phmdoctest

Release 1.4.0

Mark Taylor

Mar 22, 2022

CONTENTS:

CHAPTER

ONE

PHMDOCTEST 1.4.0

1.1 Introduction

Python syntax highlighted Markdown doctest

Command line program and Python library to test Python syntax highlighted code examples in Markdown.

- Creates a pytest Python module that tests Python examples in README and other Markdown files.
- Reads these from Markdown fenced code blocks:
  - Python interactive sessions described by doctest.
  - Python source code and expected terminal output.
- The test cases get run later by running pytest.
- Simple use case is possible with no Markdown edits at all.
- More features selected by adding HTML comment **directives** to the Markdown.
  - Set test case name.
  - Add a pytest custom marker.
  - Add a pytest.mark.skip decorator.
  - Promote names defined in a test case to module level globals.
  - Label any fenced code block for later retrieval (API).
- Configurable. Discover and process many Markdown files in a single command.
- Add inline annotations to comment out sections of code.
- Get code coverage by running pytest with coverage.
- Select Python source code blocks as setup and teardown code.
- Setup applies to code blocks and optionally to session blocks.
- An included Python library: Latest Development tools API.
  - Python function returns test file in a string. *(testfile() in main.py)*
  - Two pytest fixtures. *(tester.py)*
    1. **testfile_creator** runs *testfile()*. Use with testfile_tester.
    2. **testfile_tester** runs a pytest file with pytest's pytester in its isolated environment.
  - Runs phmdoctest and can run pytest too. *(simulator.py)*
  - Functions to read fenced code blocks from Markdown. *(tool.py)*

  - Test Markdown for Python examples. *(tool.py)*
  - Prepare directory for generated test files. *(tool.py)*
  - Extract testsuite tree and list of failing trees from JUnit XML. *(tool.py)*
- Available as the pytest plugin pytest-phmdoctest.

1.1.1 default branch status

Website | Docs | Repos | pytest | Codecov | License

Introduction | *Installation* | *Sample usage* | *Sample Usage with HTML comment directives* | *CI usage* | *–report* | *Identifying blocks* | *Directives* | *skip* | *label on code and sessions* | *label on any fenced code block* | *pytest skip* | *pytest skipif* | *setup* | *teardown* | *share-names* | *clear-names* | *pytest mark decorator* | *label skip and mark example* | *setup and teardown example* | *share-names clear-names example* | *Configuration* | *Inline annotations* | *skipping blocks with –skip* | *–skip* | *short form of –skip* | *–fail-nocode* | *–setup* | *–teardown* | *Setup example* | *Setup for sessions* | *Execution context* | *Send outfile to stdout* | *Usage* | *Run as a Python module* | *Python API* | *pytest fixtures* | *Simulate command line* | *Hints* | *Directive hints* | *Related projects*

Changes | *Contributions* | *About*

1.2 Installation

It is advisable to install in a virtual environment.

```
python -m pip install phmdoctest
```

1.3 Sample usage

Given the Markdown file *example1.md* shown in raw form here…

~~~
# This is Markdown file example1.md

## Interactive Python session (doctest)

```py
>>> print("Hello World!")
Hello World!
```

## Source Code and terminal output

Code:
```python
from enum import Enum

class Floats(Enum):
~~~

(continues on next page)

(continued from previous page)

```
    APPLES = 1
    CIDER = 2
    CHERRIES = 3
    ADUCK = 4

for floater in Floats:
    print(floater)
```
```
sample output:
```
```
Floats.APPLES
Floats.CIDER
Floats.CHERRIES
Floats.ADUCK
```
```

the command…

```
phmdoctest doc/example1.md --outfile test_example1.py
```

creates the python source code file `test_example1.py` shown here…

```
"""pytest file built from doc/example1.md"""
from phmdoctest.functions import _phm_compare_exact

def session_00001_line_6():
    r"""
    >>> print("Hello World!")
    Hello World!
    """

def test_code_14_output_28(capsys):
    from enum import Enum

    class Floats(Enum):
        APPLES = 1
        CIDER = 2
        CHERRIES = 3
        ADUCK = 4

    for floater in Floats:
        print(floater)

    _phm_expected_str = """\
Floats.APPLES
Floats.CIDER
Floats.CHERRIES
Floats.ADUCK
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
```
```

Then run a pytest command something like this in your terminal to test the Markdown session, code, and expected output blocks.

```
pytest --doctest-modules
```

Or these two commands:

```
pytest
python -m doctest test_example1.py
```

The `line_6` in the function name `session_00001_line_6` is the line number in *example1.md* of the first line of the interactive session. `00001` is a sequence number to order the doctests.

The `14` in the function name `test_code_14_output_28` is the line number of the first line of python code. `28` shows the line number of the expected terminal output.

One test case function gets generated for each:

- Markdown fenced code block interactive session
- Python-code/expected-output Markdown fenced code block pair

The `--report` option below shows the blocks discovered and how they are tested.

1.4 Sample Usage with HTML comment directives

Given the Markdown file shown in raw form here…

~~~
<!--phmdoctest-mark.skip-->
<!--phmdoctest-label test_example-->
```python
print("Hello World!")
```
```
incorrect expected output
```
~~~

the command…

```
phmdoctest tests/one_mark_skip.md --outfile test_one_mark_skip.py
```

creates the python source code file shown here…

```
"""pytest file built from tests/one_mark_skip.md"""
import pytest

from phmdoctest.functions import _phm_compare_exact


@pytest.mark.skip()
def test_example(capsys):
    print("Hello World!")

    _phm_expected_str = """\
incorrect expected output
```

(continues on next page)

(continued from previous page)

```
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
```

Run the –outfile with pytest…

```
$ pytest -vv test_one_mark_skip.py

test_one_mark_skip.py::test_example SKIPPED
```

- The HTML comments in the Markdown are phmdoctest **directives**.
- The **mark.skip** directive adds the @pytest.mark.skip() line.
- The label directive names the test case function.
- List of *Directives*
- Directives are optional.
- Markdown edits are optional.

1.5 CI usage

Test Python examples in README.md in Continuous Integration scripts. In this snippet for Linux the pytest test suite is in the **tests** folder.

```
mkdir tests/tmp
phmdoctest README.md --report --outfile tests/tmp/test_readme.py
pytest --doctest-modules -vv tests
```

This console shows testing Python examples in project.md. Look for the tmp tests at the bottom. Windows Usage on Appveyor.

See this excerpt from ci.yml *Actions usage example*. It runs on Windows, Linux, and macOS. Please find the phmdoctest command at the bottom.

No changes to README.md are needed here, look in the last job log.

1.6 report option

To see the GFM fenced code blocks in the MARKDOWN_FILE use the `--report` option like this:

```
phmdoctest doc/example2.md --report
```

which lists the fenced code blocks it found in the file *example2.md*. The `test role` column shows how each fenced code block gets tested.

```
         doc/example2.md fenced blocks
------------------------------------------------
block     line  test     TEXT or directive
type    number  role     quoted and one per line
------------------------------------------------
python       9  code
```

(continues on next page)

(continued from previous page)

```
            14  output
python      20  code
            26  output
            31  --
python      37  code
python      44  code
            51  output
yaml        59  --
text        67  --
py          75  session
python      87  code
            94  output
py         102  session
------------------------------------------------
7 test cases.
1 code blocks with no output block.
```

1.7 Identifying blocks

The PYPI commonmark project provides code to extract fenced code blocks from Markdown. Specification CommonMark Spec and website CommonMark.

Python code, expected output, and Python interactive sessions get extracted.

Only GFM fenced code blocks are considered.

A block is a session block if the info_string starts with `py` and the first line of the block starts with the session prompt: `'>>> '`.

To be treated as Python code the opening fence should start with one of these:

~~~
```python
```python3
```py3
~~~

plus the block contents can't start with `'>>> '`.

The examples use the info_strings `python` for code and `py` for sessions since they render with coloring on GitHub, readthedocs, GitHub Pages, and Python package index.

project.md has more examples of code and session blocks.

It is ok if the info string is laden with additional text, it will be ignored. The entire info string will be shown in the block type column of the report.

