
xdoctest Documentation

Release 1.3.3

Jon Crall

Apr 12, 2026

CONTENTS

1	Getting Started 0: Installation	3
2	Getting Started 1: Your first doctest	5
3	Getting Started 2: Running your doctests	7
3.1	Using the pytest interface	7
3.2	Using the native interface.	7
4	xdoctest package	11
4.1	Subpackages	11
4.1.1	xdoctest.docstr package	11
4.1.1.1	Submodules	11
4.1.1.1.1	xdoctest.docstr.docscrape_google module	11
4.1.1.1.2	xdoctest.docstr.docscrape_numpy module	16
4.1.1.2	Module contents	16
4.1.2	xdoctest.utils package	21
4.1.2.1	Submodules	21
4.1.2.1.1	xdoctest.utils.util_deprecation module	21
4.1.2.1.2	xdoctest.utils.util_import module	22
4.1.2.1.3	xdoctest.utils.util_misc module	33
4.1.2.1.4	xdoctest.utils.util_mixins module	33
4.1.2.1.5	xdoctest.utils.util_notebook module	34
4.1.2.1.6	xdoctest.utils.util_path module	37
4.1.2.1.7	xdoctest.utils.util_str module	37
4.1.2.1.8	xdoctest.utils.util_stream module	41
4.1.2.2	Module contents	43
4.2	Submodules	57
4.2.1	xdoctest.__main__ module	57
4.2.2	xdoctest._tokenize module	58
4.2.3	xdoctest.checker module	59
4.2.4	xdoctest.constants module	63
4.2.5	xdoctest.core module	64
4.2.6	xdoctest.demo module	69
4.2.7	xdoctest.directive module	71
4.2.7.1	Basic Directives	71
4.2.7.2	Advanced Directives	72
4.2.8	xdoctest.doctest_example module	81
4.2.9	xdoctest.doctest_part module	90
4.2.10	xdoctest.dynamic_analysis module	93
4.2.11	xdoctest.exceptions module	95

4.2.12	xdoctest.global_state module	96
4.2.13	xdoctest.parser module	96
	4.2.13.1 The XDoctest Parser	96
4.2.14	xdoctest.plugin module	103
4.2.15	xdoctest.runner module	103
	4.2.15.1 The Native XDoctest Runner	103
	4.2.15.2 Using the XDoctest Runner via the Terminal	103
	4.2.15.3 Using the XDoctest Runner Programmatically	103
4.2.16	xdoctest.static_analysis module	106
4.3	Module contents	118
4.3.1	Xdoctest - Execute Doctests	118
	4.3.1.1 Getting Started 0: Installation	118
	4.3.1.2 Getting Started 1: Your first doctest	118
	4.3.1.3 Getting Started 2: Running your doctests	119
	4.3.1.3.1 Using the pytest interface	120
	4.3.1.3.2 Using the native interface.	120
5	Running Doctests in Jupyter Notebooks	125
5.1	Method 1 - Inside the notebook	125
5.2	Method 2 - Outside the notebook	125
5.3	Caveats	125
6	Doctests with async code	127
6.1	ASYNC Directive	128
6.2	Caveats	129
7	Indices and tables	131
	Bibliography	133
	Python Module Index	135
	Index	137



Xdoctest is a Python package for executing tests in documentation strings!

What is a `doctest`? It is example code you write in a docstring! What is a `docstring`? Its a string you use as a comment! They get attached to Python functions and classes as metadata. They are often used to auto-generate documentation. Why is it cool? Because you can write tests while you code!

Xdoctest finds and executes your doctests for you. Just run `xdoctest <path-to-my-module>`. It plugs into `pytest` to make it easy to run on a CI. Install and run `pytest --xdoctest`.

The `xdoctest` package is a re-write of Python's builtin `doctest` module. It replaces the old regex-based parser with a new abstract-syntax-tree based parser (using Python's `ast` module). The goal is to make doctests easier to write, simpler to configure, and encourage the pattern of test driven development.

Read the docs	http://xdoctest.readthedocs.io/en/latest
Github	https://github.com/Erotemic/xdoctest
Pypi	https://pypi.org/project/xdoctest
PyCon 2020	Youtube Video and Google Slides

GETTING STARTED 0: INSTALLATION

First ensure that you have Python installed and ideally are in a virtual environment. Install xdoctest using the pip.

```
pip install xdoctest
```

Alternatively you can install xdoctest with optional packages.

```
pip install xdoctest[all]
```

This ensures that the `pygments` and `colorama` packages are installed, which are required to color terminal output.

GETTING STARTED 1: YOUR FIRST DOCTEST

If you already know how to write a doctest then you can skip to the next section. If you aren't familiar with doctests, this will help get you up to speed.

Consider the following implementation the Fibonacci function.

```
def fib(n):
    """
    Python 3: Fibonacci series up to n
    """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

We can add a “doctest” in the “docstring” as both an example and a test of the code. All we have to do is prefix the doctest code with three right chevrons `>>>`. We can also use `xdoctest` directives to control the flow of doctest execution.

```
def fib(n):
    """
    Python 3: Fibonacci series up to n

    Example:
    >>> fib(1000) # xdoctest: +SKIP
    0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
    """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

Now if this text was in a file called `fib.py` you could execute your doctest by running `xdoctest fib.py`. Note that if `fib.py` was in a package called `mymod`, you could equivalently run `xdoctest -m mymod.fib`. In other words you can all doctests in a file by passing `xdoctest` the module name or the module path.

Interestingly because this documentation is written in the `xdoctest/__init__.py` file, which is a Python file, that means we can write doctests in it. If you have `xdoctest` installed, you can use the `xdoctest` cli to execute the following code: `xdoctest -m xdoctest.__init__ __doc__:0`. Also notice that the previous doctest is skipped due to the `SKIP` directive. For more information on directives see [the docs for the xdoctest directive module](#).

```
>>> # Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(25)
0 1 1 2 3 5 8 13 21
```

GETTING STARTED 2: RUNNING YOUR DOCTESTS

There are two ways to run `xdoctest`: (1) `pytest` or (2) the native `xdoctest` interface. The native interface is less opaque and implicit, but its purpose is to run doctests. The other option is to use the widely used `pytest` package. This allows you to run both unit tests and doctests with the same command and has many other advantages.

It is recommended to use `pytest` for automatic testing (e.g. in your CI scripts), but for debugging it may be easier to use the native interface.

3.1 Using the `pytest` interface

When `pytest` is run, `xdoctest` is automatically discovered, but is disabled by default. This is because `xdoctest` needs to replace the builtin `pytest._pytest.doctest` plugin.

To enable this plugin, run `pytest` with `--xdoctest` or `--xdoc`. This can either be specified on the command line or added to your `addopts` options in the `[pytest]` section of your `pytest.ini` or `tox.ini`.

To run a specific doctest, `xdoctest` sets up `pytest` node names for these doctests using the following pattern: `<path/to/file.py>::<callname>:<num>`. For example a doctest for a function might look like this `mymod.py::funcname:0`, and a class method might look like this: `mymod.py::ClassName::method:0`

3.2 Using the native interface.

The `xdoctest` module contains a `pytest` plugin, but also contains a native command line interface (CLI). The CLI is generated using `argparse`.

For help you can run

```
xdoctest --help
```

which produces something similar to the following output:

```
usage: xdoctest [-h] [--version] [-m MODNAME] [-c COMMAND] [--style {auto,google,
↪freeform}] [--analysis {auto,static,dynamic}] [--durations DURATIONS] [--time]
           [--colored COLORED] [--nocolor] [--offset] [--report {none,cdiff,ndiff,
↪udiff,only_first_failure}] [--options OPTIONS] [--global-exec GLOBAL_EXEC]
           [--verbose VERBOSE] [--quiet] [--silent]
           [arg ...]
```

```
Xdoctest 1.0.0 - on Python - 3.9.9 (main, Jan 6 2022, 18:33:12)
[GCC 10.3.0] - discover and run doctests within a python package
```

```
positional arguments:
```

(continues on next page)

(continued from previous page)

```

    arg                Ignored if optional arguments are specified, otherwise: Defaults
↳ --modname to arg.pop(0). Defaults --command to arg.pop(0). (default: None)

optional arguments:
  -h, --help          show this help message and exit
  --version           Display version info and quit (default: False)
  -m MODNAME, --modname MODNAME
                    Module name or path. If specified positional modules are ignored
↳ (default: None)
  -c COMMAND, --command COMMAND
                    A doctest name or a command (list|all|<callname>). Defaults to
↳ all (default: None)
  --style {auto,google,freeform}
                    Choose the style of doctests that will be parsed (default: auto)
  --analysis {auto,static,dynamic}
                    How doctests are collected (default: auto)
  --durations DURATIONS
                    Specify execution times for slowest N tests.N=0 will show times
↳ for all tests (default: None)
  --time             Same as if durations=0 (default: False)
  --colored COLORED Enable or disable ANSI coloration in stdout (default: True)
  --nocolor         Disable ANSI coloration in stdout
  --offset          If True formatted source linenumbers will agree with their
↳ location in the source file. Otherwise they will be relative to the doctest itself.
↳ (default:
                    False)
  --report {none,cdiff,ndiff,udiff,only_first_failure}
                    Choose another output format for diffs on xdoctest failure
↳ (default: udiff)
  --options OPTIONS Default directive flags for doctests (default: None)
  --global-exec GLOBAL_EXEC
                    Custom Python code to execute before every test (default: None)
  --verbose VERBOSE Verbosity level. 0 is silent, 1 prints out test names, 2
↳ additionally prints test stdout, 3 additionally prints test source (default: 3)
  --quiet           sets verbosity to 1
  --silent         sets verbosity to 0

```

The xdoctest interface can be run programmatically using `xdoctest.doctest_module(path)`, which can be placed in the `__main__` section of any module as such:

```

if __name__ == '__main__':
    import xdoctest
    xdoctest.doctest_module(__file__)

```

This sets up the ability to invoke the xdoctest command line interface by invoking your module as a `main script`: `python -m <modname> <command>`, where `<modname>` is the name of your module (e.g. `foo.bar`) and `<command>` follows the following rules:

- If `<command>` is `all`, then each enabled doctest in the module is executed: `python -m <modname> all`
- If `<command>` is `list`, then the names of each enabled doctest is listed.
- If `<command>` is `dump`, then all doctests are converted into a format suitable for unit testing, and dumped to stdout (new in 0.4.0).

- If `<command>` is a “callname” (name of a function or a class and method), then that specific doctest is executed: `python -m <modname> <callname>`. Note: you can execute disabled doctests or functions without any arguments (zero-args) this way.

XDoctest is a good demonstration of itself. After pip installing xdoctest, try running xdoctest on xdoctest.

```
xdoctest xdoctest
```

If you would like a slightly less verbose output, try

```
xdoctest xdoctest --verbose=1
# or
xdoctest xdoctest --verbose=0
```

You could also consider running xdoctests tests through pytest:

```
pytest $(python -c 'import xdoctest, pathlib; print(pathlib.Path(xdoctest.__file__).parent)') --xdoctest
```

If you would like a slightly more verbose output, try

```
pytest -s --verbose --xdoctest-verbose=3 --xdoctest $(python -c 'import xdoctest, pathlib; print(pathlib.Path(xdoctest.__file__).parent)')
```

If you ran these commands, the myriad of characters that flew across your screen are lots more examples of what you can do with doctests.

You can also run doctests *inside Jupyter Notebooks*.

XDOCTEST PACKAGE

4.1 Subpackages

4.1.1 xdoctest.docstr package

4.1.1.1 Submodules

4.1.1.1.1 xdoctest.docstr.docscrape_google module

Handles parsing of information out of google style docstrings

It is not clear which of these *GoogleStyleDocs1* *GoogleStyleDocs2* is *the* standard or if there is one.

This code has been exported to a standalone package

- <https://github.com/Erotemic/googledoc>

This is similar to:

- <https://pypi.org/project/docstring-parser/>
- <https://pypi.org/project/numpydoc/>

It hasn't been decided if this will remain vendored in xdoctest or pulled in as a dependency.

References

class xdoctest.docstr.docscrape_google.DocBlock(*text*, *offset*)

Bases: `NamedTuple`

Create new instance of DocBlock(text, offset)

text: `str`

Alias for field number 0

offset: `int`

Alias for field number 1

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('text', 'offset')

classmethod **_make**(*iterable*)

Make a new DocBlock object from a sequence or iterable

`_replace(**kws)`

Return a new DocBlock object replacing specified fields with new values

`xdoctest.docstr.docscrape_google.split_google_docblocks(docstr: str) → list[tuple[str, DocBlock]]`

Breaks a docstring into parts defined by google style

Parameters

`docstr (str)` – a docstring

Returns

list of 2-tuples where the first item is a google style docstring tag and the second item is the block corresponding to that tag. The block itself is a 2-tuple where the first item is the unindented text and the second item is the line offset indicating that blocks location in the docstring.

Return type

List[Tuple[str, DocBlock]]

Note

Unknown or “freeform” sections are given a generic “__DOC__” tag. A section tag may be specified multiple times.

CommandLine

```
xdoctest xdoctest.docstr.docscrape_google split_google_docblocks:2
```

Example

```
>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> from xdoctest import utils
>>> docstr = utils.codeblock(
...     """
...     one line description
...
...     multiline
...     description
...
...     Args:
...         foo: bar
...
...     Returns:
...         None
...
...     Example:
...         >>> print('eg1')
...         eg1
...
...     Example:
...         >>> print('eg2')
...         eg2
...     """)
>>> groups = split_google_docblocks(docstr)
>>> assert len(groups) == 5
```

(continues on next page)

(continued from previous page)

```
>>> [g[0] for g in groups]
['__DOC__', 'Args', 'Returns', 'Example', 'Example']
```

Example

```
>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> docstr = split_google_docblocks.__doc__
>>> groups = split_google_docblocks(docstr)
```

Example

```
>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> from xdoctest import utils
>>> docstr = utils.codeblock(
...     """
...     a description with a leading space
...
...     Example:
...     >>> foobar
...     """)
>>> groups = split_google_docblocks(docstr)
>>> print('groups = {!r}'.format(groups))
```

Example

```
>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> from xdoctest import utils
>>> docstr = utils.codeblock(
...     """
...     Example:
...     >>> foobar
...     """)
>>> # Check that line offsets are valid if the first line is not blank
>>> groups = split_google_docblocks(docstr)
>>> offset = groups[0][1][1]
>>> print('offset = {!r}'.format(offset))
>>> assert offset == 0
>>> # Check that line offsets are valid if the first line is blank
>>> groups = split_google_docblocks(chr(10) + docstr)
>>> offset = groups[0][1][1]
>>> print('offset = {!r}'.format(offset))
>>> assert offset == 1
```

`xdoctest.docstr.docscrape_google.parse_google_args(docstr: str) → Iterator[dict[str, str]]`

Generates dictionaries of argument hints based on a google docstring

Parameters

docstr (*str*) – a google-style docstring

Yields

Dict[str, str] – dictionaries of parameter hints

Example

```
>>> docstr = parse_google_args.__doc__
>>> argdict_list = list(parse_google_args(docstr))
>>> print([sorted(d.items()) for d in argdict_list])
[[('desc', 'a google-style docstring'), ('name', 'docstr'), ('type', 'str')]]
```

`xdoctest.docstr.docscrape_google.parse_google_returns`(*docstr: str, return_annot: str | None = None*)
→ `Iterator[dict[str, str]]`

Generates dictionaries of possible return hints based on a google docstring

Parameters

- **docstr** (*str*) – a google-style docstring
- **return_annot** (*str | None*) – the return type annotation (if one exists)

Yields

`Dict[str, str]` – dictionaries of return value hints

Example

```
>>> docstr = parse_google_returns.__doc__
>>> retdict_list = list(parse_google_returns(docstr))
>>> print([sorted(d.items()) for d in retdict_list])
[[('desc', 'dictionaries of return value hints'), ('type', 'Dict[str, str]')]]
```

Example

```
>>> docstr = split_google_docblocks.__doc__
>>> retdict_list = list(parse_google_returns(docstr))
>>> print([sorted(d.items())[1] for d in retdict_list])
[('type', 'List[Tuple[str, DocBlock]]')]
```

`xdoctest.docstr.docscrape_google.parse_google_retblock`(*lines: str, return_annot: str | None = None*)
→ `Iterator[dict[str, Any]]`

Parse information out of a returns or yields block.

A returns or yeids block should be formatted as one or more '`{type}:{description}`' strings. The description can occupy multiple lines, but the indentation should increase.

Parameters

- **lines** (*str*) – unindented lines from a Returns or Yields section
- **return_annot** (*str | None*) – the return type annotation (if one exists)

Yields

`Dict[str, str]` – each dict specifies the return type and its description

Example

```
>>> # Test various ways that retlines can be written
>>> assert len(list(parse_google_retblock('list: a desc'))) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     'entire line can be desc',
```

(continues on next page)

(continued from previous page)

```

...     ' ',
...     ' if a return type annotation is given',
... ], return_annot='int'))
>>> assert len(hints) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     'bool: a description',
...     ' with a newline',
... ])))
>>> assert len(hints) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     'int or bool: a description',
...     ' ',
...     ' with a separated newline',
...     ' ',
... ])))
>>> assert len(hints) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     # Multiple types can be specified
...     'threading.Thread: a description',
...     '(int, str): a tuple of int and str',
...     'tuple: a tuple of int and str',
...     'Tuple[int, str]: a tuple of int and str',
... ])))
>>> assert len(hints) == 4
>>> # ---
>>> # If the colon is not specified nothing will be parsed
>>> # according to the "official" spec, but lets try and do it anyway
>>> hints = list(parse_google_retblock('\n'.join([
...     'list',
...     'Tuple[int, str]',
... ])))
>>> assert len(hints) == 2
>>> assert len(list(parse_google_retblock('no type, just desc'))) == 1
...

```

`xdoctest.docstr.docscrape_google.parse_google_argblock`(*lines: str, clean_desc: bool = True*) → `Iterator[dict[str, str]]`

Parse out individual items from google-style args blocks.

Parameters

- **lines** (*str*) – the unindented lines from an Args docstring section
- **clean_desc** (*bool*) – if True, will strip the description of newlines and indents. Defaults to True.

Yields

`Dict[str, str]` – A dictionary containing keys, “name”, “type”, and “desc” corresponding to an argument in the Args block.

Example

```

>>> # Test various ways that arglines can be written
>>> line_list = [
...     '',
...     'foo1 (int): a description',
...     'foo2: a description\n    with a newline',
...     'foo3 (int or str): a description',
...     'foo4 (int or threading.Thread): a description',
...     #
...     # this is sphynx-like typing style
...     'param1 (:obj:`str`, optional): ',
...     'param2 (:obj:`list` of :obj:`str`):',
...     #
...     # the Type[type] syntax is defined by the python typeing module
...     'attr1 (Optional[int]): Description of `attr1`.',
...     'attr2 (List[str]): Description of `attr2`.',
...     'attr3 (Dict[str, str]): Description of `attr3`.',
...     '*args : variable positional args description',
...     '**kwargs : keyword arguments description',
...     'malformed and unparseable',
...     'param_no_desc1', # todo: this should be parseable
...     'param_no_desc2:',
...     'param_no_desc3 ()', # todo: this should be parseable
...     'param_no_desc4 ():',
...     'param_no_desc5 (str)', # todo: this should be parseable
...     'param_no_desc6 (str):',
... ]
>>> lines = '\n'.join(line_list)
>>> argdict_list = list(parse_google_argblock(lines))
>>> # All lines except the first should be accepted
>>> assert len(argdict_list) == len(line_list) - 5
>>> assert argdict_list[1]['desc'] == 'a description with a newline'

```

4.1.1.1.2 xdoctest.docstr.docscrape_numpy module**4.1.1.2 Module contents**

`xdoctest.docstr.parse_google_argblock`(*lines: str, clean_desc: bool = True*) → `Iterator[dict[str, str]]`

Parse out individual items from google-style args blocks.

Parameters

- **lines** (*str*) – the unindented lines from an Args docstring section
- **clean_desc** (*bool*) – if True, will strip the description of newlines and indents. Defaults to True.

Yields

`Dict[str, str]` – A dictionary containing keys, “name”, “type”, and “desc” corresponding to an argument in the Args block.

Example

```

>>> # Test various ways that arglines can be written
>>> line_list = [
...     '',
...     'foo1 (int): a description',
...     'foo2: a description\n    with a newline',
...     'foo3 (int or str): a description',
...     'foo4 (int or threading.Thread): a description',
...     #
...     # this is sphynx-like typing style
...     'param1 (:obj:`str`, optional): ',
...     'param2 (:obj:`list` of :obj:`str`):',
...     #
...     # the Type[type] syntax is defined by the python typeing module
...     'attr1 (Optional[int]): Description of `attr1`.',
...     'attr2 (List[str]): Description of `attr2`.',
...     'attr3 (Dict[str, str]): Description of `attr3`.',
...     '*args : variable positional args description',
...     '**kwargs : keyword arguments description',
...     'malformed and unparseable',
...     'param_no_desc1', # todo: this should be parseable
...     'param_no_desc2:',
...     'param_no_desc3 ()', # todo: this should be parseable
...     'param_no_desc4 ():',
...     'param_no_desc5 (str)', # todo: this should be parseable
...     'param_no_desc6 (str):',
... ]
>>> lines = '\n'.join(line_list)
>>> argdict_list = list(parse_google_argblock(lines))
>>> # All lines except the first should be accepted
>>> assert len(argdict_list) == len(line_list) - 5
>>> assert argdict_list[1]['desc'] == 'a description with a newline'

```

`xdoctest.docstr.parse_google_args(docstr: str) → Iterator[dict[str, str]]`

Generates dictionaries of argument hints based on a google docstring

Parameters

docstr (*str*) – a google-style docstring

Yields

Dict[str, str] – dictionaries of parameter hints

Example

```

>>> docstr = parse_google_args.__doc__
>>> argdict_list = list(parse_google_args(docstr))
>>> print([sorted(d.items()) for d in argdict_list])
[[('desc', 'a google-style docstring'), ('name', 'docstr'), ('type', 'str')]]

```

`xdoctest.docstr.parse_google_retblock(lines: str, return_annot: str | None = None) → Iterator[dict[str, Any]]`

Parse information out of a returns or yields block.

A returns or yeids block should be formatted as one or more '`{type}:{description}`' strings. The description

can occupy multiple lines, but the indentation should increase.

Parameters

- **lines** (*str*) – unindented lines from a Returns or Yields section
- **return_annot** (*str* | *None*) – the return type annotation (if one exists)

Yields

Dict[str, str] – each dict specifies the return type and its description

Example

```
>>> # Test various ways that retlines can be written
>>> assert len(list(parse_google_retblock('list: a desc'))) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     'entire line can be desc',
...     ' ',
...     ' if a return type annotation is given',
... ]), return_annot='int'))
>>> assert len(hints) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     'bool: a description',
...     ' with a newline',
... ])))
>>> assert len(hints) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     'int or bool: a description',
...     ' ',
...     ' with a separated newline',
...     ' ',
... ])))
>>> assert len(hints) == 1
>>> # ---
>>> hints = list(parse_google_retblock('\n'.join([
...     # Multiple types can be specified
...     'threading.Thread: a description',
...     '(int, str): a tuple of int and str',
...     'tuple: a tuple of int and str',
...     'Tuple[int, str]: a tuple of int and str',
... ])))
>>> assert len(hints) == 4
>>> # ---
>>> # If the colon is not specified nothing will be parsed
>>> # according to the "official" spec, but lets try and do it anyway
>>> hints = list(parse_google_retblock('\n'.join([
...     'list',
...     'Tuple[int, str]',
... ])))
>>> assert len(hints) == 2
>>> assert len(list(parse_google_retblock('no type, just desc'))) == 1
...

```

`xdoctest.docstr.parse_google_returns(docstr: str, return_annot: str | None = None) → Iterator[dict[str, str]]`

Generates dictionaries of possible return hints based on a google docstring

Parameters

- **docstr** (*str*) – a google-style docstring
- **return_annot** (*str | None*) – the return type annotation (if one exists)

Yields

Dict[str, str] – dictionaries of return value hints

Example

```
>>> docstr = parse_google_returns.__doc__
>>> retdict_list = list(parse_google_returns(docstr))
>>> print([sorted(d.items()) for d in retdict_list])
[[('desc', 'dictionaries of return value hints'), ('type', 'Dict[str, str]')]]
```

Example

```
>>> docstr = split_google_docblocks.__doc__
>>> retdict_list = list(parse_google_returns(docstr))
>>> print([sorted(d.items())[1] for d in retdict_list])
[('type', 'List[Tuple[str, DocBlock]')]]
```

`xdoctest.docstr.split_google_docblocks(docstr: str) → list[tuple[str, DocBlock]]`

Breaks a docstring into parts defined by google style

Parameters

docstr (*str*) – a docstring

Returns

list of 2-tuples where the first item is a google style docstring tag and the second item is the block corresponding to that tag. The block itself is a 2-tuple where the first item is the unindented text and the second item is the line offset indicating that blocks location in the docstring.

Return type

List[Tuple[str, DocBlock]]

Note

Unknown or “freeform” sections are given a generic “__DOC__” tag. A section tag may be specified multiple times.

CommandLine

```
xdoctest xdoctest.docstr.docscrape_google split_google_docblocks:2
```

Example

```
>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> from xdoctest import utils
```

(continues on next page)

(continued from previous page)

```

>>> docstr = utils.codeblock(
...     """
...     one line description
...
...     multiline
...     description
...
...     Args:
...         foo: bar
...
...     Returns:
...         None
...
...     Example:
...         >>> print('eg1')
...         eg1
...
...     Example:
...         >>> print('eg2')
...         eg2
...     """)
>>> groups = split_google_docblocks(docstr)
>>> assert len(groups) == 5
>>> [g[0] for g in groups]
['__DOC__', 'Args', 'Returns', 'Example', 'Example']

```

Example

```

>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> docstr = split_google_docblocks.__doc__
>>> groups = split_google_docblocks(docstr)

```

Example

```

>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> from xdoctest import utils
>>> docstr = utils.codeblock(
...     """
...     a description with a leading space
...
...     Example:
...         >>> foobar
...     """)
>>> groups = split_google_docblocks(docstr)
>>> print('groups = {!r}'.format(groups))

```

Example

```

>>> from xdoctest.docstr.docscrape_google import * # NOQA
>>> from xdoctest import utils
>>> docstr = utils.codeblock(

```

(continues on next page)

(continued from previous page)

```

...     """
...     Example:
...         >>> foobar
...     """
>>> # Check that line offsets are valid if the first line is not blank
>>> groups = split_google_docblocks(docstr)
>>> offset = groups[0][1][1]
>>> print('offset = {!r}'.format(offset))
>>> assert offset == 0
>>> # Check that line offsets are valid if the first line is blank
>>> groups = split_google_docblocks(chr(10) + docstr)
>>> offset = groups[0][1][1]
>>> print('offset = {!r}'.format(offset))
>>> assert offset == 1

```

4.1.2 xdoctest.utils package

4.1.2.1 Submodules

4.1.2.1.1 xdoctest.utils.util_deprecation module

Utilities for helping robustly deprecate features.

`xdoctest.utils.util_deprecation.schedule_deprecation(modname: str, name: str = '?', type: str = '?', migration: str = "", deprecate: str | None = None, error: str | None = None, remove: str | None = None) → None`

Deprecation machinery to help provide users with a smoother transition.

This function provides a concise way to mark a feature as deprecated by providing a description of the deprecated feature, documentation on how to migrate away from the deprecated feature, and the versions that the feature is scheduled for deprecation and eventual removal. Based on the version of the library and the specified schedule this function will either do nothing, emit a warning, or raise an error with helpful messages for both users and developers.

Parameters

- **modname** (*str*) – The name of the underlying module associated with the feature to be deprecated. The module must already be imported and have a passable `__version__` attribute.
- **name** (*str*) – The name of the feature to deprecate. This is usually a function or argument name.
- **type** (*str*) – A description of what the feature is. This is not a formal type, but rather a prose description: e.g. “argument to my_func”.
- **migration** (*str*) – A description that lets users know what they should do instead of using the deprecated feature.
- **deprecate** (*str* | *None*) – The version when the feature is officially deprecated and this function should start to emit a deprecation warning.
- **error** (*str* | *None*) – The version when the feature is officially no longer supported, and will start to raise a `RuntimeError`.
- **remove** (*str* | *None*) – The version when the feature is completely removed. An `AssertionError` will be raised if this function is still present reminding the developer to remove the feature (or extend the remove version).

Note

The `DeprecationWarning` is not visible by default. <https://docs.python.org/3/library/warnings.html>

Example

```
>>> import sys
>>> import types
>>> import pytest
>>> dummy_module = sys.modules['dummy_module'] = types.ModuleType('dummy_module')
>>> # When less than the deprecated version this does nothing
>>> dummy_module.__version__ = '1.0.0'
>>> schedule_deprecation(
...     'dummy_module', 'myfunc', 'function', 'do something else',
...     deprecate='1.1.0', error='1.2.0', remove='1.3.0')
>>> # Now this raises warning
>>> with pytest.warns(DeprecationWarning):
...     dummy_module.__version__ = '1.1.0'
...     schedule_deprecation(
...         'dummy_module', 'myfunc', 'function', 'do something else',
...         deprecate='1.1.0', error='1.2.0', remove='1.3.0')
>>> # Now this raises an error for the user
>>> with pytest.raises(RuntimeError):
...     dummy_module.__version__ = '1.2.0'
...     schedule_deprecation(
...         'dummy_module', 'myfunc', 'function', 'do something else',
...         deprecate='1.1.0', error='1.2.0', remove='1.3.0')
>>> # Now this raises an error for the developer
>>> with pytest.raises(AssertionError):
...     dummy_module.__version__ = '1.3.0'
...     schedule_deprecation(
...         'dummy_module', 'myfunc', 'function', 'do something else',
...         deprecate='1.1.0', error='1.2.0', remove='1.3.0')
>>> # When no versions are specified, it simply emits the warning
>>> with pytest.warns(DeprecationWarning):
...     dummy_module.__version__ = '1.1.0'
...     schedule_deprecation(
...         'dummy_module', 'myfunc', 'function', 'do something else')
```

4.1.2.1.2 xdoctest.utils.util_import module

This file was autogenerated based on code in `ubelt` via `dev/port_ubelt_utils.py` in the `xdoctest` repo

`xdoctest.utils.util_import.is_modname_importable(modname: str, sys_path: list | None = None, exclude: list | None = None) → bool`

Determines if a modname is importable based on your current `sys.path`

Parameters

- **modname** (*str*) – name of module to check
- **sys_path** (*list | None*) – if specified overrides `sys.path`
- **exclude** (*list | None*) – list of directory paths. if specified prevents these directories from being searched.

