`!doctest` --- Test interactive Python examples

- Simple Usage: Checking Examples in Docstrings
- Simple Usage: Checking Examples in a Text File
- Command-line Usage
- How It Works
  - Which Docstrings Are Examined?
  - How are Docstring Examples Recognized?
  - What's the Execution Context?
  - What About Exceptions?
  - Option Flags
  - Directives
  - Warnings
- Basic API
- Unittest API
- Advanced API
  - DocTest Objects
  - Example Objects
  - DocTestFinder objects
  - DocTestParser objects
  - TestResults objects
  - DocTestRunner objects
  - OutputChecker objects
- Debugging
- Soapbox

doctest

Source code: `Lib/doctest.py`

The `!doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

Here's a complete but small example module:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
```

```
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


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

If you run `example.py` directly from the command line, `!doctest` works its magic:

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and `!doctest` prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
```

```
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
7 tests in 2 items.
7 passed.
Test passed.
$
```

That's all you need to know to start making productive use of `!doctest`! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest/test_doctest.py`.

3.13 Output is colorized by default and can be `controlled using environment variables <using-on-controlling-color>`.

Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you'll continue to do it) is to end each module `!M` with:

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

`!doctest` then examines docstrings in module `!M`.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is `***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the `-v` switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to `testmod`, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by `testmod` (so passing `-v` or not has no effect).

There is also a command line shortcut for running `testmod`, see section `doctest-cli`.

For more information on `testmod`, see section `doctest-basic-api`.

Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the `testfile` function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with `testmod`, `testfile` won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `!testmod`.

By default, `testfile` looks for files in the calling module's directory. See section `doctest-basic-api` for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod`, `testfile`'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

There is also a command line shortcut for running `testfile`, see section `doctest-cli`.

For more information on `testfile`, see section `doctest-basic-api`.

Command-line Usage

The `!doctest` module can be invoked as a script from the command line:

```
python -m doctest [-v] [-o OPTION] [-f] file [file ...]
```

doctest

`-v, --verbose`

Detailed report of all examples tried is printed to standard output, along with assorted summaries at the end:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run `testmod` on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

If the file name does not end with `.py`, `!doctest` infers that it must be run with `testfile` instead:

```
python -m doctest -v example.txt
```

`-o, --option <option>`

Option flags control various aspects of doctest's behavior, see section `doctest-options`.

3.4

`-f, --fail-fast`

This is shorthand for `-o FAIL_FAST`.

3.4

How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

None

module.__test__

doctest

In addition, there are cases when you want tests to be part of a module but not part of the help text, which requires that the tests not be included in the docstring. Doctest looks for a module-level variable called `__test__` and uses it to locate other tests. If `M.__test__` exists, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name `M.__test__.K`.

For example, place this block of code at the top of `example.py`:

```
__test__ = {
    'numbers': """
>>> factorial(6)
720


>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

The value of `example.__test__["numbers"]` will be treated as a docstring and all the tests inside it will be run. It is important to note that the value can be mapped to a function, class object, or module; if so, `!doctest` searches them recursively for docstrings, which are then scanned for tests.

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

> [!NOTE] `doctest` can only automatically discover classes and functions that are defined at the module level or inside other classes.
>
> Since nested classes and functions only exist when an outer function is called, they cannot be discovered. Define them outside to make them visible.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

single: >>>; interpreter prompt single: ...; interpreter prompt

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the `NORMALIZE_WHITESPACE` option or `directive <doctest-`

`directives>` is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom `DocTestParser` class.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

  Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- The starting column doesn't matter:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

  and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>> '` line that started the example.

What's the Execution Context?