An output block is a fenced code block that immediately follows a Python block and starts with an opening fence like this which has an empty info string.

~~~
```
~~~

A Python code block has no output if it is followed by any of:

- Python code block
- Python session block

- a fenced code block with a non-empty info string

Test code gets generated for it, but there will be no assertion statement.

1.8 Directives

Directives are HTML comments containing test generation commands. They are edited into the Markdown file immediately before a fenced code block. It is OK if other HTML comments are present. See the `<!--phmdoctest-skip-->` directive in the raw Markdown below. With the skip directive no test code will be generated from the fenced code block.

~~~
<!--phmdoctest-skip-->
<!--Another HTML comment-->
```python
print("Hello World!")
```
Expected Output
```
Hello World!
```
~~~

List of Directives

```
       Directive HTML comment      |    Use on blocks
---------------------------------- | ---------------------
<!--phmdoctest-skip-->             | code, session, output
<!--phmdoctest-label IDENTIFIER--> | code, session
<!--phmdoctest-label TEXT-->       | any
<!--phmdoctest-mark.skip-->        | code
<!--phmdoctest-mark.skipif<3.N-->  | code
<!--phmdoctest-setup-->            | code
<!--phmdoctest-teardown-->         | code
<!--phmdoctest-share-names-->      | code
<!--phmdoctest-clear-names-->      | code
<!--phmdoctest-mark.ATTRIBUTE-->   | code
```

Directive hints

1.9 skip

The skip directive or `--skip TEXT` command line option prevents code generation for the code or session block. The skip directive can be placed on an expected output block. There it prevents checking expected against actual output. *Example.*

1.10 label on code and sessions

When used on a Python code block or session the label directive changes the name of the generated test function. *Example.* Two generated tests, the first without a label, shown in pytest -v terminal output:

```
test_readme.py::test_code_93 FAILED
test_readme.py::test_beta_feature FAILED
```

1.11 label on any fenced code block

On any fenced code block, the label directive identifies the block for later retrieval by the class `phmdoctest.tool.FCBChooser()`. The `FCBChooser` is used separately from phmdoctest in a different pytest file. This allows the test developer to write additional test cases for fenced code blocks that are not handled by phmdoctest. The directive value can be any string.

~~~
# This is file doc/my_markdown_file.md

<!--phmdoctest-label my-fenced-code-block-->
```
The label directive can be placed on any fenced code block.
```
~~~

Here is Python code to fetch it:

```python
import phmdoctest.tool

chooser = phmdoctest.tool.FCBChooser("doc/my_markdown_file.md")
contents = chooser.contents(label="my-fenced-code-block")
print(contents)
```

Output:

```
The label directive can be placed on any fenced code block.
```

1.12 pytest skip

The `<!--phmdoctest-mark.skip-->` directive generates a test case with a `@pytest.mark.skip()` decorator. *Example.*

1.13 pytest skipif

The `<!--phmdoctest-mark.skipif<3.N-->` directive generates a test case with the pytest decorator `@pytest.mark.skipif(sys.version_info < (3, N), reason="requires >=py3.N")`. N is a Python minor version number. *Example.*

1.14 setup

A single Python code block can assign names visible to other code blocks by adding a setup directive or using the *–setup* command line option.

Names assigned by the setup block get copied to the test module's global namespace after the setup block runs.

Here is an example setup block from *setup.md*:

```
import math

mylist = [1, 2, 3]
a, b = 10, 11

def doubler(x):
    return x * 2
```

Using setup modifies the execution context of the Python code blocks in the Markdown file. The names `math`, `mylist`, `a`, `b`, and `doubler` are visible to the other Python code blocks. The objects can be modified. *Example.*

1.15 teardown

Selects a single Python code block that runs at test module teardown time. A teardown block can also be designated using the *–teardown* command line option. *Example.*

1.16 share-names

Names assigned by the Python code block get copied to the test module as globals after the test code runs. This happens at run time. These names are now visible to subsequent test cases generated for Python code blocks in the Markdown file. share-names modifies the execution context as described for the setup directive above. The share-names directive can be used on more than one code block. *Example.*

This directive effectively joins its Python code block to the following Python code blocks in the Markdown file.

1.17 clear-names

After the test case generated for the Python code block with the clear-names directive runs, all names that were created by one or more preceding share-names directives get deleted. The names that were shared are no longer visible. This directive also deletes the names assigned by setup. *Example.*

1.18 pytest mark decorator

The `<!--phmdoctest-mark.ATTRIBUTE-->` directive adds a @pytest.mark.ATTRIBUTE decorator to the generated test function. ATTRIBUTE is a valid Python attribute identifier. This defines a marker to pytest that is used to select and deselect tests. See the pytest documentation section "Working with custom markers". The file *mark_example.md* contains example usage of the user defined marker "slow". It generates *test_mark_example.py*

1.19 label skip and mark example

The file *directive1.md* contains example usage of label, skip, and mark directives. The command below generates *test_directive1.py*. `phmdoctest doc/directive1.md --report` produces this *report*.

```
phmdoctest doc/directive1.md --outfile test_directive1.py
```

1.20 setup and teardown example

The file *directive2.md* contains example usage of label, skip, and mark directives. The command below generates *test_directive2.py*. `phmdoctest doc/directive2.md --report` produces this *report*.

```
phmdoctest doc/directive2.md --outfile test_directive2.py
```

1.21 share-names clear-names example

The file *directive3.md* contains example usage of share-names and clear-names directives. The command below generates *test_directive3.py*. `phmdoctest doc/directive3.md --report` produces this *report*.

```
phmdoctest doc/directive3.md --outfile test_directive3.py
```

1.22 Configuration

Supply a .ini, .cfg, or .toml configuration file in place of the Markdown file. Configuration features:

- Choose Markdown files for test file generation. (glob wildcards).
- Exclude Markdown files from test file generation. (glob wildcards).
- Name the output directory.
- Removes stale test files from output directory.
- Enable printing.

Place a `[tool.phmdoctest]` section in the configuration file. *How to configure.*

1.23 Inline annotations

Inline annotations comment out sections of code. They can be added to the end of lines in Python code blocks. They should be in a comment.

- `phmdoctest:omit` comments out a section of code. The line it is on, plus following lines at greater indent get commented out.
- `phmdoctest:pass` comments out one line of code and prepends the pass statement.

Here is a snippet showing how to place `phmdoctest:pass` in the code. The second block shows the code that is generated. Note there is no # immediately before `phmdoctest:pass`. It is not required.

```
import time
def takes_too_long():
    time.sleep(100)    # delay for awhile. phmdoctest:pass
takes_too_long()
```

```
import time
def takes_too_long():
    pass  # time.sleep(100)    # delay for awhile. phmdoctest:pass
takes_too_long()
```

Use `phmdoctest:omit` on single or multi-line statements. Note the two commented out time.sleep(99). They follow and are indented more that the `if condition:`line with `phmdoctest:omit`.

```
import time                         # phmdoctest:omit

condition = True
if condition:       # phmdoctest:omit
    time.sleep(99)
    time.sleep(99)
```

```
# import time                         # phmdoctest:omit

condition = True
# if condition:       # phmdoctest:omit
#     time.sleep(99)
#     time.sleep(99)
```

Inline annotation processing counts the number of commented out sections and adds the count as the suffix `_N` to the name of the pytest function in the generated test file.

Inline annotations are similar, but less powerful than the Python standard library **doctest** directive `#doctest+SKIP`. Improper use of `phmdoctest:omit` can cause Python syntax errors.

The examples above are snippets that illustrate how to use inline annotations. Here is an example that produces a pytest file from Markdown. The command below takes *inline_example.md* and generates *test_inline_example.py*.