Returns

True if the module can be imported

Return type

bool

Example

```
>>> is_modname_importable('xdoctest')
True
>>> is_modname_importable('not_a_real_module')
False
>>> is_modname_importable('xdoctest', sys_path=[])
False
```

`xdoctest.utils.util_import._importlib_import_modpath(modpath: str | PathLike) → ModuleType`

Alternative to `import_module_from_path` using `importlib` mechanisms

Parameters

modname (*str*) – the module name.

`xdoctest.utils.util_import._importlib_modname_to_modpath(modname: str) → str`

faster version of `_syspath_modname_to_modpath()` using builtin python mechanisms, but unfortunately it doesn't play nice with `pytest`.

Parameters

modname (*str*) – the module name.

Note

This has a side effect when you import subpackages, it forces the parent package itself to import!

Example

```
>>> # xdoctest: +SKIP
>>> modname = 'xdoctest.static_analysis'
>>> _importlib_modname_to_modpath(modname)
...static_analysis.py
>>> _importlib_modname_to_modpath('json')
...json...
```

`class xdoctest.utils.util_import.PythonPathContext(dpath: str | PathLike, index: int = 0)`

Bases: `object`

Context for temporarily adding a dir to the `PYTHONPATH`.

Used in testing, and used as a helper in certain `ubelt` functions.

Warning

Even though this context manager takes precautions, this modifies `sys.path`, and things can go wrong when that happens. This is generally safe as long as nothing else you do inside of this context modifies the path. If the path is modified in this context, we will try to detect it and warn.

Variables

- **dpath** (*str* | *PathLike*) – directory to insert into the PYTHONPATH
- **index** (*int*) – position to add to. Typically either -1 or 0.

Example

```
>>> import sys
>>> with PythonPathContext('foo', -1):
>>>     assert sys.path[-1] == 'foo'
>>> assert sys.path[-1] != 'foo'
>>> with PythonPathContext('bar', 0):
>>>     assert sys.path[0] == 'bar'
>>> assert sys.path[0] != 'bar'
```

Example

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> # Mangle the path inside the context
>>> import sys
>>> self = PythonPathContext('foo', 0)
>>> self.__enter__()
>>> sys.path.insert(0, 'mangled')
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     self.__exit__(None, None, None)
```

Example

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> import sys
>>> self = PythonPathContext('foo', 0)
>>> self.__enter__()
>>> sys.path.remove('foo')
>>> import pytest
>>> with pytest.raises(RuntimeError):
>>>     self.__exit__(None, None, None)
```

Parameters

- **dpath** (*str* | *PathLike*) – directory to insert into the PYTHONPATH
- **index** (*int*) – position to add to. Typically either -1 or 0.

dpath: *str*

index: *int*

`xdoctest.utils.util_import._custom_import_modpath(modpath: str | PathLike, index: int = -1) → ModuleType`

`xdoctest.utils.util_import.import_module_from_path(modpath: str | PathLike, index: int = -1) → ModuleType`

Imports a module via a filesystem path.

This works by modifying `sys.path`, importing the module name, and then attempting to undo the change to `sys.path`. This function may produce unexpected results in the case where the imported module itself itself modifies `sys.path` or if there is another conflicting module with the same name.

Parameters

- **modpath** (*str* | *PathLike*) – Path to the module on disk or within a zipfile. Paths within a zipfile can be given by `<path-to>.zip/<path-inside-zip>.py`.
- **index** (*int*) – Location at which we modify `PYTHONPATH` if necessary. If your module name does not conflict, the safest value is `-1`, However, if there is a conflict, then use an index of `0`. The default may change to `0` in the future.

Returns

the imported module

Return type

ModuleType

References

Raises

- **IOError** - when the path to the module does not exist –
- **ImportError** - when the module is unable to be imported –

Note

If the module is part of a package, the package will be imported first. These modules may cause problems when reloading via IPython magic

This can import a module from within a zipfile. To do this `modpath` should specify the path to the zipfile and the path to the module within that zipfile separated by a colon or `pathsep`. E.g. `"/path/to/archive.zip:mymodule.pl"`

Warning

It is best to use this with paths that will not conflict with previously existing modules.

If the `modpath` conflicts with a previously existing module name. And the target module does imports of its own relative to this conflicting path. In this case, the module that was loaded first will win.

For example if you try to import `'/foo/bar/pkg/mod.py'` from the folder structure:

```
- foo/
+- bar/
  +- pkg/
    + __init__.py
    |- mod.py
    |- helper.py
```

If there exists another module named `pkg` already in `sys.modules` and `mod.py` contains the code `from . import helper`, Python will assume `helper` belongs to the `pkg` module already in `sys.modules`. This can

cause a `NameError` or worse — an incorrect helper module.

SeeAlso:

`import_module_from_name()`

Example

```
>>> # xdoctest: +SKIP("ubelt dependency")
>>> import xdoctest
>>> modpath = xdoctest.__file__
>>> module = ub.import_module_from_path(modpath)
>>> assert module is xdoctest
```

Example

```
>>> # Test importing a module from within a zipfile
>>> # xdoctest: +SKIP("ubelt dependency")
>>> import zipfile
>>> from xdoctest import utils
>>> import os
>>> from os.path import join, expanduser, normpath
>>> dpath = expanduser('~/.cache/xdoctest')
>>> dpath = utils.ensuredir(dpath)
>>> #dpath = utils.TempDir().ensure()
>>> # Write to an external module named bar
>>> external_modpath = join(dpath, 'bar.py')
>>> # For pypy support we have to write this using with
>>> with open(external_modpath, 'w') as file:
>>>     file.write('testvar = 1')
>>> internal = 'folder/bar.py'
>>> # Move the external bar module into a zipfile
>>> zippath = join(dpath, 'myzip.zip')
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(external_modpath, internal)
>>> # Import the bar module from within the zipfile
>>> modpath = zippath + ':' + internal
>>> modpath = zippath + os.path.sep + internal
>>> module = ub.import_module_from_path(modpath)
>>> assert normpath(module.__name__) == normpath('folder/bar')
>>> assert module.testvar == 1
```

Example

```
>>> import pytest
>>> # xdoctest: +SKIP("ubelt dependency")
>>> with pytest.raises(IOError):
>>>     ub.import_module_from_path('does-not-exist')
>>> with pytest.raises(IOError):
>>>     ub.import_module_from_path('does-not-exist.zip/')
```

`xdoctest.utils.util_import.import_module_from_name(modname: str) → ModuleType`

Imports a module from its string name (i.e. `__name__`)

This is a simple wrapper around `importlib.import_module()`, but is provided as a companion function to `import_module_from_path()`, which contains functionality not provided in the Python standard library.

Parameters

modname (*str*) – module name

Returns

module

Return type

ModuleType

SeeAlso:

`import_module_from_path()`

Example

```
>>> # test with modules that won't be imported in normal circumstances
>>> # todo write a test where we guarantee this
>>> # xdoctest: +SKIP("ubelt dependency")
>>> import sys
>>> modname_list = [
>>>     'pickletools',
>>>     'email.mime.text',
>>> ]
>>> #assert not any(m in sys.modules for m in modname_list)
>>> modules = [ub.import_module_from_name(modname) for modname in modname_list]
>>> assert [m.__name__ for m in modules] == modname_list
>>> assert all(m in sys.modules for m in modname_list)
```

`xdoctest.utils.util_import._parse_static_node_value(node: Any) → Any`

Extract a constant value from a node if possible

`xdoctest.utils.util_import._extension_module_tags() → list[str]`

Returns valid tags an extension module might have

Returns

List[str]

`xdoctest.utils.util_import._static_parse(varname: Any, fpath: Any) → Any`

Statically parse the a constant variable from a python file

Parameters

- **varname** (*str*) – variable name to extract
- **fpath** (*str* | *PathLike*) – path to python file to parse

Returns

the static value

Return type

Any

Example

```

>>> # xdoctest: +SKIP("ubelt dependency")
>>> dpath = ub.Path.append('tests/import/staticparse').ensuredir()
>>> fpath = (dpath / 'foo.py')
>>> fpath.write_text('a = {1: 2}')
>>> assert _static_parse('a', fpath) == {1: 2}
>>> fpath.write_text('a = 2')
>>> assert _static_parse('a', fpath) == 2
>>> fpath.write_text('a = "3"')
>>> assert _static_parse('a', fpath) == "3"
>>> fpath.write_text('a = ["3", 5, 6]')
>>> assert _static_parse('a', fpath) == ["3", 5, 6]
>>> fpath.write_text('a = ("3", 5, 6)')
>>> assert _static_parse('a', fpath) == ("3", 5, 6)
>>> fpath.write_text('b = 10' + chr(10) + 'a = None')
>>> assert _static_parse('a', fpath) is None
>>> import pytest
>>> with pytest.raises(TypeError):
>>>     fpath.write_text('a = list(range(10))')
>>>     assert _static_parse('a', fpath) is None
>>> with pytest.raises(AttributeError):
>>>     fpath.write_text('a = list(range(10))')
>>>     assert _static_parse('c', fpath) is None
>>> if sys.version_info[0:2] >= (3, 6):
>>>     # Test with type annotations
>>>     fpath.write_text('b: int = 10')
>>>     assert _static_parse('b', fpath) == 10

```

`xdoctest.utils.util_import._platform_pylib_exts()` → `tuple[str, ...]`

Returns `.so`, `.pyd`, or `.dylib` depending on linux, win or mac. On python3 return the previous with and without abi (e.g. `.cpython-35m-x86_64-linux-gnu`) flags. On python2 returns with and without multiarch.

Returns

tuple

`xdoctest.utils.util_import._syspath_modname_to_modpath(modname: Any, sys_path: Any = None, exclude: Any = None)` → `str | None`

syspath version of `modname_to_modpath`

Parameters

- **modname** (*str*) – name of module to find
- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.
- **exclude** (*List[str | PathLike] | None*) – If specified prevents these directories from being searched. Defaults to `None`.

Returns

path to the module or `None` if it does not exist.

Return type

`str | None`

Note

This is much slower than the pkgutil mechanisms.

There seems to be a change to the editable install mechanism: <https://github.com/pypa/setuptools/issues/3548> Trying to find more docs about it.

TODO: add a test where we make an editable install, regular install, standalone install, and check that we always find the right path.

Example

```
>>> print(_syspath_modname_to_modpath('xdoctest.static_analysis'))
...static_analysis.py
>>> print(_syspath_modname_to_modpath('xdoctest'))
...xdoctest
>>> print(_syspath_modname_to_modpath('json'))
...json
>>> assert _syspath_modname_to_modpath('xdoctest', sys_path=[]) is None
>>> assert _syspath_modname_to_modpath('xdoctest.static_analysis', sys_path=[]) is_
↪None
>>> assert _syspath_modname_to_modpath('json', sys_path=[]) is None
>>> assert _syspath_modname_to_modpath('this', sys_path=[]) is None
```

Example

```
>>> # test what happens when the module is not visible in the path
>>> modname = 'xdoctest.static_analysis'
>>> modpath = _syspath_modname_to_modpath(modname)
>>> exclude = [split_modpath(modpath)[0]]
>>> found = _syspath_modname_to_modpath(modname, exclude=exclude)
>>> if found is not None:
>>>     # Note: the basic form of this test may fail if there are
>>>     # multiple versions of the package installed. Try and fix that.
>>>     other = split_modpath(found)[0]
>>>     assert other not in exclude
>>>     exclude.append(other)
>>>     found = _syspath_modname_to_modpath(modname, exclude=exclude)
>>> if found is not None:
>>>     raise AssertionError(
>>>         'should not have found {}'.format(found) +
>>>         ' because we excluded: {}'.format(exclude) +
>>>         ' cwd={}'.format(os.getcwd()) +
>>>         ' sys.path={}'.format(sys.path)
>>>     )
```

`xdoctest.utils.util_import.modname_to_modpath(modname: str, hide_init: bool = True, hide_main: bool = False, sys_path: list[str | PathLike] | None = None) → str | None`

Finds the path to a python module from its name.

Determines the path to a python module without directly import it

Converts the name of a module (`__name__`) to the path (`__file__`) where it is located without importing the

module. Returns None if the module does not exist.

Parameters

- **modname** (*str*) – The name of a module in `sys.path`.
- **hide_init** (*bool*) – if False, `__init__.py` will be returned for packages. Defaults to True.
- **hide_main** (*bool*) – if False, and `hide_init` is True, `__main__.py` will be returned for packages, if it exists. Defaults to False.
- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.

Returns

modpath - path to the module, or None if it doesn't exist

Return type

`str | None`

Example

```
>>> modname = 'xdoctest.__main__'
>>> modpath = modname_to_modpath(modname, hide_main=False)
>>> assert modpath.endswith('__main__.py')
>>> modname = 'xdoctest'
>>> modpath = modname_to_modpath(modname, hide_init=False)
>>> assert modpath.endswith('__init__.py')
>>> modpath = modname_to_modpath('json')
>>> assert 'json' in modpath
```

`xdoctest.utils.util_import.normalize_modpath(modpath: str, hide_init: bool = True, hide_main: bool = False) → str`

`xdoctest.utils.util_import.normalize_modpath(modpath: PathLike, hide_init: bool = True, hide_main: bool = False) → str | PathLike`

Normalizes `__init__` and `__main__` paths.

Parameters

- **modpath** (*str | PathLike*) – path to a module
- **hide_init** (*bool*) – if True, always return package modules as `__init__.py` files otherwise always return the dpath. Defaults to True.
- **hide_main** (*bool*) – if True, always strip away main files otherwise ignore `__main__.py`. Defaults to False.

Returns

a normalized path to the module

Return type

`str | PathLike`

Note

Adds `__init__` if reasonable, but only removes `__main__` by default

Example

```
>>> from xdoctest import static_analysis as module
>>> modpath = module.__file__
>>> assert normalize_modpath(modpath) == modpath.replace('.pyc', '.py')
>>> dpath = dirname(modpath)
>>> res0 = normalize_modpath(dpath, hide_init=0, hide_main=0)
>>> res1 = normalize_modpath(dpath, hide_init=0, hide_main=1)
>>> res2 = normalize_modpath(dpath, hide_init=1, hide_main=0)
>>> res3 = normalize_modpath(dpath, hide_init=1, hide_main=1)
>>> assert res0.endswith('__init__.py')
>>> assert res1.endswith('__init__.py')
>>> assert not res2.endswith('.py')
>>> assert not res3.endswith('.py')
```

`xdoctest.utils.util_import.modpath_to_modname(modpath: str | PathLike, hide_init: bool = True, hide_main: bool = False, check: bool = True, relativeto: str | None = None) → str`

Determines importable name from file path

Converts the path to a module (`__file__`) to the importable python name (`__name__`) without importing the module.

The filename is converted to a module name, and parent directories are recursively included until a directory without an `__init__.py` file is encountered.

Parameters

- **modpath** (*str* | *os.PathLike*) – Module filepath
- **hide_init** (*bool*) – Removes the `__init__` suffix. Defaults to True.
- **hide_main** (*bool*) – Removes the `__main__` suffix. Defaults to False.
- **check** (*bool*) – If False, does not raise an error if modpath is a dir and does not contain an `__init__` file. Defaults to True.
- **relativeto** (*str* | *None*) – If specified, all checks are ignored and this is considered the path to the root module. Defaults to None.

Todo

- [] **Does this need modification to support PEP 420?**
<https://www.python.org/dev/peps/pep-0420/>

Returns

modname

Return type

str

Raises

ValueError – if check is True and the path does not exist

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = modpath.replace('.pyc', '.py')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'xdoctest.static_analysis'
```

Example

```
>>> import xdoctest
>>> assert modpath_to_modname(xdoctest.__file__.replace('.pyc', '.py')) == 'xdoctest'
↳
>>> assert modpath_to_modname(dirname(xdoctest.__file__.replace('.pyc', '.py'))) ==
↳ 'xdoctest'
```

Example

```
>>> modpath = modname_to_modpath('json')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'json'
```

Example

```
>>> modpath = '/foo/libfoobar.linux-x86_64-3.6.so'
>>> modname = modpath_to_modname(modpath, check=False)
>>> assert modname == 'libfoobar'
```

`xdoctest.utils.util_import.split_modpath(modpath: str | PathLike, check: bool = True) → tuple[str, str]`
Splits the modpath into the dir that must be in PYTHONPATH for the module to be imported and the modulepath relative to this directory.

Parameters

- **modpath** (*str*) – module filepath
- **check** (*bool*) – if False, does not raise an error if modpath is a directory and does not contain an `__init__.py` file.

Returns

(directory, rel_modpath)

Return type

Tuple[str, str]

Raises

ValueError – if modpath does not exist or is not a package

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = abspath(modpath)
>>> dpath, rel_modpath = split_modpath(modpath)
>>> recon = join(dpath, rel_modpath)
```

(continues on next page)

(continued from previous page)

```
>>> assert recon == modpath
>>> assert rel_modpath == join('xdoctest', 'static_analysis.py')
```

4.1.2.1.3 xdoctest.utils.util_misc module

Utilities that are mainly used in self-testing

class xdoctest.utils.util_misc.TempDoctest(*docstr: str, modname: str | None = None*)

Bases: `object`

Creates a temporary file containing a module-level doctest for testing

Example

```
>>> from xdoctest import core
>>> self = TempDoctest('>>> a = 1')
>>> doctests = list(core.parse_doctestables(self.modpath))
>>> assert len(doctests) == 1
```

class xdoctest.utils.util_misc.TempModule(*module_text: str, modname: str | None = None*)

Bases: `object`

Creates a temporary directory with a python module.

Example

```
>>> from xdoctest import core
>>> self = TempDoctest('>>> a = 1')
>>> doctests = list(core.parse_doctestables(self.modpath))
>>> assert len(doctests) == 1
```

print_contents() → `None`

For debugging on windows

xdoctest.utils.util_misc._run_case(*source: str, style: str = 'auto'*) → `str | None`

Runs all doctests in a source block

Parameters

source (*str*) – source code of an entire file

TODO: run case is over-duplicated and should be separated into a test utils directory

4.1.2.1.4 xdoctest.utils.util_mixins module

Port of NiceRepr from ubelt.util_mixins

class xdoctest.utils.util_mixins.NiceRepr

Bases: `object`

Defines `__str__` and `__repr__` in terms of `__nice__` function Classes that inherit `NiceRepr` must define `__nice__`

Example

```
>>> class Foo(NiceRepr):
...     pass
>>> class Bar(NiceRepr):
...     def __nice__(self):
...         return 'info'
>>> foo = Foo()
>>> bar = Bar()
>>> assert str(bar) == '<Bar(info)>'
>>> assert repr(bar).startswith('<Bar(info) at ')
>>> assert 'object at' in str(foo)
>>> assert 'object at' in repr(foo)
```

4.1.2.1.5 xdoctest.utils.util_notebook module

Utilities for handling Jupyter / IPython notebooks

This code is copied and modified from nbimporter (<https://github.com/grst/nbimporter/blob/master/nbimporter.py>) which is not actively maintained (otherwise we would use it as a dependency).

Note that using this behavior is very much discouraged, it would be far better if you maintained your reusable code in separate python modules. See <https://github.com/grst/nbimporter> for reasons.

Allow for importing of IPython Notebooks as modules from Jupyter v4.

Updated from module collated here: <https://github.com/adrn/ipython/blob/master/examples/Notebook/Importing%20Notebooks.ipynb>

Importing from a notebook is different from a module: because one typically keeps many computations and tests besides exportable defs, here we only run code which either defines a function or a class, or imports code from other modules and notebooks. This behaviour can be disabled by setting `NotebookLoader.default_options['only_defs'] = False`.

Furthermore, in order to provide per-notebook initialisation, if a special function `__nbinit__()` is defined in the notebook, it will be executed the first time an import statement is. This behaviour can be disabled by setting `NotebookLoader.default_options['run_nbinit'] = False`.

Finally, you can set the encoding of the notebooks with `NotebookLoader.default_options['encoding']`. The default is `'utf-8'`.

`xdoctest.utils.util_notebook._find_notebook(fullname, path=None)`

Find a notebook, given its fully qualified name and an optional path

This turns “foo.bar” into “foo/bar.ipynb” and tries turning “Foo_Bar” into “Foo Bar” if `Foo_Bar` does not exist.

Example

```
>>> # xdoctest: +REQUIRES(PY3, module:IPython, module:nbconvert)
>>> from xdoctest.utils.util_notebook import _find_notebook
>>> from xdoctest import utils
>>> from os.path import join, basename, splitext
>>> self = utils.TempDir()
>>> dpath = self.ensure()
>>> fpath = join(dpath, 'test_import_notebook.ipynb')
```

(continues on next page)

(continued from previous page)

```

>>> cells = ['x = 1']
>>> _make_test_notebook_fpath(fpath, cells)
>>> fullname = splitext(basename(fpath))[0]
>>> path = [dpath]
>>> _find_notebook(fullname, path)
...test_import_notebook.ipynb
>>> _find_notebook(fullname, None)
None

```

class xdoctest.utils.util_notebook.CellDeleterBases: `NodeTransformer`

Removes all nodes from an AST which are not suitable for exporting out of a notebook.

visit(*node*)

Visit a node.

class xdoctest.utils.util_notebook.NotebookLoader(*path=None*)Bases: `object`

Module Loader for Jupyter Notebooks.

default_options = {'encoding': 'utf-8', 'only_defs': False, 'run_nbinit': True}**load_module**(*fullname: str | None = None, fpath: str | PathLike | None = None*)

import a notebook as a module

xdoctest.utils.util_notebook.**import_notebook_from_path**(*ipynb_fpath: str | PathLike, only_defs: bool = False*)

Import an IPython notebook as a module from a full path and try to maintain clean sys.path variables.

Parameters

- **ipynb_fpath** (*str | PathLike*) – path to the ipython notebook file to import
- **only_defs** (*bool, default=False*) – if True ignores all non-definition statements

Example

```

>>> # xdoctest: +REQUIRES(PY3, module:IPython, module:nbconvert)
>>> from xdoctest import utils
>>> from os.path import join
>>> self = utils.TempDir()
>>> dpath = self.ensure()
>>> ipynb_fpath = join(dpath, 'test_import_notebook.ipynb')
>>> cells = [
>>>     utils.codeblock(
>>>         """
>>>         def foo():
>>>             return 'bar'
>>>         """),
>>>     utils.codeblock(
>>>         """
>>>         x = 1
>>>         """)

```

(continues on next page)

(continued from previous page)

```

>>> ]
>>> _make_test_notebook_fpath(ipynb_fpath, cells)
>>> module = import_notebook_from_path(ipynb_fpath)
>>> assert module.foo() == 'bar'
>>> assert module.x == 1

```

`xdoctest.utils.util_notebook.execute_notebook(ipynb_fpath: str | PathLike, timeout: Any = None, verbose: bool | int | None = None) → tuple[Any, dict[str, object]]`

Execute an IPython notebook in a separate kernel

Parameters

ipynb_fpath (*str* | *PathLike*) – path to the ipython notebook file to import

Returns

nb

The executed notebook.

dict: resources

Additional resources used in the conversion process.

Return type

`nbformat.notebooknode.NotebookNode`

Example

```

>>> # xdoctest: +REQUIRES(PY3, module:IPython, module:nbconvert, CPYTHON)
>>> from xdoctest import utils
>>> from os.path import join
>>> self = utils.TempDir()
>>> dpath = self.ensure()
>>> ipynb_fpath = join(dpath, 'hello_world.ipynb')
>>> _make_test_notebook_fpath(ipynb_fpath, [utils.codeblock(
>>>     """
>>>     print('hello world')
>>>     """)])
>>> nb, resources = execute_notebook(ipynb_fpath, verbose=3)
>>> print('resources = {!r}'.format(resources))
>>> print('nb = {!r}'.format(nb))
>>> for cell in nb['cells']:
>>>     if len(cell['outputs']) != 1:
>>>         import warnings
>>>         warnings.warn('expected an output, is this the issue '
>>>             'described [here](https://github.com/nteract/papermill/
↪ issues/426)?')

```

`xdoctest.utils.util_notebook._make_test_notebook_fpath(fpath: Any, cell_sources: Any) → Any`

Helper for testing

Parameters

- **fpath** (*str*) – file to write notebook to
- **cell_sources** (*List[str]*) – list of python code blocks

References

<https://stackoverflow.com/questions/38193878/create-notebook-from-code-9716279>

<https://gist.github.com/fperez/>

4.1.2.1.6 xdoctest.utils.util_path module

Utilities related to filesystem paths

class xdoctest.utils.util_path.TempDir(*persist: bool = False*)

Bases: `object`

Context for creating and cleaning up temporary files. Used in testing.

Example

```
>>> with TempDir() as self:
>>>     dpath = self.dpath
>>>     assert exists(dpath)
>>> assert not exists(dpath)
```

Example

```
>>> self = TempDir()
>>> dpath = self.ensure()
>>> assert exists(dpath)
>>> self.cleanup()
>>> assert not exists(dpath)
```

ensure() → `str`

cleanup() → `None`

xdoctest.utils.util_path.ensuredir(*dpath: str | tuple[str, ...] | list[str], mode: int = 1023*) → `str`

Ensures that directory will exist. creates new dir with sticky bits by default

Parameters

- **dpath** (*str*) – dir to ensure. Can also be a tuple to send to join
- **mode** (*int*) – octal mode of directory (default 0o1777)

Returns

path - the ensured directory

Return type

`str`

4.1.2.1.7 xdoctest.utils.util_str module

Utilities related to string manipulations

xdoctest.utils.util_str.strip_ansi(*text: str*) → `str`

Removes all ansi directives from the string.

Parameters

text (*str*)

Returns

str

References

<http://stackoverflow.com/questions/14693701/remove-ansi-filtering-out-ansi-escape-sequences> <https://stackoverflow.com/questions/13506033/>

Examples

```
>>> line = '\t\u001b[0;35mBlabla\u001b[0m      \u001b[0;36m172.18.0.2\u001b[0m'
>>> escaped_line = strip_ansi(line)
>>> assert escaped_line == '\tBlabla      172.18.0.2'
```

`xdoctest.utils.util_str.color_text(text: str, color: str | None) → str`

Colorizes text a single color using ansi tags.

Parameters

- **text** (*str*) – text to colorize
- **color** (*str*) – may be one of the following: yellow, blink, lightgray, underline, darkyellow, blue, darkblue, faint, fuchsia, black, white, red, brown, turquoise, bold, darkred, darkgreen, reset, standout, darkteal, darkgray, overline, purple, green, teal, fuchsia

Returns

colorized text.

If pygments is not installed plain text is returned.

Return type

str

Example

```
>>> import sys
>>> if sys.platform.startswith('win32'):
>>>     import pytest
>>>     pytest.skip()
>>> text = 'raw text'
>>> from xdoctest import utils
>>> from xdoctest.utils import util_str
>>> if utils.modname_to_modpath('pygments') and not util_str.NO_COLOR:
>>>     # Colors text only if pygments is installed
>>>     import pygments
>>>     print('pygments = {!r}'.format(pygments))
>>>     ansi_text1 = color_text(text, 'red')
>>>     print('ansi_text1 = {!r}'.format(ansi_text1))
>>>     ansi_text = utils.ensure_unicode(ansi_text1)
>>>     prefix = utils.ensure_unicode('\x1b[31')
>>>     print('prefix = {!r}'.format(prefix))
>>>     print('ansi_text = {!r}'.format(ansi_text))
>>>     assert ansi_text.startswith(prefix)
>>>     assert color_text(text, None) == 'raw text'
>>> else:
>>>     # Otherwise text passes through unchanged
```

(continues on next page)

(continued from previous page)

```
>>> assert color_text(text, 'red') == 'raw text'
>>> assert color_text(text, None) == 'raw text'
```

`xdoctest.utils.util_str.ensure_unicode(text: str) → str`

Casts bytes into utf8 (mostly for python2 compatibility)

Parameters

text (*str*)

Returns

str

References

<http://stackoverflow.com/questions/12561063/python-extract-data-from-file>

CommandLine

```
python -m xdoctest.utils ensure_unicode
```

Example

```
>>> assert ensure_unicode('my unicôdé string') == 'my unicôdé string'
>>> assert ensure_unicode('text1') == 'text1'
>>> assert ensure_unicode('text1'.encode('utf8')) == 'text1'
>>> assert ensure_unicode('i»¿text1'.encode('utf8')) == 'i»¿text1'
>>> import codecs
>>> assert (codecs.BOM_UTF8 + 'text»¿'.encode('utf8')).decode('utf8')
```

`xdoctest.utils.util_str.indent(text: str, prefix: str = ' ') → str`

Indents a block of text

Parameters

- **text** (*str*) – text to indent
- **prefix** (*str*) – prefix to add to each line (default = ' ')

Returns

indented text

Return type

str

CommandLine

```
python -m xdoctest.utils ensure_unicode
```

Example

```
>>> text = 'Lorem ipsum\ndolor sit amet'
>>> prefix = '    '
>>> result = indent(text, prefix)
>>> assert all(t.startswith(prefix) for t in result.split('\n'))
```

`xdoctest.utils.util_str.highlight_code`(*text: str, lexer_name: str = 'python', **kwargs: object*) → *str*
 Highlights a block of text using ansi tags based on language syntax.

