
Doctest and unittest... now they'll be merrily together

Olemis Lang (olemis@gmail.com)

Testing is a very important discipline to ensure and validate software quality. Python includes two standard modules to perform functional testing. Prior to version 2.4 both tools were unrelated, leading to scattered testing code. From this version on a `unittest` API is provided by `doctest`. The present work aims to propose some enhancements to this API in order to achieve a better `unittest-doctest` integration. Although there are more complex testing tools (e.g. `nose`) which allow performing both kinds of tests, they are outside the scope of this article. Firstly, they are not standard modules. Besides, the intent is to load and execute doctests just like if we were using `unittest`.

Even though many types of tests exist these days⁴⁶, functional testing is very important. Firstly, it validates whether software behavior matches the business rules documented in the software requirements. Besides, for continuously evolving systems and iterative development processes it is also crucial to perform regression testing. Thereby introduced defects are handled as soon as possible and the defect does not propagate to future versions. These tests are also the main building block for test-driven development and extreme programming techniques. A scripting language like Python ought to be aware of this since it is often used to write test scripts (e.g. for Java⁴⁷ and .NET⁴⁸). Because of this, the paper also covers the standard modules available nowadays in Python for functional testing.

Outline

Before explaining the whole new solution to integrate both major testing frameworks for Python, it will be helpful to talk about them separately. This can be also useful for pythoneers wishing to get a fuller understanding of the options available these days. Nevertheless, the explanation will cover only the features needed to understand the work done. Afterward they are compared so as to illustrate the need and the idea leading to their integration in version 2.4. Its usage will also be portrayed, thereby clarifying the strongest as well as the weakest points inherent to this new feature the way we know it today. These facts help to establish a motivation for the proposition presented thereafter. Firstly the novel implementation is discussed in detail. This can be useful for developers, and all those aiming to understand how an object oriented API can merge both these frameworks. Next some useful use cases are explained. They are helpful for testers because therein they will find guidelines to face some testing scenarios. All the way through the emphasis made on object orientation as well as other distinctive features are explained, and compared with respect to the current standard.

Functional tests with doctest

The standard module `doctest` was included in Python 2.1. It is perhaps the most intuitive way available in this language to write functional tests. In order to gain a deeper insight about this framework it is important to know how tests are written. Once this is fully understood, the next step is to know the API which allows running tests. Finally, it is relevant to know the elements included in the framework in order to execute one test after another, and report the results. Let's briefly talk about these topics.

46 Several types of tests are described comprehensively in G. D. Everett, R. McLeod, Jr "Software testing : testing across the entire software development life cycle" (2007) John Wiley & Sons, ISBN 978-0-471-79371-7.

47 See M. Nadel "Use Jython to build JUnit test suites" available on-line at <http://www.ibm.com/developerworks/java/library/j-jythtest.html>.

48 A. Henderson "Integrating NUnit & IronPython..." link available at <http://ironpython-urls.blogspot.com/2006/10/integrating-nunit-and-ironpython.html>.

Specifying tests with `doctest` is very easy as illustrated in Figure 1. They reside in docstrings and therefore are expressed in textual form. This means that any element able to be explicitly documented can contain such tests. Usually the documentation for one such element includes only the tests needed to ensure that its implementation fulfills its expected behavior. The syntax involved to declare them resembles interactive sessions with the Python interpreter. In fact, coders can copy the characters outputted to the console during one such session and paste them into docstrings. That would be enough to specify the tests. But in real development it is often better, and more useful, to write the test before actually implementing the code behind it. However, the former characteristic is what causes using `doctest` to feel intuitive, because every Python programmer have interacted with the interpreter. This means that anyone knowing the language can write tests.

```

1. def shuffle(seq):
2.     """
3.     >>> seq = range(10)
4.     >>> shuffle(seq)
5.     >>> seq.sort()
6.     >>> seq #doctest: +NORMALIZE_WHITESPACE
7.     [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
8.     """
9.     # code omitted
10.
11. def sample():
12.     """
13.     >>> seq = range(10)
14.     >>> sample(seq, 20) # doctest: +ELLIPSIS
15.     Traceback (most recent call last):
16.     ...
17.     ValueError: sample larger than population
18.
19.     >>> [x in seq for x in sample(seq, 5)]
20.     ... #doctest: +NORMALIZE_WHITESPACE
21.     [True, True, True, True, True]
22.     """
23.     # code omitted
24. def choice(seq):
25.     """
26.     >>> seq = range(10)
27.     >>> elem = choice(seq)
28.     >>> elem in seq
29.     True
30.     """
31.     # code omitted
32.
33. # Running doctests
34. # from the command line
35.
36. if __name__ == '__main__':
37.     import doctest
38.     doctest.testmod()

```

Figure 1: Testing functions in standard `random` module with `doctest`.⁴⁹

After the tests are written, programmers must invoke one of the functions defined inside the `doctest` module in order to execute the tests. The one most frequently used for this purpose is `testmod`, but it is also possible to use either `run_docstring_examples` or `testfile`. So... what happens immediately after they get called? The first thing that takes place after such call is test hatching. This process is usually divided into two stages. When we test modules (`testmod`), `DocTestFinder` instances are created. Their role is to enumerate the functions (methods) and classes reachable from the module specified as a parameter, and also belonging to it. For each such object found, `DocTestFinder` instances extract its docstrings and trigger the second stage: parsing. If the tests reside in text files (`testfile`) or if a single object is to be tested (`run_docstring_examples`) then parsing is started right away since there is no need to use `DocTestFinder` objects. In the first case the contents of the text file will be parsed, whereas in the second one the string bound to the object's `__doc__` attribute will be considered instead.