```
phmdoctest doc/inline_example.md --outfile test_inline_example.py
```

1.24 skipping blocks with skip option

If you don't want to generate test cases for Python blocks precede the block with a **skip** directive or use the `--skip TEXT` option. More than one **skip** directive or`--skip TEXT`is allowed.

The following describes using `--skip TEXT`. The code in each Python block gets searched for the substring `TEXT`. Zero, one or more blocks will contain the substring. These blocks will not generate test cases in the output file.

- The Python code in the fenced code block gets searched.
- The info string is **not** searched.
- Output blocks are **not** searched.
- Both Python code and session blocks get searched.
- Case is significant.

The report shows which Python blocks get skipped in the test role column, and the Python blocks that matched each –skip TEXT in the skips section.

This option makes it **very easy** to **inadvertently exclude** Python blocks from the test cases. In the event no test cases get generated, the option `--fail-nocode` described below is useful.

Three special `--skip TEXT` strings work a little differently. They select one of the first, second, or last of the Python blocks. Only Python blocks get counted.

- `--skip FIRST` skips the first Python block.
- `--skip SECOND` skips the second Python block.
- `--skip LAST` skips the final Python block.

1.25 skip option

This command using `--skip`:

```
phmdoctest doc/example2.md --skip "Python 3.7" --skip LAST --report --outfile test_
↪example2.py
```

Produces the report

```
            doc/example2.md fenced blocks
-----------------------------------------------------
block     line  test           TEXT or directive
type    number  role           quoted and one per line
```

(continues on next page)

(continued from previous page)

```
-----------------------------------------------------
python       9  code
            14  output
python      20  skip-code     "Python 3.7"
            26  skip-output
            31  --
python      37  code
python      44  code
            51  output
yaml        59  --
text        67  --
py          75  session
python      87  code
            94  output
py         102  skip-session  "LAST"
-----------------------------------------------------
5 test cases.
1 skipped code blocks.
1 skipped interactive session blocks.
1 code blocks with no output block.

  skip pattern matches (blank means no match)
------------------------------------------------
skip pattern  matching code block line number(s)
------------------------------------------------
Python 3.7    20
LAST          102
------------------------------------------------
```

creates the output file *test_example2.py*

1.26 short form of skip option

This is the same command as above using the short `-s` form of the `--skip` option in two places. It produces the same report and outfile.

```
phmdoctest doc/example2.md -s "Python 3.7" -sLAST --report --outfile test_example2.py
```

1.27 fail-nocode option

The `--fail-nocode` option produces a pytest file that will always fail when no Python code or session blocks get found.

Evem if no Python code or session blocks exist in the Markdown file a pytest file gets generated. This also happens when `--skip` eliminates all the Python code blocks. The generated pytest file will have the function `def test_nothing_passes()`.

If the option `--fail-nocode` is passed the function is `def test_nothing_fails()` which raises an assertion.

1.28 setup option

A single Python code block can assign names visible to other code blocks by giving the `--setup TEXT` option. Please see the *setup* directive above. The rules for `TEXT` are the same as for `--skip TEXT` plus…

- Only one block can match `TEXT`.
- The block cannot match a block that is skipped.
- The block cannot be a session block even though session blocks get searched for `TEXT`.
- It is ok if the block has an output block. It will be ignored.

1.29 teardown option

A single Python code block can supply code run by the pytest `teardown_module()` fixture. Use the `--teardown TEXT` option. Please see the *teardown* directive above. The rules for `TEXT` are the same as for `--setup` above except `TEXT` won't match a setup block.

1.30 Setup example

For the Markdown file *setup.md* run this command to see how the blocks get tested.

```
phmdoctest doc/setup.md --setup FIRST --teardown LAST --report
```

```
            doc/setup.md fenced blocks
-------------------------------------------------
block     line  test      TEXT or directive
type    number  role      quoted and one per line
-------------------------------------------------
python       9  setup     "FIRST"
python      20  code
            27  output
python      37  code
            42  output
python      47  code
            51  output
python      58  teardown  "LAST"
-------------------------------------------------
3 test cases.
```

This command

```
phmdoctest doc/setup.md --setup FIRST --teardown LAST --outfile test_setup.py
```

creates the test file *test_setup.py*

1.31 Setup for sessions

The pytest option `--doctest-modules` is required to run doctest on sessions. pytest runs doctests in a separate context. For more on this see *Execution context* below.

To allow sessions to see the variables assigned by the `--setup` code block, add the option `--setup-doctest`

Here is an example with setup code and sessions *setup_doctest.md*. The first part of this file is a copy of setup.md.

This command uses the short form of setup and teardown. -u for set**up** and -d for tear**down**.

```
phmdoctest doc/setup_doctest.md -u FIRST -d LAST --setup-doctest --outfile test_setup_
↪doctest.py
```

It creates the test file *test_setup_doctest.py*

1.32 Execution context

When run without `--setup`

- pytest and doctest determine the order of test case execution.
- phmdoctest assumes test code and session execution is in file order.
- Test case order is not significant.
- Code and expected output run within a function body of a pytest test case.
- If pytest is invoked with `--doctest-modules`:
  - Sessions are run in a separate doctest execution context.
  - Otherwise, sessions do not run.

1.32.1 With `--setup`

- Names assigned by setup code are visible to code blocks.
- Code blocks can modify the objects created by the setup code.
- Code block test case order is significant.
- Session order is not significant.
- If pytest is run with `--doctest-modules`:
  - pytest runs two separate contexts: one for sessions, one for code blocks.
  - setup and teardown code gets run twice, once by each context.
  - the names assigned by the setup code block are `are not` visible to the sessions.

1.32.2 With `share-names`

- Only following code blocks can modify the shared objects.
- Shared objects will **not** be visible to sessions if pytest is run with `--doctest-modules`.
- After running a code block with `clear-names`
  - Shared objects will no longer be visible.
  - Names assigned by setup code will no longer be visible.

1.32.3 With `--setup` and `--setup-doctest`

Same as the setup section plus:

- names assigned by the setup code block are visible to the sessions.
- Sessions can modify the objects created by the setup code.
- Session order is significant.
- Sessions and code blocks are still running in separate contexts isolated from each other.
- A session can't affect a code block, and a code block can't affect a session.
- Names assigned by the setup code block are globally visible to the entire test suite via the pytest doctest_namespace fixture. See hint near the end *Hints*.

1.32.4 pytest live logging demo

The live logging demos reveals pytest execution contexts. pytest Live Logs show the execution order of setup_module(), test cases, sessions, and teardown_module(). There are 2 demo invocations in the workflow action called pytest Live Log Demo. GitHub login required.

1.33 Send outfile to stdout

To redirect the above outfile to the standard output stream use one of these two commands.

Be sure to leave out `--report` when sending –outfile to standard output.

```
phmdoctest doc/example2.md -s "Python 3.7" -sLAST --outfile -
```

or

```
phmdoctest doc/example2.md -s "Python 3.7" -sLAST --outfile=-
```

1.34 Usage

phmdoctest --help

```
Usage: phmdoctest [OPTIONS] MARKDOWN_FILE

  MARKDOWN_FILE may also be .toml, .cfg, or .ini configuration file.

Options:
  --outfile TEXT       Write generated test case file to path TEXT. "-" writes
                       to stdout.

  -s, --skip TEXT      Any Python code or interactive session block that
                       contains the substring TEXT is not tested. More than
                       one --skip TEXT is ok. Double quote if TEXT contains
                       spaces. For example --skip="python 3.7" will skip every
                       Python block that contains the substring "python 3.7".
                       If TEXT is one of the 3 capitalized strings FIRST
                       SECOND LAST the first, second, or last Python code or
                       session block in the Markdown file is skipped.

  --report             Show how the Markdown fenced code blocks are used.

  --fail-nocode        This option sets behavior when the Markdown file has no
                       Python fenced code blocks or interactive session blocks
                       or if all such blocks are skipped. When this option is
                       present the generated pytest file has a test function
                       called test_nothing_fails() that will raise an
                       assertion. If this option is not present the generated
                       pytest file has test_nothing_passes() which will never
                       fail.

  -u, --setup TEXT     The Python code block that contains the substring TEXT
                       is run at test module setup time. Variables assigned at
                       the outer level are visible as globals to the other
                       Python code blocks. TEXT should match exactly one code
                       block. If TEXT is one of the 3 capitalized strings
                       FIRST SECOND LAST the first, second, or last Python
                       code or session block in the Markdown file is matched.
                       A block will not match --setup if it matches --skip, or
                       if it is a session block. Use --setup-doctest below to
                       grant Python sessions access to the globals.

  -d, --teardown TEXT  The Python code block that contains the substring TEXT
                       is run at test module teardown time. TEXT should match
                       exactly one code block. If TEXT is one of the 3
                       capitalized strings FIRST SECOND LAST the first,
                       second, or last Python code or session block in the
                       Markdown file is matched. A block will not match
                       --teardown if it matches either --skip or --setup, or
                       if it is a session block.

--setup-doctest        Make globals created by the --setup Python code block
```

(continues on next page)

(continued from previous page)

```
                      or setup directive visible to session blocks and only
                      when they are tested with the pytest --doctest-modules
                      option.  Please note that pytest runs doctests in a
                      separate context that only runs doctests. This option
                      is ignored if there is no --setup option.