Parameters

- **text** (*str*) – plain text to highlight
- **lexer_name** (*str*) – name of language
- ****kwargs** – passed to `pygments.lexers.get_lexer_by_name`

Returns

text
 [highlighted text] If `pygments` is not installed, the plain text is returned.

Return type

str

CommandLine

```
python -c "import pygments.formatters; print(list(pygments.formatters.get_all_
→formatters()))"
```

Example

```
>>> text = 'import xdoctest as xdoc; print(xdoc)'
>>> new_text = highlight_code(text)
>>> print(new_text)
```

`xdoctest.utils.util_str.add_line_numbers`(*source: list[str], start: int = 1, n_digits: int | None = None*) → *list[str]*

`xdoctest.utils.util_str.add_line_numbers`(*source: str, start: int = 1, n_digits: int | None = None*) → *str*
 Prefixes code with line numbers

Parameters

- **source** (*str | List[str]*)
- **start** (*int*)
- **n_digits** (*int | None*)

Returns

List[str] | str

Example

```
>>> print(chr(10).join(add_line_numbers(['a', 'b', 'c'])))
1 a
2 b
3 c
>>> print(add_line_numbers(chr(10).join(['a', 'b', 'c'])))
1 a
2 b
3 c
```

`xdoctest.utils.util_str.codeblock(block_str: str) → str`

Wraps multiline string blocks and returns unindented code. Useful for templated code defined in indented parts of code.

Parameters

block_str (*str*) – typically in the form of a multiline string

Returns

the unindented string

Return type

str

Example

```
>>> # Simulate an indented part of code
>>> if True:
...     # notice the indentation on this will be normal
...     codeblock_version = codeblock(
...         """
...         def foo():
...             return 'bar'
...         """
...     )
...     # notice the indentation and newlines on this will be odd
...     normal_version = (
...         def foo():
...             return 'bar'
...         '''
...     )
>>> assert normal_version != codeblock_version
>>> print('Without codeblock')
>>> print(normal_version)
>>> print('With codeblock')
>>> print(codeblock_version)
```

4.1.2.1.8 xdoctest.utils.util_stream module

Functions for capturing and redirecting IO streams.

The `CaptureStdout` captures all text sent to stdout and optionally prevents it from actually reaching stdout.

The `TeeStringIO` does the same thing but for arbitrary streams. It is how the former is implemented.

class `xdoctest.utils.util_stream.TeeStringIO(redirect: IOBase | None = None)`

Bases: `StringIO`

An IO object that writes to itself and another IO stream.

Variables

redirect (*io.IOBase*) – The other stream to write to.

Example

```
>>> redirect = io.StringIO()
>>> self = TeeStringIO(redirect)
```

isatty() → bool

Returns true if the redirect is a terminal.

Note

Needed for IPython.embed to work properly when this class is used to override stdout / stderr.

fileno() → int

Returns underlying file descriptor of the redirected IOBase object if one exists.

property encoding: str | None

Gets the encoding of the *redirect* IO object

Example

```
>>> redirect = io.StringIO()
>>> assert TeeStringIO(redirect).encoding is None
>>> assert TeeStringIO(None).encoding is None
>>> assert TeeStringIO(sys.stdout).encoding is sys.stdout.encoding
>>> redirect = io.TextIOWrapper(io.StringIO())
>>> assert TeeStringIO(redirect).encoding is redirect.encoding
```

write(msg: str) → int

Write to this and the redirected stream

flush() → None

Flush to this and the redirected stream

class xdoctest.utils.util_stream.CaptureStream

Bases: object

Generic class for capturing streaming output from stdout or stderr

class xdoctest.utils.util_stream.CaptureStdout (*suppress: bool = True, enabled: bool = True, **kwargs: object*)

Bases: CaptureStream

Context manager that captures stdout and stores it in an internal stream

Parameters

- **suppress** (*bool, default=True*) – if True, stdout is not printed while captured
- **enabled** (*bool, default=True*) – does nothing if this is False

Example

```
>>> self = CaptureStdout(suppress=True)
>>> print('dont capture the table flip (°° ')
>>> with self:
...     text = 'capture the heart '
...     print(text)
>>> print('dont capture look of disapproval _')
>>> assert isinstance(self.text, str)
>>> assert self.text == text + '\n', 'failed capture text'
```

Example

```
>>> self = CaptureStdout(suppress=False)
>>> with self:
...     print('I am captured and printed in stdout')
>>> assert self.text.strip() == 'I am captured and printed in stdout'
```

Example

```
>>> self = CaptureStdout(suppress=True, enabled=False)
>>> with self:
...     print('dont capture')
>>> assert self.text is None
```

`log_part()` → None

Log what has been captured so far

`start()` → None

`stop()` → None

Example

```
>>> CaptureStdout(enabled=False).stop()
>>> CaptureStdout(enabled=True).stop()
```

`close()` → None

4.1.2.2 Module contents

Most of these utilities exist in `ubelt`, but we copy them here to keep `xdoctest` as a package with minimal dependencies, whereas `ubelt` includes a larger set of utilities.

This `__init__` file is generated using `mkinit`:

```
mkinit xdoctest.utils
```

```
class xdoctest.utils.CaptureStdout(suppress: bool = True, enabled: bool = True, **kwargs: object)
```

Bases: `CaptureStream`

Context manager that captures stdout and stores it in an internal stream

Parameters

- **suppress** (*bool, default=True*) – if True, stdout is not printed while captured
- **enabled** (*bool, default=True*) – does nothing if this is False

Example

```
>>> self = CaptureStdout(suppress=True)
>>> print('dont capture the table flip (°° ')
>>> with self:
...     text = 'capture the heart '
...     print(text)
>>> print('dont capture look of disapproval _')
```

(continues on next page)

(continued from previous page)

```
>>> assert isinstance(self.text, str)
>>> assert self.text == text + '\n', 'failed capture text'
```

Example

```
>>> self = CaptureStdout(suppress=False)
>>> with self:
...     print('I am captured and printed in stdout')
>>> assert self.text.strip() == 'I am captured and printed in stdout'
```

Example

```
>>> self = CaptureStdout(suppress=True, enabled=False)
>>> with self:
...     print('dont capture')
>>> assert self.text is None
```

log_part() → None

Log what has been captured so far

start() → None**stop()** → None**Example**

```
>>> CaptureStdout(enabled=False).stop()
>>> CaptureStdout(enabled=True).stop()
```

close() → None**class** xdoctest.utils.CaptureStream

Bases: object

Generic class for capturing streaming output from stdout or stderr

class xdoctest.utils.NiceRepr

Bases: object

Defines `__str__` and `__repr__` in terms of `__nice__` function Classes that inherit `NiceRepr` must define `__nice__`**Example**

```
>>> class Foo(NiceRepr):
...     pass
>>> class Bar(NiceRepr):
...     def __nice__(self):
...         return 'info'
>>> foo = Foo()
>>> bar = Bar()
>>> assert str(bar) == '<Bar(info)>'
>>> assert repr(bar).startswith('<Bar(info) at ')
```

(continues on next page)

(continued from previous page)

```
>>> assert 'object at' in str(foo)
>>> assert 'object at' in repr(foo)
```

class `xdoctest.utils.PythonPathContext`(*dpath*: *str* | *PathLike*, *index*: *int* = 0)

Bases: `object`

Context for temporarily adding a dir to the PYTHONPATH.

Used in testing, and used as a helper in certain ubelt functions.

Warning

Even though this context manager takes precautions, this modifies `sys.path`, and things can go wrong when that happens. This is generally safe as long as nothing else you do inside of this context modifies the path. If the path is modified in this context, we will try to detect it and warn.

Variables

- **dpath** (*str* | *PathLike*) – directory to insert into the PYTHONPATH
- **index** (*int*) – position to add to. Typically either -1 or 0.

Example

```
>>> import sys
>>> with PythonPathContext('foo', -1):
>>>     assert sys.path[-1] == 'foo'
>>> assert sys.path[-1] != 'foo'
>>> with PythonPathContext('bar', 0):
>>>     assert sys.path[0] == 'bar'
>>> assert sys.path[0] != 'bar'
```

Example

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> # Mangle the path inside the context
>>> import sys
>>> self = PythonPathContext('foo', 0)
>>> self.__enter__()
>>> sys.path.insert(0, 'mangled')
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     self.__exit__(None, None, None)
```

Example

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> import sys
>>> self = PythonPathContext('foo', 0)
>>> self.__enter__()
>>> sys.path.remove('foo')
>>> import pytest
```

(continues on next page)

(continued from previous page)

```
>>> with pytest.raises(RuntimeError):
>>>     self.__exit__(None, None, None)
```

Parameters

- **dpath** (*str* | *PathLike*) – directory to insert into the PYTHONPATH
- **index** (*int*) – position to add to. Typically either -1 or 0.

dpath: `str`**index:** `int`

```
class xdoctest.utils.TeeStringIO(redirect: IOBase | None = None)
```

Bases: `StringIO`

An IO object that writes to itself and another IO stream.

Variables**redirect** (*io.IOBase*) – The other stream to write to.**Example**

```
>>> redirect = io.StringIO()
>>> self = TeeStringIO(redirect)
```

isatty() → `bool`

Returns true if the redirect is a terminal.

Note

Needed for IPython.embed to work properly when this class is used to override stdout / stderr.

fileno() → `int`

Returns underlying file descriptor of the redirected IOBase object if one exists.

property encoding: `str` | `None`Gets the encoding of the *redirect* IO object**Example**

```
>>> redirect = io.StringIO()
>>> assert TeeStringIO(redirect).encoding is None
>>> assert TeeStringIO(None).encoding is None
>>> assert TeeStringIO(sys.stdout).encoding is sys.stdout.encoding
>>> redirect = io.TextIOWrapper(io.StringIO())
>>> assert TeeStringIO(redirect).encoding is redirect.encoding
```

write(msg: str) → `int`

Write to this and the redirected stream

flush() → `None`

Flush to this and the redirected stream

class `xdoctest.utils.TempDir`(*persist: bool = False*)

Bases: `object`

Context for creating and cleaning up temporary files. Used in testing.

Example

```
>>> with TempDir() as self:
>>>     dpath = self.dpath
>>>     assert exists(dpath)
>>> assert not exists(dpath)
```

Example

```
>>> self = TempDir()
>>> dpath = self.ensure()
>>> assert exists(dpath)
>>> self.cleanup()
>>> assert not exists(dpath)
```

`ensure()` → `str`

`cleanup()` → `None`

class `xdoctest.utils.TempDoctest`(*docstr: str, modname: str | None = None*)

Bases: `object`

Creates a temporary file containing a module-level doctest for testing

Example

```
>>> from xdoctest import core
>>> self = TempDoctest('>>> a = 1')
>>> doctests = list(core.parse_doctestables(self.modpath))
>>> assert len(doctests) == 1
```

`xdoctest.utils.add_line_numbers`(*source: list[str], start: int = 1, n_digits: int | None = None*) → `list[str]`

`xdoctest.utils.add_line_numbers`(*source: str, start: int = 1, n_digits: int | None = None*) → `str`

Prefixes code with line numbers

Parameters

- **source** (*str | List[str]*)
- **start** (*int*)
- **n_digits** (*int | None*)

Returns

`List[str] | str`

Example

```
>>> print(chr(10).join(add_line_numbers(['a', 'b', 'c'])))
1 a
2 b
```

(continues on next page)

(continued from previous page)

```

3 c
>>> print(add_line_numbers(chr(10).join(['a', 'b', 'c'])))
1 a
2 b
3 c

```

`xdoctest.utils.codeblock(block_str: str) → str`

Wraps multiline string blocks and returns unindented code. Useful for templated code defined in indented parts of code.

Parameters

block_str (*str*) – typically in the form of a multiline string

Returns

the unindented string

Return type

`str`

Example

```

>>> # Simulate an indented part of code
>>> if True:
...     # notice the indentation on this will be normal
...     codeblock_version = codeblock(
...         """
...         def foo():
...             return 'bar'
...         """
...     )
...     # notice the indentation and newlines on this will be odd
...     normal_version = ("""
...         def foo():
...             return 'bar'
...     """)
>>> assert normal_version != codeblock_version
>>> print('Without codeblock')
>>> print(normal_version)
>>> print('With codeblock')
>>> print(codeblock_version)

```

`xdoctest.utils.color_text(text: str, color: str | None) → str`

Colorizes text a single color using ansii tags.

Parameters

- **text** (*str*) – text to colorize
- **color** (*str*) – may be one of the following: yellow, blink, lightgray, underline, darkyellow, blue, darkblue, faint, fuchsia, black, white, red, brown, turquoise, bold, darkred, darkgreen, reset, standout, darkteal, darkgray, overline, purple, green, teal, fuscia

Returns

colorized text.

If pygments is not installed plain text is returned.

Return type

str

Example

```

>>> import sys
>>> if sys.platform.startswith('win32'):
>>>     import pytest
>>>     pytest.skip()
>>> text = 'raw text'
>>> from xdoctest import utils
>>> from xdoctest.utils import util_str
>>> if utils.modname_to_modpath('pygments') and not util_str.NO_COLOR:
>>>     # Colors text only if pygments is installed
>>>     import pygments
>>>     print('pygments = {!r}'.format(pygments))
>>>     ansi_text1 = color_text(text, 'red')
>>>     print('ansi_text1 = {!r}'.format(ansi_text1))
>>>     ansi_text = utils.ensure_unicode(ansi_text1)
>>>     prefix = utils.ensure_unicode('\x1b[31')
>>>     print('prefix = {!r}'.format(prefix))
>>>     print('ansi_text = {!r}'.format(ansi_text))
>>>     assert ansi_text.startswith(prefix)
>>>     assert color_text(text, None) == 'raw text'
>>> else:
>>>     # Otherwise text passes through unchanged
>>>     assert color_text(text, 'red') == 'raw text'
>>>     assert color_text(text, None) == 'raw text'

```

xdoctest.utils.ensure_unicode(text: str) → str

Casts bytes into utf8 (mostly for python2 compatibility)

Parameters

text (str)

Returns

str

References

<http://stackoverflow.com/questions/12561063/python-extract-data-from-file>

CommandLine

```
python -m xdoctest.utils ensure_unicode
```

Example

```

>>> assert ensure_unicode('my unicôdé string') == 'my unicôdé string'
>>> assert ensure_unicode('text1') == 'text1'
>>> assert ensure_unicode('text1'.encode('utf8')) == 'text1'
>>> assert ensure_unicode('i»¿text1'.encode('utf8')) == 'i»¿text1'
>>> import codecs
>>> assert (codecs.BOM_UTF8 + 'text»¿'.encode('utf8')).decode('utf8')

```

`xdoctest.utils.ensuredir(dpath: str | tuple[str, ...] | list[str], mode: int = 1023) → str`

Ensures that directory will exist. creates new dir with sticky bits by default

Parameters

- **dpath** (*str*) – dir to ensure. Can also be a tuple to send to join
- **mode** (*int*) – octal mode of directory (default 0o1777)

Returns

path - the ensured directory

Return type

str

`xdoctest.utils.highlight_code(text: str, lexer_name: str = 'python', **kwargs: object) → str`

Highlights a block of text using ansi tags based on language syntax.

Parameters

- **text** (*str*) – plain text to highlight
- **lexer_name** (*str*) – name of language
- ****kwargs** – passed to `pygments.lexers.get_lexer_by_name`

Returns

text

[highlighted text] If pygments is not installed, the plain text is returned.

Return type

str

CommandLine

```
python -c "import pygments.formatters; print(list(pygments.formatters.get_all_
↪formatters()))"
```

Example

```
>>> text = 'import xdoctest as xdoc; print(xdoc)'
>>> new_text = highlight_code(text)
>>> print(new_text)
```

`xdoctest.utils.import_module_from_name(modname: str) → ModuleType`

Imports a module from its string name (i.e. `__name__`)

This is a simple wrapper around `importlib.import_module()`, but is provided as a companion function to `import_module_from_path()`, which contains functionality not provided in the Python standard library.

Parameters

modname (*str*) – module name

Returns

module

Return type

ModuleType

SeeAlso:

`import_module_from_path()`

Example

```

>>> # test with modules that won't be imported in normal circumstances
>>> # todo write a test where we guarantee this
>>> # xdoctest: +SKIP("ubelt dependency")
>>> import sys
>>> modname_list = [
>>>     'pickletools',
>>>     'email.mime.text',
>>> ]
>>> #assert not any(m in sys.modules for m in modname_list)
>>> modules = [ub.import_module_from_name(modname) for modname in modname_list]
>>> assert [m.__name__ for m in modules] == modname_list
>>> assert all(m in sys.modules for m in modname_list)

```

`xdoctest.utils.import_module_from_path(modpath: str | PathLike, index: int = -1) → ModuleType`

Imports a module via a filesystem path.

This works by modifying `sys.path`, importing the module name, and then attempting to undo the change to `sys.path`. This function may produce unexpected results in the case where the imported module itself itself modifies `sys.path` or if there is another conflicting module with the same name.

Parameters

- **modpath** (*str* | *PathLike*) – Path to the module on disk or within a zipfile. Paths within a zipfile can be given by `<path-to>.zip/<path-inside-zip>.py`.
- **index** (*int*) – Location at which we modify PYTHONPATH if necessary. If your module name does not conflict, the safest value is -1, However, if there is a conflict, then use an index of 0. The default may change to 0 in the future.

Returns

the imported module

Return type

ModuleType

References

Raises

- **IOError** – when the path to the module does not exist –
- **ImportError** – when the module is unable to be imported –

Note

If the module is part of a package, the package will be imported first. These modules may cause problems when reloading via IPython magic

This can import a module from within a zipfile. To do this `modpath` should specify the path to the zipfile and the path to the module within that zipfile separated by a colon or pathsep. E.g. `"/path/to/archive.zip:mymodule.pl"`

Warning

It is best to use this with paths that will not conflict with previously existing modules.

If the modpath conflicts with a previously existing module name. And the target module does imports of its own relative to this conflicting path. In this case, the module that was loaded first will win.

For example if you try to import `'/foo/bar/pkg/mod.py'` from the folder structure:

```
- foo/
+- bar/
  +- pkg/
    + __init__.py
    |- mod.py
    |- helper.py
```

If there exists another module named `pkg` already in `sys.modules` and `mod.py` contains the code `from . import helper`, Python will assume `helper` belongs to the `pkg` module already in `sys.modules`. This can cause a `NameError` or worse — an incorrect helper module.

SeeAlso:

`import_module_from_name()`

Example

```
>>> # xdoctest: +SKIP("ubelt dependency")
>>> import xdoctest
>>> modpath = xdoctest.__file__
>>> module = ub.import_module_from_path(modpath)
>>> assert module is xdoctest
```

Example

```
>>> # Test importing a module from within a zipfile
>>> # xdoctest: +SKIP("ubelt dependency")
>>> import zipfile
>>> from xdoctest import utils
>>> import os
>>> from os.path import join, expanduser, normpath
>>> dpath = expanduser('~/.cache/xdoctest')
>>> dpath = utils.ensuredir(dpath)
>>> #dpath = utils.TempDir().ensure()
>>> # Write to an external module named bar
>>> external_modpath = join(dpath, 'bar.py')
>>> # For pypy support we have to write this using with
>>> with open(external_modpath, 'w') as file:
>>>     file.write('testvar = 1')
>>> internal = 'folder/bar.py'
>>> # Move the external bar module into a zipfile
>>> zippath = join(dpath, 'myzip.zip')
>>> with zipfile.ZipFile(zippath, 'w') as myzip:
>>>     myzip.write(external_modpath, internal)
>>> # Import the bar module from within the zipfile
```

(continues on next page)

(continued from previous page)

```
>>> modpath = zippath + ':' + internal
>>> modpath = zippath + os.path.sep + internal
>>> module = ub.import_module_from_path(modpath)
>>> assert normpath(module.__name__) == normpath('folder/bar')
>>> assert module.testvar == 1
```

Example

```
>>> import pytest
>>> # xdoctest: +SKIP("ubelt dependency")
>>> with pytest.raises(IOError):
>>>     ub.import_module_from_path('does-not-exist')
>>> with pytest.raises(IOError):
>>>     ub.import_module_from_path('does-not-exist.zip/')
```

`xdoctest.utils.indent(text: str, prefix: str = ' ') → str`

Indents a block of text

Parameters

- **text** (*str*) – text to indent
- **prefix** (*str*) – prefix to add to each line (default = ' ')

Returns

indented text

Return type

`str`

CommandLine

```
python -m xdoctest.utils ensure_unicode
```

Example

```
>>> text = 'Lorem ipsum\ndolor sit amet'
>>> prefix = '    '
>>> result = indent(text, prefix)
>>> assert all(t.startswith(prefix) for t in result.split('\n'))
```

`xdoctest.utils.is_modname_importable(modname: str, sys_path: list | None = None, exclude: list | None = None) → bool`

Determines if a modname is importable based on your current `sys.path`

Parameters

- **modname** (*str*) – name of module to check
- **sys_path** (*list | None*) – if specified overrides `sys.path`
- **exclude** (*list | None*) – list of directory paths. if specified prevents these directories from being searched.

Returns

True if the module can be imported

Return type

bool

Example

```
>>> is_modname_importable('xdoctest')
True
>>> is_modname_importable('not_a_real_module')
False
>>> is_modname_importable('xdoctest', sys_path=[])
False
```

`xdoctest.utils.modname_to_modpath(modname: str, hide_init: bool = True, hide_main: bool = False, sys_path: list[str | PathLike] | None = None) → str | None`

Finds the path to a python module from its name.

Determines the path to a python module without directly import it

Converts the name of a module (`__name__`) to the path (`__file__`) where it is located without importing the module. Returns None if the module does not exist.

Parameters

- **modname** (*str*) – The name of a module in `sys_path`.
- **hide_init** (*bool*) – if False, `__init__.py` will be returned for packages. Defaults to True.
- **hide_main** (*bool*) – if False, and `hide_init` is True, `__main__.py` will be returned for packages, if it exists. Defaults to False.
- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.

Returns

modpath - path to the module, or None if it doesn't exist

Return type

str | None

Example

```
>>> modname = 'xdoctest.__main__'
>>> modpath = modname_to_modpath(modname, hide_main=False)
>>> assert modpath.endswith('__main__.py')
>>> modname = 'xdoctest'
>>> modpath = modname_to_modpath(modname, hide_init=False)
>>> assert modpath.endswith('__init__.py')
>>> modpath = modname_to_modpath('json')
>>> assert 'json' in modpath
```

`xdoctest.utils.modpath_to_modname(modpath: str | PathLike, hide_init: bool = True, hide_main: bool = False, check: bool = True, relativeto: str | None = None) → str`

Determines importable name from file path

Converts the path to a module (`__file__`) to the importable python name (`__name__`) without importing the module.

The filename is converted to a module name, and parent directories are recursively included until a directory without an `__init__.py` file is encountered.

Parameters

- **modpath** (*str* | *os.PathLike*) – Module filepath
- **hide_init** (*bool*) – Removes the `__init__` suffix. Defaults to `True`.
- **hide_main** (*bool*) – Removes the `__main__` suffix. Defaults to `False`.
- **check** (*bool*) – If `False`, does not raise an error if `modpath` is a dir and does not contain an `__init__` file. Defaults to `True`.
- **relativeto** (*str* | *None*) – If specified, all checks are ignored and this is considered the path to the root module. Defaults to `None`.

Todo

- **Does this need modification to support PEP 420?**
<https://www.python.org/dev/peps/pep-0420/>

Returns

modname

Return type

`str`

Raises

ValueError – if `check` is `True` and the path does not exist

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = modpath.replace('.pyc', '.py')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'xdoctest.static_analysis'
```

Example

```
>>> import xdoctest
>>> assert modpath_to_modname(xdoctest.__file__.replace('.pyc', '.py')) == 'xdoctest'
↪ '
>>> assert modpath_to_modname(dirname(xdoctest.__file__.replace('.pyc', '.py'))) ==
↪ 'xdoctest'
```

Example

```
>>> modpath = modname_to_modpath('json')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'json'
```

Example

```
>>> modpath = '/foo/libfoobar.linux-x86_64-3.6.so'
>>> modname = modpath_to_modname(modpath, check=False)
>>> assert modname == 'libfoobar'
```

`xdoctest.utils.normalize_modpath(modpath: str, hide_init: bool = True, hide_main: bool = False) → str`
`xdoctest.utils.normalize_modpath(modpath: PathLike, hide_init: bool = True, hide_main: bool = False) → str | PathLike`

Normalizes `__init__` and `__main__` paths.

Parameters

- **modpath** (*str* | *PathLike*) – path to a module
- **hide_init** (*bool*) – if True, always return package modules as `__init__.py` files otherwise always return the `dpath`. Defaults to True.
- **hide_main** (*bool*) – if True, always strip away main files otherwise ignore `__main__.py`. Defaults to False.

Returns

a normalized path to the module

Return type

str | *PathLike*

Note

Adds `__init__` if reasonable, but only removes `__main__` by default

Example

```
>>> from xdoctest import static_analysis as module
>>> modpath = module.__file__
>>> assert normalize_modpath(modpath) == modpath.replace('.pyc', '.py')
>>> dpath = dirname(modpath)
>>> res0 = normalize_modpath(dpath, hide_init=0, hide_main=0)
>>> res1 = normalize_modpath(dpath, hide_init=0, hide_main=1)
>>> res2 = normalize_modpath(dpath, hide_init=1, hide_main=0)
>>> res3 = normalize_modpath(dpath, hide_init=1, hide_main=1)
>>> assert res0.endswith('__init__.py')
>>> assert res1.endswith('__init__.py')
>>> assert not res2.endswith('.py')
>>> assert not res3.endswith('.py')
```

`xdoctest.utils.split_modpath(modpath: str | PathLike, check: bool = True) → tuple[str, str]`

Splits the `modpath` into the dir that must be in `PYTHONPATH` for the module to be imported and the `modulepath` relative to this directory.

Parameters

- **modpath** (*str*) – module filepath
- **check** (*bool*) – if False, does not raise an error if `modpath` is a directory and does not contain an `__init__.py` file.

Returns

(directory, rel_modpath)

Return type

Tuple[str, str]

Raises**ValueError** – if modpath does not exist or is not a package**Example**

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = abspath(modpath)
>>> dpath, rel_modpath = split_modpath(modpath)
>>> recon = join(dpath, rel_modpath)
>>> assert recon == modpath
>>> assert rel_modpath == join('xdoctest', 'static_analysis.py')
```

xdoctest.utils.strip_ansi(text: str) → str

Removes all ansi directives from the string.

Parameters**text** (str)**Returns**

str

References

<http://stackoverflow.com/questions/14693701/remove-ansi-filtering-out-ansi-escape-sequences> <https://stackoverflow.com/questions/13506033/>

Examples

```
>>> line = '\t\u001b[0;35mBlabla\u001b[0m      \u001b[0;36m172.18.0.2\u001b[0m'
>>> escaped_line = strip_ansi(line)
>>> assert escaped_line == '\tBlabla      172.18.0.2'
```

4.2 Submodules

4.2.1 xdoctest.__main__ module

Provides a simple script for running module doctests.

This should work even if the target module is unaware of xdoctest.

xdoctest.__main__.main(argv: list[str] | None = None) → int

Parameters**argv** (List[str] | None)

4.2.2 xdoctest._tokenize module

Tokenization help for Python programs.

`tokenize(readline)` is a generator that breaks a stream of bytes into Python tokens. It decodes the bytes according to PEP-0263 for determining source file encoding.

It accepts a `readline`-like method which is called repeatedly to get the next line of input (or `b''` for EOF). It generates 5-tuples with these members:

the token type (see `token.py`) the token (a string) the starting (row, column) indices of the token (a 2-tuple of ints) the ending (row, column) indices of the token (a 2-tuple of ints) the original line (string)

It is designed to match the working of the Python tokenizer exactly, except that it produces `COMMENT` tokens for comments and gives type `OP` for all operators. Additionally, all token lists start with an `ENCODING` token which tells you which encoding was used to decode the bytes stream.

`xdoctest._tokenize.ISTERMINAL(x)`

`xdoctest._tokenize.ISNONTERMINAL(x)`

`xdoctest._tokenize.ISEOF(x)`

`xdoctest._tokenize.tokenize(readline)`

The `tokenize()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `readline()` method of built-in file objects. Each call to the function should return one line of input as bytes. Alternatively, `readline` can be a callable function terminating with `StopIteration`:

```
readline = open(myfile, 'rb').__next__ # Example of alternate readline
```

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (`srow`, `scol`) of ints specifying the row and column where the token begins in the source; a 2-tuple (`erow`, `ecol`) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed is the physical line.

The first token sequence will always be an `ENCODING` token which tells you which encoding was used to decode the bytes stream.

`xdoctest._tokenize.generate_tokens(readline)`

Tokenize a source reading Python code as unicode strings.

This has the same API as `tokenize()`, except that it expects the `readline` callable to return str objects instead of bytes.

`xdoctest._tokenize.detect_encoding(readline)`

The `detect_encoding()` function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, `readline`, in the same way as the `tokenize()` generator.

It will call `readline` a maximum of twice, and return the encoding used (as a string) and a list of any lines (left as bytes) it has read in.

It detects the encoding from the presence of a utf-8 bom or an encoding cookie as specified in pep-0263. If both a bom and a cookie are present, but disagree, a `SyntaxError` will be raised. If the encoding cookie is an invalid charset, raise a `SyntaxError`. Note that if a utf-8 bom is found, 'utf-8-sig' is returned.

If no encoding is specified, then the default of 'utf-8' will be returned.

`xdoctest._tokenize.untokenize(iterable)`

Transform tokens back into Python source code. It returns a bytes object, encoded using the `ENCODING` token, which is the first token sequence output by `tokenize`.

Each element returned by the iterable must be a token sequence with at least two elements, a token number and token value. If only two tokens are passed, the resulting output is poor.

