xdoctest Documentation

Release 1.1.3

Jon Crall

Jan 30, 2024

PACKAGE LAYOUT

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

CHAPTER

ONE

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.

CHAPTER

TWO

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 code prefixed with >>> `` is skipped due to the xdoctest ``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
```

CHAPTER

THREE

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
```

(continues on next page)

(continued from previous page)

```
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)
  --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.

CHAPTER

FOUR

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: tuple

Create new instance of DocBlock(text, offset)

offset

Alias for field number 1

text

Alias for field number 0

xdoctest.docstr.docscrape_google.**split_google_docblocks**(*docstr*)

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 bock 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
>>> [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*)

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*, *return_annot=None*)

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*, *return_annot=None*)

Parse information out of a returns or yeilds 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.docscrape_google.**parse_google_argblock**(*lines*, *clean_desc=True*)

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 | None] – 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*, *clean_desc=True*)

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 | None] – 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*)

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*, *return_annot=None*)

Parse information out of a returns or yeilds 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',
```

(continues on next page)

(continued from previous page)

```
... ])))
>>> 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*, *return_annot=None*)

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*)

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 bock 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
>>> [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
```

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*, *name='?'*, *type='?'*, *migration=''*, *deprecate=None*, *error=None*, *remove=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',
```

(continues on next page)

(continued from previous page)

```
...         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*, *sys_path=None*, *exclude=None*)

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

Parameters

- **modname** (*str*) – name of module to check
- **sys_path** (*list | None, default=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
```

class xdoctest.utils.util_import.**PythonPathContext**(*dpath*, *index=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.

xdoctest.utils.util_import.**import_module_from_path**(*modpath*, *index=-1*)

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*)

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',
>>>     'lib2to3.fixes.fix_apply',
>>> ]
>>> #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.**modname_to_modpath**(*modname*, *hide_init=True*, *hide_main=False*, *sys_path=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')
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = basename(modname_to_modpath('_ctypes'))
>>> assert 'ctypes' in modpath
```

xdoctest.utils.util_import.**normalize_modpath**(*modpath*, *hide_init=True*, *hide_main=False*)

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*, *hide_init=True*, *hide_main=False*, *check=True*, *relativeto=None*)

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*) – 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

```
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = modname_to_modpath('_ctypes')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == '_ctypes'
```

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*, *check=True*)

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')
```

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*, *modname=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*, *modname=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()

For debugging on windows

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'.

class xdoctest.utils.util_notebook.**CellDeleter**

Bases: 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=None*, *fpath=None*)

import a notebook as a module

xdoctest.utils.util_notebook.**import_notebook_from_path**(*ipynb_fpath*, *only_defs=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.ipydb')
>>> cells = [
>>>     utils.codeblock(
>>>         '''
>>>         def foo():
>>>             return 'bar'
>>>         '''),
>>>     utils.codeblock(
>>>         '''
>>>         x = 1
>>>         ''')
>>> ]
>>> _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*, *timeout=None*, *verbose=None*)

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.ipydb')
>>> _make_test_notebook_fpath(ipynb_fpath, [utils.codeblock(
>>>     '''
>>>     print('hello world')
>>>     ''')])
>>> nb, resources = execute_notebook(ipynb_fpath, verbose=3)
```

(continues on next page)

(continued from previous page)

```
>>> 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)?')
```

4.1.2.1.6 xdoctest.utils.util_path module

Utilities related to filesystem paths

class xdoctest.utils.util_path.**TempDir**(*persist=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()

cleanup()

xdoctest.utils.util_path.**ensuredir**(*dpath*, *mode=1023*)

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*)

Removes all ansi directives from the string.

Parameters
text (*str*)

Returns
str

References

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

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*, *color*)

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
```

(continues on next page)

(continued from previous page)

```
>>>     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.util_str.**ensure_unicode**(*text*)