  --version           Show the version and exit.
  --help              Show this message and exit.
```

1.35 Run as a Python module

To run phmdoctest from the command line:

```
python -m phmdoctest doc/example2.md --report
```

1.36 Python API

Call **main.testfile()** to generate a pytest file in memory. Please see the Python API here. The example generates a pytest file from doc/setup.md and compares the result to doc/test_setup.py.

```
from pathlib import Path
import phmdoctest.main

generated_testfile = phmdoctest.main.testfile(
    "doc/setup.md",
    setup="FIRST",
    teardown="LAST",
)
expected = Path("doc/test_setup.py").read_text(encoding="utf-8")
assert expected == generated_testfile
```

1.37 pytest fixtures

Use fixture **testfile_creator** to generate a test file in memory. Pass the test file to fixture **testfile_tester** to run the test file in the pytester environment. Fixture API | *Example*. See more uses in tests/test_examples.py, tests/test_details.py, and tests/test_many_markdown.py. The fixtures run pytest much faster than `run_and_pytest()` below since there is no subprocess call. In the readthedocs documentation see the section Development tools API 1.4.0. pytest's pytester is suitable for pytest plugin development.

1.38 Simulate command line

To simulate a command line call to phmdoctest from within a Python script `phmdoctest.simulator` offers the function `run_and_pytest()`.

- it creates the –outfile in a temporary directory
- optionally runs pytest on the outfile
- pytest can return a JUnit XML report
- useful during development to validate the command line and prevent use of a stale –outfile

Please see the Latest Development tools API section or the docstring of the function `run_and_pytest()` in the file `simulator.py`. Pass pytest_options as a list of strings as shown below.

```
import phmdoctest.simulator

command = "phmdoctest doc/example1.md --report --outfile temporary.py"
simulator_status = phmdoctest.simulator.run_and_pytest(
    well_formed_command=command, pytest_options=["--doctest-modules", "-v"]
)
assert simulator_status.runner_status.exit_code == 0
assert simulator_status.pytest_exit_code == 0
```

1.39 Hints

- To read the Markdown file from the standard input stream. Use `-` for MARKDOWN_FILE.
- Write the test file to a temporary directory so that it is always up to date.
- In CI scripts the following shell command will create the temporary directory **tmp** in the **tests** folder on Windows, Linux, and macOS.

  ```
  python -c "from pathlib import Path; d = Path('tests') / 'tmp'; d.mkdir(mode=0o700)"
  ```

- It is easy to use –output by mistake instead of `--outfile`.
- If Python code block has no output, put assert statements in the code.
- Use pytest option `--doctest-modules` to test the sessions.
- Markdown indented code blocks (Spec section 4.4) are ignored.
- simulator_status.runner_status.exit_code == 2 is the click command line usage error.
- Since phmdoctest generates code, the input file should be from a trusted source.
- An empty code block gets given the role `del-code`. It is not tested.
- Use special TEXT values FIRST, SECOND, LAST for the command line options `--setup` and `--teardown` since they only match one block.
- The variable names `managenamespace`, `doctest_namespace`, `capsys`, and `_phm_expected_str` should not be used in Markdown Python code blocks since they may be used in generated code.
- Setup and teardown code blocks cannot have expected output.
- To have pytest collect a code block with the label directive start the value with `test_`.

- With the `--setup-doctest` option, names assigned by the setup code block are globally visible to the entire test suite. This is due to the scope of the pytest doctest_namespace fixture. Try using a separate pytest command to test just the phmdoctest test.
- The module **phmdoctest.fixture** is imported at pytest time to support setup, teardown, share-names, and clear-names features.
- The phmdoctest Markdown parser finds fenced code blocks enclosed by html `<details>` and `</details>` tags. The tags may require a preceding and trailing blank line to render correctly. See example in tests/test_details.py.
- Try redirecting phmdoctest standard output into PYPI Pygments to colorize the generated test file.

```
python -m phmdoctest project.md --outfile - | pygmentize
```

- If the –outfile is written into a folder that pre-exists in the repository, consider adding the outfile name to .gitignore. If the outfile name later changes, the change will be needed in .gitignore too.

```
# Reserved for generated test file.
tests/test_readme.py
```

1.40 Directive hints

- Only put one of setup, teardown, share-names, or clear-names on a code block.
- Only one block can be setup. Only one block can be teardown.
- The setup or teardown block can’t have an expected output block.
- Label directive does not generate a test case name on setup and teardown blocks.
- Directives displayed in the `--report` start with a dash like this: `-label test_i_ratio`.
- Code generated by Python blocks with setup and teardown directives runs at the pytest fixture `scope="module"` level.
- Code generated by Python blocks with share-names and clear-names directives are **collected** and run by pytest like any other test case.
- A malformed HTML comment ending is bad. Make sure it ends with both dashes like `-->`. Running with `--report` will expose that problem.
- The setup, teardown, share-names, and clear-names directives have logging. To see the log messages, run pytest with the option: `--log-cli-level=DEBUG --color=yes`
- There is no limit to number of blank lines after the directive HTML comment but before the fenced code block.
- The directive `<!--phmdoctest-mark.xfail-->` might be useful as an alternative to `<!--phmdoctest-mark.skip-->` for failing examples.
- The directive `<!--phmdoctest-mark.ATTRIBUTE-->` will not be effective when used with `<!--phmdoctest-setup-->` or `<!--phmdoctest-teardown-->` because pytest marks can only be applied to tests. They have no effect on fixtures. Setup and teardown use fixtures.

1.41 Related projects

- rundoc
- byexample
- sphinx.ext.doctest
- sybil
- doxec
- egtest
- pytest-phmdoctest
- pytest-codeblocks

CHAPTER

TWO

USING A CONFIGURATION FILE

Just pass the configuration filename as the first argument instead of a Markdown file. The other command line options are ignored. The config file may be formatted as '.toml', .ini. or '.cfg'. The phmdoctest configuration section `[tool.phmdoctest]` may be added to pre-existing configuration files.

A separate invocation of pytest is needed to run the generated test files.

Here are some example invocations using a configuration file that has the phmdoctest configuration section. These configuration files are in the phmdoctest repository.

```
phmdoctest pyproject.toml
pytest -v --doctest-modules .gendir-toml

phmdoctest setup.cfg
pytest -v --doctest-modules .gendir-cfg

phmdoctest tox.ini
pytest -v --doctest-modules .gendir-ini
```

You can also pass a configuration file by Python. Look for the Python API in the readthedocs documentation. See the section Development tools API 1.4.0.

This is a good starting point template section for a .toml format file.

```
[tool.phmdoctest]
# https://pypi.org/project/phmdoctest
# Writes pytest files generated from Markdown to output_directory.
# Invoke pytest separately to run the generated pytest files.

markdown_globs = [
    "README.md",
    "doc/*.md",
]
exclude_globs = [
]

output_directory = ".gendir-typical-toml"
print = ["filename", "summary"]
```

- Filenames and globs are relative to the current working directory of the shell that invokes phmdoctest.
- The output_directory can be relative or absolute.

Please see Python standard library pathlib Path.glob(pattern) for glob syntax. The ** glob pattern indicates recursive directory search. We could do the whole repository with (.toml) `markdown_globs = ["**/*.md"]`

The generated test files get written to the directory specified by `output_directory`.

`output_directory` is cleaned of all *.py files before writing new test files. Pre-existing *.py files in the output directory get renamed. If output_directory inadvertently gets pointed at a Python source directory, the renamed files can be recovered by renaming them.

The `markdown_globs` key specifies Markdown files to select for test file generation. The globs may be one per line or comma separated. Comments are OK on separate lines or at the end of a line.

The `exclude_globs` key specifies Markdown files that should not generate test files. Markdown files that don't have any Python examples get automatically excluded.

The `print` key directs printing.

- If `filename` is present the filename prints after test file generation and before writing the generated test file.
- If `summary` is present the number of test files generated is printed last.

To prevent printing everything set `print` like this:

```
# .ini, .cfg
print =

# .toml
print = []
```

Here is an example .cfg format configuration file used for testing this project. The .ini format is the same.