Round-trip invariant for full input:

Untokenized source will match input source exactly

Round-trip invariant for limited input:

```
# Output bytes will tokenize back to the input t1 = [tok[:2] for tok in tokenize(f.readline)] newcode =
untokenize(t1) readline = BytesIO(newcode).readline t2 = [tok[:2] for tok in tokenize(readline)] assert t1
== t2
```

```
class xdoctest._tokenize.TokenInfo(type, string, start, end, line)
```

Bases: *TokenInfo*

Create new instance of TokenInfo(*type, string, start, end, line*)

property exact_type

4.2.3 xdoctest.checker module

Checks for got-vs-want statements

A “got-string” is data produced by a doctest that we want to check matches some expected value.

A “want-string” is a representation of the output we expect, if the “got-string” is different than the “want-string” the doctest will fail with a *GotWantException*. A want string should come directly after a doctest and should not be prefixed by the three chevrons (`^>>>``).

There are two types of data that a doctest could “get” as a “got-string”, either the contents of standard out the value of an expression itself.

A doctest that uses stdout might look like this

```
>>> print('We expect this exact string')
We expect this exact string
```

A doctest that uses a raw expression might look like this

```
>>> def foo():
>>>     return 3
>>> foo()
3
```

In most cases it is best to use stdout to write your got-want tests because it is easier to control strings sent to stdout than it is to control the representation of expression-based “got-strings”.

```
xdoctest.checker.check_got_vs_want(want: str, got_stdout: str, got_eval: Any = <NOT_EVALUED>,
runstate: RuntimeState | None = None) → bool
```

Determines to check against either `got_stdout` or `got_eval`, and then does the comparison.

If both stdout and eval “got” outputs are specified, then the “want” target may match either value.

Parameters

- **want** (*str*) – target to match against
- **got_stdout** (*str*) – output from stdout
- **got_eval** (*str*) – output from an eval statement.
- **runstate** (*xdoctest.directive.RuntimeState | None*) – current state

Raises

GotWantException – If the “got” differs from this parts want. –

`xdoctest.checker._strip_exception_details(msg: str) → str`

Parameters

`msg (str)`

Return type

`str`

`xdoctest.checker.extract_exc_want(want: str) → str | None`

Parameters

`want (str)` – the message supplied by the user

Returns

the matchable exception part

Return type

`str`

Example

`extract_exc_want(“ Traceback (most recent call last): bar “)`

`xdoctest.checker.check_exception(exc_got: str, want: str, runstate: RuntimeState | None = None) → bool`

Checks want against an exception

Parameters

- `exc_got (str)` – the exception message
- `want (str)` – target to match against
- `runstate (xdoctest.directive.RuntimeState | None)` – current state

Raises

`GotWantException` - If the "got" differs from this parts want. -

Returns

True if got matches want

Return type

`bool`

`xdoctest.checker.check_output(got: str, want: str, runstate: RuntimeState | None = None) → bool`

Does the actual comparison between `got` and `want` as long as the check is enabled.

Parameters

- `got (str)` – text produced by the test
- `want (str)` – target to match against
- `runstate (xdoctest.directive.RuntimeState | None)` – current state

Returns

True if got matches want or if the check is disabled

Return type

`bool`

`xdoctest.checker._check_match(got: str, want: str, runstate: RuntimeState | dict) → bool`

Does the actual comparison between `got` and `want`

Parameters

- **got** (*str*) – normalized text produced by the test
- **want** (*str*) – normalized target to match against
- **runstate** (*xdoctest.directive.RuntimeState*) – current state

Returns

True if got matches want

Return type

bool

`xdoctest.checker._ellipsis_match(got: Any, want: Any) → bool`

The ellipsis matching algorithm taken directly from standard doctest.

Worst-case linear-time ellipsis matching.

Parameters

- **got** (*str*)
- **want** (*str*)

Returns

True if the text matches according to the ellipsis rule

Return type

bool

CommandLine

```
python -m xdoctest.checker _ellipsis_match
```

Example

```
>>> _ellipsis_match('aaa', 'aa...aa')
False
>>> _ellipsis_match('anything', '...')
True
>>> _ellipsis_match('prefix-anything', 'prefix-...')
True
>>> _ellipsis_match('suffix-anything', 'prefix-...')
False
>>> _ellipsis_match('foo', '... foo')
True
>>> _ellipsis_match('took=3.4s', 'took=...s')
True
>>> _ellipsis_match('best=3.4s ave=4.5s', 'best=...s ave=...s')
True
>>> _ellipsis_match('took: 1.16e-05 s\nbest=9.63e-07 s ave=1.002e-06 ± 3e-08 s\n',
>>>                      'took: ...s\nbest=...s ave=...s\n')
True
```

`xdoctest.checker.normalize(got: str, want: str, runstate: RuntimeState | Dict[str, bool] | Set[str] | OrderedDict[str, bool] | Set[str] | None = None) → tuple[str, str]`

Normalizes the got and want string based on the runtime state.

Adapted from doctest_nose_plugin.py from the nltk project:

<https://github.com/nltk/nltk>

Further extended to also support byte literals.

Parameters

- **got** (*str*) – unnormalized got str.
- **want** (*str*) – unnormalized want str.
- **runstate** (*xdoctest.directive.RuntimeState* | *None*) – current state

Returns

The normalized got and want str

Return type

Tuple[*str*, *str*]

Example

```
>>> from xdoctest.checker import * # NOQA
>>> want = "...\\n(0, 2, {'weight': 1})\\n(0, 3, {'weight': 2})"
>>> got = "(0, 2, {'weight': 1})\\n(0, 3, {'weight': 2})"
>>> normalize(got, want)
("(0, 2, {'weight': 1}) (0, 3, {'weight': 2})",
 "... (0, 2, {'weight': 1}) (0, 3, {'weight': 2})")
```

exception `xdoctest.checker.ExtractGotReprException(msg: str, orig_ex: Exception)`

Bases: `AssertionError`

Parameters

- **msg** (*str*) – The exception message
- **orig_ex** (*Exception*) – The parent exception

orig_ex: `Exception`

Exception used when we are unable to extract a string “got”

exception `xdoctest.checker.GotWantException(msg: str, got: str, want: str)`

Bases: `AssertionError`

Parameters

- **msg** (*str*) – The exception message
- **got** (*str*) – The unnormalized got str
- **want** (*str*) – The unnormalized want str

got: `str`

want: `str`

Exception used when the “got” output of a doctest differs from the expected “want” output.

_do_a_fancy_diff(*runstate: RuntimeState* | *None = None*) → `bool`

output_difference(*runstate: RuntimeState* | *None = None*, *colored: bool = True*) → `str`

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). The *runstate* contains option flags used to compare *want* and *got*.

Parameters

- **runstate** (*xdoctest.directive.RuntimeState* | *None*) – current state

- **colored** (*bool*) – if the text should be colored

Returns

formatted difference text

Return type

str

Note

This does not check if got matches want, it only outputs the raw differences. Got/Want normalization may make the differences appear more exaggerated than they are.

output_repr_difference(*runstate*: `RuntimeState` | `None = None`) → str

Constructs a repr difference with minimal normalization.

Parameters

runstate (`xdoctest.directive.RuntimeState` | `None`) – current state

Returns

formatted repr difference text

Return type

str

`xdoctest.checker.remove_blankline_marker`(*text*: str) → str

Parameters

text (str) – input text

Returns

output text

Return type

str

Example

```
>>> text1 = 'foo\n{}\nbar'.format(BLANKLINE_MARKER)
>>> text2 = '{}\nbar'.format(BLANKLINE_MARKER)
>>> text4 = 'foo\n{}'.format(BLANKLINE_MARKER)
>>> text3 = '{}'.format(BLANKLINE_MARKER)
>>> text5 = text1 + text1 + text1
>>> assert BLANKLINE_MARKER not in remove_blankline_marker(text1)
>>> assert BLANKLINE_MARKER not in remove_blankline_marker(text2)
>>> assert BLANKLINE_MARKER not in remove_blankline_marker(text3)
>>> assert BLANKLINE_MARKER not in remove_blankline_marker(text4)
>>> assert BLANKLINE_MARKER not in remove_blankline_marker(text5)
```

4.2.4 xdoctest.constants module

Defines sentinel values for internal xdoctest usage

class `xdoctest.constants._NOT_EVAL_TYPE`

Bases: `object`

This is a singleton object used as a sentinel value. The value of `NoParam` is robust to reloading, pickling, and copying. See [SO_41048643] for more details.

References

Example

```
>>> from xdoctest.constants import NOT_EVALED, _NOT_EVAL_TYPE # NOQA
>>> import copy
>>> assert not NOT_EVALED
>>> assert str(NOT_EVALED) == '<NOT_EVALED>'
>>> assert repr(NOT_EVALED) == '<NOT_EVALED>'
>>> assert NOT_EVALED(...) is None
>>> assert copy.copy(NOT_EVALED) is NOT_EVALED
>>> assert copy.deepcopy(NOT_EVALED) is NOT_EVALED
>>> assert _NOT_EVAL_TYPE() is NOT_EVALED
```

`__call__` (default: *object*) → *None*

4.2.5 xdoctest.core module

Core methods used by xdoctest runner and plugin code to statically extract doctests from a module or package.

The following is a glossary of terms and jargon used in this repo.

- `callname` - the name of a callable function, method, class etc... e.g. `myfunc`, `MyClass`, or `MyClass.some_method`.
- `got / want` - a test that produces stdout or a value to check. Whatever is produced is what you “got” and whatever is expected is what you “want”. See [xdoctest.checker](#) for more details.
- `directives` - special in-doctest comments that change the behavior of the doctests at runtime. See [xdoctest.directive](#) for more details.
- `chevrons` - the three chevrons (```>>>```) or right angle brackets are the standard prefix for a doctest, also referred to as a PS1 line in the parser.
- `zero-args` - a function that can be called without any arguments.
- `freeform style` - This is the term used to refer to a doctest that could be anywhere in the docstring. The alternative are structured doctests where they are only expected in known positions like in “Example blocks” for google and numpy style docstrings.
- `TODO` - complete this list (Make an issue or PR if there is any term you don’t immediately understand!).

`xdoctest.core.parse_freeform_docstr_examples`(*docstr: str, callname: str | None = None, modpath: str | PathLike | None = None, lineno: int = 1, fpath: str | PathLike | None = None, asone: bool = True*) → *Iterator[DocTest]*

Finds free-form doctests in a docstring. This is similar to the original doctests because these tests do not requires a google/numpy style header.

Some care is taken to avoid enabling tests that look like disabled google doctests or scripts.

Parameters

- `docstr` (*str*) – an extracted docstring
- `callname` (*str | None*) – the name of the callable (e.g. function, class, or method) that this docstring belongs to.
- `modpath` (*str | PathLike | None*) – original module the docstring is from

- **lineno** (*int*) – the line number (starting from 1) of the docstring. i.e. if you were to go to this line number in the source file the starting quotes of the docstr would be on this line. Defaults to 1.
- **fpath** (*str | PathLike | None*) – the file that the docstring is from (if the file was not a module, needed for backwards compatibility)
- **asone** (*bool*) – if False doctests are broken into multiple examples based on spacing, otherwise they are executed as a single unit. Defaults to True.

Yields

`xdoctest.doctest_example.DocTest` – doctest object

Raises

`xdoctest.exceptions.DocTestParseError` – if an error occurs in parsing

CommandLine

```
python -m xdoctest.core parse_freeform_docstr_examples
```

Example

```
>>> # TODO: move this to unit tests and make the doctest simpler
>>> from xdoctest import core
>>> from xdoctest import utils
>>> docstr = utils.codeblock(
>>>     """
>>>     freeform
>>>     >>> doctest
>>>     >>> hasmultilines
>>>     whoppie
>>>     >>> 'but this is the same doctest'
>>>
>>>     >>> secondone
>>>
>>>     Script:
>>>         >>> 'special case, dont parse me'
>>>
>>>     DisableDoctest:
>>>         >>> 'special case, dont parse me'
>>>         want
>>>
>>>     AnythingElse:
>>>         >>> 'general case, parse me'
>>>         want
>>>     """)
>>> examples = list(parse_freeform_docstr_examples(docstr, asone=True))
>>> assert len(examples) == 1
>>> examples = list(parse_freeform_docstr_examples(docstr, asone=False))
>>> assert len(examples) == 3
```

```
xdoctest.core.parse_google_docstr_examples(docstr: str, callname: str | None = None, modpath: str |
PathLike | None = None, lineno: int = 1, fpath: str |
PathLike | None = None, eager_parse: bool = True) →
Iterator[DocTest]
```

Parses Google-style doctests from a docstr and generates example objects

Parameters

- **docstr** (*str*) – an extracted docstring
- **callname** (*str* | *None*) – the name of the callable (e.g. function, class, or method) that this docstring belongs to.
- **modpath** (*str* | *PathLike* | *None*) – original module the docstring is from
- **lineno** (*int*) – the line number (starting from 1) of the docstring. i.e. if you were to go to this line number in the source file the starting quotes of the docstr would be on this line. Defaults to 1.
- **fpath** (*str* | *PathLike* | *None*) – the file that the docstring is from (if the file was not a module, needed for backwards compatibility)
- **eager_parse** (*bool*) – if True eagerly evaluate the parser inside the google example blocks. Defaults to True.

Yields

xdoctest.doctest_example.DocTest – doctest object

Raises

- *xdoctest.exceptions.MalformedDocstr* – if an error occurs in finding google blocks
- *xdoctest.exceptions.DoctestParseError* – if an error occurs in parsing

`xdoctest.core.parse_auto_docstr_examples(docstr: str, *args: Any, **kwargs: Any) → Iterator[DocTest]`

First try to parse google style, but if no tests are found use freeform style.

`xdoctest.core.parse_docstr_examples(docstr: str, callname: str | None = None, modpath: str | PathLike | None = None, lineno: int = 1, style: str = 'auto', fpath: str | PathLike | None = None, parser_kw: dict | None = None) → Iterator[DocTest]`

Parses doctests from a docstr and generates example objects. The style influences which tests are found.

Parameters

- **docstr** (*str*) – a previously extracted docstring
- **callname** (*str* | *None*) – the name of the callable (e.g. function, class, or method) that this docstring belongs to.
- **modpath** (*str* | *PathLike* | *None*) – original module the docstring is from
- **lineno** (*int*) – the line number (starting from 1) of the docstring. i.e. if you were to go to this line number in the source file the starting quotes of the docstr would be on this line. Defaults to 1.
- **style** (*str*) – expected doctest style, which can be “google”, “freeform”, or “auto”. Defaults to ‘auto’.
- **fpath** (*str* | *PathLike* | *None*) – the file that the docstring is from (if the file was not a module, needed for backwards compatibility)
- **parser_kw** (*dict* | *None*) – passed to the parser as keyword args

Yields

xdoctest.doctest_example.DocTest – parsed example

CommandLine

```
python -m xdoctest.core parse_docstr_examples
```

Example

```
>>> from xdoctest.core import *
>>> from xdoctest import utils
>>> docstr = utils.codeblock(
...     """
...     >>> 1 + 1 # xdoctest: +SKIP
...     2
...     >>> 2 + 2
...     4
...     """)
>>> examples = list(parse_docstr_examples(docstr, 'name', fpath='foo.txt', style=
→ 'freeform'))
>>> print(len(examples))
1
>>> examples = list(parse_docstr_examples(docstr, fpath='foo.txt'))
```

`xdoctest.core._rectify_to_modpath(modpath_or_name)`

if `modpath_or_name` is a name, statically converts it to a path

`xdoctest.core.package_calldefs(pkg_identifier: str | PathLike | ModuleType, exclude: list[str] = [], ignore_syntax_errors: bool = True, analysis: str = 'auto') → Iterator[tuple[dict[str, CallDefNode], Any]]`

Statically generates all callable definitions in a module or package

Parameters

- **pkg_identifier** (*str* | *PathLike* | *ModuleType*) – path to or name of the module to be tested (or the live module itself, which is not recommended)
- **exclude** (*List[str]*) – glob-patterns of file names to exclude
- **ignore_syntax_errors** (*bool*) – if False raise an error when syntax errors occur in a doctest Defaults to True.
- **analysis** (*str*) – if 'static', only static analysis is used to parse call definitions. If 'auto', uses dynamic analysis for compiled python extensions, but static analysis elsewhere, if 'dynamic', then dynamic analysis is used to parse all calldefs. Defaults to 'auto'.

Yields

Tuple[Dict[str, xdoctest.static_analysis.CallDefNode], str | ModuleType] -

- `item[0]`: the mapping of callnames-to-calldefs
- `item[1]`: the path to the file containing the doctest (usually a module) or the module itself

Example

```
>>> pkg_identifier = 'xdoctest.core'
>>> testables = list(package_calldefs(pkg_identifier))
>>> assert len(testables) == 1
>>> calldefs, modpath = testables[0]
```

(continues on next page)

(continued from previous page)

```
>>> assert util_import.modpath_to_modname(modpath) == pkg_identifier
>>> assert 'package_calldefs' in calldefs
```

`xdoctest.core.parse_calldefs`(*module_identifier*: *str* | *ModuleType*, *analysis*: *str* = 'auto') → *dict*[*str*, *CallDefNode*] | *None*

Parse calldefs from a single module using either static or dynamic analysis.

Parameters

- **module_identifier** (*str* | *ModuleType*) – path to or name of the module to be tested (or the live module itself, which is not recommended)
- **analysis** (*str*; *default*='auto') – if 'static', only static analysis is used to parse call definitions. If 'auto', uses dynamic analysis for compiled python extensions, but static analysis elsewhere, if 'dynamic', then dynamic analysis is used to parse all calldefs.

Returns

the mapping of callnames-to-calldefs within the module.

Return type

Dict[*str*, *xdoctest.static_analysis.CallDefNode*]

`xdoctest.core.parse_doctestables`(*module_identifier*: *str* | *PathLike* | *ModuleType*, *exclude*: *list*[*str*] = [], *style*: *str* = 'auto', *ignore_syntax_errors*: *bool* = *True*, *parser_kw*: *dict* = {}, *analysis*: *str* = 'auto') → *Iterator*[*DocTest*]

Parses all doctests within top-level callables of a module and generates example objects. The style influences which tests are found.

Parameters

- **module_identifier** (*str* | *PathLike* | *ModuleType*) – path or name of a module or a module itself (we prefer a path)
- **exclude** (*List*[*str*]) – glob-patterns of file names to exclude
- **style** (*str*) – expected doctest style (e.g. google, freeform, auto)
- **ignore_syntax_errors** (*bool*, *default*=*True*) – if *False* raise an error when syntax errors
- **parser_kw** – extra args passed to the parser
- **analysis** (*str*, *default*='auto') – if 'static', only static analysis is used to parse call definitions. If 'auto', uses dynamic analysis for compiled python extensions, but static analysis elsewhere, if 'dynamic', then dynamic analysis is used to parse all calldefs.

Yields

xdoctest.doctest_example.DocTest – parsed doctest example objects

CommandLine

```
python -m xdoctest.core parse_doctestables
```

Example

```
>>> module_identifier = 'xdoctest.core'
>>> testables = list(parse_doctestables(module_identifier))
>>> this_example = None
>>> for example in testables:
```

(continues on next page)

(continued from previous page)

```

>>> # print(example)
>>> if example.callname == 'parse_doctestables':
>>>     this_example = example
>>> assert this_example is not None
>>> assert this_example.callname == 'parse_doctestables'

```

Example

```

>>> from xdoctest import utils
>>> docstr = utils.codeblock(
...     """
...     >>> 1 + 1 # xdoctest: +SKIP
...     2
...     >>> 2 + 2
...     4
...     """)
>>> temp = utils.TempDoctest(docstr, 'test_modfile')
>>> modpath = temp.modpath
>>> examples = list(parse_doctestables(modpath, style='freeform'))
>>> print(len(examples))
1

```

4.2.6 xdoctest.demo module

This file contains quick demonstrations of how to use xdoctest

CommandLine

```

xdoctest -m xdoctest.demo

xdoctest -m xdoctest.demo --verbose 0
xdoctest -m xdoctest.demo --silent
xdoctest -m xdoctest.demo --quiet

```

`xdoctest.demo.myfunc()` → int

Demonstrates how to write a doctest. Prefix with >>> and ideally place in an *Example:* block. You can also change Example, Ignore will Prefix with >>> and ideally place in an *Example:* block.

CommandLine

```

# it would be nice if sphinx.ext.napoleon could handle this
xdoctest -m xdoctest.demo myfunc

```

Example

```

>>> result = myfunc()
>>> assert result == 123

```

```

class xdoctest.demo.MyClass(*args: object, **kw: object)
    Bases: object

```

Example

```
>>> self = MyClass.demo()
>>> print('self.data = {!r}'.format(self.data))
```

Example

```
>>> # xdoctest: +REQUIRES(--fail)
>>> raise Exception
```

classmethod `demo(**kw: object) → MyClass`

CommandLine

```
xdoctest -m xdoctest.demo MyClass.demo
xdoctest -m xdoctest.demo MyClass.demo --say
```

Example

```
>>> print('starting my doctest')
>>> self = MyClass.demo(demo='thats my demo')
>>> # xdoc: +REQUIRES(--say)
>>> print('self.data = {!r}'.format(self.data))
```

static `always_fails() → None`

CommandLine

```
xdoctest -m xdoctest.demo MyClass.always_fails
xdoctest -m xdoctest.demo MyClass.always_fails --fail
xdoctest -m xdoctest.demo MyClass.always_fails --fail --really

xdoctest -m xdoctest.demo MyClass.always_fails:0 --fail
xdoctest -m xdoctest.demo MyClass.always_fails:1 --fail
xdoctest -m xdoctest.demo MyClass.always_fails:2 --fail
xdoctest -m xdoctest.demo MyClass.always_fails:3 --fail --really
```

Example

```
>>> # xdoctest: +REQUIRES(--fail)
>>> raise Exception('doctest always fails')
```

Example

```
>>> # xdoctest: +REQUIRES(--fail)
>>> MyClass.demo().always_fails()
```

Example

```
>>> # xdoctest: +REQUIRES(--fail)
>>> print('there is no way to fail')
There are so many ways to fail
```

Example

```
>>> # xdoctest: +REQUIRES(--fail)
>>> # xdoctest: +REQUIRES(--really)
>>> raise Exception # xdoctest: +SKIP
>>> print('did you know') # xdoctest: +IGNORE_WANT
directives are useful
>>> print('match this')
...
>>> print('match this') # xdoctest: -ELLIPSIS
...
```

4.2.7 xdoctest.directive module

Directives special comments that influence the runtime behavior of doctests. There are two types of directives: block and inline

Block directives are specified on their own line and influence the behavior of multiple lines of code.

Inline directives are specified after in the same line of code and only influence that line / repl part.

4.2.7.1 Basic Directives

Basic directives correspond directly to an xdoctest runtime state attribute. These can be modified by directly using the xdoctest directive syntax. The following documents all supported basic directives.

The basic directives and their defaults are as follows:

- DONT_ACCEPT_BLANKLINE: False,
- ELLIPSIS: True,
- IGNORE_WHITESPACE: False,
- IGNORE_EXCEPTION_DETAIL: False,
- NORMALIZE_WHITESPACE: True,
- IGNORE_WANT: False,
- NORMALIZE_REPR: True,
- REPORT_CDIF: False,
- REPORT_NDIFF: False,
- REPORT_UDIFF: True,
- ASYNC: False,
- SKIP: False


Use `-` to disable a directive that is enabled by default, e.g. `# xdoctest: -ELLIPSIS`, or use `+` to enable a directive that is disabled by default, e.g. `# xdoctest +SKIP`.

4.2.7.2 Advanced Directives

Advanced directives may take arguments, be conditional, or modify the runtime state in complex ways. For instance, whereas most directives modify a boolean value in the runtime state, the advanced `REQUIRES` directive either adds or removes a value from a set of unmet requirements. Doctests will only run if there are no unmet requirements.

Currently the only advanced directive is `REQUIRES(.)`. Multiple arguments may be specified, by separating them with commas. The currently available arguments allow you to condition on:

- Special operating system / python implementation / python version tags, via: `WIN32`, `LINUX`, `DARWIN`, `POSIX`, `NT`, `JAVA`, `CPYTHON`, `IRONPYTHON`, `JYTHON`, `PYPY`, `PY2`, `PY3`. (e.g. `# xdoctest +REQUIRES(WIN32)`)
- Command line flags, via: `--<someflag>`, (e.g. `# xdoctest +REQUIRES(--verbose)`)
- If a python module is installed, via: `module:<modname>`, (e.g. `# xdoctest +REQUIRES(module:numpy)`)
- Environment variables, via: `env:<varname>==<val>`, (e.g. `# xdoctest +REQUIRES(env:MYENVIRON==1)`)

 Todo

- [] Directive for Python version: e.g. `xdoctest: +REQUIRES(Python>=3.7)`
- [] Directive for module version: e.g. `xdoctest: +REQUIRES(module:rich>=1.0)`
- [] Customize directive.
- [] Add `SKIPIF` directive

Customized Requirements Design:

- Allow user to specify a customized requirement on the CLI or environ. e.g. `XDOCTEST_CUSTOM_MY_REQUIRE="import torch; torch.cuda.is_available()"`
Then `xdoctest: +REQUIRES(custom:MY_REQUIRE)` would invoke it and enable the missing requirement if that snippet ended with a truthy or falsy value

CommandLine

```
python -m xdoctest.directive __doc__
```

The following example shows how the `+SKIP` directives may be used to bypass certain places in the code.

Example

```
>>> # An inline directive appears on the same line as a command and
>>> # only applies to the current line.
>>> raise AssertionError('this will not be run (a)') # xdoctest: +SKIP
>>> print('This line will print: (A)')
>>> print('This line will print: (B)')
>>> # However, if a directive appears on its own line, then it applies
>>> # too all subsequent lines.
>>> # xdoctest: +SKIP()
>>> raise AssertionError('this will not be run (b)')
>>> print('This line will not print: (A)')
>>> # Note, that SKIP is simply a state and can be disabled to allow
>>> # the program to continue executing.
>>> # xdoctest: -SKIP
```

(continues on next page)

(continued from previous page)

```

>>> print('This line will print: (C)')
>>> print('This line will print: (D)')
>>> # This applies to inline directives as well
>>> # xdoctest: +SKIP("an assertion would occur")
>>> raise AssertionError('this will not be run (c)')
>>> print('This line will print: (E)') # xdoctest: -SKIP
>>> raise AssertionError('this will not be run (d)')
>>> # xdoctest: -SKIP("a reason can be given as an argument")
>>> print('This line will print: (F)')

```

This next examples illustrates how to use the advanced +REQUIRES() directive. Note, the REQUIRES and SKIP states are independent.

Example

```

>>> import sys
>>> plat = sys.platform
>>> count = 0
>>> # xdoctest: +REQUIRES(WIN32)
>>> assert plat.startswith('win32'), 'this only runs on windows'
>>> count += 1
>>> # xdoctest: -REQUIRES(WIN32)
>>> # xdoctest: +REQUIRES(LINUX)
>>> assert plat.startswith('linux'), 'this only runs on linux'
>>> count += 1
>>> # xdoctest: -REQUIRES(LINUX)
>>> # xdoctest: +REQUIRES(DARWIN)
>>> assert plat.startswith('darwin'), 'this only runs on osx'
>>> count += 1
>>> # xdoctest: -REQUIRES(DARWIN)
>>> print(count)
>>> import sys
>>> if any(plat.startswith(n) for n in {'linux', 'win32', 'darwin'}):
>>>     assert count == 1, 'Exactly one of the above parts should have run'
>>> else:
>>>     assert count == 0, 'Nothing should have run on plat={}'.format(plat)
>>> # xdoctest: +REQUIRES(--verbose)
>>> print('This is only printed if you run with --verbose')

```

Example

```

>>> # New in 0.7.3: the requires directive can accept module names
>>> # xdoctest: +REQUIRES(module:foobar)

```

`xdoctest.directive.named(key: str, pattern: str) → str`

helper for regex

Parameters

- **key** (*str*)
- **pattern** (*str*)

Returns

str

class xdoctest.directive.**RuntimeStateDict**

Bases: `TypedDict`

TypedDict representing the xdoctest runtime state.

This defines the structure of the runtime state dictionary used by the directive system to control doctest behavior.

DONT_ACCEPT_BLANKLINE: `bool`

ELLIPSIS: `bool`

IGNORE_WHITESPACE: `bool`

IGNORE_EXCEPTION_DETAIL: `bool`

NORMALIZE_WHITESPACE: `bool`

IGNORE_WANT: `bool`

NORMALIZE_REPR: `bool`

REPORT_CDIF: `bool`

REPORT_NDIFF: `bool`

REPORT_UDIFF: `bool`

ASYNC: `bool`

SKIP: `bool`

REQUIRES: `set[str]`

class xdoctest.directive.**Effect**(*action, key, value*)

Bases: `tuple`

Create new instance of Effect(action, key, value)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('action', 'key', 'value')

classmethod **_make**(*iterable*)

Make a new Effect object from a sequence or iterable

_replace(***kws*)

Return a new Effect object replacing specified fields with new values

action

Alias for field number 0

key

Alias for field number 1

value

Alias for field number 2

class `xdoctest.directive.RuntimeState`(*default_state*: RuntimeStateDict | None = None)

Bases: *NiceRepr*

Maintains the runtime state for a single `run()` of an example

Inline directives are pushed and popped after the line is run. Otherwise directives persist until another directive disables it.