Then you may ask... why do we need parsing in this case? Well, mainly because tests are intermingled with documentation, most of which is usually written in natural language. That's why the outcome of the previous step are the input to `DocTestParser` objects. Their duty is to separate the text representing interactive sessions from intervening text. Interactive sessions look like a collection of statements inputted by the programmer followed by the result outputted by the interpreter in consequence of its execution. The instances of the `Example` class encapsulate such information (statement to execute + expected interpreter output). `DocTestParser` objects are thus also responsible for creating the `Example` instances

⁴⁹ Throughout the text the different frameworks are compared against versions of the test code provided in <http://www.python.org/doc/2.5.1/lib/minimal-example.html>.

representing the interactive sessions they found.

After all parsing has been performed, the resulting `Example` objects are put together into a container. Actually `DocTest` objects are liable for that, besides referencing a namespace (global namespace) which will be used in the next phase: test execution.

The processing class used to execute and verify the interactive examples in a `DocTest` is `DocTestRunner`. In order to do so, it uses an output handler (a callable object). Firstly, instances of the later class notify that testing is about to start. Next, they extract the `Example` objects included in the incoming `DocTest`. Each example is later on compiled. `DocTestRunner` requests the interpreter to execute them in the context of the namespace held by the `DocTest` instance under testing. The requesting object also gathers the interpreter output. At this time `OutputChecker` instances get into action.

`OutputChecker` instances compare the aforesaid output with the expected result (contained in `Example` objects). If both of them match, success is reported by the runner. If they do not match then failure notification occurs. A third case is still possible if an unexpected exception is raised during the execution. All these situations are reported by the aforementioned output handler, `sys.stdout.write` being the default one.

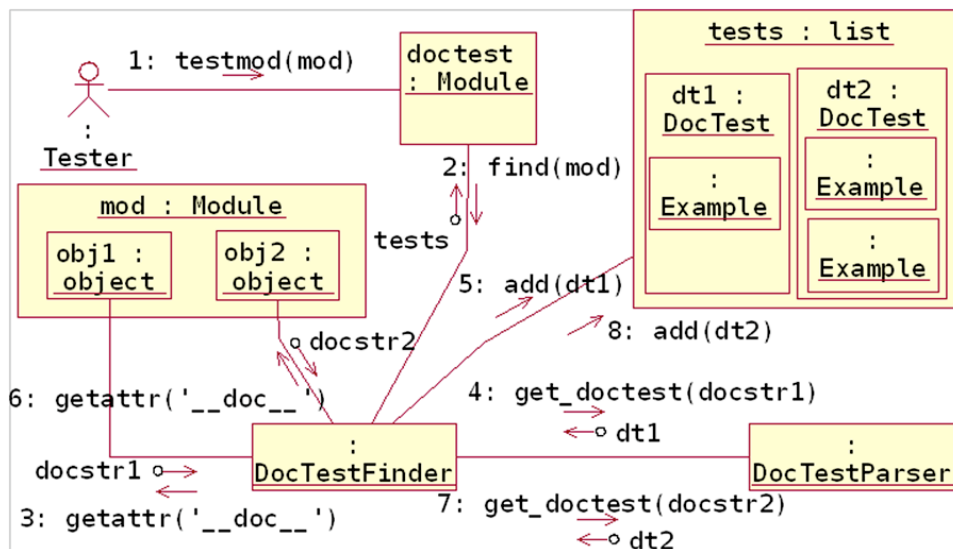


Figure 2: Retrieving `DocTest` instances from module docstrings.

The term unexpected exception was used before because by using `doctest` it is also possible to specify that the expected behavior is an exception to be raised. The documentation for function `sample` in Figure 1 shows how this is done. This kind of tests follows the same philosophy of being similar to interactive sessions. However, in case of exceptions the interpreter also outputs some fields which depend on the execution context and can vary from time to time. The use of ellipsis is compulsory under these conditions. They are wildcard characters which tell output checkers that a portion of the resulting string has to be ignored because either it is not important for the check to perform, or perhaps because this segment can vary from one time to another. Ellipsis ought to be used in together with “boolean” *option flags*. The comments in lines 6 and 14 in Figure 1 portray how to set them. They allow to customize how `OutputCheckers` verify the examples and the contents of the subsequent report. A `DocTestRunner` can use another output checker so there is space to write tests using a different syntax which supports user needs.

Assessing doctest

A direct consequence of the aforesaid layout for tests is that they can be considered as a behavioral description of the element which is to be tested. In this case, the same language used to code the feature is used to sketch its behavior, so there is no need for further

translations to get executable testing artifacts. The tests also tell us explicitly how to use the different functions, classes, methods and so on. It is also clear what should be expected in return. Since all this information is located in docstrings, the documentation of these objects is enhanced. Example usage is available, being possible to record it in the form of a simple tutorial combining source code written in Python, together with explanations written in natural language. This is practical because “sometimes a single line of code is better than one thousand words”.

