated
[<TableError object at ...>]
```

10.2.4 Formatting Errors

Now that we can parse the tables and evaluate them, we need to be able to display the results in a readable fashion.

```
def format_table_errors(document):
    for region in document:
        if not isinstance(region.evaluated, TableErrors):
            continue

        # if there were no errors, there is nothing to report
        if not region.evaluated:
            continue

        messages = []
        for error in region.evaluated:
            messages.append('%s, line %d: expected %r, got %r instead.' % (
                error.location, error.lineno, error.expected, error.got))

        sep = '\n    '
        header = 'when evaluating table at %s, line %d' % (
            document.location, region.lineno)
        region.formatted = header + sep + sep.join(messages)
```

We can see how the results are formatted.

```
>>> format_table_errors(document)
>>> six.print_(region.formatted, end='')
when evaluating table at fake.txt, line 6
    fake.txt, line 11: expected True, got False instead.
    fake.txt, line 13: expected False, got True instead.
```

10.2.5 All Together Now

All the pieces (parsing, evaluating, and formatting) are available now, so we just have to put them together into a single "Manuel" object.

```
class Manuel(manuel.Manuel):
    def __init__(self):
        manuel.Manuel.__init__(self, [parse_tables], [evaluate_table],
            [format_table_errors])
```

Now we can create a fresh document and tell it to do all the above steps (parse, evaluate, format) using an instance of our plug-in.

```
>>> m = Manuel()
>>> document = manuel.Document(source_with_errors, location='fake.txt')
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted(), end='')
when evaluating table at fake.txt, line 6
```

(continues on next page)

(continued from previous page)

```
    fake.txt, line 11: expected True, got False instead.
    fake.txt, line 13: expected False, got True instead.
```

Of course, if there were no errors, nothing would be reported:

```
>>> document = manuel.Document(source, location='fake.txt')
>>> document.process_with(m, globs={})
>>> six.print_(document.formatted())
```

If we wanted to use instances of our Manuel object in a test, we would follow the directions in *Getting Started*, importing Manuel from the module where we placed the code, just like any other Manuel plug-in.

10.3 Fixed Bugs

Here are demonstrations of various bugs that have been fixed in Manuel. If you encounter a bug in a previous version of Manuel, check here in the newest version to see if your bug has been addressed.

10.3.1 Start and End Coinciding

If a line of text matches both a "start" and "end" regular expression, no exception should be raised.

```
>>> source = """\
... Blah, blah.
...
... xxx
... some text
... xxx
...
... """
>>> import manuel
>>> document = manuel.Document(source)
>>> import re
>>> start = end = re.compile(r'^xxx$', re.MULTILINE)
>>> document.find_regions(start, end)
[<manuel.Region object at ...]
```

10.3.2 Code-block Options

The code-block handler didn't originally allow reST options, so blocks like the one below would generate a syntax error during parsing.

```
class Foo(object):
    pass
```

```
import manuel.codeblock
m = manuel.codeblock.Manuel()
manuel.Document(source).parse_with(m)
```

10.3.3 Code-block options with hyphens

The code-block handler reST option parsing used to not allow for options with hyphens in their name, so blocks like this one would generate a syntax error:

```
class Foo(object):
    pass
```

```
import manuel.codeblock
m = manuel.codeblock.Manuel()
manuel.Document(source).parse_with(m)
```

10.3.4 Empty documents

While empty documents aren't useful, they are still documents containing no tests, and shouldn't break the test suite.

```
>>> document = manuel.Document('')
>>> document.source
'\n'
```

10.3.5 Glob lifecycle

Anything put into the globs during a doctest run should still be in there afterward.

```
>>> a
1
```

```
>>> b = 2
```

```
import manuel.doctest
m = manuel.doctest.Manuel()
globs = {'a': 1}
document = manuel.Document(source)
document.process_with(m, globs=globs)
```

The doctest in the *source* variable ran with no errors.

```
>>> six.print_(document.formatted())
```

And now the globs dictionary reflects the changes made when the doctest ran.

```
>>> globs['b']
2
```

10.3.6 zope.testing.module

At one point, because of the way manuel.doctest handles glob dictionaries, zope.testing.module didn't work.

We need a globs dictionary.

```
>>> globs = {'foo': 1}
```

To call the setUp and tearDown functions, we need to set up a fake test object that uses our globs dict from above.

```
class FakeTest(object):
    def __init__(self):
        self.globs = globs

test = FakeTest()
```

Now we will use the globs as a module.

```
>>> import zope.testing.module
>>> zope.testing.module.setUp(test, 'fake')
```

Now if we run this test through Manuel, the fake module machinery works.

> The items put into the globs before the test are here.
>
> ```
> >>> import fake
> >>> fake.foo
> 1
> ```
>
> And if we create new bindings, they appear in the module too.
>
> ```
> >>> bar = 2
> >>> fake.bar
> 2
> ```

```
import manuel.doctest
m = manuel.doctest.Manuel()
document = manuel.Document(source)
document.process_with(m, globs=globs)
```

The doctest in the *source* variable ran with no errors.

```
>>> six.print_(document.formatted())
```

We should clean up now.

```
>>> import zope.testing.module
>>> zope.testing.module.tearDown(test)
```

10.3.7 Debug flag and adding instances

The unittest integration (manuel.testing) sets the debug attribute on Manuel objects. Manuel instances that result from adding instances together need to have the debug value passed to each Manuel instances that was added together.

```
>>> m1 = manuel.Manuel()
>>> m2 = manuel.Manuel()
```

The debug flag starts off false…

```
>>> m1.debug
False
>>> m2.debug
False
```

…but if we set it add the two instances together and set the flag on on the resulting instance, the other one gets the value too.

```
>>> m3 = m1 + m2
>>> m3.debug = True
```

```
>>> m1.debug
True
>>> m2.debug
True
>>> m3.debug
True
```

10.3.8 TestCase id methods

Twisted's testrunner, trial, makes use of the id method of TestCase instances in a way that requires it to be a meaningful string.

For manuel.testing.TestCase instances, this used to return None. As you can see below, the manuel.testing.TestCase.shortDescription is now returned instead:

```
>>> from manuel.testing import TestCase
>>> m = manuel.Manuel()
>>> six.print_(TestCase(m, manuel.RegionContainer(), None).id())
<memory>
```

10.3.9 DocTestRunner peaks at sys.argv

A (bad) feature of DocTestRunner (and its subclass DebugRunner) is that it will turn on "verbose" mode if sys.argv contains "-v". This means that if you pass -v to a test runner that then invokes Manuel, all tests would fail because extra junk was inserted into the doctest output. That is, before I fixed it. Now, manuel.doctest.Manuel passes "verbose = False" to the DocTestRunner constructor which disables the functionality.

We can ensure that the verbose mode is always disabled by creating test standins for DocTestRunner and DebugRunner that capture their constructor arguments.

```
import doctest
import manuel.doctest
class FauxDocTestRunner(object):
    def __init__(self, **kws):
        self.kws = kws
try:
    manuel.doctest.DocTestRunner = FauxDocTestRunner
    manuel.doctest.DebugRunner = FauxDocTestRunner

    m = manuel.doctest.Manuel()

finally:
    manuel.doctest.DocTestRunner = doctest.DocTestRunner
    manuel.doctest.DebugRunner = doctest.DebugRunner
```

Now, with the Manuel object instantiated we can verify that verbose is off for both test runners.

```
>>> m.runner.kws['verbose']
False
```

```
>>> m.debug_runner.kws['verbose']
False
```