By default, each time `!doctest` finds a docstring to test, it uses a *shallow copy* of `!M`'s globals, so that running tests doesn't change the module's real globals, and so that one test in `!M` can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in `!M`, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to `testmod` or `testfile` instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[^1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That doctest succeeds if `ValueError` is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with `ValueError`) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's `ELLIPSIS` option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether `ValueError` is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the `IGNORE_EXCEPTION_DETAIL` doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some `SyntaxError`s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a `!SyntaxError` that omits the traceback header, you will need to manually add the traceback header line to your test example.

single: ^ (caret); marker

- For some exceptions, Python displays the position of the error using `^` markers and tildes:

  ```
  >>> 1 + None
    File "<stdin>", line 1
      1 + None
      ~~^~~~~
  TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
  ```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the `^` marker in the wrong location:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be `bitwise ORed <bitwise>` together and passed to various functions. The names can also be used in `doctest directives <doctest-directives>`, and may be passed to the doctest command line interface via the `-o` option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

DONT_ACCEPT_TRUE_FOR_1

By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is considered to be a match, and similarly for `0` versus `False`. When `DONT_ACCEPT_TRUE_FOR_1` is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

single: <BLANKLINE>

DONT_ACCEPT_BLANKLINE

By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When `DONT_ACCEPT_BLANKLINE` is specified, this substitution is not allowed.

NORMALIZE_WHITESPACE

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. `NORMALIZE_WHITESPACE` is especially

useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

single: ...; in doctests

ELLIPSIS

When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that `.*` is prone to in regular expressions.

IGNORE_EXCEPTION_DETAIL

When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don't match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a `TypeError` is raised instead. It will also ignore any fully qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

Note that `ELLIPSIS` can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

3.2 `IGNORE_EXCEPTION_DETAIL` now also ignores any information relating to the module containing the exception under test.

SKIP

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.

COMPARISON_FLAGS

A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

REPORT_UDIFF

When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

REPORT_CDIFF

When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

REPORT_NDIFF

When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

REPORT_ONLY_FIRST_FAILURE

When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When `REPORT_ONLY_FIRST_FAILURE` is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

FAIL_FAST

When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

REPORTING_FLAGS

A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn't useful unless you intend to extend `!doctest` internals via subclassing:

register_optionflag(name)

Create a new option flag with a given name, and return the new flag's integer value. `register_optionflag` can be used when subclassing `OutputChecker` or `DocTestRunner` to create new options that are supported by your subclasses. `register_optionflag` should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

single: # (hash); in doctests single: + (plus); in doctests single: - (minus); in doctests

Directives

Doctest directives may be used to modify the `option flags <doctest-options>` for an individual example. Doctest directives are special Python comments following an example's source code:

doctest directive: "#" "doctest:" directive_options directive_options: directive_option ("," directive_option)* directive_option: on_or_off directive_option_name on_or_off: "+" | "-" directive_option_name: "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...

Whitespace is not allowed between the `+` or `-` and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use `+` to enable the named behavior, or `-` to disable it.

For example, this test passes:

>>> print(list(range(20))) # doctest: +NORMALIZE_WHITESPACE [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

>>> print(list(range(20))) # doctest: +ELLIPSIS [0, 1, ..., 18, 19]

Multiple directives can be used on a single physical line, separated by commas:

>>> print(list(range(20))) # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE [0, 1, ..., 18, 19]

If multiple directive comments are used for a single example, then they are combined:

>>> print(list(range(20))) # doctest: +ELLIPSIS ... # doctest: +NORMALIZE_WHITESPACE [0, 1, ..., 18, 19]

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40))) ... # doctest: +ELLIPSIS [0, ..., 4, 10, ..., 19, 30, ..., 39]

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via `+` in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via `-` in a directive can be useful.

Warnings

`!doctest` is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a set, Python doesn't guarantee that the element is printed in any particular order, so a test like :

```
>>> foo()
{"spam", "eggs"}
```

is vulnerable! One workaround is to do :

```
>>> foo() == {"spam", "eggs"}
True
```

instead. Another is to do :

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

>>> id(1.0) # certain to fail some of the time # doctest: +SKIP 7948648 >>> class C: pass >>> C() # the default repr() for instances embeds an address # doctest: +SKIP <C object at 0x00AC18F0>

The `ELLIPSIS` directive gives a nice approach for the last example:

>>> C() # doctest: +ELLIPSIS <C object at 0x...>

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for some floating-point calculations, and C libraries vary widely in quality here. :

```
>>> 1000**0.1  # risky
1.9952623149688797
>>> round(1000**0.1, 9) # safer
1.995262315
>>> print(f'{1000**0.1:.4f}') # much safer
1.9953
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

Basic API

The functions `testmod` and `testfile` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections `doctest-simple-testmod` and `doctest-simple-testfile`.

testfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=DocTestParser(), encoding=None)

All arguments except *filename* are optional, and should be specified in keyword form.

Test examples in the file named *filename*. Return `(failure_count, test_count)`.

Optional argument *module_relative* specifies how the filename should be interpreted:

- If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like `dict.update`: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in `sys.argv`.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* (default value `0`) takes the `bitwise OR <bitwise>` of option flags. See section `doctest-options`.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

testmod(m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module `__main__` if *m* is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return `(failure_count, test_count)`.

Optional argument *name* gives the name of the module; by default, or if `None`, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize <DocTestRunner.summarize>` in conjunction with `testmod` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function `testfile` above, except that *globs* defaults to `m.__dict__`.

run_docstring_examples(f, globs, verbose=False, name="NoName", compileflags=None, optionflags=0)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to `"NoName"`.

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of

future features found in *globs*.

Optional argument *optionflags* works as for function `testfile` above.

Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. `!doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a `load_tests <load_tests-protocol>` function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating `unittest.TestSuite` instances from text files and modules with doctests:

DocFileSuite(*paths, module_relative=True, package=None, setUp=None, tearDown=None, globs=None, optionflags=0, parser=DocTestParser(), encoding=None)

Convert doctest tests from one or more text files to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. Each file is run as a separate unit test, and each example in a file is run as a `subtest <subtests>`. If any example in a file fails, then the synthesized unit test fails. The traceback for failure or error contains the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a file are skipped, then the synthesized unit test is also marked as skipped.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To

ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a `DocTest` object. The *setUp* function can access the test globals as the `~DocTest.globs` attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a `DocTest` object. The *tearDown* function can access the test globals as the `~DocTest.globs` attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section `doctest-options`. See function `set_unittest_reportflags` below for a better way to set reporting options.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using `DocFileSuite`.

3.15 Run each example as a `subtest <subtests>`.

DocTestSuite(module=None, globs=None, extraglobs=None, test_finder=None, setUp=None, tearDown=None, optionflags=0, checker=None)

Convert doctest tests for a module to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. Each docstring is run as a separate unit test, and each example in a docstring is run as a `subtest <subtests>`. If any of the doctests fail, then the synthesized unit test fails. The traceback for failure or error contains the name of the file containing the test and a (sometimes approximate)

line number. If all the examples in a docstring are skipped, then the synthesized unit test is also marked as skipped.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is the module's `~module.__dict__`.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the `DocTestFinder` object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function `DocFileSuite` above, but they are called for each docstring.

This function uses the same search technique as `testmod`.

3.5 `DocTestSuite` returns an empty `unittest.TestSuite` if *module* contains no docstrings instead of raising `ValueError`.

3.15 Run each example as a `subtest <subtests>`.

3.16 Report every example, as in verbose mode, if the test runner reports more than the test names, i.e. its `~unittest.TestResult.verbosity` is 3 or higher (for example with `python -m unittest -vv`).

Under the covers, `DocTestSuite` creates a `unittest.TestSuite` out of `!doctest.DocTestCase` instances, and `!DocTestCase` is a subclass of `unittest.TestCase`. `!DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of `unittest` integration.

Similarly, `DocFileSuite` creates a `unittest.TestSuite` out of `!doctest.DocFileCase` instances, and `!DocFileCase` is a subclass of `!DocTestCase`.

So both ways of creating a `unittest.TestSuite` run instances of `!DocTestCase`. This is important for a subtle reason: when you run `!doctest` functions yourself, you can control the `!doctest` options in use directly, by passing option flags to `!doctest` functions. However, if you're writing a `unittest` framework, `!unittest` ultimately controls when and how tests get run. The framework author typically wants to control `!doctest` reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through `!unittest` to `!doctest` test runners.

For this reason, `!doctest` also supports a notion of `!doctest` reporting flags specific to `unittest` support, via this function:

set_unittest_reportflags(flags)

Set the `!doctest` reporting flags to use.

Argument *flags* takes the `bitwise OR <bitwise>` of option flags. See section `doctest-options`. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module `unittest`: the `!runTest` method of `!DocTestCase` looks at the option flags specified for the test case when the `!DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `!doctest`'s `!unittest` reporting flags are `bitwise ORed <bitwise>` into the option flags, and the option flags so augmented are passed to the `DocTestRunner` instance created to run the doctest. If any reporting flags were specified when the `!DocTestCase` instance was constructed, `!doctest`'s `!unittest` reporting flags are ignored.

The value of the `unittest` reporting flags in effect before the function was called is returned by the function.

Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- `Example`: A single Python `statement`, paired with its expected output.
- `DocTest`: A collection of `Example`s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- `DocTestFinder`: Finds all docstrings in a given module, and uses a `DocTestParser` to create a `DocTest` from every docstring that contains interactive examples.
- `DocTestParser`: Creates a `DocTest` object from a string (such as an object's docstring).
- `DocTestRunner`: Executes the examples in a `DocTest`, and uses an `OutputChecker` to verify their output.
- `OutputChecker`: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:

```
list of:
+------+                   +---------+
|module| --DocTestFinder-> | DocTest | --DocTestRunner-> results
+------+    |        ^     +---------+     |       ^    (printed)
|        |     | Example |     |       |
v        |     |   ...   |     v       |
DocTestParser   | Example |   OutputChecker
+---------+
```

DocTest Objects

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

`DocTest` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of `Example` objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in `globs` after the test is run.

name

A string name identifying the `DocTest`. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this `DocTest` was extracted from; or `None` if the filename is unknown, or if the `!DocTest` was not extracted from a file.

lineno

The line number within `filename` where this `DocTest` begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or `None` if the string is unavailable, or if the test was not extracted from a string.

Example Objects

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

`Example` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). `want` ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg

The exception message generated by the example, if the example is expected to generate an exception; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only`. `exc_msg` ends with a newline unless it's `None`. The constructor adds a newline if needed.

lineno

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options

A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`'s `optionflags <doctest-options>`). By default, no options are set.

DocTestFinder objects

A processing class used to extract the `DocTest`s that are relevant to a given object, from its docstring and the docstrings of its contained objects. `DocTest`s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the `DocTestParser` object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then `DocTestFinder.find` will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then `DocTestFinder.find` will include tests for objects with empty docstrings.

`DocTestFinder` defines the following method:

find(obj[, name][, module][, globs][, extraglobs])

Return a list of the `DocTest`s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned `DocTest`s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.

- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each `DocTest` is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each `!DocTest`. If *globs* is not specified, then it defaults to the module's `~module.__dict__`, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser objects

A processing class used to extract interactive examples from a string, and use them to create a `DocTest` object.

`DocTestParser` defines the following methods:

get_doctest(string, globs, name, filename, lineno)

Extract all doctest examples from the given string, and collect them into a `DocTest` object.

globs, *name*, *filename*, and *lineno* are attributes for the new `!DocTest` object. See the documentation for `DocTest` for more information.

get_examples(string, name='<string>')

Extract all doctest examples from the given string, and return them as a list of `Example` objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(string, name='<string>')

Divide the given string into examples and intervening text, and return them as a list of alternating `Example`s and strings. Line numbers for the `!Example`s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

TestResults objects

failed

Number of failed tests.

attempted

Number of attempted tests.

skipped

Number of skipped tests.

3.13

DocTestRunner objects

A processing class used to execute and verify the interactive examples in a `DocTest`.

The comparison between expected outputs and actual outputs is done by an `OutputChecker`. This comparison may be customized with a number of option flags; see section `doctest-options` for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of `!OutputChecker` to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to `run`; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods `report_skip`, `report_start`, `report_success`, `report_unexpected_exception`, and `report_failure`.

The optional keyword argument *checker* specifies the `OutputChecker` object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the `DocTestRunner`'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section `doctest-options`.

The test runner accumulates statistics. The aggregated number of attempted, failed and skipped examples is also available via the `tries`, `failures` and `skips` attributes. The `run` and `summarize` methods return a `TestResults` instance.

`DocTestRunner` defines the following methods:

report_skip(out, test, example)

Report that the given example was skipped. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run`.

3.15

report_start(out, test, example)

Report that the test runner is about to process the given example. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run`.

report_success(out, test, example, got)

Report that the given example ran successfully. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run`.