Another benefit of `doctest` is its encapsulation. Coders need not to know what happens behind the scene so as to test the different examples. This contributes a lot to make `doctest` a user-friendly tool. On the other hand, its results are truthful since it relies on the interpreter to execute the examples. So whatever you see happening during the test is what you actually get once the statements in the docstrings are executed in a real scenario. This could be the case when other coders use the library code being checked to build their own systems.

However some enhancements to `doctest` are still possible. For example, `DocTest` class is meant to represent a *Composite* but is not implemented considering this pattern. This implies that it's not easy to put together different instances of this class, as well as instances of other classes representing other kinds of functional tests. This also means that it could be difficult or unnatural to mix in the examples found in different modules in order to test them altogether. In this case scattered test reports could be obtained while testing a populated package, for example. Furthermore:

“test setup has to be either copied or hidden away from the test, making the overall environment harder to understand.”⁵⁰

Another valuable characteristic is that different subclasses of those used by `doctest` in test execution can allow for customization of the whole process. However the API supplied for this purpose restrains this flexibility and, for example, only allows sending reports to the standard output. This means that test upshots for individual examples are intermingled. Therefore if they were of interest, `sys.stdout` needs to be redirected perhaps via `StringIO`. Besides further analyses are made difficult since test reports ought to be parsed.

PyUnit tests in action

The module `unittest` is included among the standard modules since version 2.1. It inherits a long tradition started by Kent Beck's *Smalltalk testing framework* (a.k.a. *SUnit*), and followed by others like *JUnit* (Java), *CppUnit* (C++), *JSUnit* (JavaScript), *HUnit* (Haskell), *mUnit* (MATLAB), *utMySQL* (MySQL), *NUnit*, *csUnit* (.NET programming languages), and many other implementations for at least 51 programming languages of different nature. Python could not be the exception.

The notion of *test cases* is central to the testing process defined by *xUnit* frameworks. In a simplistic way they could be viewed as the most atomic unit of testing. Broadly speaking, they define the features under test and how to carry out the assessment. The most naive way to implement a test is to extend the `TestCase` class and redefine its `runTest` method. Every statement to check has to be asserted using the methods available in the prior base class. If no such assertion fails then the test succeeds; otherwise, it is reported as a failure. Failures represent anticipated problems. Once again a third case is possible when an unanticipated exception (one not caught by `assertRaises` method) is raised while conducting the test. These conditions are reported as errors, and are more catastrophic since they are unchecked bugs.

Even a simple application can demand several functional tests. That's the reason for having the `TestSuite` class. Its instances group the test cases being part of a testing scenario. These suites are implemented according to the Composite pattern. Therefore they can contain other suites, as well as separate test cases. Infinite nesting levels are possible in theory. A test suite can be built explicitly by hand. Nonetheless, in real life there will be several

⁵⁰ Wikipedia page for `doctest`, available on-line at <http://en.wikipedia.org/wiki/doctest>.

classes whose methods define tests over the target application. Suites construction would be very tedious in these situations. Therefore `unittest` includes the `TestLoader` class. Its objects automatically gather the tests defined within a module or class. In modules containing many `TestCase` descendants, the tests found within each such class are returned wrapped in a `TestSuite`. Well, what happens once loaders receive classes as input?

Table 1: *TestCase* methods to check for and report failures.

Method Name	Description
<code>fail</code>	Signals a test failure unconditionally.
<code>assert_</code>	Signals a test failure if an expected condition is not met. A message describing the failure can be supplied.
<code>assertEqual</code>	Test that two values are equal. If not, the test will fail with a given explanation.
<code>assertNotEqual</code>	Test that two values are not equal. If they do compare equal, the test will fail with a given explanation.
<code>assertAlmostEqual</code>	Used to test for equality of two instances of inexact types like float. An explanation may be given.
<code>assertNotAlmostEqual</code>	Used to test if two instances of inexact types differ so much as to be considered different. An explanation may be given.
<code>assertRaises</code>	Test that an exception of a given type (or any of a group of exceptions) is raised when a callable is invoked with known positional and keyword arguments.
<code>failIf</code>	Signals a test failure if an abnormal condition is met. A message describing the failure can be supplied.

The fact is that `unittest` supports test fixtures. In practice, as software evolves testers find groups of similar test cases. Often they require the same initialization and cleanup. If this is the case a single `TestCase` inheritor can contain multiple test methods. When a test loader is about to build a suite out of a `TestCase` descendant, it examines whether it contains any method whose name starts with the “test” prefix like shown in Figure 3 lines 10, 18, and 22. If this is not the case, only `runTest` is executed as explained before. Otherwise, for each method the loader creates one instance of the class in order to execute the former. The method name is bound to the `_testMethodName` attribute of this instance. All the objects thus created are collected into a test suite.

Having nothing but a test suite is not enough. Its test cases should be executed. Test runners are used for this purpose. They decide how to test the suite and what to do with test results. `PyUnit` incorporates the class `TextTestRunner` in order to output the results in textual form to a file-like object (`sys.stderr` by default).

