Literate Testing:
Automated Testing with doctest

Jim Fulton, jim@zope.com
Tim Peters, tim@zope.com

PyCon 2004

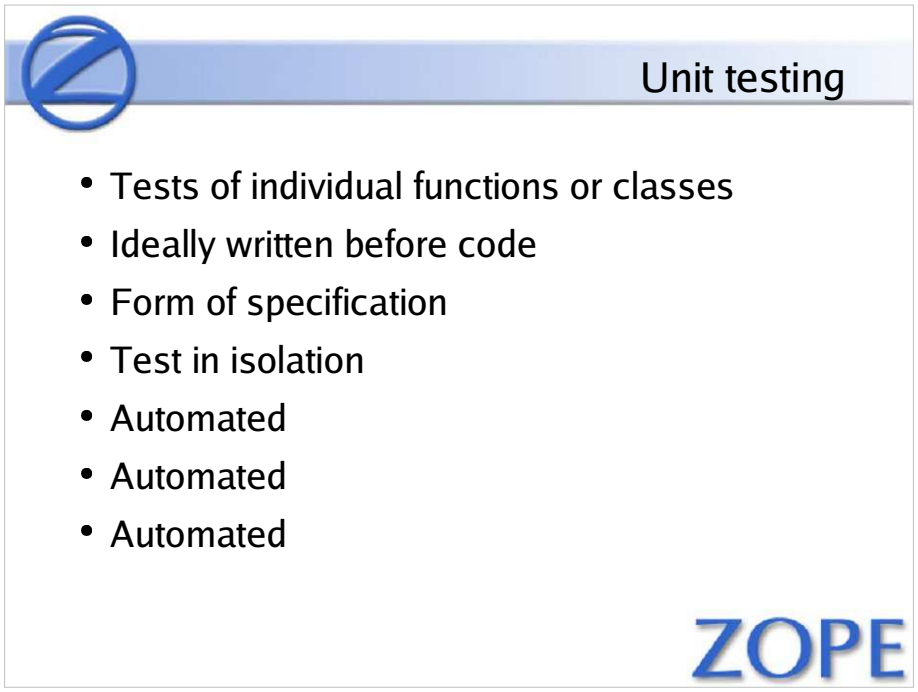
Unit testing is an important practice for improving the quality of software and for enabling refactoring. The standard Python unit testing framework, PyUnit, was based on the existing Java unit testing framework, JUnit. The framework provides a testing applications programming interface (API) through inheritance that supports test set up and provides various ways of making assertions about tests. It's common for testing code to contain more testing API calls than application code. Descriptive text is provided as comments and is usually in short supply.

Doctest is a system for writing tests within Python documentation strings. The emphasis is on documentation. Tests are provided as example code, set off with Python prompts. Doctest tests lend themselves toward a literate form of test code.

In Python 2.3, a new feature was added to Python to create unit tests from doctest doc strings. This talk provides an overview of doctest, shows how to create unit tests with doctest, and compares and contrasts regular PyUnit unit tests and doctest-based unit tests.

Tim Peters is the original author of doctest, and has been active in core Python development since Python 0.9.1.

Jim Fulton is the chief architect of Zope and the Zope object database, and has been a Python contributor since 1994.

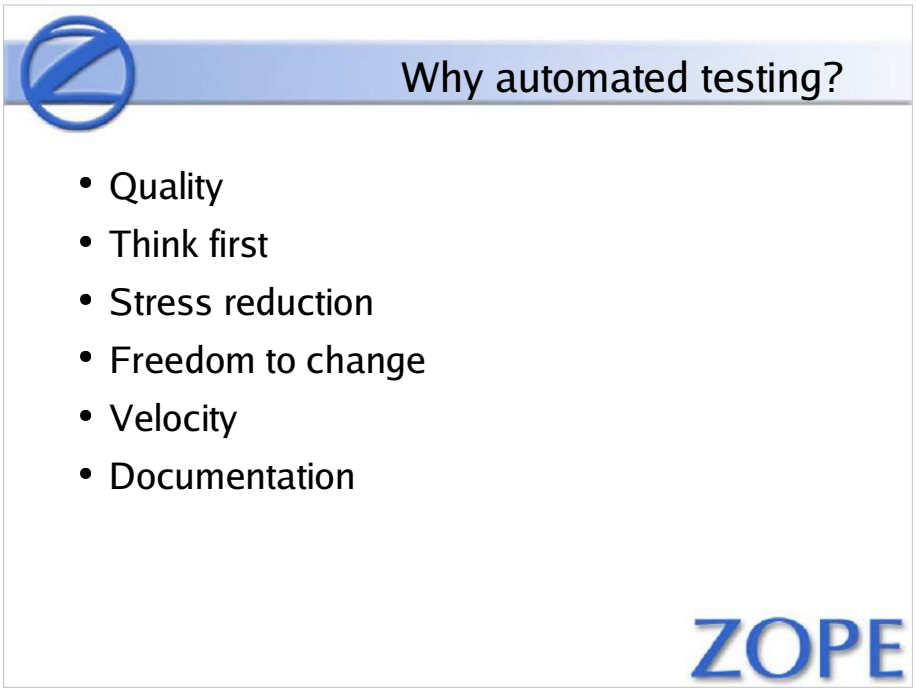


A unit test should ideally test only one "thing". A test should test one feature of a class or function.

Ideally, unit tests are written before code. The unit tests thus form a sort of specification for the code. In practice, this works best when the desired behavior of something is well understood ahead of time.

Perhaps the most important feature of unit tests is that they are automated. It's easy to run, and re-run, and re-run, collections of unit tests.

In fact, we will often use the unit testing framework to create tests that are not technically unit tests.



Testing gives us gives us greater confidence that the software does what it's supposed to do. Automated testing makes it easier to write and run tests, so we do more of it.