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 ünicôdé strįng') == 'my ünicôdé strįng'
>>> assert ensure_unicode('text1') == 'text1'
>>> assert ensure_unicode('text1'.encode('utf8')) == 'text1'
>>> assert ensure_unicode('ï»¿text1'.encode('utf8')) == 'ï»¿text1'
>>> import codecs
>>> assert (codecs.BOM_UTF8 + 'text»¿'.encode('utf8')).decode('utf8')
```

xdoctest.utils.util_str.**indent**(*text*, *prefix=' '*)

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*, *lexer_name='python'*, ***kwargs*)

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*, *start=1*, *n_digits=None*)

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*)

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=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()

Returns true of the redirect is a terminal.

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

fileno()

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

property encoding

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*)

Write to this and the redirected stream

flush()

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=True*, *enabled=True*, ***kwargs*)

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()

Log what has been captured so far

start()

stop()

Example

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

close()

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=True*, *enabled=True*, ***kwargs*)

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()

Log what has been captured so far

start()

stop()

Example

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

close()

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

class xdoctest.utils.**PythonPathContext**(*dpath*, *index=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.

class xdoctest.utils.**TeeStringIO**(*redirect=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()

Returns true of the redirect is a terminal.

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

fileno()

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

property encoding

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*)

Write to this and the redirected stream

flush()

Flush to this and the redirected stream

class xdoctest.utils.**TempDir**(*persist=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()

cleanup()

class xdoctest.utils.**TempDoctest**(*docstr*, *modname=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*, *start=1*, *n_digits=None*)

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.**codeblock**(*block_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*, *color*)

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*)

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 ünicôdé strįng') == 'my ünicôdé strįng'
>>> assert ensure_unicode('text1') == 'text1'
>>> assert ensure_unicode('text1'.encode('utf8')) == 'text1'
>>> assert ensure_unicode('ï»¿text1'.encode('utf8')) == 'ï»¿text1'
>>> import codecs
>>> assert (codecs.BOM_UTF8 + 'text»¿'.encode('utf8')).decode('utf8')
```

xdoctest.utils.**ensuredir**(*dpath*, *mode=1023*)

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*, *lexer_name='python'*, ***kwargs*)

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*)

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',
>>>     'lib2to3.fixes.fix_apply',
>>> ]
>>> #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*, *index=-1*)

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.**indent**(*text*, *prefix=' '*)

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*, *sys_path=None*, *exclude=None*)

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

Parameters

- **modname** (*str*) – name of module to check
- **sys_path** (*list | None, default=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*, *hide_init=True*, *hide_main=False*, *sys_path=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')
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = basename(modname_to_modpath('_ctypes'))
>>> assert 'ctypes' in modpath
```

xdoctest.utils.**modpath_to_modname**(*modpath*, *hide_init=True*, *hide_main=False*, *check=True*, *relativeto=None*)

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*) – 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

```
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = modname_to_modpath('_ctypes')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == '_ctypes'
```

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*, *hide_init=True*, *hide_main=False*)

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*, *check=True*)

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*)

Removes all ansi directives from the string.

Parameters
text (*str*)

Returns
str

References

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

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.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 cheverons (``>>> ``).

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*, *got_stdout*, *got_eval=<NOT_EVALED>*, *runstate=None*)

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.**extract_exc_want**(*want*)

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*, *want*, *runstate=None*)

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*, *want*, *runstate=None*)

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.**normalize**(*got*, *want*, *runstate=None*)

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*, *orig_ex*)

Bases: AssertionError

Exception used when we are unable to extract a string "got"

Parameters

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

exception xdoctest.checker.**GotWantException**(*msg*, *got*, *want*)

Bases: AssertionError

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

Parameters

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

output_difference(*runstate=None*, *colored=True*)

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=None*)

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*)

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)
```

4.2.2 xdoctest.constants module

Defines sentinel values for internal xdoctest usage

4.2.3 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 cheverons (``>>> ``) or right angle brakets 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*, *callname=None*, *modpath=None*, *lineno=1*, *fpath=None*, *asone=True*)

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*, *callname=None*, *modpath=None*, *lineno=1*, *fpath=None*, *eager_parse=True*)

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

xdoctest.core.**parse_docstr_examples**(*docstr*, *callname=None*, *modpath=None*, *lineno=1*, *style='auto'*, *fpath=None*, *parser_kw=None*)
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
...     ''')
```

(continues on next page)

(continued from previous page)

```
>>> 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.**package_calldefs**(*pkg_identifier*, *exclude=[]*, *ignore_syntax_errors=True*, *analysis='auto'*)

Statically generates all callable definitions in a module or package

Parameters

- **pkg_identifier** (*str | 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]
>>> assert util_import.modpath_to_modname(modpath) == pkg_identifier
>>> assert 'package_calldefs' in calldefs
```

xdoctest.core.**parse_calldefs**(*module_identifier*, *analysis='auto'*)

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*, *exclude=[]*, *style='auto'*, *ignore_syntax_errors=True*, *parser_kw={}*, *analysis='auto'*)

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:
>>>     # 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
```

(continues on next page)

(continued from previous page)

```
>>> examples = list(parse_doctestables(modpath, style='freeform'))
>>> print(len(examples))
1
```

4.2.4 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**()

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*, ***kw*)

Bases: object

Example

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

Example

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

classmethod demo(***kw*)

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()