The first things runners do is to create an instance of the class `TestResult`. Later, `TestCase` instances contained into the target test suite are processed one by one. Immediately before each test is actually performed, the test case's `setUp` method gets invoked. Since test cases derived from the same fixture share this method, the common initialization steps should be coded therein. Afterward, the test method bound to the test case instance gets called. The framework monitors whether a failure, an error or a successful test occurs. In any case, the result is stowed into the `TestResult` object created for this run. Immediately after the test method has been called and the result recorded, the `TestCase.tearDown` method gets executed. Since instances of the same fixture also share this method, it can be used to release the resources allocated from within `setUp`.

```

1. from random import shuffle,
2.     choice, sample
3. import unittest
4.
5. class TestSequenceFunctions(
6.     unittest.TestCase):
7.
8.     def setUp(self):
9.         self.seq = range(10)
10.
11.    def testshuffle(self):
12.        # make sure the shuffled
13.        # sequence does not lose
14.        # any elements
15.        shuffle(self.seq)
16.        self.seq.sort()
17.        self.assertEqual(self.seq,
18.            range(10))
19.
20.    def testchoice(self):
21.        element = choice(self.seq)
22.        self.assert_(element in self.seq)
23.
24.    def testsample(self):
25.        self.assertRaises(ValueError,
26.            sample, self.seq, 20)
27.        for elem in sample(self.seq, 5):
28.            self.assert_(elem in self.seq)
29.
30. if __name__ == '__main__':
31.     unittest.main() # Command-line test
32.
33. def explicit_test_run():
34.     # Finer level of control to run tests
35.     loader = unittest.TestLoader()
36.     suite = loader.loadTestsFromModule(
37.         sys.modules[__name__])
38.     unittest.TextTestRunner(
39.         verbosity=2).run(suite)

```

Figure 3: Writing `unittest` test cases for the `random` module.

Assessing `unittest`.

Testing frameworks like `unittest` are quite popular since long time ago. One of the reasons behind this success is perhaps that it shows a mature object design supported by a high pattern density. Only the `TestCase` class is involved in at least four design patterns. This implies that this tool is easier to use, but harder to change. For instance, test suites allow different kinds of test cases to be tested altogether. Test cases of variate nature can dwell inside a single suite. A larger group can be formed after appending this same suite to another one perhaps containing other arbitrary suites. Regardless of their possibly different nature, they all are attached and tested the same way. Consequently it is possible to say that `unittest` encourages easy assembly and smooth integration.

Another key feature is the code reuse made possible thanks to an object oriented API. This is possible mainly to a deep separation of concerns among test retrieval, test procedure, test execution, and finally result gathering for later analysis. Firstly, this means that custom test loaders can load test cases from diverse sources, can be represented in different formats, or even follow different conventions. Separately, the system under test can be assessed in many different ways. To do so it is only necessary to add new test methods to `TestCase` descendants. Besides the testing process may be performed in dissimilar manners without interfering with the test code that actually checks the target system. To illustrate this point let's consider the example of the peer library `JUnit`. It is possible to run the same test suites reporting the outcomes in text mode via `junit.textui.TestRunner`, or graphical mode via one of `junit.awtui.TestRunner` or `junit.swingui.TestRunner`. Examples of special runners are those used by IDEs (e.g. Eclipse) to represent a test run in their interface.

It is also possible to customize the way test outcomes are stored by using personalized `TestResult` subclasses. Therefore besides volatile storage, either RDBMS, ORM, proprietary files, or anything else can be used for this purposes. A test repository being part of the project measurements could hence be deployed. The data gathered this way might give support to test analyses which can illustrate continuous displays and evolution of project status, the capacity to progress towards goals, and the efficacy of the development process. Enterprises interested in moving their CMM⁵¹ level up, can take advantage by automating key process areas from levels 2 (*Repeatable*) to 5 (*Optimizing*).

Nonetheless people usually spends far more time reading test code than actually writing it. That's why the challenge is writing readable tests⁵². In this respect `unittest` code can be hard to understand, demanding from the reader previous knowledge about the framework. Test

⁵¹ Capability Maturity Model

⁵² J. Fulton, T. Peters "Literate Testing: Automated Testing with doctest" (2004), *PyCon 2004*.

code is usually separated from source code, which can possibly difficult this task even more. This testing toolkit by itself makes no contribution to software documentation either. Another controversial topic is the way exceptions are asserted. Maybe this is the most notorious case illustrating that test code does not look like client code⁵². In this respect `doctest` seems to be more natural.

The gathering

Before Python 2.4, `doctest` included the `Tester` class. It provided simple means to combine the doctests retrieved from different modules (e.g. a package), and test them thoroughly. From version 2.4 and on, this class has been deprecated. Now it is clear that both frameworks complement each other. The weaknesses of the former turn into strengths of the later, as explained before. At the moment, the `unittest` API provided by `doctest` makes possible to create test suites from modules and text files containing doctests. The later can be combined with tests from multiple sources. Consequently `unittest` runners can run them altogether at once.

```

33. # Override the statements in Figure 1 from line 33 on.
34. import doctest, unittest
35.
36. if __name__ == '__main__':
37.     unittest.main(defaultTest='suite')
38.
39. def suite():
40.     return doctest.DocTestSuite(sys.modules[__name__])
41.
42. def run_tests():
43.     # Finer level of control to run tests
44.     unittest.TextTestRunner(verbosity=2).run(suite())

```

Figure 4: Verifying doctests using the standard `unittest` API.

Suites are created from doctests out of modules via the `DocFileSuite` function. It accepts the `setUp` and `tearDown` optional parameters. Both should be bound to a function object. In this case a `DocTest` object is built as formerly explained with the help of `DocTestFinder`. Next it is wrapped by an instance of `DocTestCase` (a `TestCase` descendant) and a regular suite containing it is returned. During this process the aforementioned parameters are bound to attributes of the new `DocTestCase` instance. Besides the `unittest`-oriented API consents testers to specify custom instances of `DocTestParser` to extract `doctest Examples` out of docstrings. Personalized `OutputCheckers` are welcomed as well in order to match differently the interpreter output against the expected result. They both are supplied in the form of keyword arguments to `DocFileSuite`. This feature is definitely an enhancement over the preceding API.