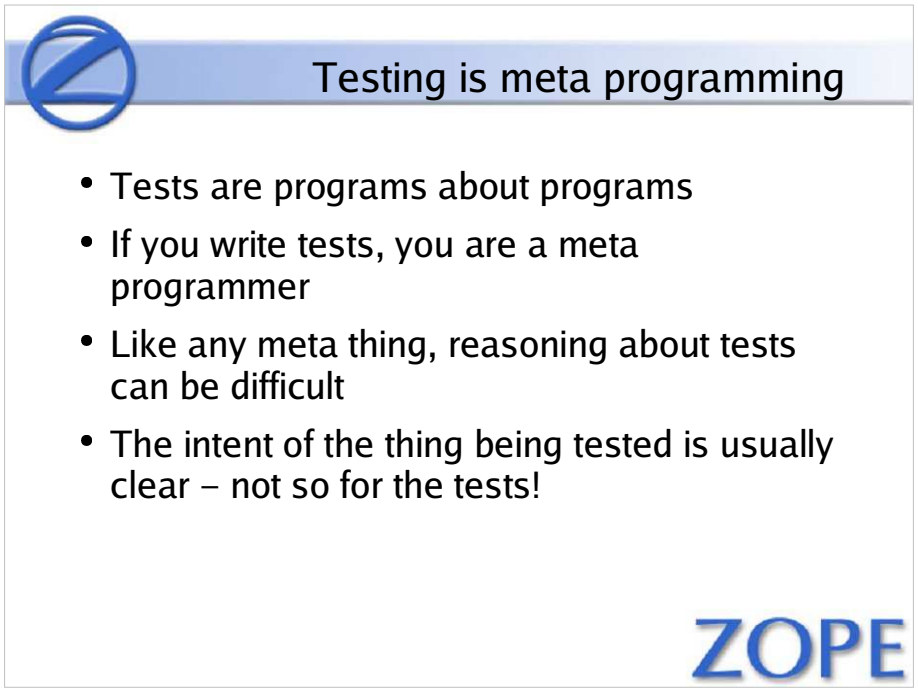
Testing encourages us to think first. We think about what the software is supposed to do before we start writing the software.

Automated testing reduces stress, because we know we can rerun tests at any time to be sure that changes haven't broken something without us knowing about it.

We're free to make small changes knowing that tests will catch breakage. Even large changes are safer, although we might need to change tests. The tests we have to change help us identify impacts on other systems that might be affected by the changes.

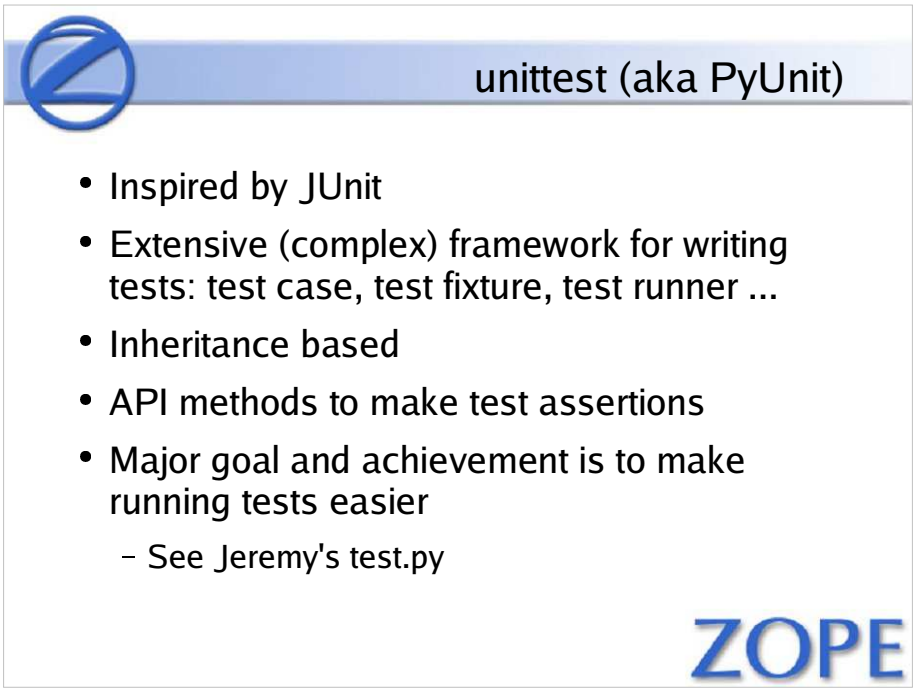
Tests ultimately allow us to create things quicker, because we aren't constantly fixing things or having to deal with poor code that we're afraid to fix.

Tests can help document how components are used by providing examples.

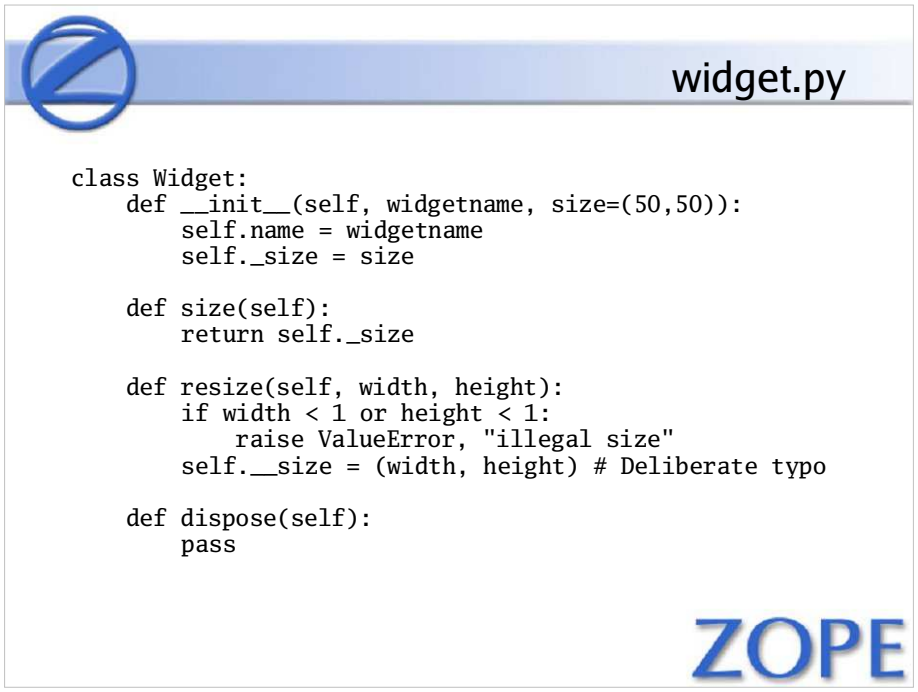


In our opinion, the greatest challenge when writing tests is writing tests that are understandable!