CommandLine

```
xdoctest -m xdoctest.directive RuntimeState
```

Example

```
>>> from xdoctest.directive import *
>>> runstate = RuntimeState()
>>> assert not runstate['IGNORE_WHITESPACE']
>>> # Directives modify the runtime state
>>> directives = list(Directive.extract('# xdoc: -ELLIPSIS, +IGNORE_WHITESPACE'))
>>> runstate.update(directives)
>>> assert not runstate['ELLIPSIS']
>>> assert runstate['IGNORE_WHITESPACE']
>>> # Inline directives only persist until the next update
>>> directives = [Directive('IGNORE_WHITESPACE', False, inline=True)]
>>> runstate.update(directives)
>>> assert not runstate['IGNORE_WHITESPACE']
>>> runstate.update({})
>>> assert runstate['IGNORE_WHITESPACE']
```

Example

```
>>> # xdoc: +IGNORE_WHITESPACE
>>> print(str(RuntimeState()))
<RuntimeState({
  ASYNC: False,
  DONT_ACCEPT_BLANKLINE: False,
  ELLIPSIS: True,
  IGNORE_EXCEPTION_DETAIL: False,
  IGNORE_WANT: False,
  IGNORE_WHITESPACE: False,
  NORMALIZE_REPR: True,
  NORMALIZE_WHITESPACE: True,
  REPORT_CDIFF: False,
  REPORT_NDIFF: False,
  REPORT_UDIFF: True,
  REQUIRES: set(...),
  SKIP: False
})>
```

Parameters

default_state (*None* | *dict*) – starting default state, if unspecified falls back to the global DEFAULT_RUNTIME_STATE

_global_state: *RuntimeStateDict*

_inline_state: *dict[str, Any]*

to_dict() → *OrderedDict[str, bool | set[str]]*

Returns

OrderedDict

set_report_style(*reportchoice: Literal['udiff', 'ndiff', 'cdiff'], state: RuntimeStateDict | None = None*) → *None*

Parameters

- **reportchoice** (*ReportStyle*) – name of report style. Must be one of ‘udiff’, ‘ndiff’, or ‘cdiff’.
- **state** (*None* | *RuntimeStateDict*) – if unspecified defaults to the global state

Example

```
>>> from xdoctest.directive import *
>>> runstate = RuntimeState()
>>> assert runstate['REPORT_UDIFF']
>>> runstate.set_report_style('ndiff')
>>> assert not runstate['REPORT_UDIFF']
>>> assert runstate['REPORT_NDIFF']
```

update(*directives: list[Directive]*) → *None*

Update the runtime state given a set of directives

Parameters

directives (*list[Directive]*) – list of directives. The effects method is used to update this object.

class xdoctest.directive.**Directive**(*name: str, positive: bool = True, args: list[str] | None = None, inline: bool | None = None*)

Bases: *NiceRepr*

Directives modify the runtime state.

Parameters

- **name** (*str*) – The name of the directive
- **positive** (*bool*) – if it is enabling / disabling
- **args** (*list[str]*) – arguments given to the directive
- **inline** (*bool | None*) – True if this is an inline directive (i.e. only impacts a single line)

classmethod **extract**(*text: str*) → *Iterator[Directive]*

Parses directives from a line or repl line

Parameters

text (*str*) – must correspond to exactly one PS1 line and its PS2 followups.

Yields

Directive – directive - the parsed directives

Note

The original doctest module sometimes yielded false positives for a directive pattern. Because xdoctest is parsing the text, this issue does not occur.

Example

```
>>> from xdoctest.directive import Directive, RuntimeState
>>> state = RuntimeState()
>>> assert len(state['REQUIRES']) == 0
>>> extracted1 = list(Directive.extract('# xdoctest: +REQUIRES(CPYTHON)'))
>>> extracted2 = list(Directive.extract('# xdoctest: +REQUIRES(PYPY)'))
>>> print('extracted1 = {!r}'.format(extracted1))
>>> print('extracted2 = {!r}'.format(extracted2))
>>> effect1 = extracted1[0].effects()[0]
>>> effect2 = extracted2[0].effects()[0]
>>> print('effect1 = {!r}'.format(effect1))
>>> print('effect2 = {!r}'.format(effect2))
>>> assert effect1.value == 'CPYTHON'
>>> assert effect2.value == 'PYPY'
>>> # At least one of these will not be satisfied
>>> assert effect1.action == 'set.add' or effect2.action == 'set.add'
>>> state.update(extracted1)
>>> state.update(extracted2)
>>> print('state = {!r}'.format(state))
>>> assert len(state['REQUIRES']) > 0
```

Example

```
>>> from xdoctest.directive import Directive
>>> text = '# xdoc: + SKIP'
>>> print(', '.join(list(map(str, Directive.extract(text)))))
<Directive(+SKIP)>
```

```
>>> # Directive with args
>>> text = '# xdoctest: requires(--show)'
>>> print(', '.join(list(map(str, Directive.extract(text)))))
<Directive(+REQUIRES(--show))>
```

```
>>> # Malformatted directives are ignored
>>> # xdoctest: +REQUIRES(module:pytest)
>>> text = '# xdoctest: does_not_exist, skip'
>>> import pytest
>>> with pytest.warns(Warning) as record:
>>>     print(', '.join(list(map(str, Directive.extract(text)))))
<Directive(+SKIP)>
```

```
>>> # Two directives in one line
>>> text = '# xdoctest: +ELLIPSIS, -NORMALIZE_WHITESPACE'
>>> print(', '.join(list(map(str, Directive.extract(text)))))
<Directive(+ELLIPSIS)>, <Directive(-NORMALIZE_WHITESPACE)>
```

```
>>> # Make sure commas inside parens are not split
>>> text = '# xdoctest: +REQUIRES(module:foo,module:bar)'
>>> print(', '.join(list(map(str, Directive.extract(text)))))
<Directive(+REQUIRES(module:foo, module:bar)>
```

Example

```
>>> from xdoctest.directive import Directive, RuntimeState
>>> any(Directive.extract(' # xdoctest: skip'))
True
>>> any(Directive.extract(' # badprefix: not-a-directive'))
False
>>> any(Directive.extract(' # xdoctest: skip'))
True
>>> any(Directive.extract(' # badprefix: not-a-directive'))
False
```

`_unpack_args(num: int) → list[str] | None`

`effect(argv: list[str] | None = None, environ: dict[str, str] | None = None) → Effect`

`effects(argv: list[str] | None = None, environ: dict[str, str] | None = None) → list[Effect]`

Returns how this directive modifies a RuntimeState object

This is called by `RuntimeState.update()` to update itself

Parameters

- **argv** (`list[str] | None`) – Command line the directive is interpreted in the context of. If unspecified, uses `sys.argv`.
- **environ** (`dict[str, str] | None`) – Environment variables the directive is interpreted in the context of. If unspecified, uses `os.environ`.

Returns

list of named tuples containing:

action (str): code indicating how to update key (str): name of runtime state item to modify
value (object): value to modify with

Return type

`list[Effect]`

CommandLine

```
xdoctest -m xdoctest.directive Directive.effects
```

Example

```
>>> Directive('SKIP').effects()[0]
Effect(action='assign', key='SKIP', value=True)
>>> Directive('SKIP', inline=True).effects()[0]
Effect(action='assign', key='SKIP', value=True)
>>> Directive('REQUIRES', args=['-s']).effects(argv=['-s'])[0]
Effect(action='noop', key='REQUIRES', value='-s')
```

(continues on next page)

(continued from previous page)

```
>>> Directive('REQUIRES', args=['-s']).effects(argv=[])[0]
Effect(action='set.add', key='REQUIRES', value='-s')
>>> Directive('ELLIPSIS', args=['-s']).effects(argv=[])[0]
Effect(action='assign', key='ELLIPSIS', value=True)
```

Doctest

```
>>> # requirement directive with module
>>> directive = list(Directive.extract('# xdoctest: requires(module:xdoctest)
↳'))[0]
>>> print('directive = {}'.format(directive))
>>> print('directive.effects() = {}'.format(directive.effects()[0]))
directive = <Directive(+REQUIRES(module:xdoctest))>
directive.effects() = Effect(action='noop', key='REQUIRES', value=
↳'module:xdoctest')
```

```
>>> directive = list(Directive.extract('# xdoctest: requires(module:notamodule)
↳'))[0]
>>> print('directive = {}'.format(directive))
>>> print('directive.effects() = {}'.format(directive.effects()[0]))
directive = <Directive(+REQUIRES(module:notamodule))>
directive.effects() = Effect(action='set.add', key='REQUIRES', value=
↳'module:notamodule')
```

```
>>> directive = list(Directive.extract('# xdoctest: requires(env:F00==1)'))[0]
>>> print('directive = {}'.format(directive))
>>> print('directive.effects() = {}'.format(directive.effects(environ={})[0]))
directive = <Directive(+REQUIRES(env:F00==1))>
directive.effects() = Effect(action='set.add', key='REQUIRES', value='env:F00==1
↳')
```

```
>>> directive = list(Directive.extract('# xdoctest: requires(env:F00==1)'))[0]
>>> print('directive = {}'.format(directive))
>>> print('directive.effects() = {}'.format(directive.effects(environ={'F00': '1
↳'})[0]))
directive = <Directive(+REQUIRES(env:F00==1))>
directive.effects() = Effect(action='noop', key='REQUIRES', value='env:F00==1')
```

```
>>> # requirement directive with two args
>>> directive = list(Directive.extract('# xdoctest: requires(--show,
↳module:xdoctest)'))[0]
>>> print('directive = {}'.format(directive))
>>> for effect in directive.effects():
>>>     print('effect = {}'.format(effect))
directive = <Directive(+REQUIRES(--show, module:xdoctest))>
effect = Effect(action='set.add', key='REQUIRES', value='--show')
effect = Effect(action='noop', key='REQUIRES', value='module:xdoctest')
```

`xdoctest.directive._split_opstr(optstr: str) → list[str]`

Simplified balanced paren logic to only split commas outside of parens

Parameters

opstr (*str*) – the command, which may contain more than one directive

Returns

individual directive optstrings

Return type

list[str]

Example

```
>>> optstr = '+FOO, REQUIRES(foo,bar), +ELLIPSIS'
>>> _split_opstr(optstr)
['+FOO', 'REQUIRES(foo,bar)', '+ELLIPSIS']
```

xdoctest.directive._is_requires_satisfied(*arg: str, argv: list[str] | None = None, environ: dict[str, str] | None = None*) → bool

Determines if the argument to a REQUIRES directive is satisfied

Parameters

- **arg** (*str*) – condition code
- **argv** (*list[str] | None*) – cmdline if arg is cmd code usually sys.argv
- **environ** (*dict[str, str] | None*) – environment variables usually os.environ

Returns

flag - True if the requirement is met

Return type

bool

Example

```
>>> from xdoctest.directive import * # NOQA
>>> from xdoctest.directive import _is_requires_satisfied
>>> _is_requires_satisfied('PY2', argv=[])
>>> _is_requires_satisfied('PY3', argv=[])
>>> _is_requires_satisfied('cpython', argv=[])
>>> _is_requires_satisfied('pypy', argv=[])
>>> _is_requires_satisfied('nt', argv=[])
>>> _is_requires_satisfied('linux', argv=[])
```

```
>>> _is_requires_satisfied('env:FOO', argv=[], environ={'FOO': '1'})
True
>>> _is_requires_satisfied('env:FOO==1', argv=[], environ={'FOO': '1'})
True
>>> _is_requires_satisfied('env:FOO==T', argv=[], environ={'FOO': '1'})
False
>>> _is_requires_satisfied('env:BAR', argv=[], environ={'FOO': '1'})
False
>>> _is_requires_satisfied('env:BAR==1', argv=[], environ={'FOO': '1'})
False
>>> _is_requires_satisfied('env:BAR!=1', argv=[], environ={'FOO': '1'})
True
>>> _is_requires_satisfied('env:BAR!=1', argv=[], environ={'BAR': '0'})
```

(continues on next page)

(continued from previous page)

```
True
>>> _is_requires_satisfied('env:BAR!=1')
...

```

```
>>> # xdoctest: +REQUIRES(module:pytest)
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     _is_requires_satisfied('badflag:BAR==1', [])

```

```
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     _is_requires_satisfied('env:BAR>=1', argv=[], environ={'BAR': '0'})

```

`xdoctest.directive._module_exists(modname: Any) → bool`

Parameters

`modname` (*str*)

Returns

bool

`xdoctest.directive.parse_directive_optstr(optpart: str, inline: None | bool = None) → Directive | None`

Parses the information in the directive from the “optpart”

optstrs are:

optionally prefixed with + (default) or - comma separated may contain one paren enclosed argument (experimental) all spaces are ignored

Parameters

- **optpart** (*str*) – the string corresponding to the operation
- **inline** (*None | bool*) – True if the directive only applies to a single line.

Returns

the parsed directive (or None if parsing failed)

Return type

Directive | None

Example

```
>>> print(str(parse_directive_optstr('+IGNORE_WHITESPACE')))
<Directive(+IGNORE_WHITESPACE)>

```

4.2.8 xdoctest.doctest_example module

This module defines the main class that holds a DocTest example

`class xdoctest.doctest_example.DocTestConfig(*args: Any, **kwargs: Any)`

Bases: `dict`

Doctest configuration

Static configuration for collection, execution, and reporting doctests. Note dynamic directives are not managed by DoctestConfig, they use RuntimeState.

`_populate_from_cli` (*ns*)

`_update_argparse_cli` (*add_argument: Callable[[...], Any], prefix: str | list[str] | None = None, defaults: dict[str, Any] = {}*) → None

Updates a pytest or argparse CLI

Parameters

add_argument (*callable*) – the parser.add_argument function

`getvalue` (*key: str, given: Any = None*) → object

Parameters

- **key** (*str*) – The configuration key
- **given** (*Any*) – A user override

Returns

if given is None returns the configured value

Return type

Any

```
class xdoctest.doctest_example.DocTest(docsrc: str, modpath: str | PathLike | None = None, callname: str | None = None, num: int = 0, lineno: int = 1, fpath: str | PathLike | None = None, block_type: str | None = None, mode: str = 'pytest')
```

Bases: `object`

Holds information necessary to execute and verify a doctest

Variables

- **docsrc** (*str*) – doctest source code
- **modpath** (*str | PathLike | None*) – module the source was read from
- **callname** (*str*) – name of the function/method/class/module being tested
- **num** (*int*) – the index of the doctest in the docstring. (i.e. this object refers to the num-th doctest within a docstring)
- **lineno** (*int*) – The line (starting from 1) in the file that the doctest begins on. (i.e. if you were to go to this line in the file, the first line of the doctest should be on this line).
- **fpath** (*PathLike*) – Typically the same as modpath, only specified for non-python files (e.g. rst files).
- **block_type** (*str | None*) – Hint indicating the type of docstring block. Can be ('Example', 'Doctest', 'Script', 'Benchmark', 'zero-arg', etc..).
- **mode** (*str*) – Hint at what created / is running this doctest. This impacts how results are presented and what doctests are skipped. Can be “native” or “pytest”. Defaults to “pytest”.
- **config** (`DoctestConfig`) – configuration for running / checking the doctest
- **module** (*ModuleType | None*) – a reference to the module that contains the doctest
- **modname** (*str*) – name of the module that contains the doctest.

- **failed_tb_lineno** (*int* | *None*) – Line number a failure occurred on.
- **exc_info** (*None* | *tuple*[*type*[*BaseException*], *BaseException*, *types.TracebackType*] | *tuple*[*None*, *None*, *None*]) – traceback of a failure if one occurred.
- **failed_part** (*None* | *DoctestPart*) – the part containing the failure if one occurred.
- **warn_list** (*list*) – from `warnings.catch_warnings()`
- **logged_evals** (*OrderedDict*) – Mapping from part index to what they evaluated to (if anything)
- **logged_stdout** (*OrderedDict*) – Mapping from part index to captured stdout.
- **global_namespace** (*dict*) – globals visible to the doctest

CommandLine

```
xdoctest -m xdoctest.doctest_example DocTest
```

Example

```
>>> from xdoctest import core
>>> from xdoctest import doctest_example
>>> import os
>>> modpath = doctest_example.__file__.replace('.pyc', '.py')
>>> modpath = os.path.realpath(modpath)
>>> testables = core.parse_doctestables(modpath)
>>> for test in testables:
>>>     if test.callname == 'DocTest':
>>>         self = test
>>>         break
>>> assert self.num == 0
>>> assert self.modpath == modpath
>>> print(self)
<DocTest(xdoctest.doctest_example DocTest:0 ln ...)>
```

Parameters

- **docsrc** (*str*) – the text of the doctest
- **modpath** (*str* | *PathLike* | *None*)
- **callname** (*str* | *None*)
- **num** (*int*)
- **lineno** (*int*)
- **fpath** (*str* | *None*)
- **block_type** (*str* | *None*)
- **mode** (*str*)

```
UNKNOWN_MODNAME = '<modname?>'
```

```
UNKNOWN_MODPATH = '<modpath?>'
```

```
UNKNOWN_CALLNAME = '<callname?>'
UNKNOWN_FPATH = '<fpath?>'
module: types.ModuleType | None
fpath: str | os.PathLike | None
modname: str | None
docsrc: str | None
lineno: int | None
num: int | None
_parts: list['DoctestPart'] | None
failed_tb_lineno: int | None
exc_info: tuple[type[BaseException], BaseException, types.TracebackType] |
tuple[None, None, None] | None
failed_part: 'DoctestPart' | str | None
warn_list: list | None
_partfilename: str | None
failed_tb_part: 'DoctestPart' | None
_partfilename_to_part: dict[str, 'DoctestPart']
logged_evals: OrderedDict[int, Any] | None
logged_stdout: OrderedDict[int, str | None] | None
_unmatched_stdout: list[str] | None
_skipped_parts: list | None
_runstate: Any
global_namespace: dict[str, Any]
```

`is_disabled(pytest=False) → bool`

Checks for comment directives on the first line of the doctest

A doctest is force-disabled if it starts with any of the following patterns

- `>>> # DISABLE_DOCTEST`
- `>>> # SCRIPT`
- `>>> # UNSTABLE`
- `>>> # FAILING`

And if running in pytest, you can also use

- `>>> import pytest; pytest.skip()`

Note

modern versions of xdoctest contain directives like `# xdoctest: +SKIP`, which are a better way to do this.

Todo

Robustly deprecate these non-standard ways of disabling a doctest. Generate a warning for several versions if they are used, and indicate what the replacement strategy is. Then raise an error for several more versions before finally removing this code.

Return type

`bool`

property `unique_callname`: `str`

A key that references this doctest given its module

Returns

`str`

property `node`: `str`

A key that references this doctest within pytest

Returns

`str`

property `valid_testnames`: `set[str]`

A set of `callname` and `unique_callname`

Returns

`Set[str]`

`wants()` → `Generator[str, None, None]`

Returns a list of the populated wants

Yields

`str`

`format_parts`(*linenos*: `bool = True`, *colored*: `bool | None = None`, *want*: `bool = True`, *offset_linenos*: `bool | None = None`, *prefix*: `bool = True`)

Used by `format_src()`

Parameters

- **linenos** (`bool`) – show line numbers
- **colored** (`bool | None`) – pygmentize the code
- **want** (`bool`) – include the want value if it exists
- **offset_linenos** (`bool`) – if True include the line offset relative to the source file
- **prefix** (`bool`) – if False, exclude the doctest ``>>>`` prefix

`format_src`(*linenos*: `bool = True`, *colored*: `bool | None = None`, *want*: `bool = True`, *offset_linenos*: `bool | None = None`, *prefix*: `bool = True`) → `str`

Adds prefix and line numbers to a doctest

Parameters

- **linenos** (*bool*) – if True, adds line numbers to output
- **colored** (*bool*) – if True highlight text with ansi colors. Default is specified in the config.
- **want** (*bool*) – if True includes “want” lines (default False).
- **offset_linenos** (*bool*) – if True offset line numbers to agree with their position in the source text file (default False).
- **prefix** (*bool*) – if False, exclude the doctest ``>>> `` prefix

Returns

str

Example

```
>>> from xdoctest.core import *
>>> from xdoctest import core
>>> testables = parse_doctestables(core.__file__)
>>> self = next(testables)
>>> self._parse()
>>> print(self.format_src())
>>> print(self.format_src(linenos=False, colored=False))
>>> assert not self.is_disabled()
```

_parse() → None

Divide the given string into examples and intervening text.

Returns

None

Example

```
>>> s = 'I am a dummy example with three parts'
>>> x = 10
>>> print(s)
I am a dummy example with three parts
>>> s = 'My purpose it so demonstrate how wants work here'
>>> print('The new want applies ONLY to stdout')
>>> print('given before the last want')
>>> '''
    this wont hurt the test at all
    even though its multiline '''
>>> y = 20
The new want applies ONLY to stdout
given before the last want
>>> # Parts from previous examples are executed in the same context
>>> print(x + y)
30
```

this is simply text, and doesnt apply to the previous doctest the <BLANKLINE> directive is still in effect.

Example

```

>>> from xdoctest import parser
>>> from xdoctest.docstr import docsrape_google
>>> from xdoctest import doctest_example
>>> DocTest = doctest_example.DocTest
>>> docstr = DocTest._parse.__doc__
>>> blocks = docsrape_google.split_google_docblocks(docstr)
>>> doclineno = DocTest._parse.__code__.co_firstlineno
>>> key, (docsrc, offset) = blocks[-2]
>>> lineno = doclineno + offset
>>> self = DocTest(docsrc, doctest_example.__file__, '_parse', 0,
>>>                lineno)
>>> self._parse()
>>> assert len(self._parts) >= 3
>>> #p1, p2, p3 = self._parts
>>> self.run()

```

`_import_module()` → None

After this point we are in dynamic analysis mode, in most cases xdoctest should have been in static-analysis-only mode.

Returns

None

`static _extract_future_flags(namespace: Mapping)` → int

Return the compiler-flags associated with the future features that have been imported into the given namespace (i.e. globals).

Returns

int

`_test_globals()`

`_partfilename_for(partno: int)` → str

Construct a synthetic filename for a specific doctest part.

We can't use one filename for the entire doctest because traceback frames only record a filename + line number. If every compiled part shares the same filename, then when a traceback points into code defined in an earlier part, we cannot tell which part that line number belongs to.

Giving each part its own pseudo-filename makes traceback ownership unambiguous.

`anything_ran()` → bool

Returns

bool

`run(verbose: int | None | bool = None, on_error: str | None = None)` → dict[str, Any]

Executes the doctest, checks the results, reports the outcome.

Parameters

- **`verbose`** (*int*) – verbosity level
- **`on_error`** (*str*) – can be 'raise' or 'return'

Returns

summary

Return type

Dict

_check_or_defer_part_output(*part*: DoctestPart, *got_stdout*: str | None, *got_eval*: Any, *runstate*: directive.RuntimeState) → None

Apply the configured output contract for one executed part.

With the default configuration, parts without a local want may defer stdout for later trailing matching, while parts with a local want are checked immediately. The *deferred_output_matching* knob disables the deferred-trailing behavior, and *optional_want* requires output producing parts to have a local want unless *IGNORE_WANT* is active. Any part with *IGNORE_WANT* active is treated as a boundary and does not contribute output to later matching.

property globs

Alias for `global_namespace` for pytest 8.0 compatibility

property cmdline: str

A cli-instruction that can be used to execute *this* doctest.

Return type

str

property _block_prefix

_pre_run(*verbose*: bool | int) → None

failed_line_offset() → int | None

Determine which line in the doctest failed.

failed_lineno() → int | None

Returns

int | None

repr_failure(*with_tb*: Any = True) → list[str]

Constructs lines detailing information about a failed doctest

Parameters

with_tb (bool) – if True include the traceback

Returns

List[str]

CommandLine

```
python -m xdoctest.core DocTest.repr_failure:0
python -m xdoctest.core DocTest.repr_failure:1
python -m xdoctest.core DocTest.repr_failure:2
```

Example

```
>>> from xdoctest.core import *
>>> docstr = utils.codeblock(
    '''
    >>> x = 1
    >>> print(x + 1)
    2
```

(continues on next page)

(continued from previous page)

```

>>> print(x + 3)
3
>>> print(x + 100)
101
'''
>>> parsekw = dict(fpath='foo.txt', callname='bar', lineno=42)
>>> self = list(parse_docstr_examples(docstr, **parsekw))[0]
>>> summary = self.run(on_error='return', verbose=0)
>>> print('[res]' + '\n[res]'.join(self.repr_failure()))

```

Example

```

>>> from xdoctest.core import *
>>> docstr = utils.codeblock(
    r'''
>>> 1
1
>>> print('. .\n. .') # xdoc: -NORMALIZE_WHITESPACE
. .
. .
'''
>>> parsekw = dict(fpath='foo.txt', callname='bar', lineno=42)
>>> self = list(parse_docstr_examples(docstr, **parsekw))[0]
>>> summary = self.run(on_error='return', verbose=1)
>>> print('[res]' + '\n[res]'.join(self.repr_failure()))

```

Example

```

>>> from xdoctest.core import *
>>> docstr = utils.codeblock(
    '''
>>> assert True
>>> assert False
>>> x = 100
'''
>>> self = list(parse_docstr_examples(docstr))[0]
>>> summary = self.run(on_error='return', verbose=0)
>>> print('[res]' + '\n[res]'.join(self.repr_failure()))

```

`_print_captured()` → None

`_color(text: str, color: str, enabled: bool | None = None)`

conditionally color text based on config and flags

`_post_run(verbose: bool | int) → dict[str, Any]`

Returns

summary

Return type

Dict

`xdoctest.doctest_example._traverse_traceback(tb)`

4.2.9 xdoctest.doctest_part module

Simple storage container used to store a single executable part of a doctest example. Multiple parts are typically stored in a `xdoctest.doctest_example.Doctest`, which manages execution of each part.

```
class xdoctest.doctest_part.DoctestPart(exec_lines: list[str], want_lines: list[str] | None = None,
                                         line_offset: int = 0, orig_lines: list[str] | None = None,
                                         directives: list | None = None, partno: int | None = None)
```

Bases: `object`

Parameters

- **exec_lines** (*List[str]*) – executable lines in this part
- **want_lines** (*List[str] | None*) – lines that the result of the execution should match
- **line_offset** (*int*) – line number relative to the start of the doctest
- **orig_lines** (*List[str] | None*) – The original text parsed into exec and want. This is only used in formatting and may be removed in the future.
- **directives** (*list | None*) – directives that this part will apply before being run. If unspecified, these will be extracted.
- **partno** (*int | None*) – identifies the part number in the larger example

exec_lines: `list[str]`

want_lines: `list[str] | None`

line_offset: `int`

orig_lines: `list[str] | None`

partno: `int | None`

compile_mode: `str`

The result of parsing that represents a “logical block” of code. If a want statement is defined, it is stored here.

Variables

- **exec_lines** (*List[str]*) – executable lines in this part
- **want_lines** (*List[str] | None*) – lines that the result of the execution should match
- **line_offset** (*int*) – line number relative to the start of the doctest
- **orig_lines** (*List[str] | None*) – the original text parsed into exec and want
- **directives** (*list | None*) – directives that this part will apply before being run
- **partno** (*int | None*) – identifies the part number in the larger example
- **compile_mode** (*str*) – mode passed to compile.

property n_lines: `int`

Returns: `int`: number of lines in the entire source (i.e. `exec + want`)

property n_exec_lines: `int`

Returns: `int`: number of executable lines (excluding `want`)

property n_want_lines: `int`

Returns: `int`: number of lines in the “want” statement.

property source: `str`

Returns: `str`: A single block of text representing the source code.

compilable_source() → `str`

Use this to build the string for compile. Takes care of a corner case.

Returns

`str`

has_any_code() → `bool`

Heuristic to check if there is any runnable code in this doctest. We currently just check that not every line is a comment, which helps the runner count a test as skipped if only lines with comments “ran”.

Returns

`bool`

property directives: `list[Directive]`

Returns

The extracted or provided directives to be applied.

Return type

`List[directive.Directive]`

Example

```
>>> self = DoctestPart(['
>>> print(', '.join(list(map(str, self.directives))))
<Directive(+SKIP)>
```

property want: `str | None`

Returns: `str | None`: what the test is expected to produce

check(*got_stdout: str, got_eval: str | _NOT_EVAL_TYPE = <NOT_EVALUED>, runstate: RuntimeState | None = None, unmatched: list | None = None*) → `None`

Check if the “got” output obtained by running this test matches the “want” target. Note there are two types of “got” output: (1) output from stdout and (2) eval'd output. If both are specified, then want may match either value.

Parameters

- **got_stdout** (*str*) – output from stdout
- **got_eval** (*str*) – output from an eval statement
- **runstate** (*directive.RuntimeState*) – runner options
- **unmatched** (*list*) – if specified, the want statement is allowed to match any trailing sequence of unmatched output and `got_stdout` from this doctest part.

Raises

xdoctest.checker.GotWantException - If the "got" differs from this parts want. -

Example

```

>>> # xdoctest: +REQUIRES(module:pytest)
>>> import pytest
>>> got_stdout = 'more text\n'
>>> unmatched = ['some text\n']
>>> self = DoctestPart(None, want_lines=['some text', 'more text'])
>>> self.check(got_stdout, unmatched=unmatched)
>>> # Leading junk doesnt matter if we match a trailing sequence
>>> self.check(got_stdout, unmatched=['junk\n'] + unmatched)
>>> # fail when want doesnt match any trailing sequence
>>> with pytest.raises(checker.GotWantException):
>>>     self.check(got_stdout)
>>> with pytest.raises(checker.GotWantException):
>>>     self.check(got_stdout, ['some text\n', 'junk\n'])

```

format_part(*linenos*: *bool* = *True*, *want*: *bool* = *True*, *startline*: *int* = *1*, *n_digits*: *int* | *None* = *None*, *colored*: *bool* = *False*, *partnos*: *bool* = *False*, *prefix*: *bool* = *True*) → *str*

Customizable formatting of the source and want for this doctest.