Running the new test case is a process which reuses the prior `DocTestRunner` class. However, before the execution the function supplied in the `setUp` argument is invoked with the wrapped `DocTest` instance as its sole parameter. After the test is performed the same happens with the priorly mentioned `tearDown` argument, thereby imitating test fixtures. The outcome given by the `doctest` runner is stored into a character string through `StringIO` objects. If success was not accomplished, a failure is reported to the `unittest` runner carrying out the global test. The `doctest` details are provided as the descriptive message. That's why all the issues highlighted for `DocTestRunner` reports are also valid in this context. Eventually both `unittest` and `doctest` formats will be interleaved. This could be annoying. Long reports might be confusing especially in view of the fact that multiple summaries are made. Moreover the number of individual `doctests` that failed or behaved erroneously are not considered for the final statistics reported by the `unittest` runner. A single failure abbreviates them all.

The function `DocFileSuite` rescues us when doctests lie within text files. The whole procedure is very similar to the one already explained. A characteristic common to both these functions is that they have extensive signatures. The API itself lacks on object orientation, and is not compliant with `unittest` test loaders. The main difference is that `DocFileCase` objects are used instead of `DocTestCase`. Nothing new happens in practice since the former only overrides cosmetic features of the later.

One object oriented API to join them

After analyzing the former arguments the focus turned out to evolve the API available to retrieve test cases from doctests. This new interfaces aims to allow programmers to write doctests the same way they have done so far, but handle the tests like `unittest` users do. Hence it mostly reuses both frameworks. It also overrides the classic `doctest` interface and the one given for Python 2.4, but reuses important implementation details. So they can be considered as a useful facilitator for this work.

Table 2: Mapping from `unittest` integration API items to `doctest`'s.

doctest	Standard unittest API	OO unittest API
Example		<code>DocTestCase</code>
<code>DocTest</code>	<code>DocTestCase</code>	<code>DocTestSuite</code>
<code>DocTestFinder</code>	<code>DocTestSuite</code> function	<code>DocTestLoader</code>
<code>DocTestRunner</code>	<code>DocTestRunner</code>	<code>_Doc2UnitTestRunner</code>
<code>testmod</code> function	<code>unittest.TextTestRunner</code>	<code>unittest.TextTestRunner</code>

The main variation introduced with respect to the 2.4 version is how legacy `doctest` classes map to `unittest` concepts. Since one goal was to gather separately the information resulting from testing individual `Example` instances (i.e. single statements), the unit of testing could be no longer bound to `DocTest` object like before. Rather than this, the target for new test cases are `Example` objects. Let's explain the whole in more detail.

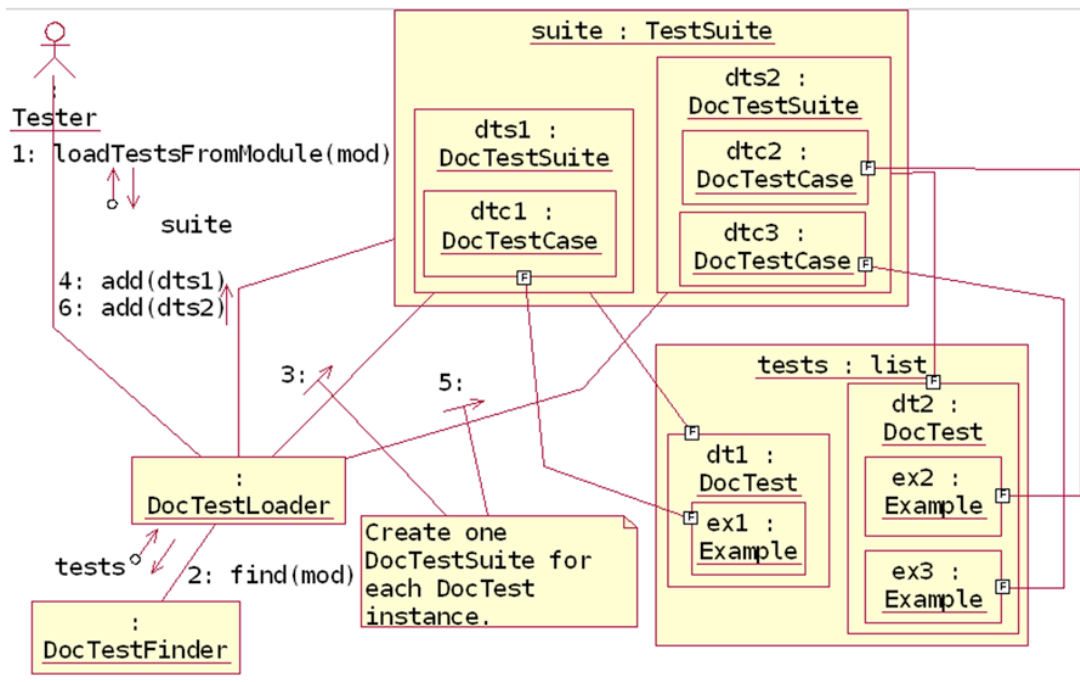


Figure 5: Retrieving test cases from doctests with loaders.