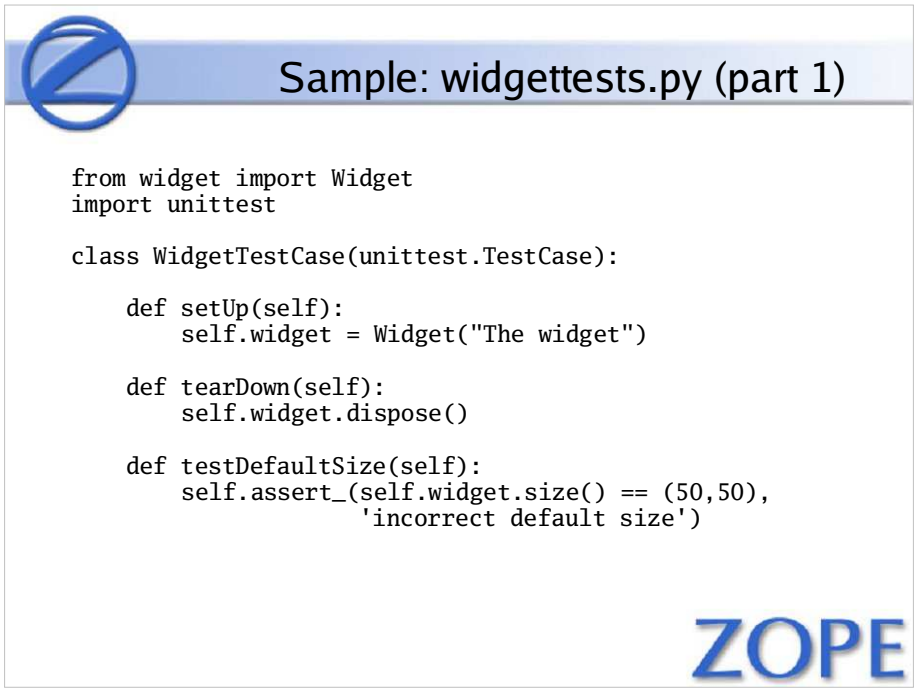
This is critical when tests break or need to be refactored



In many ways, the framework is too rich. There are too many ways to do things.



This is a simple example class. Notice that it has an intentional error.

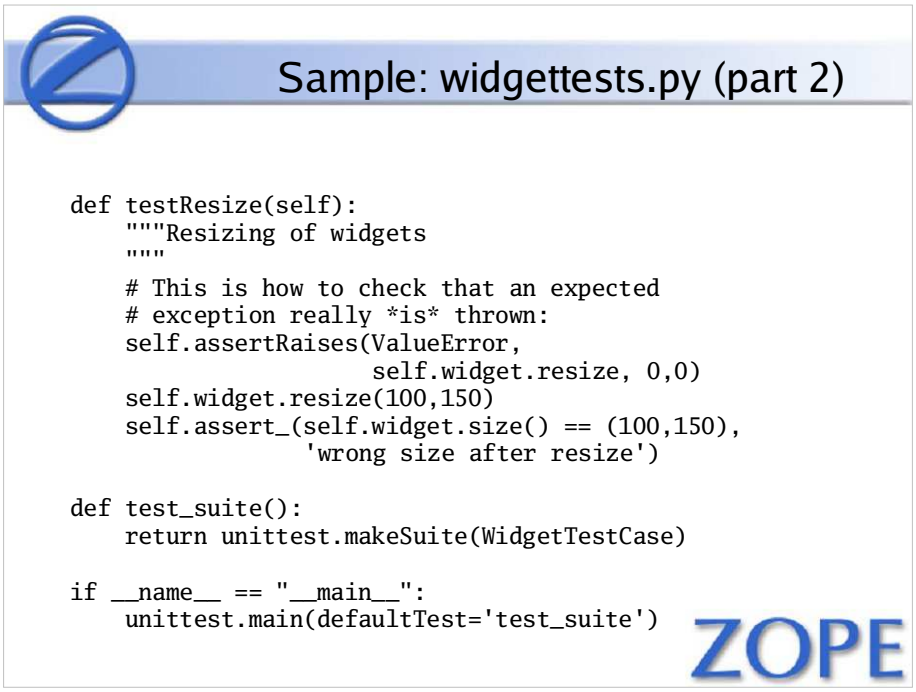


We create test cases by subclassing `TestCase` with methods whose names begin with "test". The test methods execute code being tested and use API functions to test results:

- `assert_` tests whether something is true
- `failUnless` is a synonym of `assert_`
- `assertEqual` test whether two things are equal
- `assertRaises` tests whether a particular exception is raised

`setUp` and `tearDown` methods can be used to abstract common code for set up and tear down.

It is **critical** that tests leave their environment in the same state that they found it in.

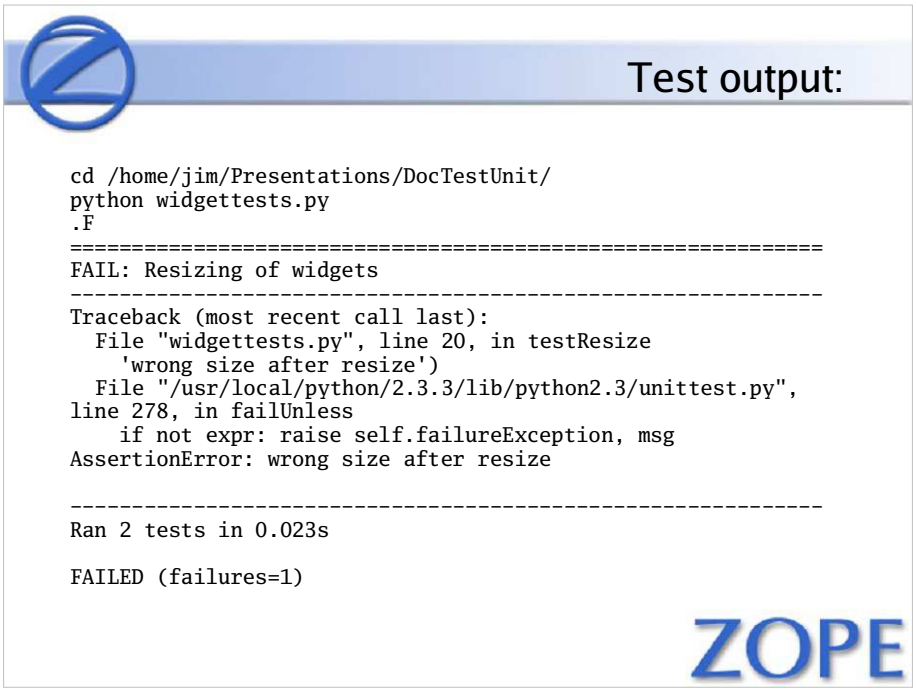


Here we see an example of testing that an exception is raised.