```
# tests/generate.cfg
[tool.phmdoctest]
# https://pypi.org/project/phmdoctest
# Writes pytest files generated from Markdown to output_directory.
# Invoke pytest separately to run the generated pytest files.
markdown_globs =
    # Refer to Python standard library Path.glob(pattern)
    project.md
    doc/*.md
    tests/managenamespace.md  # inline comments are ok
    tests/one_code_block.md
    tests/output_has_blank_lines.md
    tests/setup_only.md
    tests/twentysix_session_blocks.md

exclude_globs =
    # Don't test files matching globs below:
    # Reason- needs command line args.
    doc/setup.md
    doc/setup_doctest.md
    # Reason- contains an already generated test file.
    doc/*_raw.md
    doc/*_py.md
    # Reason- need to register markers to avoid PytestUnknownMarkWarning.
    doc/mark_example.md

output_directory = .gendir-suite-cfg
print = filename, summary
```

This is the equivalent .toml format.

```
# tests/generate.toml
[tool.phmdoctest]
# https://pypi.org/project/phmdoctest
# Writes pytest files generated from Markdown to output_directory.
# Invoke pytest separately to run the generated pytest files.

markdown_globs = [
    # Refer to Python standard library Path.glob(pattern)
    "project.md",
    "doc/*.md",
    "tests/managenamespace.md",
    "tests/one_code_block.md",
    "tests/output_has_blank_lines.md",
    "tests/setup_only.md",
    "tests/twentysix_session_blocks.md",
]
exclude_globs = [
    # Don't test files matching globs below:
    # Reason- needs command line args.
    "doc/setup.md",
    "doc/setup_doctest.md",
    # Reason- contains an already generated test file.
    "doc/*_raw.md",
    "doc/*_py.md",
    # Reason- need to register markers to avoid PytestUnknownMarkWarning.
    "doc/mark_example.md",
]

output_directory = ".gendir-suite-toml"
print = ["filename", "summary"]
```

CHAPTER

THREE

RECENT CHANGES

1.4.0 - 2022-03-19

- Add feature to generate test files using a configuration file.
- Add `<--phmdoctest-mark.ATTRIBUTE-->` directive.
- Add tool to check for Python examples.
- Add tool to prepare a generated test file output directory.
- Bugfix- issue- pytest not required for installation.
- Combined CI script install.yml into ci.yml.
- Close open files in test_readthedocs_python_version().

1.3.0 - 2021-11-08

- Add main.testfile().
- Add testfile_creator and testfile_tester fixtures.
- Bugfix- Issue- Generated test name has output_NN when skip directive on output block.
- Bugfix- Issue- mark.skipif example code causes pytest AST fail at assertion rewrite time. Happens on skipped Python version. Replaced with code that compiles on the skipped version.
- Drop Python 3.6 add Python 3.10.

tests:

- Add mode=0o700 to mkdir() calls in test .yml files.
- Run tests in virtual envirionments. ci.yml.
- Add test_details.py.
- Add Appveyor to CI to show pytest items.
- Rework requirements files. Add tests.
- Refactor new fixtures to conftest.py.
- Rework/refactor quick_links test logic.
- Add test to find trailing spaces in sources.
- Tox no longer used in test suite.

docs:

- Bugfix- Issue- Markdown header level out of sequence.
- Loosen doc dependencies.

- Fenced code block info_string pycon -> py.
- Sphinx with myst_parser for docs.

style:

- Style/pep8/inspection fixes.
- Path and open changes.
- Remove trailing spaces from ~25 files.

1.2.1 - 2021-07-07

- Bugfix- #16, #15, Issue- Simulator subprocess failed on win venv.
- Code Quality fixes: assert –> raise.
- Make fenced code block info_strings compatible with GitHub pages.
- Restored tox.ini.

1.2.0 - 2021-06-09

- Add inline annotations.
- Reformat code style with black.
- Rework setup.py/setup.cfg.
- Remove tox.
- Fix bad example in README.md.

1.1.1 - 2021-05-14

- Bugfix- Pull Request #6, Issue #8 –outfile missing `import pytest`.
- Documentation typo fixes.

1.1.0 - 2021-05-12

- Add test directives taken from HTML comments in .md.
- Implement setup/teardown with Pytest fixtures.
- Use difflib.ndiff to show unexpected output.
- Add simulator feature to return JUnitXML from pytest.

1.0.1 - 2020-12-16

- Bugfix- Issue #4- pytest fails in pypy3 if using –setup, –setup-doctest.
- Removed pytest –strict option since not needed.

1.0.0 - 2020-07-12

- New feature to do setup and teardown code block.

0.1.0 - 2020-06-14

- New feature to handle Python interactive sessions.

0.0.6 - 2020-06-07

- Bugfix- Issue- Skip pattern matching start of code ignored.

0.0.5 - 2020-04-20

- Bugfix- Issue- Won’t fail if Python code block doesn’t print.

- Bugfix- Issue- README CI example missing “install:”.
- Add Development tools API section to the documentation.
- Pin phmdoctest dependency version ranges in setup.py.

0.0.4 - 2020-04-02

- Changes to build documentation on readthedocs.org.
- Inspection fixes.

0.0.3 - 2020-03-18

- Initial upload to Python Package Index.

CHAPTER

FOUR

DEVELOPMENT TOOLS API VERSION 1.4.0

4.1 Generate a pytest file.

`phmdoctest.main.testfile`(*markdown_file: str = '', *, skips: Optional[List[str]] = None, fail_nocode: bool = False, setup: Optional[str] = None, teardown: Optional[str] = None, setup_doctest: bool = False, built_from: str = ''*) → str

Run with callers keyword arguments and default values.

Return a string that contains the generated pytest file. The parameters are described by the command line help. Each string in the list skips is described by the –skip TEXT command line option.

Parameters **`markdown_file`** – Path to the Markdown input file.

Keyword Arguments

- **`skips`** – List[str]. Do not test blocks with substring TEXT.
- **`fail_nocode`** – Markdown file with no code blocks generates a failing test.
- **`setup`** – Run block with substring TEXT at test module setup time.
- **`teardown`** – Run block with substring TEXT at test module teardown time.
- **`setup_doctest`** – Make globals created by the setup Python code block or setup directive visible to Python interactive session >>> blocks. The globals are set at Pytest Session scope and are visible to all tests run by –doctest-modules.
- **`built_from`** – Text that follows "built from" in test file's docstring. When empty string the docstring built from text is derived from markdown_file.

Returns String containing the contents of the generated pytest file.

4.2 Generate pytest files using a configuration file.

`phmdoctest.main.generate_using`(*config_file: pathlib.Path*) → None

Generate test files as directed by configuration file.

See the "Using a configuration file" section of the documentation.

- The *markdown_globs* key specifies Markdown files to select for test file generation.
- The *exclude_globs* key specifies Markdown files that should not generate test files. Markdown files that don't have any Python examples are automatically excluded.
- The generated test files get written to the directory specified by *output_directory*.
- The *print* key directs printing.

Parameters **`config_file`** – Path to the .cfg, .ini, or .toml configuration file.

4.3 Test with Pytest fixtures.

`phmdoctest.tester.`**`testfile_creator`**(*pytestconfig*)

Fixture creates the pytest test file contents from the Markdown file.

A Markdown file, in a folder relative to the pytest command line invocation directory is processed by phmdoctest to create a pytest file which is returned as a string.

This fixture is needed to produce the pytest test file when using the fixture testfile_tester below because pytester changes the current working directory when it is activated.

The fixture injects a function with the following signature. Please consult the source in tester.py. The returned function calls phmdoctest.main.testfile() to generate the pytest test file from Markdown.

Parameters **`markdown_file`** – Path to the Markdown input file. This file name is relative to the working directory of the command that invokes pytest. Typically this is the root of the repository.

Keyword Arguments

- **`skips`** – List[str]. Do not test blocks with substring TEXT.
- **`fail_nocode`** – Markdown file with no code blocks generates a failing test.
- **`setup`** – Run block with substring TEXT at test module setup time.
- **`teardown`** – Run block with substring TEXT at test module teardown time.
- **`setup_doctest`** – Make globals created by the setup Python code block or setup directive visible to Python interactive session >>> blocks. Caution: The globals are set at Pytest Session scope and are visible to all tests run by –doctest-modules.

Returns String containing the contents of the generated pytest file.

`phmdoctest.tester.`**`testfile_tester`**(*pytester*)

Fixture runs pytester.runpytest with the caller's pytest file string.

Stores the caller's pytest test file *contents* in a temporary directory hosted by pytest. Run pytest on it using *pytest_options*. Typically the caller runs testfile_creator to generate the test file. See example usage in the files tests/test_examples.py and tests/test_details.py.

The fixture injects a function with the following signature. Please consult the source in tester.py.

- pytester requires conftest.py in tests folder with pytest_plugins = ["pytester"]
- Requires pytest >= 6.2.

Parameters

- **`contents`** – String containing the contents of a pytest test file.
- **`testfile_name`** – Name given to the test file when it is stored in the pytester temporary directory.
- **`pytest_options`** – List of strings of pytest command line options that are passed to pytester.

Returns pytest RunResult returned by pytester.runpytest().

4.4 Simulate the command line.

`phmdoctest.simulator.run_and_pytest`(*well_formed_command: str*, *pytest_options: Optional[List[str]] = None*, *junit_family: Optional[str] = None*) → phmdoctest.simulator.SimulatorStatus

Simulate a phmdoctest command, optionally run pytest.

If a filename is provided by the `--outfile` option, the command is rewritten replacing the OUTFILE with a path to a temporary directory and a synthesized filename.

To run pytest on an `--outfile`, pass a list of zero or more pytest_options. pytest is run in a subprocess.

The PYPI package pytest must be installed separately since pytest is not required to install phmdoctest. Use this command: `pip install pytest`

Returns SimulatorStatus object. SimulatorStatus.runner_status is the CliRunner.invoke return value.

If an outfile is streamed to stdout a copy of it is found in simulator_status.runner_status.stdout.

If calling run_and_pytest() from a pytest file, try adding the pytest option `--capture=tee-sys` to the command running pytest on the file.

For example on a checkout of phmdoctest the command line:

```
python -m pytest tests -v --capture=tee-sys
```

will print the outputs from the subprocess.run() invocations of pytest on the `--outfile` written to the temporary directory. A wild guess would be that the subprocess inherited changes made to the parent by –capture=tee-sys.

Parameters

- **`well_formed_command`** –
  - starts with phmdoctest
  - followed by MARKDOWN_FILE
  - ends with `--outfile` OUTFILE (if needed)
  - all other options are between MARKDOWN_FILE and `--outfile` for example: `phmdoctest MARKDOWN_FILE --skip FIRST --outfile OUTFILE`
- **`pytest_options`** – List of strings like this: `["--doctest-modules", "-v"]`. Set to empty list to run pytest with no options. Set to None to skip pytest.
- **`junit_family`** – Configures the format of the Pytest generated JUnit XML string returned in SimulatorStatus. The value is used for the Pytest configuration option of the same name. Set to None or the empty string to skip XML generation.

Returns SimulatorStatus containing runner_status, outfile, pytest_exit_code, and generated JUnit XML.

4.5 Read contents of Markdown fenced code blocks.

`class phmdoctest.tool.FCBChooser(markdown_filename: str)`
Select labeled fenced code block from the Markdown file.

`FCBChooser.__init__(markdown_filename: str)`
Gather labelled Markdown fenced code blocks in the file.

Parameters **`markdown_filename`** – Path to the Markdown file as a string.

`FCBChooser.contents(label: str = '')` → str
Return contents of the labeled fenced code block with label.

Parameters **`label`** – Value of label directive placed on the fenced code block in the Markdown file.

Returns Contents of the labeled fenced code block as a string or empty string if the label is not found. Fenced code block strings typically end with a newline.

`class phmdoctest.tool.LabeledFCB(label, line, contents)`
Describes a fenced code block that has a label directive. (collections.namedtuple).

Parameters

- **`label`** – The label directive's value.
- **`line`** – Markdown file line number of block contents.
- **`contents`** – Fenced code block contents.

`phmdoctest.tool.labeled_fenced_code_blocks(markdown_filename: str)` → List[*phmdoctest.tool.LabeledFCB*]
Return Markdown fenced code blocks that have label directives.

Label directives are placed immediately before a fenced code block in the Markdown source file. The directive can be placed before any fenced code block. The label directive is the HTML comment `<!--phmdoctest-label VALUE-->` where VALUE is typically a legal Python identifier. The space must be present before VALUE. The label directive is also used to name the test function in generated code. When used that way, it must be a valid Python identifier. If there is more than one label directive on the block, the label value that occurs earliest in the file is used.

Parameters **`markdown_filename`** – Path to the Markdown file as a string.

Returns

List of LabeledFCB objects.

LabeledFCB is a NamedTuple with these fields:

- label is the value of a label directive placed in a HTML comment before the fenced code block.
- line is the line number in the Markdown file where the block starts.
- contents is the fenced code block contents as a string.

`phmdoctest.tool.fenced_code_blocks(markdown_filename: str)` → List[str]
Return Markdown fenced code block contents as a list of strings.

Parameters **`markdown_filename`** – Path to the Markdown file as a string.

Returns List of strings, one for the contents of each Markdown fenced code block.

`phmdoctest.tool.fenced_block_nodes(fp: IO[str])` → List[commonmark.node.Node]
Get markdown fenced code blocks as list of Node objects.

Deprecation Watch: This function may be labelled as deprecated in a future version of phmdoctest. It returns a data type defined by the commonmark package.

Parameters `fp` – file object returned by open().

Returns List of commonmark.node.Node objects.

4.6 Get elements from test suite JUnit XML output.

`phmdoctest.tool.`**`extract_testsuite`**`(`*junit_xml_string: str*`)` → Tuple[Optional[xml.etree.ElementTree.Element], List[xml.etree.ElementTree.Element]]

Return testsuite tree and list of failing trees from JUnit XML.

Parameters **`junit_xml_string`** – String containing JUnit xml returned by pytest or phmdoctest.simulator.run_and_pytest().

Returns tuple testsuite tree, list of failed test case trees

4.7 Check a Markdown file for Python examples.

`class` `phmdoctest.tool.`**`PythonExamples`**`(`*has_code*, *has_session*`)`

Presence of Python fenced code blocks in Markdown. (collections.namedtuple)

Parameters

- **`has_code`** – True if detected at least one fenced code block with Python code.
- **`has_session`** – True if detected at least one fenced code block with Python interactive session (doctest).

`phmdoctest.tool.`**`detect_python_examples`**`(`*markdown_path: pathlib.Path*`)` → *phmdoctest.tool.PythonExamples*

Return whether .md has any Python highlighted fenced code blocks.

This includes Python code blocks and Python doctest interactive session blocks. These blocks may or may not generate test cases once processed by phmdoctest.test_file() and collected.

- We don't care here if the code block is followed by expected output.
- This logic does not check if the block has any phmdoctest skip, mark.skip, or mark.skipif directives.
- This logic does not check if the block would be skipped by a phmdoctest command line –skip option.

Parameters **`markdown_path`** – pathlib.Path of input Markdown file.

4.8 Prepare directory for generated test files.

`phmdoctest.tool.`**`wipe_testfile_directory`**(*target_dir: pathlib.Path*) → None

Create and/or clean target_dir directory to receive generated testfiles.

Create target_dir if needed for writing generated pytest files. Prevent future use of pre-existing .py files in target_dir.

- The FILENAME.py files found in target_dir are renamed to noFILENAME.sav.
- If a noFILENAME.sav already exists it is not modified.
- Files in target_dir with other extensions are not modified.
- A FILENAME.py pre-existing in target_dir is only renamed and not deleted. This allows for recovery of .py files when target_dir gets pointed by mistake to a directory with Python source files.

Parameters **`target_dir`** – pathlib.Path of destination directory for generated test files.

CHAPTER

FIVE

ABOUT THE DOCUMENTATION

README.md at the project root serves as the:

- project home page
- PYPI long description
- user manual

Design considerations:

- Most text is in the README.
- Python Package Index long description taken from the README.
- README is at the GitHub repository root.
- Examples in the README are fully syntax highlighted.
- Building a static copy of the documentation for offline use.
- No visible raw ReStructured text in the README rendered by GitHub.

5.1 Implementation

- GitHub pages hosts the project website.
- GitHub hosts the repository and renders README.md.
- readthedocs.org hosts the HTML and creates the PDF for offline use.
- Nearly everything is in README.md. These aren't:
  - index.rst - Top level of the Sphinx documentation.
  - about.md - About the documentation (this page).
  - api.rst - Development tools API generated by Sphinx autodoc and napoleon.
  - recent_changes.md
  - CONTRIBUTING.md

5.2 Tools

- GitHub Pages
- Sphinx
- myst_parser

myst_parser enables Sphinx to parse Markdown files.

5.3 Files

These files are at the project root:

- _config.yml
- .readthedocs.yml
- index.rst
- README.md
- conf.py

GitHub page build consumes _config.yml.

Since conf.py is at the project root Sphinx searches the entire project for document source files. Additional **exclude_patterns** keep out unwanted document source files.

The files below in the doc folder are not part of the documentation:

- make_wrapped_examples.py
- livelog.py
- livelog_test_assertion.py
- livelog_bad_session.py

5.3.1 Read the Docs hosting

readthedocs.org hosts the Sphinx documentation. **doc/requirements.txt** lists the build dependencies.

CHAPTER

SIX

CONTRIBUTING

- Create an issue or submit a pull request forked from the develop branch.
- For pull requests please refer to steps 1-6 at the top of Contributing to Simple Icons

Preconditions for pull request merge:

- For bug fixes a test that fails.
- Documentation and test updates for features.

CHAPTER SEVEN

LIST OF EXAMPLES

These pages are referenced by relative links in README.md.

7.1 Snippet of use in GitHub Actions

(This view is bad on github.io since it removes the variable references in the .yml script. View here instead.)