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
...
```

4.2.5 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.

4.2.5.1 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_CDIFF`: False,
- `REPORT_NDIFF`: False,
- `REPORT_UDIFF`: True,
- `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`.

4.2.5.2 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
```

(continues on next page)

(continued from previous page)

```
>>> # the program to continue executing.
>>> # xdoctest: -SKIP
>>> 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*, *pattern*)

helper for regex

Parameters

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

Returns

str

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

Bases: tuple

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

action

Alias for field number 0

key

Alias for field number 1

value

Alias for field number 2

class xdoctest.directive.**RuntimeState**(*default_state=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)]
```

(continues on next page)

(continued from previous page)

```
>>> runstate.update(directives)
>>> assert not runstate['IGNORE_WHITESPACE']
>>> runstate.update({})
>>> assert runstate['IGNORE_WHITESPACE']
```

Example

```
>>> # xdoc: +IGNORE_WHITESPACE
>>> print(str(RuntimeState()))
<RuntimeState({
    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

`to_dict()`

Returns

OrderedDict

`set_report_style(reportchoice, state=None)`

Parameters

- **reportchoice** (*str*) – name of report style
- **state** (*None | Dict*) – 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*)

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*, *positive=True*, *args=[]*, *inline=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*)

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
```

effect(*argv=None*, *environ=None*)

effects(*argv=None*, *environ=None*)

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')
>>> 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:FOO==1)'))[0]
>>> print('directive = {}'.format(directive))
```

(continues on next page)

(continued from previous page)

```
>>> print('directive.effects() = {}'.format(directive.effects(environ={})[0]))
directive = <Directive(+REQUIRES(env:FOO==1))>
directive.effects() = Effect(action='set.add', key='REQUIRES', value='env:FOO==1
↪')
```

```
>>> directive = list(Directive.extract('# xdoctest: requires(env:FOO==1)'))[0]
>>> print('directive = {}'.format(directive))
>>> print('directive.effects() = {}'.format(directive.effects(environ={'FOO': '1
↪'})[0]))
directive = <Directive(+REQUIRES(env:FOO==1))>
directive.effects() = Effect(action='noop', key='REQUIRES', value='env:FOO==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 = {!r}'.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.**parse_directive_optstr**(*optpart*, *inline=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

Return type

Directive

Example

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

4.2.6 xdoctest.doctest_example module

This module defines the main class that holds a DocTest example

class xdoctest.doctest_example.**DoctestConfig**(**args*, ***kwargs*)

Bases: dict

Doctest configuration

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

getvalue(*key*, *given=None*)

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*, *modpath=None*, *callname=None*, *num=0*, *lineno=1*, *fpath=None*, *block_type=None*, *mode='pytest'*)

Bases: object

Holds information necessary to execute and verify a doctest

Variables

- **docsrc** (*str*) – doctest source code
- **modpath** (*str | PathLike*) – 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 | TracebackType*) – 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?>'

is_disabled(*pytest=False*)

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

A key that references this doctest given its module

Returns
str

property node

A key that references this doctest within pytest

Returns
str

property valid_testnames

A set of callname and unique_callname

Returns
Set[str]

wants()

Returns a list of the populated wants

Yields
str

format_parts(*linenos=True*, *colored=None*, *want=True*, *offset_linenos=None*, *prefix=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=True*, *colored=None*, *want=True*, *offset_linenos=None*, *prefix=True*)

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()
```

anything_ran()

Returns

bool

run(*verbose=None*, *on_error=None*)

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

property globs

Alias for `global_namespace` for pytest 8.0 compatability

property cmdline

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

Return type
str

failed_line_offset()

Determine which line in the doctest failed.

Returns
int | None

failed_lineno()

Returns
int | None

repr_failure(*with_tb=True*)

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
    >>> 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()))
```

4.2.7 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*, *want_lines=None*, *line_offset=0*, *orig_lines=None*, *directives=None*, *partno=None*)

Bases: `object`

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.

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

`property n_lines`

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

`property n_exec_lines`

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

`property n_want_lines`

Returns: int: number of lines in the "want" statement.