In the example, we define a test_suite function that computes a test suite, which is a collection of tests. We pass the name of this function to unittest.main. We could have avoided defining the `test_suite` function and let unittest.main find the tests for us:

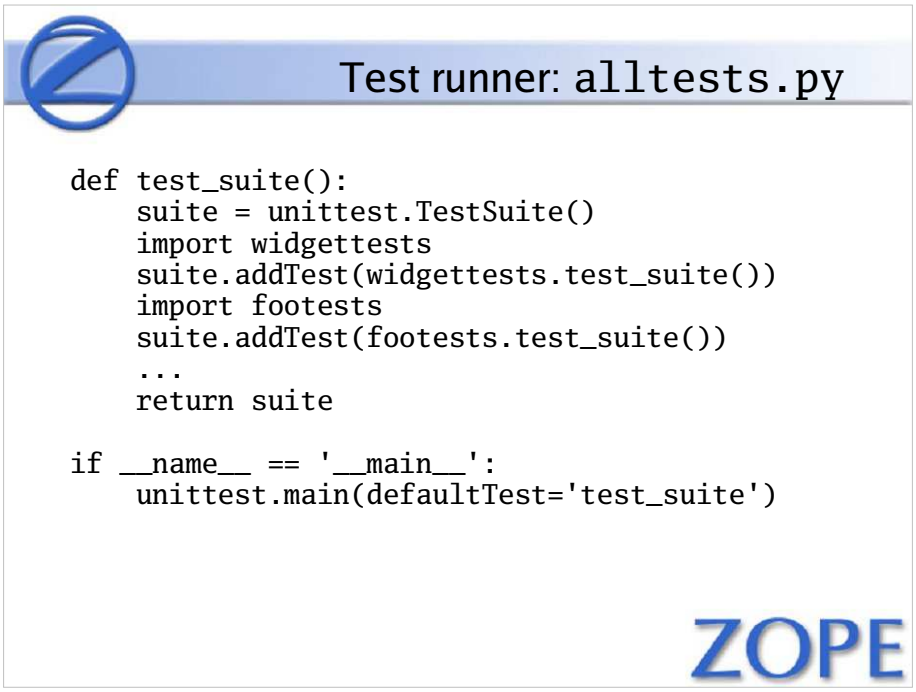
```
if __name__ = '__main__':
     inttest.main()
```

but defining a test_suite function makes it easier to define aggregate test runner scripts later.



In this example, the test output includes a summary line that shows a dot, an "F", or an "E", for each test, depending on whether it ran successfully, failed, or raised an unexpected exception. The output also shows details of any failures or errors.

The ability to run many tests together provides a lot of the power of `unittest`.

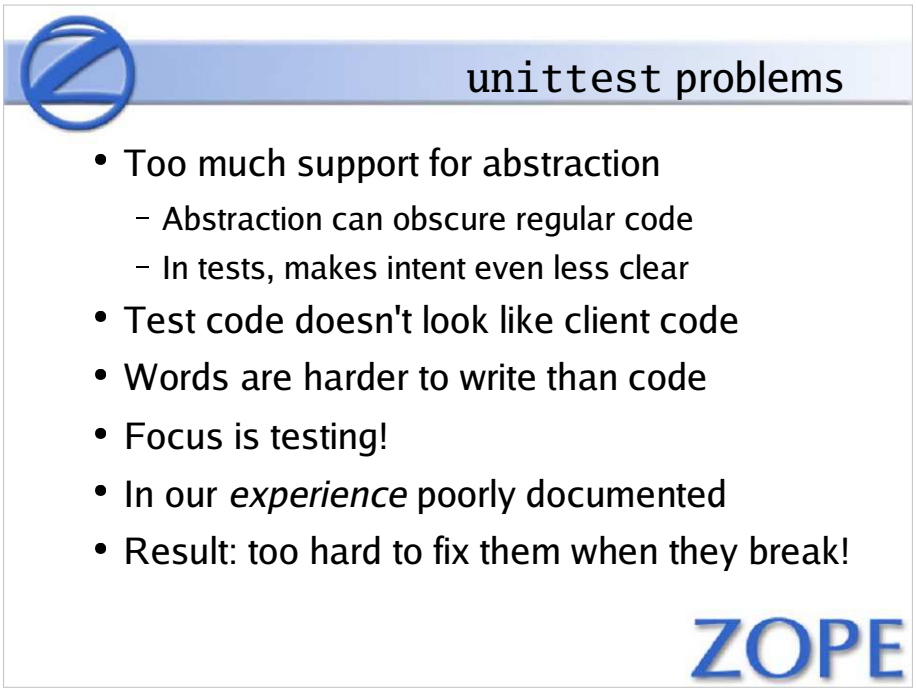


We often have multiple test files. We want to be able to run all the tests at once easily. We can do this by creating a collection test module that simply imports all of the other test modules and runs them at once. Every time you write a new test module, you need to modify the test runner.

For the Zope project, we have evolved a super test runner that automatically finds tests in a source tree. This test runner has a number of important features beyond simple test collection, including:

- post-mortem debugging support
- easy specification of test subsets to run via command line arguments
- code coverage analysis
- profiling
- analysis of memory leaks

We need to figure out how to make this available for other projects.

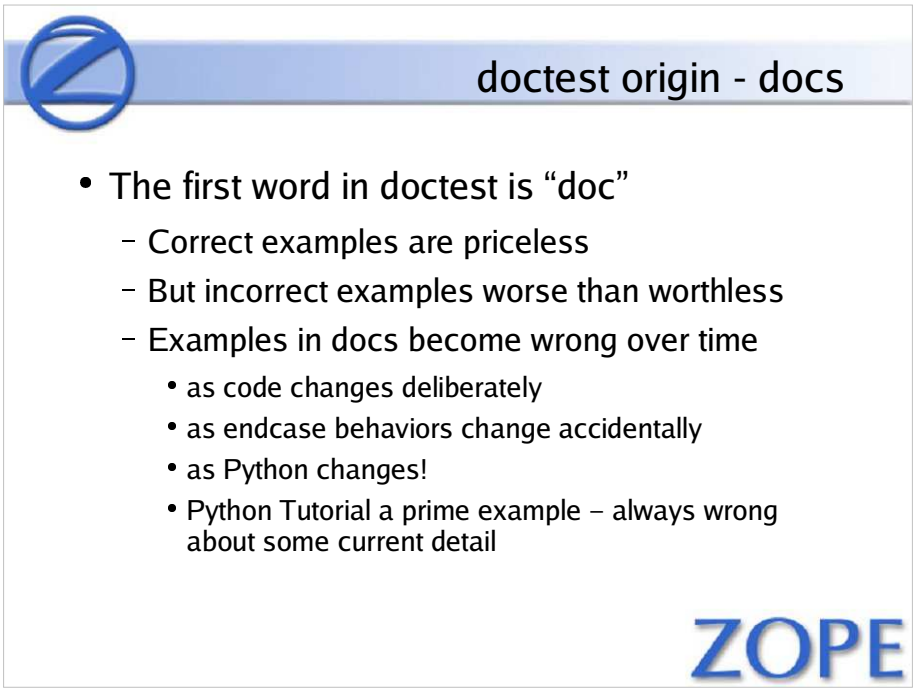


Because the testing framework uses inheritance, tests often use inheritance to abstract common set up or other testing code.