```yaml
jobs:
  os:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.x
      uses: actions/setup-python@v2
      with:
        python-version: 3.x
    - name: Windows Venv
      run: |
        python -m venv ${{ github.workspace }}\env
        ${{ github.workspace }}\env\Scripts\Activate.ps1
        python -m pip --version
      if: startswith(runner.os, 'Windows')
    - name: Linux/macOS Venv
      run: |
        python -m venv ${{ github.workspace }}/env
        source ${{ github.workspace }}/env/bin/activate
        python -m pip --version
      if: startswith(runner.os, 'Linux') || startswith(runner.os, 'macOS')
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        python -m pip install --no-deps "."
        python -m pip install -r requirements.txt
        python -m pip install -r tests/requirements.txt
    - name: Tests
      run: |
        python -c "from pathlib import Path; d = Path('tests') / 'tmp'; d.
↪mkdir(mode=0o700)"
```

(continues on next page)

(continued from previous page)

```
        phmdoctest project.md --report --outfile tests/tmp/test_project.py
        pytest --doctest-modules -vv tests
```

7.2 This is Markdown file example1.md

7.2.1 Interactive Python session (doctest)

```
>>> print("Hello World!")
Hello World!
```

7.2.2 Source Code and terminal output

Code:

```
from enum import Enum

class Floats(Enum):
    APPLES = 1
    CIDER = 2
    CHERRIES = 3
    ADUCK = 4

for floater in Floats:
    print(floater)
```

sample output:

```
Floats.APPLES
Floats.CIDER
Floats.CHERRIES
Floats.ADUCK
```

7.3 This is Markdown file example2.md

7.3.1 Fenced code block expected output block pair.

In order for phmdoctest to work with Python source code and terminal output add print statements to the source code to produce the expected output.

Example code adapted from the Python Tutorial:

```
squares = [1, 4, 9, 16, 25]
print(squares)
```

expected output:

```
[1, 4, 9, 16, 25]
```

7.3.2 Another fenced code block expected output block pair.

Example code adapted from What's new in Python:

```
# Formatted string literals require Python 3.7
name = "Fred"
print(f"He said his name is {name}.")
```

expected output:

```
He said his name is Fred.
```

7.3.3 Here is a second fenced code block with no info string.

```
doesn't have an info string
```

7.3.4 Here are two Python code blocks in a row and one output block at the end.

The first one:

```
a, b = 0, 1
while a < 1000:
    print(a, end=",")
    a, b = b, a + b
```

The second one. This means the preceding code block has no output block.

```
words = ["cat", "window", "defenestrate"]
for w in words:
    print(w, len(w))
```

The expected output block for the second code block:

```
cat 3
window 6
defenestrate 12
```

7.3.5 A fenced code block with yaml info string.

```
dist: xenial
language: python
sudo: false
```

7.3.6 A fenced block with text info string

```
some text
```

7.3.7 A doctest session

Here is a Python interactive session. It is described by the Python Standard Library module doctest. Note there is no need for an empty line at the end of the session.

```
>>> a = "Greetings Planet!"
>>> a
'Greetings Planet!'
>>> b = 12
>>> b
12
```

7.3.8 One more code plus expected output pair.

Example borrowed from Python Standard Library datetime documentation.

```
from datetime import date

d = date.fromordinal(730920)  # 730920th day after 1. 1. 0001
print(d)
```

```
2002-03-11
```

7.3.9 Another doctest session (skipped in test_example2.py)

Example borrowed from Python Standard Library fractions documentation.

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
```

7.4 doc/test_example2.py

```
"""pytest file built from doc/example2.md"""
from phmdoctest.functions import _phm_compare_exact


def test_code_9_output_14(capsys):
    squares = [1, 4, 9, 16, 25]
    print(squares)

    _phm_expected_str = """\
[1, 4, 9, 16, 25]
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_37():
    a, b = 0, 1
    while a < 1000:
        print(a, end=",")
        a, b = b, a + b

    # Caution- no assertions.


def test_code_44_output_51(capsys):
    words = ["cat", "window", "defenestrate"]
    for w in words:
        print(w, len(w))

    _phm_expected_str = """\
cat 3
window 6
defenestrate 12
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def session_00001_line_75():
    r"""
    >>> a = "Greetings Planet!"
    >>> a
    'Greetings Planet!'
    >>> b = 12
    >>> b
    12
    """


def test_code_87_output_94(capsys):
    from datetime import date
```

(continues on next page)

(continued from previous page)

```
    d = date.fromordinal(730920)  # 730920th day after 1. 1. 0001
    print(d)

    _phm_expected_str = """\
2002-03-11
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.5 This is Markdown file directive1.md

Directives are HTML comments and are not rendered. To see the directives press Edit on GitHub and then the Raw button.

7.5.1 skip directive. No test case gets generated.

It is OK to put a directive above pre-existing HTML comments. The HTML comments are not visible in the rendered Markdown.

```
assert False
```

7.5.2 skip directive on an expected output block.

Generates a test case that runs the code block but does not check the expected output.

```
from datetime import date

date.today()
```

```
datetime.date(2021, 4, 18)
```

7.5.3 skip directive on Python session.

No test case gets generated.

```
>>> print("Hello World!")
incorrect expected output should fail
if test case is generated
```

7.5.4 mark.skip directive with label directive.

- Use `mark.skip` on Python code blocks. A test case gets generated with a @pytest.mark.skip() decorator.
- On a code block the label directive gives the function name of the generated test case.

```
print("testing @pytest.mark.skip().")
```

```
incorrect expected output
```

7.5.5 mark.skipif directive.

Use mark.skipif on Python code blocks. A test case gets generated with a @pytest.mark.skipif(. . .) decorator. This test case will only run when Python is version 3.8 or higher. int.as_integer_ratio() is new in Python 3.8.

```
b = 10
print(b.as_integer_ratio())
```

```
(10, 1)
```

7.5.6 label directive on a session.

This will generate a test case called `doctest_print_coffee()`. It does not start with test_ to avoid collection as a test item.

```
>>> print("coffee")
coffee
```

7.6 doc/directive1.md

~~~
# This is Markdown file directive1.md

Directives are HTML comments and are not rendered.
To see the directives press Edit on GitHub and then
the Raw button.

## skip directive. No test case gets generated.
It is OK to put a directive above pre-existing HTML comments.
The HTML comments are not visible
in the rendered Markdown.

<!--phmdoctest-skip-->
<!-- OK if there is more than one HTML comment here -->
<!-- OK if there is a HTML comment here -->
```python
assert False
```
~~~

(continues on next page)

(continued from previous page)

~~~
## skip directive on an expected output block.
Generates a test case that runs the code block but does
not check the expected output.
```python
from datetime import date

date.today()
```

<!--phmdoctest-skip-->
```
datetime.date(2021, 4, 18)
```

## skip directive on Python session.

No test case gets generated.
<!--phmdoctest-skip-->
```py
>>> print("Hello World!")
incorrect expected output should fail
if test case is generated
```

## mark.skip directive with label directive.
- Use `mark.skip` on Python code blocks.
  A test case gets generated with a @pytest.mark.skip()
  decorator.
- On a code block the label directive gives the
  function name of the generated test case.