Parameters

- **linenos** (*bool*) – show line numbers
- **want** (*bool*) – include the want value if it exists
- **startline** (*int*) – offsets the line numbering
- **n_digits** (*int* | *None*) – number of digits to use for line numbers
- **colored** (*bool*) – pygmentize the code
- **partnos** (*bool*) – if *True*, shows the part number in the string
- **prefix** (*bool*) – if *False*, exclude the doctest ``>>> `` prefix

Returns

pretty text suitable for printing

Return type

str

CommandLine

```
python -m xdoctest.doctest_part DoctestPart.format_part
```

Example

```

>>> from xdoctest.parser import *
>>> self = DoctestPart(exec_lines=['print(123)'],
>>>                    want_lines=['123'], line_offset=0, partno=1)
>>> # xdoctest: -NORMALIZE_WHITESPACE
>>> print(self.format_part(partnos=True))
(p1) 1 >>> print(123)
      123

```

Example

```
>>> from xdoctest.parser import *
>>> self = DoctestPart(exec_lines=['print(123)'],
>>>                    want_lines=['123'], line_offset=0, partno=1)
>>> # xdoctest: -NORMALIZE_WHITESPACE
>>> print(self.format_part(partnos=False, prefix=False,
>>>                       linenos=False, want=False))
print(123)
```

4.2.10 xdoctest.dynamic_analysis module

Utilities for dynamically inspecting code

`xdoctest.dynamic_analysis.parse_dynamic_calldefs(modpath_or_module: str | os.PathLike | types.ModuleType) → dict[str, 'CallDefNode']`

Dynamic parsing of module doctestable items.

Unlike static parsing this forces execution of the module code before test-time, however the former is limited to plain-text python files whereas this can discover doctests in binary extension libraries.

Parameters

modpath_or_module (*str* | *PathLike* | *ModuleType*) – path to module or the module itself

Returns

mapping from callnames to CallDefNodes, which contain
info about the item with the doctest.

Return type

`Dict[str, xdoctest.static_analysis.CallDefNode]`

CommandLine

```
python -m xdoctest.dynamic_analysis parse_dynamic_calldefs
```

Example

```
>>> from xdoctest import dynamic_analysis
>>> module = dynamic_analysis
>>> calldefs = parse_dynamic_calldefs(module.__file__)
>>> for key, calldef in sorted(calldefs.items()):
...     print('key = {!r}'.format(key))
...     print(' * calldef.callname = {}'.format(calldef.callname))
...     if calldef.docstr is None:
...         print(' * len(calldef.docstr) = {}'.format(calldef.docstr))
...     else:
...         print(' * len(calldef.docstr) = {}'.format(len(calldef.docstr)))
```

`xdoctest.dynamic_analysis.get_stack_frame(n: int = 0, strict: bool = True) → FrameType`

Gets the current stack frame or any of its ancestors dynamically

Parameters

- **n** (*int*) – n=0 means the frame you called this function in. n=1 is the parent frame.
- **strict** (*bool*) – (default = True)

Returns

frame_cur

Return type

FrameType

Example

```
>>> frame_cur = get_stack_frame(n=0)
>>> print('frame_cur = %r' % (frame_cur,))
>>> assert frame_cur.f_globals['frame_cur'] is frame_cur
```

xdoctest.dynamic_analysis.get_parent_frame(*n: int = 0*) → FrameType

Returns the frame of that called you. This is equivalent to *get_stack_frame(n=1)*

Parameters

n (*int*) – *n=0* means the frame you called this function in. *n=1* is the parent frame.

Returns

parent_frame

Return type

FrameType

Example

```
>>> root0 = get_stack_frame(n=0)
>>> def foo():
>>>     child = get_stack_frame(n=0)
>>>     root1 = get_parent_frame(n=0)
>>>     root2 = get_stack_frame(n=1)
>>>     return child, root1, root2
>>> # Note this wont work in IPython because several
>>> # frames will be inserted between here and foo
>>> child, root1, root2 = foo()
>>> print('root0 = %r' % (root0,))
>>> print('root1 = %r' % (root1,))
>>> print('root2 = %r' % (root2,))
>>> print('child = %r' % (child,))
>>> assert root0 == root1
>>> assert root1 == root2
>>> assert child != root1
```

xdoctest.dynamic_analysis.iter_module_doctestables(*module: ModuleType*) → Iterator[tuple[str, Any]]

Yields doctestable objects that belong to a live python module

Parameters

module (*ModuleType*) – live python module

Yields

Tuple[str, callable] – (funcname, func) doctestable

CommandLine

```
python -m xdoctest.dynamic_analysis iter_module_doctestables
```

Example

```
>>> from xdoctest import dynamic_analysis
>>> module = dynamic_analysis
>>> doctestable_list = list(iter_module_doctestables(module))
>>> items = sorted([str(item) for item in doctestable_list])
>>> print('[' + '\n'.join(items) + ']')
```

`xdoctest.dynamic_analysis.is_defined_by_module`(*item: Any, module: Any*) → bool

Check if item is directly defined by a module.

This check may not always work, especially for decorated functions.

Parameters

- **item** (*object*) – item to check
- **module** (*ModuleType*) – module to check against

CommandLine

```
xdoctest -m xdoctest.dynamic_analysis is_defined_by_module
```

Example

```
>>> from xdoctest import dynamic_analysis
>>> item = dynamic_analysis.is_defined_by_module
>>> module = dynamic_analysis
>>> assert is_defined_by_module(item, module)
>>> item = dynamic_analysis.inspect
>>> assert not is_defined_by_module(item, module)
>>> item = dynamic_analysis.inspect.ismodule
>>> assert not is_defined_by_module(item, module)
>>> assert not is_defined_by_module(print, module)
>>> # xdoctest: +REQUIRES(CPython)
>>> import _ctypes
>>> item = _ctypes.Array
>>> module = _ctypes
>>> assert is_defined_by_module(item, module)
>>> item = _ctypes.CFuncPtr.restype
>>> module = _ctypes
>>> assert is_defined_by_module(item, module)
```

4.2.11 xdoctest.exceptions module

Define errors that may be raised by xdoctest

exception `xdoctest.exceptions.MalformedDocstr`

Bases: `Exception`

Exception raised when the docstring itself does not conform to the expected style (e.g. google / numpy).

exception `xdoctest.exceptions.ExistingEventLoopError`

Bases: `Exception`

Exception raised when the docstring uses a top level await and the test is already running in an event loop.

exception `xdoctest.exceptions.DoctestParseError`(*msg: str, string: str | None = None, info: Any | None = None, orig_ex: Exception | None = None*)

Bases: `Exception`

Exception raised when doctest code has an error.

Parameters

- **msg** (*str*) – error message
- **string** (*str | None*) – the string that failed
- **info** (*Any | None*) – extra information
- **orig_ex** (*Exception | None*) – The underlying exception

exception `xdoctest.exceptions.ExitTestException`

Bases: `Exception`

exception `xdoctest.exceptions.IncompleteParseError`

Bases: `SyntaxError`

Used when something goes wrong in the xdoctest parser

`xdoctest.exceptions._resolve_skipped_exception_class()` → *Any*

`xdoctest.exceptions.Skipped`

alias of `LocalSkipped`

4.2.12 `xdoctest.global_state` module

Global state initialized at import time. Used for hidden arguments and developer features.

`xdoctest.global_state._boolean_environ(key: str)` → `bool`

Parameters

key (*str*)

Returns

`bool`

4.2.13 `xdoctest.parser` module

4.2.13.1 The XDoctest Parser

This parses a docstring into one or more “doctest part” *after* the docstrings have been extracted from the source code by either static or dynamic means.

Terms and definitions:

logical block:

a snippet of code that can be executed by itself if given the correct global / local variable context.

PS1:

The original meaning is “Prompt String 1”. For details see: [\[SE32096\]](#) [\[BashPS1\]](#) [\[CustomPrompt\]](#) [\[GeekPrompt\]](#). In the context of xdoctest, instead of referring to the prompt prefix, we use PS1 to refer to a line that starts a “logical block” of code. In the original doctest module these all had to be

prefixed with “>>>”. In xdoctest the prefix is used to simply denote the code is part of a doctest. It does not necessarily mean a new “logical block” is starting.

PS2:

The original meaning is “Prompt String 2”. In the context of xdoctest, instead of referring to the prompt prefix, we use PS2 to refer to a line that continues a “logical block” of code. In the original doctest module these all had to be prefixed with “...”. However, xdoctest uses parsing to automatically determine this.

want statement:

Lines directly after a logical block of code in a doctest indicating the desired result of executing the previous block.

While I do believe this AST-based code is a significant improvement over the RE-based builtin doctest parser, I acknowledge that I’m not an AST expert and there is room for improvement here.

References

`class xdoctest.parser.DoctestParser(simulate_repl: bool = False)`

Bases: `object`

Breaks docstrings into parts using the `parse` method.

Example

```
>>> from xdoctest.parser import * # NOQA
>>> parser = DoctestParser()
>>> doctest_parts = parser.parse(
>>>     """
>>>     >>> j = 0
>>>     >>> for i in range(10):
>>>         >>>     j += 1
>>>         >>> print(j)
>>>     10
>>>     """.rstrip('\n'))
>>> print('\n'.join(list(map(str, doctest_parts))))
<DoctestPart(ln 0, src="j = 0...", want=None)>
<DoctestPart(ln 3, src="print(j)...", want="10...")>
```

Example

```
>>> # Having multiline strings in doctests can be nice
>>> string = utils.codeblock(
>>>     """
>>>     >>> name = 'name'
>>>     'anything'
>>>     """)
>>> self = DoctestParser()
>>> doctest_parts = self.parse(string)
>>> print('\n'.join(list(map(str, doctest_parts))))
```

Parameters

`simulate_repl` (*bool*) – if True each line will be treated as its own doctest. This more closely mimics the original doctest module. Defaults to False.

`parse(string: str, info: dict | None = None) → list[DoctestPart | str]`

Divide the given string into examples and interleaving text.

Parameters

- **string** (*str*) – The docstring that may contain one or more doctests.
- **info** (*dict | None*) – info about where the string came from in case of an error

Returns

a list of *DoctestPart* objects and intervening text in the input docstring.

Return type

List[xdoctest.doctest_part.DoctestPart | str]

CommandLine

```
python -m xdoctest.parser DoctestParser.parse
```

Example

```
>>> docstr = '''
>>>     A simple docstring contains text followed by an example.
>>>     >>> numbers = [1, 2, 3, 4]
>>>     >>> thirds = [x / 3 for x in numbers]
>>>     >>> print(thirds)
>>>     [0.33 0.66 1 1.33]
>>> '''
>>> from xdoctest import parser
>>> self = parser.DoctestParser()
>>> results = self.parse(docstr)
>>> assert len(results) == 3
>>> for index, result in enumerate(results):
>>>     print(f'results[{index}] = {result!r}')
results[0] = '\nA simple docstring contains text followed by an example.'
results[1] = <DoctestPart(ln 2, src="numbers ...", want=None) at ...>
results[2] = <DoctestPart(ln 4, src="print(th...", want="[0.33 0...)" at ...>
```

Example

```
>>> s = 'I am a dummy example with two parts'
>>> x = 10
>>> print(s)
I am a dummy example with two parts
>>> s = 'My purpose it so demonstrate how wants work here'
>>> print('The new want applies ONLY to stdout')
>>> print('given before the last want')
>>> '''
        this wont hurt the test at all
        even though its multiline '''
>>> y = 20
The new want applies ONLY to stdout
given before the last want
>>> # Parts from previous examples are executed in the same context
```

(continues on next page)

(continued from previous page)

```
>>> print(x + y)
30
```

this is simply text, and doesn't apply to the previous doctest the <BLANKLINE> directive is still in effect.

Example

```
>>> from xdoctest.parser import * # NOQA
>>> from xdoctest import parser
>>> from xdoctest.docstr import docsrape_google
>>> from xdoctest import core
>>> self = parser.DoctestParser()
>>> docstr = self.parse.__doc__
>>> blocks = docsrape_google.split_google_docblocks(docstr)
>>> doclineno = self.parse.__func__.__code__.co_firstlineno
>>> key, (string, offset) = blocks[-2]
>>> self._label_docsrc_lines(string)
>>> doctest_parts = self.parse(string)
>>> # each part with a want-string needs to be broken in two
>>> assert len(doctest_parts) == 6
>>> len(doctest_parts)
```

`_package_groups`(*grouped_lines*)

`_package_chunk`(*raw_source_lines*, *raw_want_lines*, *lineno=0*)

if `self.simulate_repl` is True, then each statement is broken into its own part. Otherwise, statements are grouped by the closest *want* statement.

Todo

- [] EXCEPT IN CASES OF EXPLICIT CONTINUATION

Example

```
>>> from xdoctest.parser import *
>>> raw_source_lines = ['>>> "string"']
>>> raw_want_lines = ['string']
>>> self = DoctestParser()
>>> part, = self._package_chunk(raw_source_lines, raw_want_lines)
>>> part.source
"string"
>>> part.want
'string'
```

`_group_labeled_lines`(*labeled_lines*) → list[list | tuple | str]

Group labeled lines into logical parts to be executed together

Returns

A list of parts. Text parts are just returned as a list of lines. Executable parts are returned as a tuple of source lines and an optional “want” statement.

Return type

List[List[str] | Tuple[List[str], str]]

`_locate_ps1_linenos`(*source_lines: list[str]*) → tuple[list[int], str]

Determines which lines in the source begin a “logical block” of code.

Parameters

source_lines (*List[str]*) – lines belonging only to the doctest src these will be unindented, prefixed, and without any want.

Returns

linenos is the first value a list of indices indicating which lines are considered “PS1” and mode_hint, the second value, is a flag indicating if the final line should be considered for a got/want assertion.

Return type

Tuple[List[int], bool]

Example

```
>>> self = DoctestParser()
>>> source_lines = ['>>> def foo():', '>>>     return 0', '>>> 3']
>>> linenos, mode_hint = self._locate_ps1_linenos(source_lines)
>>> assert linenos == [0, 2]
>>> assert mode_hint == 'eval'
```

Example

```
>>> from xdoctest.parser import * # NOQA
>>> self = DoctestParser()
>>> source_lines = ['>>> x = [1, 2, ', '>>> 3, 4]', '>>> print(len(x))']
>>> linenos, mode_hint = self._locate_ps1_linenos(source_lines)
>>> assert linenos == [0, 2]
>>> assert mode_hint == 'eval'
```

Example

```
>>> from xdoctest.parser import * # NOQA
>>> self = DoctestParser()
>>> source_lines = [
>>>     '>>> x = 1',
>>>     '>>> try: raise Exception',
>>>     '>>> except Exception: pass',
>>>     '...',
>>> ]
>>> linenos, mode_hint = self._locate_ps1_linenos(source_lines)
>>> assert linenos == [0, 1]
>>> assert mode_hint == 'exec'
```

Example

```
>>> from xdoctest.parser import * # NOQA
>>> self = DoctestParser()
```

(continues on next page)

(continued from previous page)

```
>>> source_lines = [
>>>     '>>> import os; print(os)',
>>>     '...',
>>> ]
>>> linenos, mode_hint = self._locate_ps1_linenos(source_lines)
>>> assert linenos == [0]
>>> assert mode_hint == 'single'
```

Example

```
>>> # We should ensure that decorators are PS1 lines
>>> from xdoctest.parser import * # NOQA
>>> self = DoctestParser()
>>> source_lines = [
>>>     '>>> # foo',
>>>     '>>> @foo',
>>>     '... def bar():',
>>>     '...     ...',
>>> ]
>>> linenos, mode_hint = self._locate_ps1_linenos(source_lines)
>>> print(f'linenos={linenos}')
>>> assert linenos == [0, 1]
```

`_label_docsrc_lines(string: str) → list[tuple[str, str]]`

Give each line in the docstring a label so we can distinguish what parts are text, what parts are code, and what parts are “want” string.

Parameters

string (*str*) – doctest source

Returns

labeled_lines - the above source broken

up by lines, each with a label indicating its type for later use in parsing.

Return type

List[Tuple[str, str]]

Todo

- [] Sphinx does not parse this doctest properly

Example

```
>>> from xdoctest.parser import *
>>> # Having multiline strings in doctests can be nice
>>> string = utils.codeblock(
>>>     '''
>>>     text
>>>     >>> items = ['also', 'nice', 'to', 'not', 'worry',
>>>                 'about', '...', 'vs', '>>>']
```

(continues on next page)

(continued from previous page)

```
... print('but its still allowed')
but its still allowed
```

more text "")

```
>>> self = DoctestParser()
>>> labeled = self._label_docsrc_lines(string)
>>> expected = [
>>>     ('text', 'text'),
>>>     ('dsrc', ">>> items = ['also', 'nice', 'to', 'not', 'worry',"],
>>>     ('dsrc', ">>>         'about', '...', 'vs', '>>>']"),
>>>     ('dcnt', "... print('but its still allowed')"),
>>>     ('want', 'but its still allowed'),
>>>     ('text', ''),
>>>     ('text', 'more text')
>>> ]
>>> assert labeled == expected
```

`xdoctest.parser._min_indentation(s)`

Return the minimum indentation of any non-blank line in *s*

`xdoctest.parser._complete_source(line, state_indent, line_iter)`

helper remove lines from the iterator if they are needed to complete source

This uses `static.is_balanced_statement()` to do the heavy lifting

Example

```
>>> from xdoctest.parser import * # NOQA
>>> from xdoctest.parser import _complete_source
>>> state_indent = 0
>>> line = '>>> x = { # The line is not finished'
>>> remain_lines = ['>>> 1:2,', '>>> 3:4,', '>>> 5:6}', '>>> y = 7']
>>> line_iter = enumerate(remain_lines, start=1)
>>> finished = list(_complete_source(line, state_indent, line_iter))
>>> final = chr(10).join([t[1] for t in finished])
>>> print(final)
```

`xdoctest.parser._iterthree(items, pad_value=None)`

Iterate over a sliding window of size 3 with None padding on both sides.

Example

```
>>> from xdoctest.parser import *
>>> print(list(_iterthree([])))
>>> print(list(_iterthree(range(1))))
>>> print(list(_iterthree([1, 2])))
>>> print(list(_iterthree([1, 2, 3])))
>>> print(list(_iterthree(range(4))))
>>> print(list(_iterthree(range(7))))
```

`xdoctest.parser._hasprefix(line, prefixes) → bool`
 helper prefix test

4.2.14 xdoctest.plugin module

4.2.15 xdoctest.runner module

4.2.15.1 The Native XDoctest Runner

This file defines the native xdoctest interface to the collecting, executing, and summarizing the results of running doctests. This is an alternative to running through pytest.

4.2.15.2 Using the XDoctest Runner via the Terminal

This interface is exposed via the `xdoctest.__main__` script and can be invoked on any module via: `python -m xdoctest <modname>`, where `<modname>` is the path to. For example to run the tests in this module you could execute:

```
python -m xdoctest xdoctest.runner all
```

For more details see:

```
python -m xdoctest --help
```

4.2.15.3 Using the XDoctest Runner Programmatically

This interface may also be run programmatically using `xdoctest.doctest_module(path)`, which can be placed in the `__main__` section of any module as such:

```
if __name__ == '__main__':
    import xdoctest as xdoc
    xdoc.doctest_module(__file__)
```

This allows you to invoke the runner on a specific module by simply running that module as the main module. Via: `python -m <modname> <command>`. For example, the this module ends with the previous code, which means you can run the doctests as such:

```
python -m xdoctest.runner list
```

`xdoctest.runner.log(msg: str, verbose: bool | int, level: int = 1) → None`

Simple conditional print logger

Parameters

- **msg** (*str*) – message to print
- **verbose** (*int*) – verbosity level, higher means more is print
- **level** (*int*) – verbosity level at which this is printed. 0 is always, 1 is info, 2 is verbose, 3 is very-verbose.

`xdoctest.runner.doctest_callable(func: Callable[[...], Any]) → None`

Executes doctests an in-memory function or class.

Parameters

- **func** (*callable*) – live method or class for which we will run its doctests.

Example

```
>>> def inception(text):
>>>     """
>>>     Example:
>>>         >>> inception("I heard you liked doctests")
>>>     """
>>>     print(text)
>>> func = inception
>>> doctest_callable(func)
```

`xdoctest.runner.gather_doctests`(*doctest_identifiers*, *style='auto'*, *analysis='auto'*, *verbose=None*) → *None*

`xdoctest.runner.doctest_module`(*module_identifier: str | ModuleType | None = None*, *command: str | None = None*, *argv: list[str] | None = None*, *exclude: list[str] = []*, *style: str = 'auto'*, *verbose: int | None = None*, *config: dict[str, Any] | None = None*, *durations: int | None = None*, *analysis: str = 'auto'*) → *dict[str, Any]*

Executes requested google-style doctests in a package or module. Main entry point into the testing framework.

Parameters

- **module_identifier** (*str | ModuleType | None*) – The name of / path to the module, or the live module itself. If not specified, dynamic analysis will be used to introspect the module that called this function and that module will be used. This can also contain the callname followed by the `::` token.
- **command** (*str*) – determines which doctests to run. if `command` is `None`, this is determined by parsing `sys.argv`. Value values are ‘all’ - find and run all tests in a module ‘list’ - list the tests in a module ‘dump’ - dumps tests to stdout
- **argv** (*List[str] | None*) – if specified, command line flags that might influence behavior. if `None` uses `sys.argv`. SeeAlso `:func:_update_argparse_cli` SeeAlso `:func:doctest_example.DoctestConfig._update_argparse_cli`
- **style** (*str*) – Determines how doctests are recognized and grouped. Can be freeform, google, or auto.
- **verbose** (*int | None*) – Verbosity level. 0 - disables all text 1 - minimal printing 3 - verbose printing
- **exclude** (*List[str]*) – ignores any modname matching any of these glob-like patterns
- **config** (*Dict[str, object]*) – modifies each examples configuration. Defaults and expected keys are documented in `xdoctest.doctest_example.DoctestConfig`
- **durations** (*int | None*) – if specified report top N slowest tests
- **analysis** (*str*) – determines if doctests are found using static or dynamic analysis.

Returns

`run_summary`

Return type

`Dict[str, Any]`

Example

```
>>> modname = 'xdoctest.dynamic_analysis'
>>> result = doctest_module(modname, 'list', argv=[''])
```

Example

```
>>> # xdoctest: +SKIP
>>> # Demonstrate different ways "module_identifier" can be specified
>>> #
>>> # Using a module name
>>> result = doctest_module('xdoctest.static_analysis')
>>> #
>>> # Using a module path
>>> result = doctest_module(os.expandpath('~/.code/xdoctest/src/xdoctest/static_
->analysis.py'))
>>> #
>>> # Using a module itself
>>> from xdoctest import runner
>>> result = doctest_module(runner)
>>> #
>>> # Using a module name and a specific callname
>>> from xdoctest import runner
>>> result = doctest_module('xdoctest.static_analysis::parse_static_value')
```

`xdoctest.runner._auto_disable_failing_tests_hook(context) → None`

Experimental feature to modify code based on failing tests. This should likely be moved to its own submodule.

`xdoctest.runner._convert_to_test_module(enabled_examples: list[DocTest]) → str`

Logic for the “dumps” command.

Converts all doctests to unit tests that can exist in a standalone module

`xdoctest.runner.undefined_names(sourcecode: str) → set[str]`

Parses source code for undefined names

Parameters

sourcecode (*str*) – code to check for unused names

Returns

the unused variable names

Return type

Set[str]

Example

```
>>> # xdoctest: +REQUIRES(module:pyflakes)
>>> print(undefined_names('x = y'))
{'y'}
```

`xdoctest.runner._print_summary_report(run_summary, parse_warnlist, n_seconds, enabled_examples, durations, config=None, _log=None) → None`

Summary report formatting and printing

`xdoctest.runner._gather_zero_arg_examples(modpath)`

Find functions in *modpath* args with no args (so we can automatically make a dummy docstring).

`xdoctest.runner._run_examples(enabled_examples, verbose, config=None, _log=None)`

Internal helper, loops over each example, runs it, returns a summary

`xdoctest.runner._parse_commandline(command=None, style='auto', verbose=None, argv=None)`

`xdoctest.runner._update_argparse_cli(add_argument, prefix=None) → None`

Update the CLI with arguments that control how doctests are collected and how aggregate results are reported.

4.2.16 xdoctest.static_analysis module

The core logic that allows for xdoctest to parse source statically

class `xdoctest.static_analysis.CallDefNode`(*callname: str, lineno: int | None, docstr: str | None, doclineneno: int | None, doclineneno_end: int | None, args: arguments | None = None*)

Bases: `object`

Variables

lineno_end (*None | int*) – the line number the docstring ends on (if known)

Parameters

- **callname** (*str*) – the name of the item containing the docstring.
- **lineno** (*int*) – the line number the item containing the docstring.
- **docstr** (*str*) – the docstring itself
- **doclineneno** (*int*) – the line number (1 based) the docstring begins on
- **doclineneno_end** (*int*) – the line number (1 based) the docstring ends on
- **args** (*None | ast.arguments*) – arguments from static analysis `TopLevelVisitor`.

class `xdoctest.static_analysis.TopLevelVisitor`(*source: str | None = None*)

Bases: `NodeVisitor`

Parses top-level function names and docstrings

For other `visit_<classname>` values see [\[MeetTheNodes\]](#).

References

CommandLine

```
python -m xdoctest.static_analysis TopLevelVisitor
```

Variables

- **calldefs** (*OrderedDict*)
- **source** (*None | str*)
- **sourcelines** (*None | List[str]*)
- **assignments** (*list*)

Example

```

>>> from xdoctest.static_analysis import * # NOQA
>>> from xdoctest import utils
>>> source = utils.codeblock(
    '''
        def foo():
            """ my docstring """
            def subfunc():
                pass
        def bar():
            pass
        class Spam:
            def eggs(self):
                pass
            @staticmethod
            def hams():
                pass
            @property
            def jams(self):
                return 3
            @jams.setter
            def jams2(self, x):
                print('ignoring')
            @jams.deleter
            def jams(self, x):
                print('ignoring')
    ''')
>>> self = TopLevelVisitor.parse(source)
>>> callnames = set(self.calldefs.keys())
>>> assert callnames == {
>>>     'foo', 'bar', 'Spam', 'Spam.eggs', 'Spam.hams',
>>>     'Spam.jams'}
>>> assert self.calldefs['foo'].docstr.strip() == 'my docstring'
>>> assert 'subfunc' not in self.calldefs

```

Parameters

source (*None* | *str*)

classmethod parse(*source: str*) → *TopLevelVisitor*

main entry point

executes parsing algorithm and populates self.calldefs

Parameters

source (*str*)

syntax_tree() → *AST*

creates the abstract syntax tree

Return type

ast.Module

process_finished(*node: AST* | *int*) → *None*

process (get ending lineno) for everything marked as finished

Parameters

node (*ast.AST*)

visit(*node: AST*) → None

Parameters

node (*ast.AST*)

_visit_generic_FunctionDef(*node: FunctionDef | AsyncFunctionDef*) → None

visit_FunctionDef(*node: FunctionDef*) → None

Parameters

node (*ast.FunctionDef*)

visit_AsyncFunctionDef(*node: AsyncFunctionDef*) → None

Parameters

node (*ast.AsyncFunctionDef*)

visit_ClassDef(*node: ClassDef*) → None

Parameters

node (*ast.ClassDef*)

visit_Module(*node: Module*) → None

Parameters

node (*ast.Module*)

visit_Assign(*node: Assign*) → None

Parameters

node (*ast.Assign*)

visit_If(*node: If*) → None

Parameters

node (*ast.If*)

_docnode_line_workaround(*docnode: Expr | stmt*) → tuple[int, int]

Find the start and ending line numbers of a docstring

Parameters

docnode (*ast.Expr*)

Returns

tuple[int, int]

CommandLine

```
xdoctest -m xdoctest.static_analysis TopLevelVisitor._docnode_line_workaround
```

Example

```
>>> from xdoctest.static_analysis import * # NOQA
>>> sq = chr(39) # single quote
>>> dq = chr(34) # double quote
>>> source = utils.codeblock(
    '''
```

(continues on next page)

(continued from previous page)

```

def func0():
    {ddd} docstr0 {ddd}
def func1():
    {ddd}
    docstr1 {ddd}
def func2():
    {ddd} docstr2
    {ddd}
def func3():
    {ddd}
    docstr3
    {ddd} # foobar
def func5():
    {ddd}pathological case
    {sss} # {ddd} # {sss} # {ddd} # {ddd}
def func6():
    " single quoted docstr "
def func7():
    r{ddd}
    raw line
    {ddd}
'''').format(ddd=dq * 3, sss=sq * 3)
>>> self = TopLevelVisitor(source)
>>> func_nodes = self.syntax_tree().body
>>> print(utils.add_line_numbers(utils.highlight_code(source), start=1))
>>> wants = [
>>>     (2, 2),
>>>     (4, 5),
>>>     (7, 8),
>>>     (10, 12),
>>>     (14, 15),
>>>     (17, 17),
>>>     (19, 21),
>>> ]
>>> for i, func_node in enumerate(func_nodes):
>>>     docnode = func_node.body[0]
>>>     got = self._docnode_line_workaround(docnode)
>>>     want = wants[i]
>>>     print('got = {!r}'.format(got))
>>>     print('want = {!r}'.format(want))
>>>     assert got == want

```

classmethod `_find_docstr_endpos_workaround`(*docstr: str, sourcelines: list[str], startpos: int*) → *tuple[int, int]*

Like `docstr_line_workaround`, but works from the top-down instead of bottom-up. This is for pypy.

Given a docstring, its original source lines, and where the start position is, this function finds the end-position of the docstr

Example

```

>>> fmtkw = dict(sss=chr(39) * 3, ddd=chr(34) * 3)
>>> source = utils.codeblock(
    '''
        {ddd}
        docstr0
        {ddd}
    '''.format(**fmtkw))
>>> sourcelines = source.splitlines()
>>> docstr = eval(source, {}, {})
>>> startpos = 0
>>> start, stop = TopLevelVisitor._find_docstr_endpos_workaround(docstr,
↳sourcelines, startpos)
>>> assert (start, stop) == (0, 2)
>>> #
>>> source = utils.codeblock(
    '''
        "docstr0"
    '''.format(**fmtkw))
>>> sourcelines = source.splitlines()
>>> docstr = eval(source, {}, {})
>>> startpos = 0
>>> start, stop = TopLevelVisitor._find_docstr_endpos_workaround(docstr,
↳sourcelines, startpos)
>>> assert (start, stop) == (0, 0)

```

`_find_docstr_startpos_workaround`(*docstr*: *str*, *sourcelines*: *list[str]*, *endpos*: *int*) → *tuple[int, int]*

Find the which sourcelines contain the docstring

Parameters

- **docstr** (*str*) – the extracted docstring.
- **sourcelines** (*list*) – a list of all lines in the file. We assume the docstring exists as a pure string literal in the source. In other words, no postprocessing via `split`, `format`, or any other dynamic programmatic modification should be made to the docstrings. Python's docstring extractor assumes this as well.
- **endpos** (*int*) – line position (starting at 0) the docstring ends on. Note: positions are 0 based but `linenos` are 1 based.