Abstraction can be good, but it can also make code harder to read. To understand what something is doing, you have to look in multiple places. Deciding how much to abstract is a **tradeoff**. Increasing abstraction increases reuse, reduces the amount of code that has to be maintained, but abstraction also reduces clarity. Because tests are meta, clarity is more important than for normal code and abstraction can be quite harmful and should be used with caution.

Test code is filled with calls to the testing framework. These calls are often dominant. To understand what's being done, you have to tease out the application logic from the testing logic.

In our experience, unit tests tend to have few words or explanations. There are probably a number of reasons for this, including our own inexperience. We think a dominant reason is the mind set. The `unittest` framework puts emphasis on testing. We think an emphasis on documentation can make tests more readable.



This slide and the two following outline doctest's author's original motivations. Tim was writing a large package of mathematical functions at the time, which tend to have very clear input-output requirements. Incremental development using IDLE was the norm: write a function, try it in the shell, fix surprises and repeat. The inspiration for doctest came when Tim noticed that his best tests were thrown away at the end of each IDLE session – and also his best docs, since the entire intended behavior of a mathematical function is often clear from a handful of concrete examples (for example, if a simple factorial implementation performs as desired at arguments -1, 0, 1, 2. 3 and 10, it's probably correct everywhere).

So the first doctests really were written by pasting pieces of an interactive Python session into docstrings. Of course that approach doesn't scale well to applications as large as Zope, so it's more common now to type Python examples and hoped-for output directly into a docstring, inserting sys.PS1 and sys.PS2 strings as manual markup.

doctest origin – interactive testing

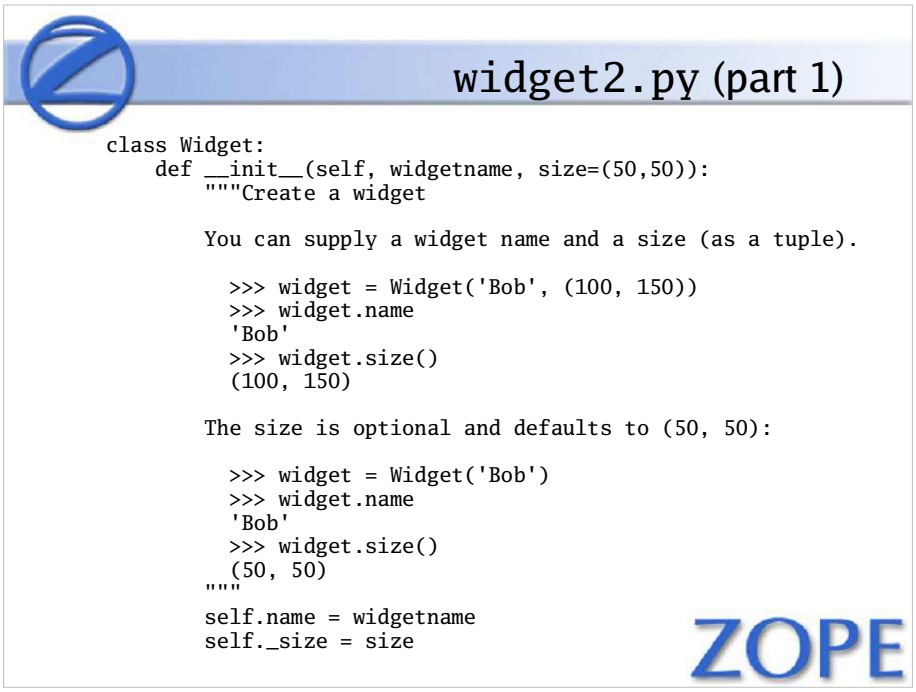
- “Code a little, test a little”
  - A Python shell (cmdline, IDLE, Emacs, PythonWin) makes iterative interactive development easy
  - Flesh out the code, test in the shell, repeat
    - interesting cases
    - oddball cases
    - error cases
  - You won't try all of them again on the next iteration, and they're lost forever when the shell closes

ZOPE

doctest origin – put tests in the docs

- Solution: paste those tests into the docs!
  - doctest ensures the examples run exactly as advertised forever after
  - Of course they won't, and then doctest tells you
  - Improves coverage by verifying normal, endcase and error cases
  - Improves docs by demonstrating examples of each – examples are often clearer than prose
  - Examples + prose is ideal: “literate testing”, “executable docs”

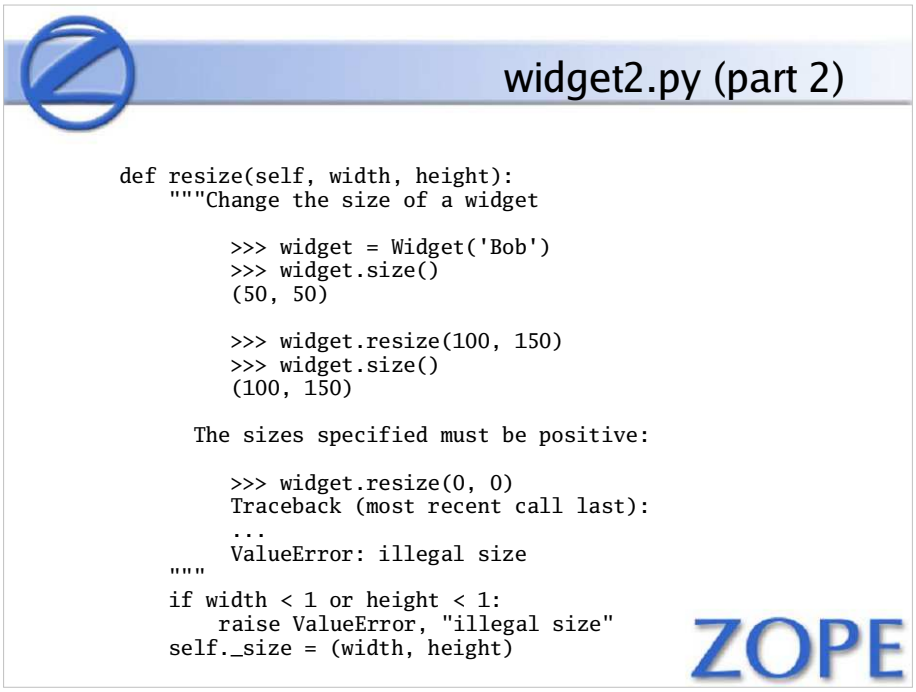
ZOPE



Here, we've added doc strings to the code. These doc strings include examples of how to use the widget's methods. The examples show mini-interpreter sessions, showing input and output. There are no test assertions other than expected output.