`property source`

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

`compilable_source()`

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

Returns
str

`has_any_code()`

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`

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`

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

check(*got_stdout*, *got_eval=<NOT_EVALED>*, *runstate=None*, *unmatched=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) evaled 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=True*, *want=True*, *startline=1*, *n_digits=None*, *colored=False*, *partnos=False*, *prefix=True*)

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*) – 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)
```

4.2.8 xdoctest.dynamic_analysis module

Utilities for dynamically inspecting code

xdoctest.dynamic_analysis.**parse_dynamic_calldefs**(*modpath_or_module*)

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=0*, *strict=True*)

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=0*)

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*)

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*, *module*)

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)
```

4.2.9 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.**DoctestParseError**(*msg*, *string=None*, *info=None*, *orig_ex=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 exceptoin

exception xdoctest.exceptions.**ExitTestException**

Bases: Exception

exception xdoctest.exceptions.**IncompleteParseError**

Bases: SyntaxError

Used when something goes wrong in the xdoctest parser

4.2.10 xdoctest.global_state module

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

4.2.11 xdoctest.parser module

4.2.11.1 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=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
```

(continues on next page)

(continued from previous page)

```
>>>     '''.lstrip('\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*, *info=None*)

Divide the given string into examples and interleaving text.

Parameters

- **string** (*str*) – string representing the doctest
- **info** (*dict | None*) – info about where the string came from in case of an error

Returns

a list of *DoctestPart* objects

Return type

List[*xdoctest.doctest_part.DoctestPart*]

CommandLine

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

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')
>>> '''
```

(continues on next page)

(continued from previous page)

```
    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.parser import *  # NOQA
>>> from xdoctest import parser
>>> from xdoctest.docstr import docscrape_google
>>> from xdoctest import core
>>> self = parser.DoctestParser()
>>> docstr = self.parse.__doc__
>>> blocks = docscrape_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)
```

4.2.12 xdoctest.plugin module

4.2.13 xdoctest.runner module

4.2.13.1 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.

4.2.13.2 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
```

4.2.13.3 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*, *verbose*, *level=1*)

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*)

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*)

xdoctest.runner.**doctest_module**(*module_identifier=None*, *command=None*, *argv=None*, *exclude=[]*, *style='auto'*, *verbose=None*, *config=None*, *durations=None*, *analysis='auto'*)

Executes requestsed 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 beharior. 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.**undefined_names**(*sourcecode*)

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'}
```

4.2.14 xdoctest.static_analysis module

The core logic that allows for xdoctest to parse source statically

class xdoctest.static_analysis.**CallDefNode**(*callname*, *lineno*, *docstr*, *doclineno*, *doclineno_end*, *args=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
- **doclineno** (*int*) – the line number (1 based) the docstring begins on
- **doclineno_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=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(object):
            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*)

main entry point

executes parsing algorithm and populates self.calldefs

Parameters
source (*str*)

syntax_tree()

creates the abstract syntax tree

Return type
ast.Module

process_finished(*node*)

process (get ending lineno) for everything marked as finished

Parameters
node (*ast.AST*)

visit(*node*)

Parameters
node (*ast.AST*)

visit_FunctionDef(*node*)

Parameters
node (*ast.FunctionDef*)

visit_ClassDef(*node*)

Parameters
node (*ast.ClassDef*)

visit_Module(*node*)

Parameters
node (*ast.Module*)

visit_Assign(*node*)

Parameters
node (*ast.Assign*)

visit_If(*node*)

Parameters
node (*ast.If*)

xdoctest.static_analysis.**parse_static_calldefs**(*source=None*, *fpath=None*)

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=None*, *fpath=None*)

xdoctest.static_analysis.**parse_static_value**(*key*, *source=None*, *fpath=None*)

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*, *with_pkg=False*, *with_mod=True*, *followlinks=True*, *recursive=True*, *with_libs=False*, *check=True*)

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
>>> assert 'xdoctest' not in names
>>> print('\n'.join(names))
```

xdoctest.static_analysis.**is_balanced_statement**(*lines*, *only_tokens=False*, *reraise=0*)

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'])),",
>>>     '    ],',
>>> ]
>>> 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*)

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
        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.**six_axt_parse**(*source_block*, *filename='<source_block>'*, *compatible=True*)

Python 2/3 compatible replacement for ast.parse(source_block, filename='<source_block>')

Parameters

- **source** (*str*)
- **filename** (*str*)
- **compatible** (*bool*)

Returns
ast.Module | types.CodeType

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 code prefixed with >>> `` is skipped due to the xdoctest ``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␣
```

(continues on next page)

(continued from previous page)

```
↪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.

exception xdoctest.**DoctestParseError**(*msg*, *string=None*, *info=None*, *orig_ex=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 exceptoin

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).

xdoctest.**doctest_module**(*module_identifier=None*, *command=None*, *argv=None*, *exclude=[]*, *style='auto'*, *verbose=None*, *config=None*, *durations=None*, *analysis='auto'*)

Executes requestsed 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 beharior. 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*)

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)
```

CHAPTER

FIVE

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

INDEX

G

H

I

K

L

M

T

U

V

W

X