A radical new feature is that test cases are no longer loaded through functions. They have been replaced with the class `DocTestLoader`. It is a novel test loader introduced to achieve better object orientation and conform to `unittest` rules. The loading process starts when legacy `DocTestFinder` functionality is reused. And here we have the first feature contributing to flexibility. The type of finder to use for this purposes can be specified when loaders are created. The subsequent step is to wrap the resulting `DocTest` instances with specialized suites and group the later into the `TestSuite` object returned by the loading process. The aforementioned specialized suites are represented by the also new `DocTestSuite` class and its

descendants. And yes, since the suite type used in practice is bound to loaders' `docTestSuiteClass` attribute, subtypes of `DocTestLoader` can override this value and instantiate some other suites. Testers can thereby introduce their own features to meet particular needs. This solution is inspired in the usage given to the `suiteClass` attribute in `TestLoader` class.

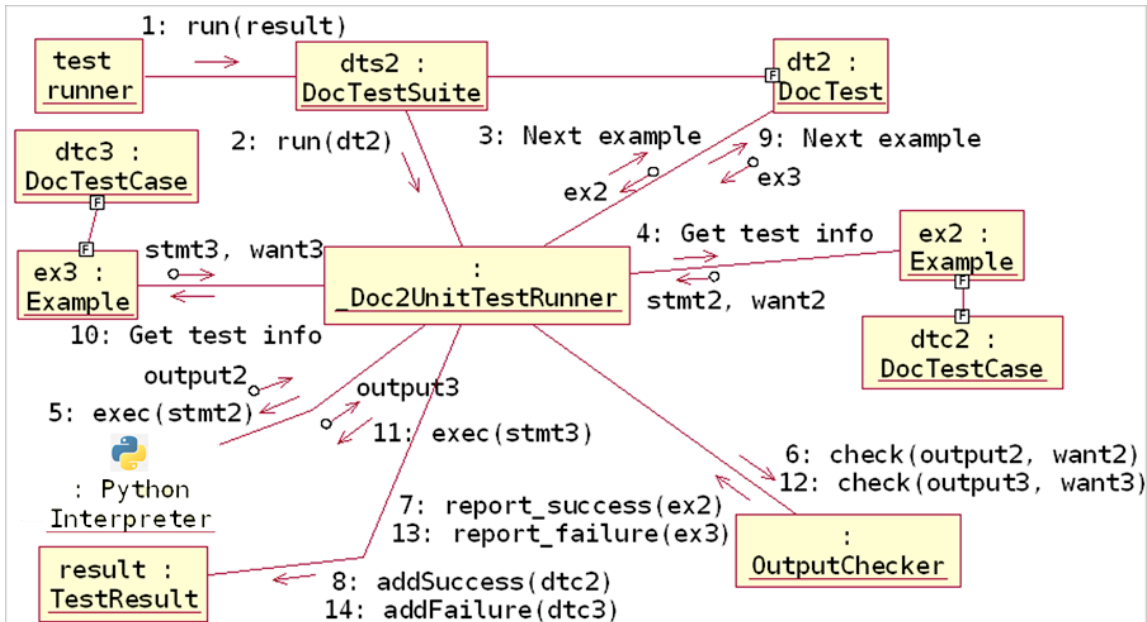


Figure 6: The custom-made `doctest` run yields more precise reports.

`DocTestSuite` class has a determining role in the integration and acts like `DocTest` peer. It maps a `unittest` run carried out by an enclosing `unittest` runner to the run needed to verify `doctest` examples. Thus It matches the *Adapter* pattern, being `DocTestRunner` the *adaptee*. In the background a tailored `DocTestRunner` descendant (`_Doc2UnitTestRunner`) executes and verifies interactive examples. Nonetheless before doing so, the suite instantiates a test case for each `Example` contained within the `DocTest` wrapped by itself. A bidirectional association is established among both examples and test cases. The novel `DocTestCase` class is used by default. Once again it is possible to supersede this decision by overriding the type object (i.e. `DocTestCase` subtype) bound to the `docTestCaseClass` attribute of the suite.

Next, when the interactive examples are executed and verified, the *Adapter* pattern is also employed to hook `_Doc2UnitTestRunner` report methods. This allows to record the outcomes in `TestResult` objects. Testers can refine the default runner by overriding the type object (i.e. a subtype of `_Doc2UnitTestRunner`) bound to the `docRunnerClass` attribute of the suite (preferably by sub-classing `DocTestSuite`).

Use cases

So far the focus has been placed in explaining how the different classes collaborate to achieve the desired goal. Let's dedicate some time to illustrate how to use the novel API.

Basic usage

There are some changes with respect to running tests by using classic `doctest`, Python 2.4 `unittest` API, and the current solution. Assuming the same functions from line 1 to 32 in Figure 1 have been declared, Figure 7 shows how doctests are run with the object-oriented API. Observe there is almost no difference with respect to `unittest` usage.

```

33. # Override the statements in Figure 1 from line 33 on.
34. import doctest
35.
36. if __name__ == '__main__':
37.     doctest.main() # Command-line test
38.
39. def run_tests():
40.     # Finer level of control to run tests
41.     suite = doctest.DocTestLoader().loadTestsFromModule(sys.modules[__name__])
42.     unittest.TextTestRunner(verbosity=2).run(suite)

```

Figure 7: Using the object oriented API to run tests.