Sometimes, we do have to resort to assertions because we can't show the output, either because the output would be too long or wouldn't be suitable for other reasons. In cases like this, we'll simply evaluate a boolean expression, as in:

```
>>> foo() == expected
True
```



Here we see an error example. We actually show the traceback that would result from a call. We can elide parts of the traceback output we aren't interested in.

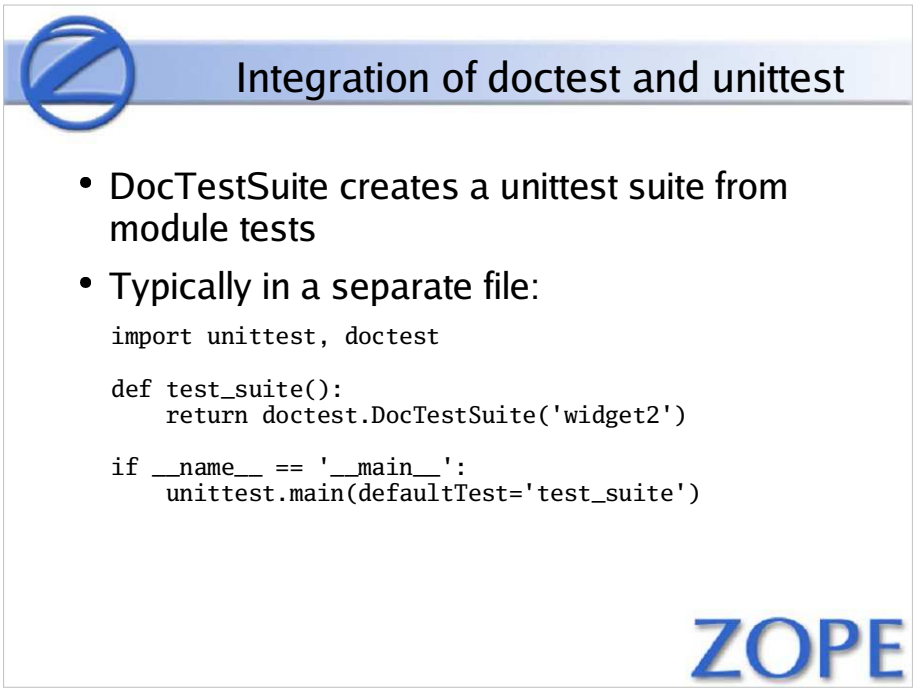
For completeness, here are some additional methods from widget2.py that we haven't shown in the slides:

```
def size(self):
    """Return the current widget size

      >>> widget = Widget('Bob', (100, 150))
      >>> widget.size()
      (100, 150)

    """
    return self._size

def dispose(self):
    pass
```

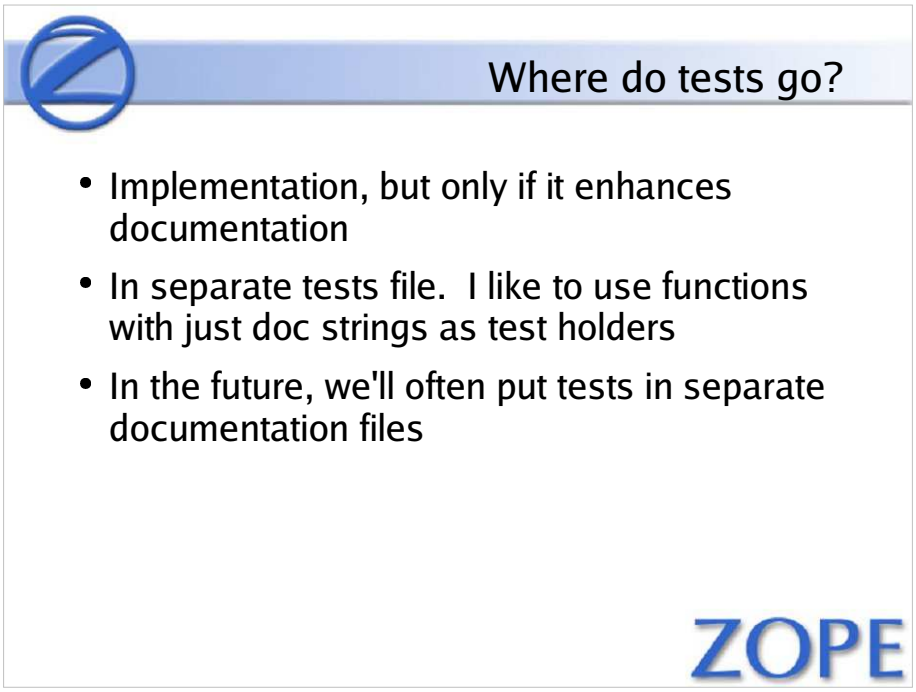


You can pass a module or a module name to DocTestSuite. You can omit an argument altogether, in which case the current module is tested.

So what changed?

- Focus on documentation
- No testing calls
  - what you see is what the client does
  - Less noise
- Connected to implementation
- Don't need to learn an API to start writing tests
- The focus on documentation gets much more important as things get sophisticated!