report_failure(out, test, example, got)

Report that the given example failed. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run`.

report_unexpected_exception(out, test, example, exc_info)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by `sys.exc_info`). *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run`.

run(test, compileflags=None, out=None, clear_globs=True)

Run the examples in *test* (a `DocTest` object), and display the results using the writer function *out*. Return a `TestResults` instance.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the `!DocTestRunner.report_\*` methods.

summarize(verbose=None)

Print a summary of all the test cases that have been run by this DocTestRunner, and return a `TestResults` instance.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the `DocTestRunner`'s verbosity is used.

`DocTestParser` has the following attributes:

tries

Number of attempted examples.

failures

Number of failed examples.

skips

Number of skipped examples.

3.13

OutputChecker objects

A class used to check the whether the actual output from a doctest example matches the expected output. `OutputChecker` defines two methods: `check_output`, which compares a given pair of outputs, and returns `True` if they match; and `output_difference`, which returns a string describing the differences between two outputs.

`OutputChecker` defines the following methods:

check_output(want, got, optionflags)

Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section `doctest-options` for more information about option flags.

output_difference(example, got, optionflags)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, `pdb`.
- The `DebugRunner` class is a subclass of `DocTestRunner` that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The `unittest` cases generated by `DocTestSuite` support the `debug` method defined by `unittest.TestCase`.
- You can add a call to `pdb.set_trace` in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
```

```
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

script_from_examples(s)

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example, :

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

testsource(module, name)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for `script_from_examples` above. For example, if module `a.py` contains a top-level function `!f`, then :

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `!f`'s docstring, with doctests converted to code, and the rest placed in comments.

debug(module, name, pm=False)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function `testsource` above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, `pdb`.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via `pdb.post_mortem`, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate `exec` call to `pdb.run`.

debug_src(src, pm=False, globs=None)

Debug the doctests in a string.

This is like function `debug` above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function `debug` above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The `DebugRunner` class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially `DebugRunner`'s docstring (which is a doctest!) for more details:

A subclass of `DocTestRunner` that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an `UnexpectedException` exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a `DocTestFailure` exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for `DocTestRunner` in section `doctest-advanced-api`.

There are two exceptions that may be raised by `DebugRunner` instances:

DocTestFailure(test, example, got)

An exception raised by `DocTestRunner` to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

`DocTestFailure` defines the following attributes:

DocTestFailure.test

The `DocTest` object that was being run when the example failed.

DocTestFailure.example

The `Example` that failed.

DocTestFailure.got

The example's actual output.

UnexpectedException(test, example, exc_info)

An exception raised by `DocTestRunner` to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

`UnexpectedException` defines the following attributes:

UnexpectedException.test

The `DocTest` object that was being run when the example failed.

UnexpectedException.example

The `Example` that failed.

UnexpectedException.exc_info

A tuple containing information about the unexpected exception, as returned by `sys.exc_info`.

Soapbox

As mentioned in the introduction, `!doctest` has grown to have three primary uses:

1. Checking examples in docstrings.

2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned---it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my `!doctest` examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using `testfile` or `DocFileSuite`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `~module.__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
```

```
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

Footnotes

[^1]: Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

□□□ #2
□ □□□ □□□ 2026-10-31 17:08:23 UTC
□ □□□ □□□ 2026-10-31 17:08:26 UTC