Using optional doctest features

Anybody can wonder “How can I control doctest's behavior via option flags?”. The initializer in `DocTestLoader` accepts the extra keyword arguments shown in Figure 8. These arguments allow the use of tailored output checkers, as well as the use of legacy `doctest` options. All these parameters flow from `DocTestLoader` to `DocTestSuites`, where they are stored. At test runtime, the `_Doc2TestUnitRunner` object involved employs them to perform the `doctest` run. Besides the keyword arguments for option flags, it is also possible to supply in to the initializer a `dict` to be used as the globals when executing examples. Parameterized doctests are also possible by supplying in another `dict` to be merged into the globals used to execute examples.

```

1. import doctest, doctest
2. from __future__ import CO_FUTURE_WITH_STATEMENT
3.
4. loader = doctest.DocTestLoader(
5.     DocTestFinderSubClass(), # compatible with previous DocTestFinder subtypes
6.     {'glob1': 1, 'glob2': 2, 'glob3': 3}, # globals used when executing examples
7.     {'extra1': 1, 'extra2': 2}, # extra globals to execute examples
8.     optionflags = doctest.REPORT_UDIFF, # doctest options flags to use
9.     checker = MyOwnCheckerClass(), # override how examples are verified
10.    runopts = dict(
11.        compileflags = CO_FUTURE_WITH_STATEMENT, # options to compile examples
12.        clear_globals = True) # clear global namespace after testing
13. )

```

Figure 8: Specifying doctest optional features.

Combining test cases and doctests

“What if my module contains both doctests and test cases?” Thanks to the object oriented nature of the present API, a simple solution is at hand. Firstly, we need a loader whose purpose is to retrieve various types of tests and assemble them into a test suite. An implementation resembling the *Chain of Responsibility* pattern is shown in Figure 9c. An instance of this loader holding a legacy `TestLoader` and a `DocTestLoader` does what we need.

Since testers might frequently face this situation in practice, the `MultiTestLoader` class has been included into the API. It is important to notice that a single step is needed to set up the testing scenario. Besides this all happens like we are used to with `unittest` loaders. Hence this style encourages uniformity. The class `MultiTestLoader` is extremely reusable, even in other contexts.

```

1. def choice(seq):
2.     """
3.     >>> seq = range(10)
4.     >>> elem = choice(seq)
5.     >>> elem in seq
6.     True
7.     """
8.     # code omitted
9.
10. def shuffle(seq):
11.     # code omitted
12.
13. def sample():
14.     """
15.     >>> seq = range(10)
16.     >>> sample(seq, 20)
17.     ... #doctest: +ELLIPSIS
18.     Traceback (...):
19.     ...
20.     ValueError: ...
21.     >>> [x in seq for x in \
22.     ...     sample(seq, 5)]
23.     ... #doctest: +NORMALIZE_WHITESPACE
24.     [True, True, True, True, True]
25.     """
26.     # code omitted
27.
28. class TestSequenceFunctions(
29.     unittest.TestCase):
30.
31.     def setUp(self):
32.         self.seq = range(10)
33.
34.     def testshuffle(self):
35.         shuffle(self.seq)
36.         self.seq.sort()
37.         self.assertEqual(
38.             self.seq, range(10))
39.
40.     def testsample(self):
41.         self.assertEqual([],
42.             sample(self.seq, 0))

```

a) Writing the tests

```

43. import unittest
44. from doctest import DocTestLoader,
45.     main, MultiTestLoader
46.
47. loaders = [unittest.defaultTestLoader,
48.            DocTestLoader()]
49.
50. if __name__ == '__main__':
51.     main(testloader=
52.         MultiTestLoader(loaders))
53.
54. def run_tests():
55.     # Finer level of control to run tests
56.     loader = MultiTestLoader(loaders)
57.     suite = loader.loadTestsFromModule(
58.         sys.modules[__name__])
59.     unittest.TextTestRunner(
60.         verbosity=2).run(suite)

```

b) Running doctests and test cases altogether

```

1. class MultiTestLoader(
2.     unittest.TestLoader):
3.     def __init__(self, loaders= []):
4.         self.loaders= loaders
5.
6.     def loadTestsFromModule(self,
7.         module):
8.         return self.suiteClass(
9.             [loader.loadTestsFromModule(
10.                 module) for loader in
11.                 self.loaders])
12.     # further code omitted

```

c) A loader to retrieve different kinds of tests

Figure 9: Asserting doctests and test cases found in a single module.

Defining fixtures

The concept of fixtures pioneered by *xUnit* frameworks can be used to hide away test setup code from docstrings, thereby obtaining more concise documentation. Nevertheless at the same time readability might be jeopardized at some extent. Since test cases work at the example level in this solution, the legacy `setUp` and `tearDown` methods are executed respectively before and after the interpreter executes each example.

The current `doctest unittest` API behaves differently. It calls fixture methods once before and after asserting all the examples. However the same behavior can be obtained using test cases implemented with test patterns like *Shared Fixture* and *Chained Tests*⁵³.