ZOPE



We like to put lots of examples in the implementation code:

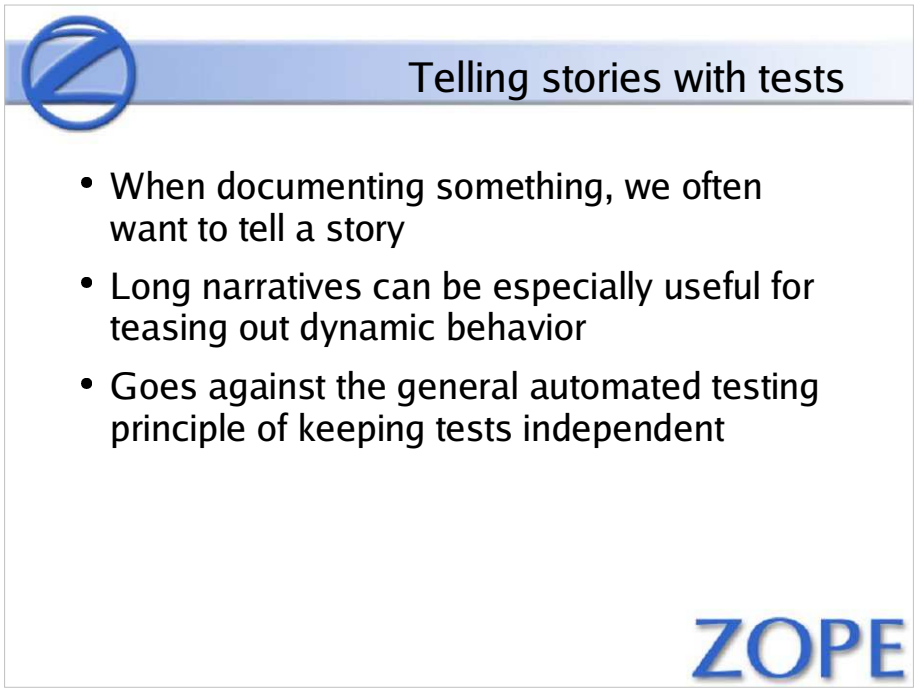
• The examples help to document the individual methods.

• It's easier for us to make sure the important methods are tested if the tests are with the method implementations.

Many people find that much documentation in the implementation is intrusive. The rule of thumb is to only add tests to the implementation code if they don't detract from reading the code; however, this is subjective. Folding editors can make it easy to hide the documentation, but not everyone uses folding editors.

We'll often put doc tests in separate test files. The files will have a number of functions with empty bodies, except for doc strings.

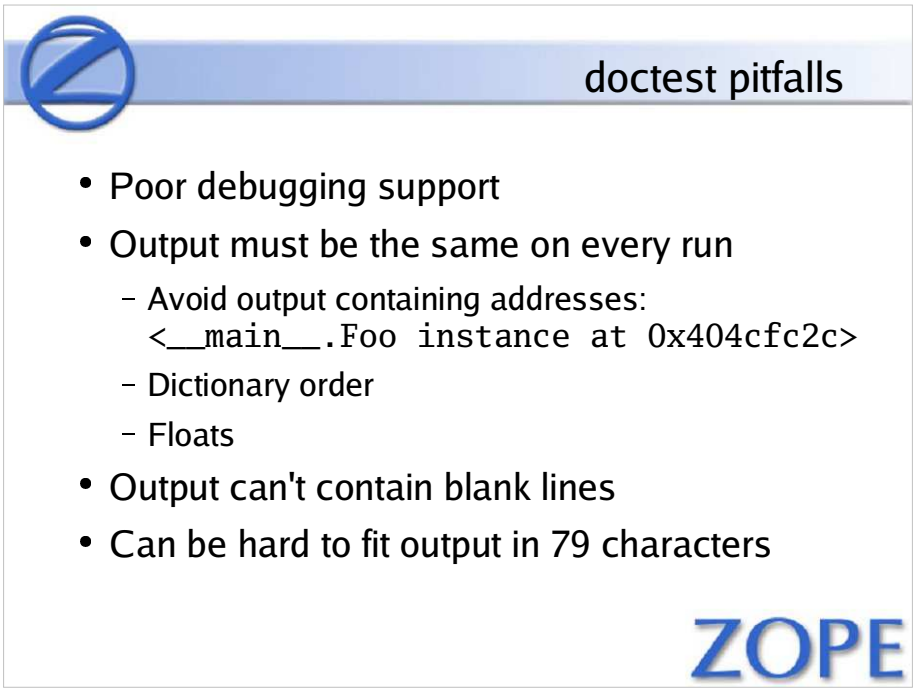
Lately, we've been experimenting with putting tests in separate documentation (.txt) files. This works very well, especially when the primary purpose of the file is documentation.



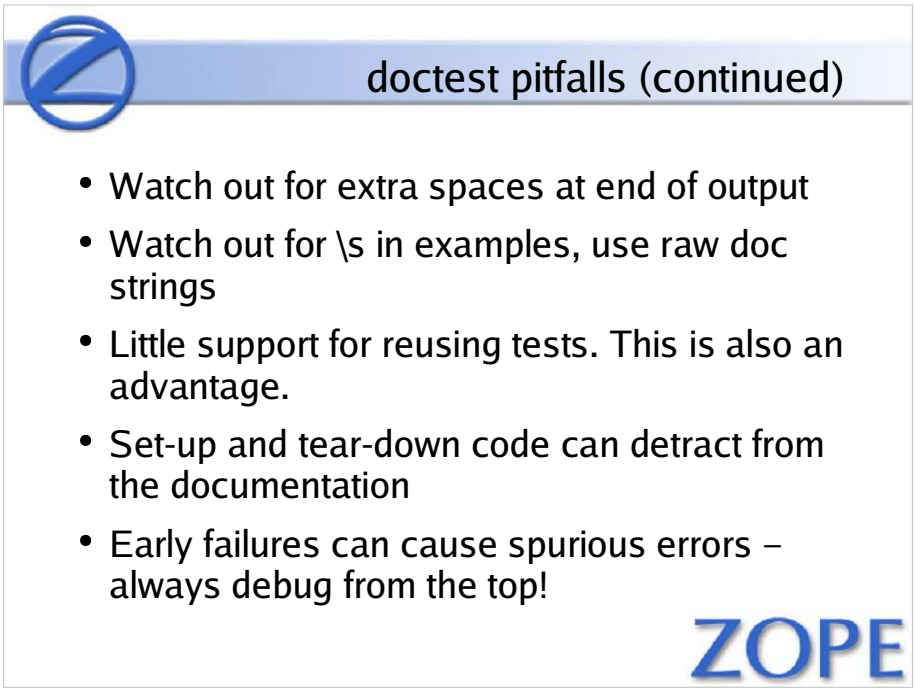
We often want to build up a story, showing how to use something in a step-by-step fashion.

Sometimes, you need to see how different methods interact to understand the dynamic behavior of a component. Static descriptions of individual methods don't tell the whole story. Attempts to explain the methods in isolation can be bewildering to the user. Attempts to test methods in isolation, when the methods are, in fact interrelated, can be bewildering to the test reader.

Building up a story that exercises many methods goes against a principal of automated tests that says that tests should be independent. A failure early on in a set of operations can cause spurious errors later on. This can indeed be a problem, but the added clarity gained from the story sometimes outweighs the inconvenience of spurious failures.