Returns**start, stop:**

start: the line position (0 based) the docstring starts on stop: the line position (0 based) that the docstring stops

such that `sourcelines[start:stop]` will contain the docstring

Return type

`tuple[int, int]`

CommandLine

```
python -m xdoctest xdoctest.static_analysis TopLevelVisitor._find_docstr_
↳startpos_workaround
python -m xdoctest xdoctest.static_analysis TopLevelVisitor._find_docstr_
↳startpos_workaround --debug
```

Example

```
>>> # xdoctest: +REQUIRES(CPython)
>>> # This function is a specific workaround for a CPython bug.
>>> from xdoctest.static_analysis import *
>>> sq = chr(39) # single quote
>>> dq = chr(34) # double quote
>>> source = utils.codeblock(
'''
def func0():
    {ddd} docstr0 {ddd}
def func1():
    {ddd}
    docstr1 {ddd}
def func2():
    {ddd} docstr2
    {ddd}
def func3():
    {ddd}
    docstr3
    {ddd} # foobar
def func5():
    {ddd}pathological case
    {sss} # {ddd} # {sss} # {ddd} # {ddd}
def func6():
    " single quoted docstr "
def func7():
    r{ddd}
    raw line
    {ddd}
''').format(ddd=dq * 3, sss=sq * 3)
>>> print(utils.add_line_numbers(utils.highlight_code(source), start=0))
>>> targets = [
>>>     (1, 2),
>>>     (3, 5),
>>>     (6, 8),
>>>     (9, 12),
>>>     (13, 15),
>>>     (16, 17),
>>>     (18, 21),
>>> ]
>>> self = TopLevelVisitor.parse(source)
>>> pt = ast.parse(source.encode('utf8'))
>>> sourcelines = source.splitlines()
>>> # PYPY docnode.lineno specify the startpos of a docstring not
>>> # the end.
```

(continues on next page)

(continued from previous page)

```

>>> print('\n\n====\n\n')
>>> #for i in [0, 1]:
>>> for i in range(len(targets)):
>>>     print('-----')
>>>     funcnode = pt.body[i]
>>>     print('funcnode = {!r}'.format(funcnode))
>>>     docnode = funcnode.body[0]
>>>     print('funcnode.__dict__ = {!r}'.format(funcnode.__dict__))
>>>     print('docnode = {!r}'.format(docnode))
>>>     print('docnode.value = {!r}'.format(docnode.value))
>>>     print('docnode.value.__dict__ = {!r}'.format(docnode.value.__dict__))
>>>     if IS_PY_GE_312:
>>>         print('docnode.value.value = {!r}'.format(docnode.value.value))
>>>     else:
>>>         print('docnode.value.s = {!r}'.format(docnode.value.s))
>>>         print('docnode.lineno = {!r}'.format(docnode.lineno))
>>>         print('docnode.col_offset = {!r}'.format(docnode.col_offset))
>>>         print('docnode = {!r}'.format(docnode))
>>>         #import IPython
>>>         #IPython.embed()
>>>         docstr = ast.get_docstring(funcnode, clean=False)
>>>         print('len(docstr) = {}'.format(len(docstr)))
>>>         endpos = docnode.lineno - 1
>>>         if hasattr(docnode, 'end_lineno'):
>>>             endpos = docnode.end_lineno - 1
>>>         print('endpos = {}'.format(endpos))
>>>         start, end = self._find_docstr_startpos_workaround(docstr, sourcelines,
└endpos)
>>>         print('i = {}'.format(i))
>>>         print('got = {}, {}'.format(start, end))
>>>         print('want = {}, {}'.format(*targets[i]))
>>>         if targets[i] != (start, end):
>>>             print('---')
>>>             print(docstr)
>>>             print('---')
>>>             print('sourcelines = [\n{}\n]'.format(', \n'.join(list(map(repr,
└enumerate(sourcelines))))))
>>>             print('endpos = {}'.format(endpos))
>>>             raise AssertionError('docstr workaround is failing')
>>>         print('-----')

```

`_get_docstring`(*node*: *AsyncFunctionDef* | *FunctionDef* | *ClassDef* | *Module*) → tuple[str | None, int | None, int | None]

CommandLine

```
xdoctest -m xdoctest.static_analysis.py TopLevelVisitor._get_docstring
```

Example

```
>>> source = utils.codeblock(
'''
def foo():
    'docstr'
''')
>>> self = TopLevelVisitor(source)
>>> node = self.syntax_tree().body[0]
>>> self._get_docstring(node)
('docstr', 2, 2)
```

`xdoctest.static_analysis.parse_static_calldefs`(*source*: *str* | *None* = *None*, *fpath*: *str* | *PathLike* | *None* = *None*) → *dict*[*str*, *CallDefNode*]

Statically finds top-level callable functions and methods in python source

Parameters

- **source** (*str*) – python text
- **fpath** (*str*) – filepath to read if source is not specified

Returns

mapping from callnames to **CallDefNodes**, which contain info about the item with the doctest.

Return type

Dict[*str*, *CallDefNode*]

Example

```
>>> from xdoctest import static_analysis
>>> fpath = static_analysis.__file__.replace('.pyc', '.py')
>>> calldefs = parse_static_calldefs(fpath=fpath)
>>> assert 'parse_static_calldefs' in calldefs
```

`xdoctest.static_analysis.parse_calldefs`(*source*: *str* | *None* = *None*, *fpath*: *str* | *PathLike* | *None* = *None*) → *dict*[*str*, *CallDefNode*]

`xdoctest.static_analysis._parse_static_node_value`(*node*: *AST*) → *Any*

Extract a constant value from a node if possible

`xdoctest.static_analysis.parse_static_value`(*key*: *str*, *source*: *str* | *bytes* | *None* = *None*, *fpath*: *str* | *None* = *None*) → *object*

Statically parse a constant variable's value from python code.

TODO: This does not belong here. Move this to an external static analysis library.

Parameters

- **key** (*str*) – name of the variable
- **source** (*str*) – python text
- **fpath** (*str*) – filepath to read if source is not specified

Returns

object

Example

```

>>> from xdoctest.static_analysis import parse_static_value
>>> key = 'foo'
>>> source = 'foo = 123'
>>> assert parse_static_value(key, source=source) == 123
>>> source = 'foo = "123"'
>>> assert parse_static_value(key, source=source) == '123'
>>> source = 'foo = [1, 2, 3]'
>>> assert parse_static_value(key, source=source) == [1, 2, 3]
>>> source = 'foo = (1, 2, "3")'
>>> assert parse_static_value(key, source=source) == (1, 2, "3")
>>> source = 'foo = {1: 2, 3: 4}'
>>> assert parse_static_value(key, source=source) == {1: 2, 3: 4}
>>> source = 'foo = None'
>>> assert parse_static_value(key, source=source) == None
>>> #parse_static_value('bar', source=source)
>>> #parse_static_value('bar', source='foo=1; bar = [1, foo]')

```

`xdoctest.static_analysis.package_modpaths`(*pkgpath*: *str*, *with_pkg*: *bool* = *False*, *with_mod*: *bool* = *True*, *followlinks*: *bool* = *True*, *recursive*: *bool* = *True*, *with_libs*: *bool* = *False*, *check*: *bool* = *True*) → *Iterator*[*str*]

Finds sub-packages and sub-modules belonging to a package.

Parameters

- **pkgpath** (*str*) – path to a module or package
- **with_pkg** (*bool*) – if *True* includes package `__init__` files (default = *False*)
- **with_mod** (*bool*) – if *True* includes module files (default = *True*)
- **exclude** (*list*) – ignores any module that matches any of these patterns
- **recursive** (*bool*) – if *False*, then only child modules are included
- **with_libs** (*bool*) – if *True* then compiled shared libs will be returned as well
- **check** (*bool*) – if *False*, then then *pkgpath* is considered a module even if it does not contain an `__init__` file.

Yields

str – module names belonging to the package

References

<http://stackoverflow.com/questions/1707709/list-modules-in-py-package>

Example

```

>>> from xdoctest.static_analysis import *
>>> pkgpath = modname_to_modpath('xdoctest')
>>> paths = list(package_modpaths(pkgpath))
>>> print('\n'.join(paths))
>>> names = list(map(modpath_to_modname, paths))
>>> assert 'xdoctest.core' in names
>>> assert 'xdoctest.__main__' in names

```

(continues on next page)

(continued from previous page)

```
>>> assert 'xdoctest' not in names
>>> print('\n'.join(names))
```

`xdoctest.static_analysis.is_balanced_statement`(*lines: list[str]*, *only_tokens: bool = False*, *reraise: int = 0*) → bool

Checks if the lines have balanced braces and quotes.

Parameters

lines (*List[str]*) – list of strings, one for each line

Returns

True if the statement is balanced, otherwise False

Return type

bool

CommandLine

```
xdoctest -m xdoctest.static_analysis is_balanced_statement:0
```

References

<https://stackoverflow.com/questions/46061949/parse-until-complete>

Example

```
>>> from xdoctest.static_analysis import * # NOQA
>>> assert is_balanced_statement(['print(foobar)'])
>>> assert is_balanced_statement(['foo = bar']) is True
>>> assert is_balanced_statement(['foo = (') is False
>>> assert is_balanced_statement(['foo = (, ''(')']) is True
>>> assert is_balanced_statement(
...     ['foo = (, ''''', ')']''''', ')']) is True
>>> assert is_balanced_statement(
...     ['foo = ', ''''', ')']''''', ')']) is False
>>> #assert is_balanced_statement(['foo = '] is False
>>> #assert is_balanced_statement(['== '] is False
>>> lines = ['def foo():', ' ', '    x = 1', 'assert True', '']
>>> assert is_balanced_statement(lines)
```

Example

```
>>> from xdoctest.static_analysis import *
>>> source_parts = [
>>>     'setup(',
>>>     "    name='extension',",
>>>     '    ext_modules=[',
>>>     '        CppExtension(',
>>>     "            name='extension',",
>>>     "            sources=['extension.cpp'],",
>>>     "            extra_compile_args=['-g'])",
>>>     '    ],',
```

(continues on next page)

(continued from previous page)

```

>>> ]
>>> print('\n'.join(source_parts))
>>> assert not is_balanced_statement(source_parts)
>>> source_parts = [
>>>     'setup(',
>>>     "     name='extension',"
>>>     '     ext_modules=[',
>>>     '         CppExtension(',
>>>     "         name='extension',"
>>>     "         sources=['extension.cpp',"
>>>     "         extra_compile_args=['-g'])",
>>>     '     ],',
>>>     '     cmdclass={',
>>>     "         'build_ext': BuildExtension",
>>>     '     }')',
>>> ]
>>> print('\n'.join(source_parts))
>>> assert is_balanced_statement(source_parts)

```

Example

```

>>> lines = ['try: raise Exception']
>>> is_balanced_statement(lines, only_tokens=1)
True
>>> is_balanced_statement(lines, only_tokens=0)
False

```

Example

```

>>> # Cause a failure case on 3.12
>>> from xdoctest.static_analysis import *
>>> lines = ['3, 4]', 'print(len(x))']
>>> is_balanced_statement(lines, only_tokens=1)
False

```

`xdoctest.static_analysis.extract_comments`(*source: str | list[str]*) → `Iterator[str]`

Returns the text in each comment in a block of python code. Uses `tokenize` to account for quotations.

Parameters

source (*str | List[str]*)

CommandLine

```
python -m xdoctest.static_analysis extract_comments
```

Example

```

>>> from xdoctest import utils
>>> source = utils.codeblock(
>>>     '''
>>>     # comment 1

```

(continues on next page)

(continued from previous page)

```

    a = '# not a comment' # comment 2
    c = 3
    '''
>>> comments = list(extract_comments(source))
>>> assert comments == ['# comment 1', '# comment 2']
>>> comments = list(extract_comments(source.splitlines()))
>>> assert comments == ['# comment 1', '# comment 2']

```

`xdoctest.static_analysis._strip_hashtag_comments_and_newlines`(*source: str | list[str]*) → *str*

Removes hashtag comments from underlying source

Parameters

source (*str | List[str]*)

CommandLine

```
xdoctest -m xdoctest.static_analysis _strip_hashtag_comments_and_newlines
```

Todo

would be better if this was some sort of configurable minify API

Example

```

>>> from xdoctest.static_analysis import _strip_hashtag_comments_and_newlines
>>> from xdoctest import utils
>>> fmtkw = dict(sss=chr(39) * 3, ddd=chr(34) * 3)
>>> source = utils.codeblock(
>>>     '''
>>>     # comment 1
>>>     a = '# not a comment' # comment 2

```

```

multiline_string = {ddd}
one
{ddd} b = [
    1, # foo
    # bar 3,
] c = 3 """.format(**fmtkw)

```

```

>>> non_comments = _strip_hashtag_comments_and_newlines(source)
>>> print(non_comments)
>>> assert non_comments.count(chr(10)) == 10
>>> assert non_comments.count('#') == 1

```

4.3 Module contents



4.3.1 Xdoctest - Execute Doctests

Xdoctest is a Python package for executing tests in documentation strings!

What is a `doctest`? It is example code you write in a docstring! What is a `docstring`? Its a string you use as a comment! They get attached to Python functions and classes as metadata. They are often used to auto-generate documentation. Why is it cool? Because you can write tests while you code!

Xdoctest finds and executes your doctests for you. Just run `xdoctest <path-to-my-module>`. It plugs into `pytest` to make it easy to run on a CI. Install and run `pytest --xdoctest`.

The `xdoctest` package is a re-write of Python's builtin `doctest` module. It replaces the old regex-based parser with a new abstract-syntax-tree based parser (using Python's `ast` module). The goal is to make doctests easier to write, simpler to configure, and encourage the pattern of test driven development.

Read the docs	http://xdoctest.readthedocs.io/en/latest
Github	https://github.com/Erotemic/xdoctest
Pypi	https://pypi.org/project/xdoctest
PyCon 2020	Youtube Video and Google Slides

4.3.1.1 Getting Started 0: Installation

First ensure that you have Python installed and ideally are in a virtual environment. Install `xdoctest` using the `pip`.

```
pip install xdoctest
```

Alternatively you can install `xdoctest` with optional packages.

```
pip install xdoctest[all]
```

This ensures that the `pygments` and `colorama` packages are installed, which are required to color terminal output.

4.3.1.2 Getting Started 1: Your first doctest

If you already know how to write a doctest then you can skip to the next section. If you aren't familiar with doctests, this will help get you up to speed.

Consider the following implementation the Fibonacci function.

```
def fib(n):  
    """  
    Python 3: Fibonacci series up to n  
    """  
    a, b = 0, 1
```

(continues on next page)

(continued from previous page)

```

while a < n:
    print(a, end=' ')
    a, b = b, a+b
print()

```

We can add a “doctest” in the “docstring” as both an example and a test of the code. All we have to do is prefix the doctest code with three right chevrons ```>>>```. We can also use xdoctest directives to control the flow of doctest execution.

```

def fib(n):
    """
    Python 3: Fibonacci series up to n

    Example:
    >>> fib(1000) # xdoctest: +SKIP
    0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
    """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

```

Now if this text was in a file called `fib.py` you could execute your doctest by running `xdoctest fib.py`. Note that if `fib.py` was in a package called `mymod`, you could equivalently run `xdoctest -m mymod.fib`. In other words you can all doctests in a file by passing xdoctest the module name or the module path.

Interestingly because this documentation is written in the `xdoctest/__init__.py` file, which is a Python file, that means we can write doctests in it. If you have xdoctest installed, you can use the xdoctest cli to execute the following code: `xdoctest -m xdoctest.__init__ __doc__:0`. Also notice that the previous doctest is skipped due to the SKIP directive. For more information on directives see the docs for the xdoctest directive module.

```

>>> # Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(25)
0 1 1 2 3 5 8 13 21

```

4.3.1.3 Getting Started 2: Running your doctests

There are two ways to run xdoctest: (1) `pytest` or (2) the native `xdoctest` interface. The native interface is less opaque and implicit, but its purpose is to run doctests. The other option is to use the widely used `pytest` package. This allows you to run both unit tests and doctests with the same command and has many other advantages.

It is recommended to use `pytest` for automatic testing (e.g. in your CI scripts), but for debugging it may be easier to use the native interface.

4.3.1.3.1 Using the pytest interface

When `pytest` is run, `xdoctest` is automatically discovered, but is disabled by default. This is because `xdoctest` needs to replace the builtin `pytest._pytest.doctest` plugin.

To enable this plugin, run `pytest` with `--xdoctest` or `--xdoc`. This can either be specified on the command line or added to your `addopts` options in the `[pytest]` section of your `pytest.ini` or `tox.ini`.

To run a specific doctest, `xdoctest` sets up `pytest` node names for these doctests using the following pattern: `<path/to/file.py>::<callname>:<num>`. For example a doctest for a function might look like this `mymod.py::funcname:0`, and a class method might look like this: `mymod.py::ClassName::method:0`

4.3.1.3.2 Using the native interface.

The `xdoctest` module contains a `pytest` plugin, but also contains a native command line interface (CLI). The CLI is generated using `argparse`.

For help you can run

```
xdoctest --help
```

which produces something similar to the following output:

```
usage: xdoctest [-h] [--version] [-m MODNAME] [-c COMMAND] [--style {auto,google,
↪freeform}] [--analysis {auto,static,dynamic}] [--durations DURATIONS] [--time]
           [--colored COLORED] [--nocolor] [--offset] [--report {none,cdiff,ndiff,
↪udiff,only_first_failure}] [--options OPTIONS] [--global-exec GLOBAL_EXEC]
           [--verbose VERBOSE] [--quiet] [--silent]
           [arg ...]

Xdoctest 1.0.0 - on Python - 3.9.9 (main, Jan 6 2022, 18:33:12)
[GCC 10.3.0] - discover and run doctests within a python package

positional arguments:
  arg                    Ignored if optional arguments are specified, otherwise: Defaults_
↪--modname to arg.pop(0). Defaults --command to arg.pop(0). (default: None)

optional arguments:
  -h, --help            show this help message and exit
  --version             Display version info and quit (default: False)
  -m MODNAME, --modname MODNAME
                        Module name or path. If specified positional modules are ignored_
↪(default: None)
  -c COMMAND, --command COMMAND
                        A doctest name or a command (list|all|<callname>). Defaults to_
↪all (default: None)
  --style {auto,google,freeform}
                        Choose the style of doctests that will be parsed (default: auto)
  --analysis {auto,static,dynamic}
                        How doctests are collected (default: auto)
  --durations DURATIONS
                        Specify execution times for slowest N tests.N=0 will show times_
↪for all tests (default: None)
  --time               Same as if durations=0 (default: False)
  --colored COLORED   Enable or disable ANSI coloration in stdout (default: True)
```

(continues on next page)

(continued from previous page)

```

--nocolor          Disable ANSI coloration in stdout
--offset           If True formatted source line numbers will agree with their
↳ location in the source file. Otherwise they will be relative to the doctest itself.
↳ (default:
                    False)
--report {none,cdiff,ndiff,udiff,only_first_failure}
                    Choose another output format for diffs on xdoctest failure.
↳ (default: udiff)
--options OPTIONS  Default directive flags for doctests (default: None)
--global-exec GLOBAL_EXEC
                    Custom Python code to execute before every test (default: None)
--verbose VERBOSE  Verbosity level. 0 is silent, 1 prints out test names, 2
↳ additionally prints test stdout, 3 additionally prints test source (default: 3)
--quiet           sets verbosity to 1
--silent          sets verbosity to 0

```

The xdoctest interface can be run programmatically using `xdoctest.doctest_module(path)`, which can be placed in the `__main__` section of any module as such:

```

if __name__ == '__main__':
    import xdoctest
    xdoctest.doctest_module(__file__)

```

This sets up the ability to invoke the xdoctest command line interface by invoking your module as a `main script`: `python -m <modname> <command>`, where `<modname>` is the name of your module (e.g. `foo.bar`) and `command` follows the following rules:

- If `<command>` is `all`, then each enabled doctest in the module is executed: `python -m <modname> all`
- If `<command>` is `list`, then the names of each enabled doctest is listed.
- If `<command>` is `dump`, then all doctests are converted into a format suitable for unit testing, and dumped to stdout (new in 0.4.0).
- If `<command>` is a “callname” (name of a function or a class and method), then that specific doctest is executed: `python -m <modname> <callname>`. Note: you can execute disabled doctests or functions without any arguments (zero-args) this way.

XDoctest is a good demonstration of itself. After pip installing xdoctest, try running xdoctest on xdoctest.

```
xdoctest xdoctest
```

If you would like a slightly less verbose output, try

```

xdoctest xdoctest --verbose=1
# or
xdoctest xdoctest --verbose=0

```

You could also consider running xdoctests tests through pytest:

```

pytest $(python -c 'import xdoctest, pathlib; print(pathlib.Path(xdoctest.__file__).
↳ parent)') --xdoctest

```

If you would like a slightly more verbose output, try

```
pytest -s --verbose --xdoctest-verbose=3 --xdoctest $(python -c 'import xdoctest,
↳ pathlib; print(pathlib.Path(xdoctest.__file__).parent)')
```

If you ran these commands, the myriad of characters that flew across your screen are lots more examples of what you can do with doctests.

You can also run doctests *inside Jupyter Notebooks*.

exception `xdoctest.DoctestParseError`(*msg: str, string: str | None = None, info: Any | None = None, orig_ex: Exception | None = None*)

Bases: `Exception`

Exception raised when doctest code has an error.

Parameters

- **msg** (*str*) – error message
- **string** (*str | None*) – the string that failed
- **info** (*Any | None*) – extra information
- **orig_ex** (*Exception | None*) – The underlying exception

exception `xdoctest.ExitTestException`

Bases: `Exception`

exception `xdoctest.MalformedDocstr`

Bases: `Exception`

Exception raised when the docstring itself does not conform to the expected style (e.g. google / numpy).

exception `xdoctest.ExistingEventLoopError`

Bases: `Exception`

Exception raised when the docstring uses a top level await and the test is already running in an event loop.

`xdoctest.doctest_module`(*module_identifier: str | ModuleType | None = None, command: str | None = None, argv: list[str] | None = None, exclude: list[str] = [], style: str = 'auto', verbose: int | None = None, config: dict[str, Any] | None = None, durations: int | None = None, analysis: str = 'auto') → dict[str, Any]*

Executes requested google-style doctests in a package or module. Main entry point into the testing framework.

Parameters

- **module_identifier** (*str | ModuleType | None*) – The name of / path to the module, or the live module itself. If not specified, dynamic analysis will be used to introspect the module that called this function and that module will be used. This can also contain the callname followed by the `::` token.
- **command** (*str*) – determines which doctests to run. if command is None, this is determined by parsing `sys.argv`. Value values are ‘all’ - find and run all tests in a module ‘list’ - list the tests in a module ‘dump’ - dumps tests to stdout
- **argv** (*List[str] | None*) – if specified, command line flags that might influence behavior. if None uses `sys.argv`. SeeAlso `:func:_update_argparse_cli` SeeAlso `:func:doctest_example.DoctestConfig._update_argparse_cli`
- **style** (*str*) – Determines how doctests are recognized and grouped. Can be freeform, google, or auto.

- **verbose** (*int | None*) – Verbosity level. 0 - disables all text 1 - minimal printing 3 - verbose printing
- **exclude** (*List[str]*) – ignores any modname matching any of these glob-like patterns
- **config** (*Dict[str, object]*) – modifies each examples configuration. Defaults and expected keys are documented in `xdoctest.doctest_example.DoctestConfig`
- **durations** (*int | None*) – if specified report top N slowest tests
- **analysis** (*str*) – determines if doctests are found using static or dynamic analysis.

Returns

run_summary

Return type

Dict[str, Any]

Example

```
>>> modname = 'xdoctest.dynamic_analysis'
>>> result = doctest_module(modname, 'list', argv=[''])
```

Example

```
>>> # xdoctest: +SKIP
>>> # Demonstrate different ways "module_identifier" can be specified
>>> #
>>> # Using a module name
>>> result = doctest_module('xdoctest.static_analysis')
>>> #
>>> # Using a module path
>>> result = doctest_module(os.expandpath('~/.code/xdoctest/src/xdoctest/static_
↪analysis.py'))
>>> #
>>> # Using a module itself
>>> from xdoctest import runner
>>> result = doctest_module(runner)
>>> #
>>> # Using a module name and a specific callname
>>> from xdoctest import runner
>>> result = doctest_module('xdoctest.static_analysis::parse_static_value')
```

`xdoctest.doctest_callable(func: Callable[[...], Any]) → None`

Executes doctests an in-memory function or class.

Parameters

func (*callable*) – live method or class for which we will run its doctests.

Example

```
>>> def inception(text):
>>>     """
>>>     Example:
>>>         >>> inception("I heard you liked doctests")
>>>     """
```

(continues on next page)

(continued from previous page)

```
>>> print(text)
>>> func = inception
>>> doctest_callable(func)
```

RUNNING DOCTESTS IN JUPYTER NOTEBOOKS

You can run doctests within a Jupyter notebook in two ways:

5.1 Method 1 - Inside the notebook

Either insert this cell into your notebook:

```
if __name__ == '__main__':  
    import xdoctest  
    xdoctest.doctest_module()
```

This will execute any doctests for callables that are in the top-level namespace of the notebook. While you don't have to include the `if __name__` block, it is better practice because it will prevent issues if you also wish to use "Method 2".

5.2 Method 2 - Outside the notebook

An alternative way to run would be using the `xdoctest` command line tool and pointing to the notebook file.

```
xdoctest path/to/notebook.ipynb
```

This will execute *every* cell in the notebook and then execute the doctest of any defined callable with a doctest.

5.3 Caveats

WARNING: in both of the above methods, when you execute doctests it will include any function / class that was defined in the notebook, but also *any external library callable with a doctest that you import directly!* Therefore it is best to (1) never use `from <module> import *` statements (in general using `import *` is bad practice) and (2) prefer using functions via their module name rather than importing directory. For example instead of `from numpy import array; x = array([1])` use `import numpy as np; x = np.array([1])`.

Lastly, it is important to note that Jupyter notebooks are great for prototyping and exploration, but in practice storing algorithm and utilities in Jupyter notebooks is not sustainable (for some of these reasons). Reusable code should eventually be refactored into a proper pip-installable Python package where the top level directory contains a `setup.py` and a folder with a name corresponding to the module name and containing an `__init__.py` file and any other package python files. However, if you write your original Jupyter code with doctests, then when you port your code to a proper package the automated tests come with it! (And the above warning does *not* apply to statically parsed python packages)

DOCTESTS WITH ASYNC CODE

Python 3.5 introduced *async* functions. These are functions that run within an “event loop” that is not blocked if those functions perform blocking IO. It’s similar to writing multi-threaded code but with the advantage of not having to worry about thread safety. For more information see [the python docs](#).

Asynchronous python code examples using `asyncio` are supported at the top level by `xdoctest`. This means that your code examples do not have to wrap every small snippet in a function and call `asyncio.run()`. `xdoctest` handles that for you keeping the examples simple and easy to follow.

For example **without** `xdoctest` your code example would have to be written like this:

```
>>> import yourlibrary
>>> import asyncio
>>> async def connect_and_get_running_info_wrapper():
...     server = await yourlibrary.connect_to_server("example_server_url")
...     running_info = await server.get_running_info()
...     return server, running_info
...
>>> server, running_info = asyncio.run(connect_and_get_running_info_wrapper())
>>> running_info.restarted_at
01:00

>>> async def restart_and_get_running_info_wrapper(server):
...     await server.restart()
...     return await server.get_running_info()
...
>>> running_info = asyncio.run(restart_and_get_running_info_wrapper(server))
>>> running_info.restarted_at
13:15
```

Now **with** `xdoctest` this can now be written like this:

```
>>> import yourlibrary
>>> server = await yourlibrary.connect_to_server("example_server_url")
>>> running_info = await server.get_running_info()
>>> running_info.restarted_at
01:00

>>> await server.restart()
>>> running_info = await server.get_running_info()
>>> running_info.restarted_at
13:15
```

The improvement in brevity is obvious but even more so if you are writing longer examples where you want to maintain and reuse variables between each test output step.

Note

If you don't want to utilise this feature for your async examples you don't have to. Just don't write code examples with top level awaits.

Caution

Each code block with top level awaits runs in its own `asyncio.run()`. This means that all tasks created in such a block will be cancelled when it finishes. If you want more asyncio REPL-like behavior, see the next section.

6.1 ASYNC Directive

By default, xdoctest separates code blocks with top level awaits and blocks without. The former will have a running asyncio event loop, but the latter will not. This can be undesirable when you need to multitask or use the same event loop for all blocks.

```
>>> import yourlibrary
>>> import asyncio
>>> task = asyncio.create_task(yourlibrary.send_message()) # fails!
>>> # ...do something else...
>>> result = await task # never be reached
```

To solve this problem, since 1.3.0 xdoctest has a new basic directive, ASYNC. Just enable the directive at the beginning of your code example and you will get the asyncio REPL behavior.

```
>>> # xdoctest: +ASYNC
>>> import yourlibrary
>>> import asyncio
>>> task = asyncio.create_task(yourlibrary.send_message()) # ok
>>> # ...do something else...
>>> result = await task # will be reached
```

Of course, you can use the directive to cover certain places in your code too. With this you can demonstrate the behavior of your functions both inside and outside `asyncio.run()` in a single example.

```
>>> import yourlibrary
>>> yourlibrary.in_async_context() # xdoctest: +ASYNC
True
>>> yourlibrary.in_async_context() # xdoctest: -ASYNC
False
```

You may also find it convenient to enable the directive for all tests by default, in order to avoid boilerplate. For this, as with any other basic directive, you can use `--options ASYNC` for the native interface and `--xdoctest-options ASYNC` for the pytest interface.