⁵³ Test patterns are presented in G. Meszaros "XUnit test patterns : refactoring test code" (2007), Addison-Wesley, ISBN 0-13-149505-4.

```

13. """
14. range(10) is assigned to seq before
15. executing each statement.
16.
17. >>> shuffle(seq); seq.sort(); seq
18. ... #doctest: +NORMALIZE_WHITESPACE
19. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
20.
21. >>> sample(seq, 20) # doctest: +ELLIPSIS
22. Traceback (most recent call last):
23. ...
24. ValueError: sample larger than population
25.
26. >>> [x in seq for x in sample(seq, 5)]
27. ... #doctest: +NORMALIZE_WHITESPACE
28. [True, True, True, True, True]
29.
30. >>> choice(seq) in seq
31. True
32. """

```

a) Concise docstrings

```

33. from doctest import main,
34.     DocTestCase, DocTestSuite,
35.     DocTestLoader
36.
37. class RandomTestLoader(
38.     DocTestLoader):
39.     class doctestSuiteClass(
40.         DocTestSuite):
41.         class docTestCaseClass(
42.             DocTestCase):
43.             def setUp(self):
44.                 exec 'seq = range(20)' in \
45.                     self._dt.globs
46.
47. # self._dt returns the DocTest
48. # object containing the Example
49.
50. if __name__ == '__main__':
51.     main(loader=RandomTestLoader())

```

b) Compact fixture code

Figure 10: Per example preparation and cleanup actions.

Conclusions

In 2007 Python has been considered by TIOBE as the language of the year⁵⁴. According to the same source in January 2008 it has scaled up to the sixth place among the most popular programming languages. It is also considered as the glue language by excellence, and the community behind it is undoubtedly healthy. Besides being a recognition to the work made by many since years ago, all these arguments are so moving for new developers captured by its beauty. Nonetheless there is still a place for enhancements.

The standard module `doctest` is one example of such a beauty, whereas `unittest` is a typical case of strength, flexibility and stability. The later is full of design patterns, and sustains a large number of testing patterns. The former reflects the strong support provided for meta-programming in Python. It is related to many well-known *xUnit* patterns, especially test automation patterns (e.g. *Data-Driven Test*, *Recorded Test*, *Scripted Test*), and result verification patterns (e.g. *Behavior Verification*, *State Verification*, *Delta Assertion*)⁵³.

The main ambition of the present work is to run the `doctest` machinery while performing `unittest` runs. This has been accomplished after interleaving a layer which reconciles their respective interfaces. Given the object oriented nature of the solution, it is not bizarre that core classes be subjects of the *Adapter* pattern. Notably, `_Doc2UnitTestRunner` takes part twice. It is also the main gateway between `doctest` and `unittest` in our solution. This confirms the fact that pattern density gets higher around key classes. As already said, the novel API also allows the use of many *xUnit* testing patterns while testing doctests.

Perhaps the strongest arguments in favor of this solution are related to its contribution to automated test analysis. This discipline is very important because it is a powerful indicator of a project's progress towards its goals. The number of attempted test cases over time highlights how effective the testing activities perform. Otherwise it could be found that they do not behave accordingly to the test plan. Test analysis can be helpful to adjust schedules, assign tasks, track defects, prioritize goals, monitor the development process, discover root causes, and many other dissimilar activities. In all cases, the main input consists of test results. The process of obtaining detailed information via the current `unittest` API is more complex in view of the fact that a full `doctest` report is stored. First, we need to extract the report from a `TestResult` instance. Next, useful information is retrieved through parsing. In our case, the same can be done by directly inspecting `TestResult` objects. This could ease tasks contributing to CMM key process areas like *Software Project Tracking and Oversight*,

⁵⁴ News found at TIOBE's home page <http://www.tiobe.com/>.

Software Quality Assurance (Repeatable level), Peer reviews, Software Product Engineering, Organization Process Focus, Intergroup Coordination (Defined level), Quantitative Process Management, Software Quality Management (Managed level), and finally Defect Prevention (Optimizing level).

```
F..F.FF
=====
FAIL: testshuffle (__main__.TestSequenceFunctions)
-----
Traceback (most recent call last):
  File "<stdin>", line 8, in testElems
AssertionError: [0, 1, 3, 4, 5, 6, 7, 8, 9] != [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

=====
FAIL: __main__.choice (line 5)
-----
AssertionError: Failed example:
  elem in seq
Expected:
  True
Got:
  False

=====
FAIL: __main__.sample (line 4)
-----
AssertionError: Failed example:
  sample(seq, 20) # doctest: +ELLIPSIS
Expected:
  Traceback (most recent call last):
    ...
  ValueError: sample larger than population
Got:
  []

=====
FAIL: __main__.sample (line 9)
-----
AssertionError: Failed example:
  [x in seq for x in sample(seq, 5)]
  #doctest: +NORMALIZE_WHITESPACE
Expected:
  [True, True, True, True, True]
Got:
  []

=====
Ran 7 tests in 0.015s

FAILED (failures=4)
```

Figure 11: Test report obtained with the object-oriented [unittest](#) API.

The implementation also contains plenty of interesting ideas, even for cosmetic features. For example, Meyer's *principle of uniform access* was exercised while coding [DocTestCase](#). Built-in [property](#) objects assisted in hiding the complex computation to derive test method's names from the corresponding [Example](#)'s attributes.

Acknowledgments

The thorough reviews made by the editors were crucial to enhance the contents and catch some mistakes. Their dedication and timeliness were highly valuable. The emphasis made on readability while writing the code snippets, is mainly due to Medardo Rodríguez. Moreover, the anonymous reviewers offered concise and very helpful comments.