It's hard to debug doctest failures. (It's hard to debug unittest failures using the tools provided with unittest as well.) The doctest module does provide a debugger that lets you debug individual doc strings. There isn't currently any support for post-mortem debugging, and pdb.set_trace doesn't work due to the way that doctest treats sys.stdout.



Doctest compares output literally. A common mistake is to get extra trailing spaces in expected output when copying and pasting.

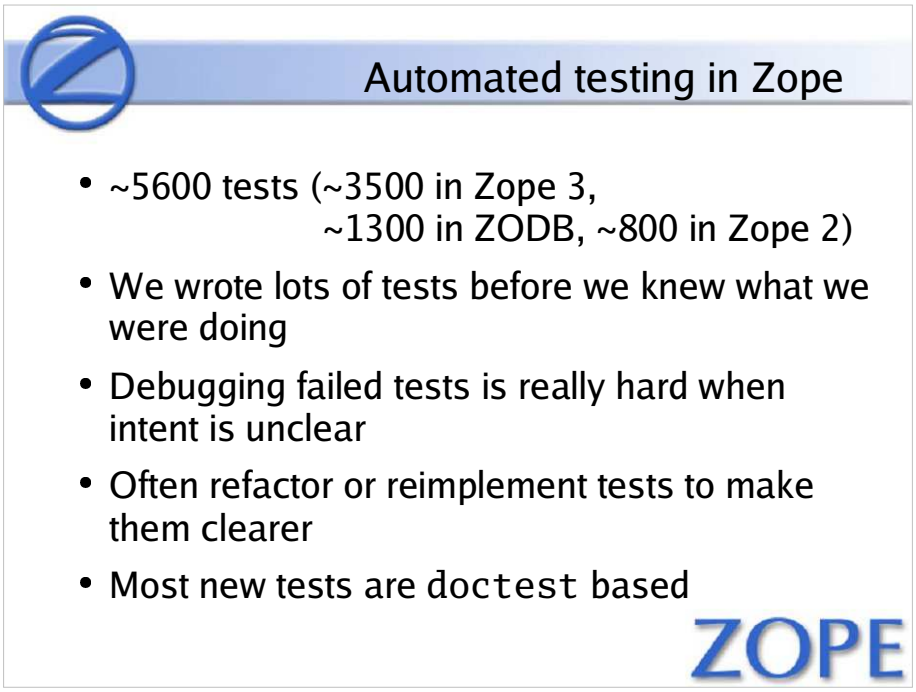
If doc strings contain backslashes, they will be interpreted by Python before doctest sees them. Whenever you have backslashes in your doc strings, you'll want to use raw strings, as in:

```
def foo():
      r""" ....
```

An advantage of doctest is that it doesn't support inheritance What you see is what you get. Sometimes you really want to be able to reuse complex tests. The doctest module provides some low-level facilities that would allow this, but the unittest integration doesn't expose them.

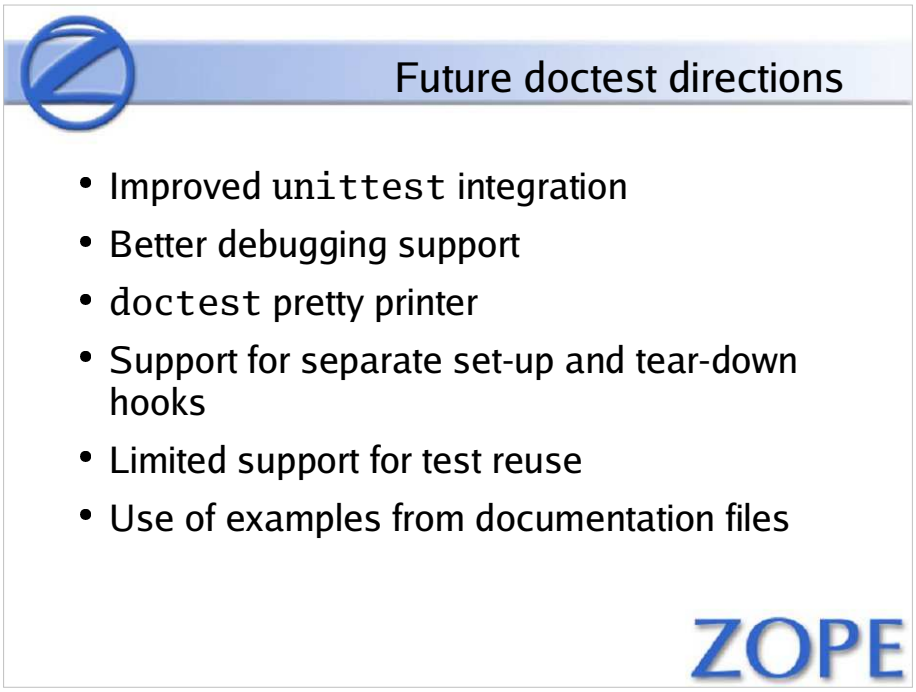
You can simply call set-up and tear-down code in your examples. This code could fit well into the documentation, however, it may be needed mainly to support testing and thus detract from the story being told. This is especially true of tear-down code. In cases like this, it might be nice to be able to separate the set-up pr tear-down code from the examples.

Doctest executes all of the examples, even if early examples fail in some fashion. For this reason, you always want to address failures from top to bottom.



Writing good tests is hard. Early on, we thought the most important thing was the number of tests and test coverage. Test coverage is important, but test readability is much more important.

In the Zope 3 project, we've refactored and restructured mercilessly. Our tests made this possible, but we've often experienced great pain trying to figure out test failures when the intent of the failing tests wasn't clear.



The unittest integration that appeared in Python 2.3 was a first cut. It didn't provide enough meta-data for the test runner to generate helpful test summaries. We've since fixed this and will include the fixes is future releases.

We plan to add support for post-mortem debugging and for use of pdb.set_trace within examples or code called from examples. We've implemented some of this in Zope already.

We would like to add a doctest-specific pretty printer that:

• Sorts dictionary items

• Normalizes (or removes) obvious addresses

• Normalizes floats

• Removes blank lines and trailing spaces

• Provides easier width control

We plan to add set-up and tear-down hooks to the DocTestSuite constructor. This is available in Zope 3.

We plan to provide basic support for test reuse by allowing additional module variables to be passed to the test suite constructor.

We have begin experimenting with creating tests from examples in separate documentation (e.g. .txt) files. We plan to integrate this in doctest's unittest support.