6.2 Caveats

- Consumers reading your documentation may not be familiar with async concepts. It could be helpful to mention in your docs that the code examples should be run in an event loop or in a REPL that supports top-level `await`. (IPython supports this by default. For the standard Python REPL, use `python -m asyncio`.)
- Using top level awaits in tests that are already running in an event loop is not supported.
- Only python's native asyncio library is supported for top level awaits.

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

- [GoogleStyleDocs1] https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html#example-google
- [GoogleStyleDocs2] http://www.sphinx-doc.org/en/stable/ext/example_google.html#example-google
- [SO_67631] <https://stackoverflow.com/questions/67631/import-module-given-path>
- [SO_67631] <https://stackoverflow.com/questions/67631/import-module-given-path>
- [SE32096] <https://unix.stackexchange.com/questions/32096/why-is-bashs-prompt-variable-called-ps1>
- [BashPS1] <https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#index-PS1>
- [CustomPrompt] https://wiki.archlinux.org/title/Bash/Prompt_customization
- [GeekPrompt] https://web.archive.org/web/20230824025647/https://www.thegeekstuff.com/2008/09/bash-shell-take-control-of-ps1-ps2-ps3-ps4-and-prompt_command/
- [MeetTheNodes] <http://greentreesnakes.readthedocs.io/en/latest/nodes.html>

PYTHON MODULE INDEX

X

- xdoctest, 118
- xdoctest.__init__, 1
- xdoctest.__main__, 57
- xdoctest._tokenize, 58
- xdoctest.checker, 59
- xdoctest.constants, 63
- xdoctest.core, 64
- xdoctest.demo, 69
- xdoctest.directive, 71
- xdoctest.docstr, 16
- xdoctest.docstr.docscrape_google, 11
- xdoctest.docstr.docscrape_numpy, 16
- xdoctest.doctest_example, 81
- xdoctest.doctest_part, 90
- xdoctest.dynamic_analysis, 93
- xdoctest.exceptions, 95
- xdoctest.global_state, 96
- xdoctest.parser, 96
- xdoctest.runner, 103
- xdoctest.static_analysis, 106
- xdoctest.utils, 43
- xdoctest.utils.util_deprecation, 21
- xdoctest.utils.util_import, 22
- xdoctest.utils.util_misc, 33
- xdoctest.utils.util_mixins, 33
- xdoctest.utils.util_notebook, 34
- xdoctest.utils.util_path, 37
- xdoctest.utils.util_str, 37
- xdoctest.utils.util_stream, 41

Symbols

- `_NOT_EVAL_TYPE` (class in `xdoctest.constants`), 63
- `__call__()` (`xdoctest.constants._NOT_EVAL_TYPE` method), 64
- `_asdict()` (`xdoctest.directive.Effect` method), 74
- `_asdict()` (`xdoctest.docstr.docscrape_google.DocBlock` method), 11
- `_auto_disable_failing_tests_hook()` (in module `xdoctest.runner`), 105
- `_block_prefix` (`xdoctest.doctest_example.DocTest` property), 88
- `_boolean_envIRON()` (in module `xdoctest.global_state`), 96
- `_check_match()` (in module `xdoctest.checker`), 60
- `_check_or_defer_part_output()` (`xdoctest.doctest_example.DocTest` method), 88
- `_color()` (`xdoctest.doctest_example.DocTest` method), 89
- `_complete_source()` (in module `xdoctest.parser`), 102
- `_convert_to_test_module()` (in module `xdoctest.runner`), 105
- `_custom_import_modpath()` (in module `xdoctest.utils.util_import`), 24
- `_do_a_fancy_diff()` (`xdoctest.checker.GotWantException` method), 62
- `_docnode_line_workaround()` (`xdoctest.static_analysis.TopLevelVisitor` method), 108
- `_ellipsis_match()` (in module `xdoctest.checker`), 61
- `_extension_module_tags()` (in module `xdoctest.utils.util_import`), 27
- `_extract_future_flags()` (`xdoctest.doctest_example.DocTest` static method), 87
- `_field_defaults` (`xdoctest.directive.Effect` attribute), 74
- `_field_defaults` (`xdoctest.docstr.docscrape_google.DocBlock` attribute), 11
- `_fields` (`xdoctest.directive.Effect` attribute), 74
- `_fields` (`xdoctest.docstr.docscrape_google.DocBlock` attribute), 11
- `_find_docstr_endpos_workaround()` (`xdoctest.static_analysis.TopLevelVisitor` class method), 109
- `_find_docstr_startpos_workaround()` (`xdoctest.static_analysis.TopLevelVisitor` method), 110
- `_find_notebook()` (in module `xdoctest.utils.util_notebook`), 34
- `_gather_zero_arg_examples()` (in module `xdoctest.runner`), 105
- `_get_docstring()` (`xdoctest.static_analysis.TopLevelVisitor` method), 112
- `_global_state` (`xdoctest.directive.RuntimeState` attribute), 76
- `_group_labeled_lines()` (`xdoctest.parser.DoctestParser` method), 99
- `_hasprefix()` (in module `xdoctest.parser`), 102
- `_import_module()` (`xdoctest.doctest_example.DocTest` method), 87
- `_importlib_import_modpath()` (in module `xdoctest.utils.util_import`), 23
- `_importlib_modname_to_modpath()` (in module `xdoctest.utils.util_import`), 23
- `inline_state` (`xdoctest.directive.RuntimeState` attribute), 76
- `_is_requires_satisfied()` (in module `xdoctest.directive`), 80
- `_iterthree()` (in module `xdoctest.parser`), 102
- `_label_docsrc_lines()` (`xdoctest.parser.DoctestParser` method), 101
- `_locate_ps1_linenos()` (`xdoctest.parser.DoctestParser` method), 100
- `_make()` (`xdoctest.directive.Effect` class method), 74
- `_make()` (`xdoctest.docstr.docscrape_google.DocBlock` class method), 11
- `_make_test_notebook_fpath()` (in module `xdoctest.utils.util_notebook`), 36
- `_min_indentation()` (in module `xdoctest.parser`), 102
- `_module_exists()` (in module `xdoctest.directive`), 81
- `_package_chunk()` (`xdoctest.parser.DoctestParser`

method), 99
 _package_groups() (*xdoctest.parser.DocTestParser method*), 99
 _parse() (*xdoctest.doctest_example.DocTest method*), 86
 _parse_commandline() (*in module xdoctest.runner*), 106
 _parse_static_node_value() (*in module xdoctest.static_analysis*), 113
 _parse_static_node_value() (*in module xdoctest.utils.util_import*), 27
 _partfilename (*xdoctest.doctest_example.DocTest attribute*), 84
 _partfilename_for() (*xdoctest.doctest_example.DocTest method*), 87
 _partfilename_to_part (*xdoctest.doctest_example.DocTest attribute*), 84
 _parts (*xdoctest.doctest_example.DocTest attribute*), 84
 _platform_pylib_exts() (*in module xdoctest.utils.util_import*), 28
 _populate_from_cli() (*xdoctest.doctest_example.DocTestConfig method*), 82
 _post_run() (*xdoctest.doctest_example.DocTest method*), 89
 _pre_run() (*xdoctest.doctest_example.DocTest method*), 88
 _print_captured() (*xdoctest.doctest_example.DocTest method*), 89
 _print_summary_report() (*in module xdoctest.runner*), 105
 _rectify_to_modpath() (*in module xdoctest.core*), 67
 _replace() (*xdoctest.directive.Effect method*), 74
 _replace() (*xdoctest.docstr.docscrape_google.DocBlock method*), 11
 _resolve_skipped_exception_class() (*in module xdoctest.exceptions*), 96
 _run_case() (*in module xdoctest.utils.util_misc*), 33
 _run_examples() (*in module xdoctest.runner*), 106
 _runstate (*xdoctest.doctest_example.DocTest attribute*), 84
 _skipped_parts (*xdoctest.doctest_example.DocTest attribute*), 84
 _split_opstr() (*in module xdoctest.directive*), 79
 _static_parse() (*in module xdoctest.utils.util_import*), 27
 _strip_exception_details() (*in module xdoctest.checker*), 59
 _strip_hashtag_comments_and_newlines() (*in module xdoctest.static_analysis*), 117
 _syspath_modname_to_modpath() (*in module xdoctest.utils.util_import*), 28
 _test_globals() (*xdoctest.doctest_example.DocTest method*), 87
 _traverse_traceback() (*in module xdoctest.doctest_example*), 89
 _unmatched_stdout (*xdoctest.doctest_example.DocTest attribute*), 84
 _unpack_args() (*xdoctest.directive.Directive method*), 78
 _update_argparse_cli() (*in module xdoctest.runner*), 106
 _update_argparse_cli() (*xdoctest.doctest_example.DocTestConfig method*), 82
 _visit_generic_FunctionDef() (*xdoctest.static_analysis.TopLevelVisitor method*), 108

A

action (*xdoctest.directive.Effect attribute*), 74
 add_line_numbers() (*in module xdoctest.utils*), 47
 add_line_numbers() (*in module xdoctest.utils.util_str*), 40
 always_fails() (*xdoctest.demo.MyClass static method*), 70
 anything_ran() (*xdoctest.doctest_example.DocTest method*), 87
 ASYNC (*xdoctest.directive.RuntimeStateDict attribute*), 74

C

CallDefNode (*class in xdoctest.static_analysis*), 106
 CaptureStdout (*class in xdoctest.utils*), 43
 CaptureStdout (*class in xdoctest.utils.util_stream*), 42
 CaptureStream (*class in xdoctest.utils*), 44
 CaptureStream (*class in xdoctest.utils.util_stream*), 42
 CellDeleter (*class in xdoctest.utils.util_notebook*), 35
 check() (*xdoctest.doctest_part.DocTestPart method*), 91
 check_exception() (*in module xdoctest.checker*), 60
 check_got_vs_want() (*in module xdoctest.checker*), 59
 check_output() (*in module xdoctest.checker*), 60
 cleanup() (*xdoctest.utils.TempDir method*), 47
 cleanup() (*xdoctest.utils.util_path.TempDir method*), 37
 close() (*xdoctest.utils.CaptureStdout method*), 44
 close() (*xdoctest.utils.util_stream.CaptureStdout method*), 43
 cmdline (*xdoctest.doctest_example.DocTest property*), 88
 codeblock() (*in module xdoctest.utils*), 48
 codeblock() (*in module xdoctest.utils.util_str*), 40
 color_text() (*in module xdoctest.utils*), 48
 color_text() (*in module xdoctest.utils.util_str*), 38
 compilable_source() (*xdoctest.doctest_part.DocTestPart method*), 91
 compile_mode (*xdoctest.doctest_part.DocTestPart attribute*), 90

D

- default_options (*xdoctest.utils.util_notebook.NotebookLoader* attribute), 35
 - demo() (*xdoctest.demo.MyClass* class method), 70
 - detect_encoding() (in module *xdoctest._tokenize*), 58
 - Directive (class in *xdoctest.directive*), 76
 - directives (*xdoctest.doctest_part.DoctestPart* property), 91
 - DocBlock (class in *xdoctest.docstr.docscrape_google*), 11
 - docsrc (*xdoctest.doctest_example.DocTest* attribute), 84
 - DocTest (class in *xdoctest.doctest_example*), 82
 - doctest_callable() (in module *xdoctest*), 123
 - doctest_callable() (in module *xdoctest.runner*), 103
 - doctest_module() (in module *xdoctest*), 122
 - doctest_module() (in module *xdoctest.runner*), 104
 - DoctestConfig (class in *xdoctest.doctest_example*), 81
 - DoctestParseError, 96, 122
 - DoctestParser (class in *xdoctest.parser*), 97
 - DoctestPart (class in *xdoctest.doctest_part*), 90
 - DONT_ACCEPT_BLANKLINE (*xdoctest.directive.RuntimeStateDict* attribute), 74
 - dpath (*xdoctest.utils.PythonPathContext* attribute), 46
 - dpath (*xdoctest.utils.util_import.PythonPathContext* attribute), 24
- ## E
- Effect (class in *xdoctest.directive*), 74
 - effect() (*xdoctest.directive.Directive* method), 78
 - effects() (*xdoctest.directive.Directive* method), 78
 - ELLIPSIS (*xdoctest.directive.RuntimeStateDict* attribute), 74
 - encoding (*xdoctest.utils.TeeStringIO* property), 46
 - encoding (*xdoctest.utils.util_stream.TeeStringIO* property), 42
 - ensure() (*xdoctest.utils.TempDir* method), 47
 - ensure() (*xdoctest.utils.util_path.TempDir* method), 37
 - ensure_unicode() (in module *xdoctest.utils*), 49
 - ensure_unicode() (in module *xdoctest.utils.util_str*), 39
 - ensuredir() (in module *xdoctest.utils*), 49
 - ensuredir() (in module *xdoctest.utils.util_path*), 37
 - exact_type (*xdoctest._tokenize.TokenInfo* property), 59
 - exc_info (*xdoctest.doctest_example.DocTest* attribute), 84
 - exec_lines (*xdoctest.doctest_part.DoctestPart* attribute), 90
 - execute_notebook() (in module *xdoctest.utils.util_notebook*), 36
 - ExistingEventLoopError, 95, 122
 - ExitTestException, 96, 122
 - extract() (*xdoctest.directive.Directive* class method), 76
 - extract_comments() (in module *xdoctest.static_analysis*), 116
 - extract_exc_want() (in module *xdoctest.checker*), 60
 - ExtractGotReprException, 62
- ## F
- failed_line_offset() (*xdoctest.doctest_example.DocTest* method), 88
 - failed_lineno() (*xdoctest.doctest_example.DocTest* method), 88
 - failed_part (*xdoctest.doctest_example.DocTest* attribute), 84
 - failed_tb_lineno (*xdoctest.doctest_example.DocTest* attribute), 84
 - failed_tb_part (*xdoctest.doctest_example.DocTest* attribute), 84
 - fileno() (*xdoctest.utils.TeeStringIO* method), 46
 - fileno() (*xdoctest.utils.util_stream.TeeStringIO* method), 42
 - flush() (*xdoctest.utils.TeeStringIO* method), 46
 - flush() (*xdoctest.utils.util_stream.TeeStringIO* method), 42
 - format_part() (*xdoctest.doctest_part.DoctestPart* method), 92
 - format_parts() (*xdoctest.doctest_example.DocTest* method), 85
 - format_src() (*xdoctest.doctest_example.DocTest* method), 85
 - fpath (*xdoctest.doctest_example.DocTest* attribute), 84
- ## G
- gather_doctests() (in module *xdoctest.runner*), 104
 - generate_tokens() (in module *xdoctest._tokenize*), 58
 - get_parent_frame() (in module *xdoctest.dynamic_analysis*), 94
 - get_stack_frame() (in module *xdoctest.dynamic_analysis*), 93
 - getvalue() (*xdoctest.doctest_example.DoctestConfig* method), 82
 - global_namespace (*xdoctest.doctest_example.DocTest* attribute), 84
 - globs (*xdoctest.doctest_example.DocTest* property), 88
 - got (*xdoctest.checker.GotWantException* attribute), 62
 - GotWantException, 62
- ## H
- has_any_code() (*xdoctest.doctest_part.DoctestPart* method), 91
 - highlight_code() (in module *xdoctest.utils*), 50
 - highlight_code() (in module *xdoctest.utils.util_str*), 39
- ## I
- IGNORE_EXCEPTION_DETAIL

- `(xdoctest.directive.RuntimeStateDict attribute)`, 74
 - `IGNORE_WANT` (*xdoctest.directive.RuntimeStateDict attribute*), 74
 - `IGNORE_WHITESPACE` (*xdoctest.directive.RuntimeStateDict attribute*), 74
 - `import_module_from_name()` (in module *xdoctest.utils*), 50
 - `import_module_from_name()` (in module *xdoctest.utils.util_import*), 26
 - `import_module_from_path()` (in module *xdoctest.utils*), 51
 - `import_module_from_path()` (in module *xdoctest.utils.util_import*), 24
 - `import_notebook_from_path()` (in module *xdoctest.utils.util_notebook*), 35
 - `IncompleteParseError`, 96
 - `indent()` (in module *xdoctest.utils*), 53
 - `indent()` (in module *xdoctest.utils.util_str*), 39
 - `index` (*xdoctest.utils.PythonPathContext attribute*), 46
 - `index` (*xdoctest.utils.util_import.PythonPathContext attribute*), 24
 - `is_balanced_statement()` (in module *xdoctest.static_analysis*), 115
 - `is_defined_by_module()` (in module *xdoctest.dynamic_analysis*), 95
 - `is_disabled()` (*xdoctest.doctest_example.DocTest method*), 84
 - `is_modname_importable()` (in module *xdoctest.utils*), 53
 - `is_modname_importable()` (in module *xdoctest.utils.util_import*), 22
 - `isatty()` (*xdoctest.utils.TeeStringIO method*), 46
 - `isatty()` (*xdoctest.utils.util_stream.TeeStringIO method*), 41
 - `ISEOF()` (in module *xdoctest._tokenize*), 58
 - `ISNONTERMINAL()` (in module *xdoctest._tokenize*), 58
 - `ISTERMINAL()` (in module *xdoctest._tokenize*), 58
 - `iter_module_doctestables()` (in module *xdoctest.dynamic_analysis*), 94
- K**
- `key` (*xdoctest.directive.Effect attribute*), 74
- L**
- `line_offset` (*xdoctest.doctest_part.DoctestPart attribute*), 90
 - `lineno` (*xdoctest.doctest_example.DocTest attribute*), 84
 - `load_module()` (*xdoctest.utils.util_notebook.NotebookLoader method*), 35
 - `log()` (in module *xdoctest.runner*), 103
 - `log_part()` (*xdoctest.utils.CaptureStdout method*), 44
 - `log_part()` (*xdoctest.utils.util_stream.CaptureStdout method*), 43
 - `logged_evals` (*xdoctest.doctest_example.DocTest attribute*), 84
 - `logged_stdout` (*xdoctest.doctest_example.DocTest attribute*), 84
- M**
- `main()` (in module *xdoctest.__main__*), 57
 - `MalformedDocstr`, 95, 122
 - `modname` (*xdoctest.doctest_example.DocTest attribute*), 84
 - `modname_to_modpath()` (in module *xdoctest.utils*), 54
 - `modname_to_modpath()` (in module *xdoctest.utils.util_import*), 29
 - `modpath_to_modname()` (in module *xdoctest.utils*), 54
 - `modpath_to_modname()` (in module *xdoctest.utils.util_import*), 31
 - module
 - xdoctest*, 118
 - xdoctest.__init__*, 1
 - xdoctest.__main__*, 57
 - xdoctest._tokenize*, 58
 - xdoctest.checker*, 59
 - xdoctest.constants*, 63
 - xdoctest.core*, 64
 - xdoctest.demo*, 69
 - xdoctest.directive*, 71
 - xdoctest.docstr*, 16
 - xdoctest.docstr.docscrape_google*, 11
 - xdoctest.docstr.docscrape_numpy*, 16
 - xdoctest.doctest_example*, 81
 - xdoctest.doctest_part*, 90
 - xdoctest.dynamic_analysis*, 93
 - xdoctest.exceptions*, 95
 - xdoctest.global_state*, 96
 - xdoctest.parser*, 96
 - xdoctest.runner*, 103
 - xdoctest.static_analysis*, 106
 - xdoctest.utils*, 43
 - xdoctest.utils.util_deprecation*, 21
 - xdoctest.utils.util_import*, 22
 - xdoctest.utils.util_misc*, 33
 - xdoctest.utils.util_mixins*, 33
 - xdoctest.utils.util_notebook*, 34
 - xdoctest.utils.util_path*, 37
 - xdoctest.utils.util_str*, 37
 - xdoctest.utils.util_stream*, 41
 - module (*xdoctest.doctest_example.DocTest attribute*), 84
 - `MyClass` (class in *xdoctest.demo*), 69
 - `myfunc()` (in module *xdoctest.demo*), 69
- N**
- `n_exec_lines` (*xdoctest.doctest_part.DoctestPart property*), 90

- `n_lines` (*xdoctest.doctest_part.DoctestPart* property), 90
- `n_want_lines` (*xdoctest.doctest_part.DoctestPart* property), 90
- `named()` (in module *xdoctest.directive*), 73
- `NiceRepr` (class in *xdoctest.utils*), 44
- `NiceRepr` (class in *xdoctest.utils.util_mixins*), 33
- `node` (*xdoctest.doctest_example.DocTest* property), 85
- `normalize()` (in module *xdoctest.checker*), 61
- `normalize_modpath()` (in module *xdoctest.utils*), 56
- `normalize_modpath()` (in module *xdoctest.utils.util_import*), 30
- `NORMALIZE_REPR` (*xdoctest.directive.RuntimeStateDict* attribute), 74
- `NORMALIZE_WHITESPACE` (*xdoctest.directive.RuntimeStateDict* attribute), 74
- `NotebookLoader` (class in *xdoctest.utils.util_notebook*), 35
- `num` (*xdoctest.doctest_example.DocTest* attribute), 84
- ## O
- `offset` (*xdoctest.docstr.docscrape_google.DocBlock* attribute), 11
- `orig_ex` (*xdoctest.checker.ExtractGotReprException* attribute), 62
- `orig_lines` (*xdoctest.doctest_part.DoctestPart* attribute), 90
- `output_difference()` (*xdoctest.checker.GotWantException* method), 62
- `output_repr_difference()` (*xdoctest.checker.GotWantException* method), 63
- ## P
- `package_calldefs()` (in module *xdoctest.core*), 67
- `package_modpaths()` (in module *xdoctest.static_analysis*), 114
- `parse()` (*xdoctest.parser.DoctestParser* method), 97
- `parse()` (*xdoctest.static_analysis.TopLevelVisitor* class method), 107
- `parse_auto_docstr_examples()` (in module *xdoctest.core*), 66
- `parse_calldefs()` (in module *xdoctest.core*), 68
- `parse_calldefs()` (in module *xdoctest.static_analysis*), 113
- `parse_directive_optstr()` (in module *xdoctest.directive*), 81
- `parse_docstr_examples()` (in module *xdoctest.core*), 66
- `parse_doctestables()` (in module *xdoctest.core*), 68
- `parse_dynamic_calldefs()` (in module *xdoctest.dynamic_analysis*), 93
- `parse_freeform_docstr_examples()` (in module *xdoctest.core*), 64
- `parse_google_argblock()` (in module *xdoctest.docstr*), 16
- `parse_google_argblock()` (in module *xdoctest.docstr.docscrape_google*), 15
- `parse_google_args()` (in module *xdoctest.docstr*), 17
- `parse_google_args()` (in module *xdoctest.docstr.docscrape_google*), 13
- `parse_google_docstr_examples()` (in module *xdoctest.core*), 65
- `parse_google_retblock()` (in module *xdoctest.docstr*), 17
- `parse_google_retblock()` (in module *xdoctest.docstr.docscrape_google*), 14
- `parse_google_returns()` (in module *xdoctest.docstr*), 18
- `parse_google_returns()` (in module *xdoctest.docstr.docscrape_google*), 14
- `parse_static_calldefs()` (in module *xdoctest.static_analysis*), 113
- `parse_static_value()` (in module *xdoctest.static_analysis*), 113
- `partno` (*xdoctest.doctest_part.DoctestPart* attribute), 90
- `print_contents()` (*xdoctest.utils.util_misc.TempModule* method), 33
- `process_finished()` (*xdoctest.static_analysis.TopLevelVisitor* method), 107
- `PythonPathContext` (class in *xdoctest.utils*), 45
- `PythonPathContext` (class in *xdoctest.utils.util_import*), 23
- ## R
- `remove_blankline_marker()` (in module *xdoctest.checker*), 63
- `REPORT_CDIF` (*xdoctest.directive.RuntimeStateDict* attribute), 74
- `REPORT_NDIFF` (*xdoctest.directive.RuntimeStateDict* attribute), 74
- `REPORT_UDIFF` (*xdoctest.directive.RuntimeStateDict* attribute), 74
- `repr_failure()` (*xdoctest.doctest_example.DocTest* method), 88
- `REQUIRES` (*xdoctest.directive.RuntimeStateDict* attribute), 74
- `run()` (*xdoctest.doctest_example.DocTest* method), 87
- `RuntimeState` (class in *xdoctest.directive*), 75
- `RuntimeStateDict` (class in *xdoctest.directive*), 74
- ## S
- `schedule_deprecation()` (in module *xdoctest.utils.util_deprecation*), 21
- `set_report_style()` (*xdoctest.directive.RuntimeState* method), 76

SKIP (*xdoctest.directive.RuntimeStateDict* attribute), 74
 Skipped (in module *xdoctest.exceptions*), 96
 source (*xdoctest.doctest_part.DoctestPart* property), 90
 split_google_docblocks() (in module *xdoctest.docstr*), 19
 split_google_docblocks() (in module *xdoctest.docstr.docscrape_google*), 12
 split_modpath() (in module *xdoctest.utils*), 56
 split_modpath() (in module *xdoctest.utils.util_import*), 32
 start() (*xdoctest.utils.CaptureStdout* method), 44
 start() (*xdoctest.utils.util_stream.CaptureStdout* method), 43
 stop() (*xdoctest.utils.CaptureStdout* method), 44
 stop() (*xdoctest.utils.util_stream.CaptureStdout* method), 43
 strip_ansi() (in module *xdoctest.utils*), 57
 strip_ansi() (in module *xdoctest.utils.util_str*), 37
 syntax_tree() (*xdoctest.static_analysis.TopLevelVisitor* method), 107

T

TeeStringIO (class in *xdoctest.utils*), 46
 TeeStringIO (class in *xdoctest.utils.util_stream*), 41
 TempDir (class in *xdoctest.utils*), 46
 TempDir (class in *xdoctest.utils.util_path*), 37
 TempDoctest (class in *xdoctest.utils*), 47
 TempDoctest (class in *xdoctest.utils.util_misc*), 33
 TempModule (class in *xdoctest.utils.util_misc*), 33
 text (*xdoctest.docstr.docscrape_google.DocBlock* attribute), 11
 to_dict() (*xdoctest.directive.RuntimeState* method), 76
 TokenInfo (class in *xdoctest._tokenize*), 59
 tokenize() (in module *xdoctest._tokenize*), 58
 TopLevelVisitor (class in *xdoctest.static_analysis*), 106

U

undefined_names() (in module *xdoctest.runner*), 105
 unique_callname (*xdoctest.doctest_example.DocTest* property), 85
 UNKNOWN_CALLNAME (*xdoctest.doctest_example.DocTest* attribute), 83
 UNKNOWN_FPATH (*xdoctest.doctest_example.DocTest* attribute), 84
 UNKNOWN_MODNAME (*xdoctest.doctest_example.DocTest* attribute), 83
 UNKNOWN_MODPATH (*xdoctest.doctest_example.DocTest* attribute), 83
 untokenize() (in module *xdoctest._tokenize*), 58
 update() (*xdoctest.directive.RuntimeState* method), 76

V

valid_testnames (*xdoctest.doctest_example.DocTest*

property), 85
 value (*xdoctest.directive.Effect* attribute), 74
 visit() (*xdoctest.static_analysis.TopLevelVisitor* method), 108
 visit() (*xdoctest.utils.util_notebook.CellDeleter* method), 35
 visit_Assign() (*xdoctest.static_analysis.TopLevelVisitor* method), 108
 visit_AsyncFunctionDef() (*xdoctest.static_analysis.TopLevelVisitor* method), 108
 visit_ClassDef() (*xdoctest.static_analysis.TopLevelVisitor* method), 108
 visit_FunctionDef() (*xdoctest.static_analysis.TopLevelVisitor* method), 108
 visit_If() (*xdoctest.static_analysis.TopLevelVisitor* method), 108
 visit_Module() (*xdoctest.static_analysis.TopLevelVisitor* method), 108

W

want (*xdoctest.checker.GotWantException* attribute), 62
 want (*xdoctest.doctest_part.DoctestPart* property), 91
 want_lines (*xdoctest.doctest_part.DoctestPart* attribute), 90
 wants() (*xdoctest.doctest_example.DocTest* method), 85
 warn_list (*xdoctest.doctest_example.DocTest* attribute), 84
 write() (*xdoctest.utils.TeeStringIO* method), 46
 write() (*xdoctest.utils.util_stream.TeeStringIO* method), 42

X

xdoctest
 module, 118
 xdoctest.__init__
 module, 1
 xdoctest.__main__
 module, 57
 xdoctest._tokenize
 module, 58
 xdoctest.checker
 module, 59
 xdoctest.constants
 module, 63
 xdoctest.core
 module, 64
 xdoctest.demo
 module, 69
 xdoctest.directive
 module, 71
 xdoctest.docstr
 module, 16

- xdoctest.docstr.docscrape_google
 - module, 11
- xdoctest.docstr.docscrape_numpy
 - module, 16
- xdoctest.doctest_example
 - module, 81
- xdoctest.doctest_part
 - module, 90
- xdoctest.dynamic_analysis
 - module, 93
- xdoctest.exceptions
 - module, 95
- xdoctest.global_state
 - module, 96
- xdoctest.parser
 - module, 96
- xdoctest.runner
 - module, 103
- xdoctest.static_analysis
 - module, 106
- xdoctest.utils
 - module, 43
- xdoctest.utils.util_deprecation
 - module, 21
- xdoctest.utils.util_import
 - module, 22
- xdoctest.utils.util_misc
 - module, 33
- xdoctest.utils.util_mixins
 - module, 33
- xdoctest.utils.util_notebook
 - module, 34
- xdoctest.utils.util_path
 - module, 37
- xdoctest.utils.util_str
 - module, 37
- xdoctest.utils.util_stream
 - module, 41