<!--phmdoctest-mark.skip-->
<!--phmdoctest-label test_mark_skip-->
```python
print("testing @pytest.mark.skip().")
```
```
incorrect expected output
```

## mark.skipif directive.

Use mark.skipif on Python code blocks.
A test case gets generated with a @pytest.mark.skipif(...)
decorator.  This test case will only run when Python
is version 3.8 or higher. int.as_integer_ratio() is new in
Python 3.8.

<!--phmdoctest-label test_i_ratio-->
<!--phmdoctest-mark.skipif<3.8-->
```python
b = 10
~~~

(continues on next page)

(continued from previous page)

~~~
print(b.as_integer_ratio())
```
```
(10, 1)
```


## label directive on a session.
This will generate a test case called `doctest_print_coffee()`.
It does not start with test_ to avoid collection as a test item.
<!--phmdoctest-label doctest_print_coffee-->
```py
>>> print("coffee")
coffee
```
~~~

The above fenced code block contains the contents of a Markdown file. It shows the HTML comments which are not visible in rendered Markdown. It is included in the documentation as an example raw Markdown file.

7.7 doc/directive1_report.txt

```
             doc/directive1.md fenced blocks
---------------------------------------------------------
block     line  test          TEXT or directive
type    number  role          quoted and one per line
---------------------------------------------------------
python      16  skip-code     -skip
python      23  code
            30  skip-output   -skip
py          38  skip-session  -skip
python      53  code          -mark.skip
                              -label test_mark_skip
            56  output
python      70  code          -label test_i_ratio
                              -mark.skipif<3.8
            74  output
py          82  session       -label doctest_print_coffee
---------------------------------------------------------
4 test cases.
1 skipped code blocks.
1 skipped interactive session blocks.
1 code blocks with no output block.
```

The above fenced code block contains the contents of a plain text file. It is included in the documentation as an example text file.

7.8 doc/test_directive1.py

```python
"""pytest file built from doc/directive1.md"""
import sys

import pytest

from phmdoctest.functions import _phm_compare_exact


def test_code_23():
    from datetime import date

    date.today()

    # Caution- no assertions.


@pytest.mark.skip()
def test_mark_skip(capsys):
    print("testing @pytest.mark.skip().")

    _phm_expected_str = """\
incorrect expected output
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


@pytest.mark.skipif(sys.version_info < (3, 8), reason="requires >=py3.8")
def test_i_ratio(capsys):
    b = 10
    print(b.as_integer_ratio())

    _phm_expected_str = """\
(10, 1)
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def doctest_print_coffee():
    r"""
    >>> print("coffee")
    coffee
    """
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.9 This is Markdown file directive2.md

Directives are HTML comments and are not rendered. To see the directives press Edit on GitHub and then the Raw button.

7.9.1 This will be marked as the setup code.

The setup logic makes the names assigned here global to the test module. The code assigns the **names** math, mylist, a, b, and the function doubler(). Setup code does not have an output block. Note the `<!--phmdoctest-setup-->` directive in the Markdown file.

```
import math

mylist = [1, 2, 3]
a, b = 10, 11

def doubler(x):
    return x * 2
```

7.9.2 This test case shows the setup names are visible.

```
print("math.pi=", round(math.pi, 3))
print(mylist)
print(a, b)
print("doubler(16)=", doubler(16))
```

expected output:

```
math.pi= 3.142
[1, 2, 3]
10 11
doubler(16)= 32
```

7.9.3 This test case modifies mylist.

The objects created by the –setup code can be modified and blocks run afterward will see the changes.

```
mylist.append(4)
print(mylist)
```

expected output:

```
[1, 2, 3, 4]
```

7.9.4 This test case sees the modified mylist.

```
print(mylist == [1, 2, 3, 4])
```

expected output:

```
True
```

7.9.5 This will be marked as the teardown code.

Teardown code does not have an output block. Note `<!--phmdoctest-teardown-->` directive in the Markdown file.

```
mylist.clear()
assert not mylist, "mylist was not emptied"
```

7.10 doc/directive2.md

~~~
# This is Markdown file directive2.md

Directives are HTML comments and are not rendered.
To see the directives press Edit on GitHub and then
the Raw button.

## This will be marked as the setup code.
The setup logic makes the names assigned here global to the test module.
The code assigns the **names** math, mylist, a, b, and the function doubler().
Setup code does not have an output block.
Note the `<!--phmdoctest-setup-->` directive in the Markdown file.
<!--phmdoctest-setup-->
```python
import math

mylist = [1, 2, 3]
a, b = 10, 11

def doubler(x):
    return x * 2
```


## This test case shows the setup names are visible.
```python
print("math.pi=", round(math.pi, 3))
print(mylist)
print(a, b)
print("doubler(16)=", doubler(16))
```
expected output:
```
math.pi= 3.142
~~~

(continues on next page)

(continued from previous page)

~~~
[1, 2, 3]
10 11
doubler(16)= 32
```

## This test case modifies mylist.
The objects created by the --setup code can be modified
and blocks run afterward will see the changes.
```python
mylist.append(4)
print(mylist)
```
expected output:
```
[1, 2, 3, 4]
```

## This test case sees the modified mylist.
```python
print(mylist == [1, 2, 3, 4])
```
expected output:
```
True
```

## This will be marked as the teardown code.
Teardown code does not have an output block.
Note `<!--phmdoctest-teardown-->` directive in the Markdown file.
<!--phmdoctest-teardown-->
```python
mylist.clear()
assert not mylist, "mylist was not emptied"
```
~~~

The above fenced code block contains the contents of a Markdown file. It shows the HTML comments which are not visible in rendered Markdown. It is included in the documentation as an example raw Markdown file.

7.11 doc/directive2_report.txt

```
         doc/directive2.md fenced blocks
--------------------------------------------------
block     line  test      TEXT or directive
type    number  role      quoted and one per line
--------------------------------------------------
python      14  setup     -setup
python      25  code
            32  output
python      42  code
            47  output
```

(continues on next page)

(continued from previous page)

```
python      52  code
            56  output
python      64  teardown  -teardown
-------------------------------------------------
3 test cases.
```

The above fenced code block contains the contents of a plain text file. It is included in the documentation as an example text file.

7.12 doc/test_directive2.py

```
"""pytest file built from doc/directive2.md"""
import pytest

from phmdoctest.fixture import managenamespace
from phmdoctest.functions import _phm_compare_exact


@pytest.fixture(scope="module")
def _phm_setup_teardown(managenamespace):
    # setup code line 14.
    import math

    mylist = [1, 2, 3]
    a, b = 10, 11

    def doubler(x):
        return x * 2

    managenamespace(operation="update", additions=locals())
    yield
    # teardown code line 64.
    mylist.clear()
    assert not mylist, "mylist was not emptied"

    managenamespace(operation="clear")


pytestmark = pytest.mark.usefixtures("_phm_setup_teardown")


def test_code_25_output_32(capsys):
    print("math.pi=", round(math.pi, 3))
    print(mylist)
    print(a, b)
    print("doubler(16)=", doubler(16))

    _phm_expected_str = """\
math.pi= 3.142
[1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
10 11
doubler(16)= 32
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_42_output_47(capsys):
    mylist.append(4)
    print(mylist)

    _phm_expected_str = """\
[1, 2, 3, 4]
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_52_output_56(capsys):
    print(mylist == [1, 2, 3, 4])

    _phm_expected_str = """\
True
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.13 This is Markdown file directive3.md

Directives are HTML comments and are not rendered. To see the directives press Edit on GitHub and then the Raw button.

7.13.1 share-names and clear-names directives.

First a normal test case with no directives. This generates a test case. The name `not_shared` is local to the function test_code_13_output_17().

```
not_shared = "Hello World!"
print(not_shared)
```

```
Hello World!
```

This verifies `not_shared` is not visible.

```
try:
    print(not_shared)
except NameError:
    pass
else:
    assert False, "did not get expected NameError"
```

7.13.2 Share the names assigned here with later Python code blocks.

The share-names directive makes the names assigned here global to the test module. The names are visible to all Python code blocks occurring later in the Markdown source file. The code assigns the names string, x, y, z, grades, and the function incrementer(). Place the `<!--phmdoctest-share-names-->` directive in the Markdown file.

```
import string

x, y, z = 77, 88, 99

def incrementer(x):
    return x + 1

grades = ["A", "B", "C"]
```

7.13.3 This test case shows the shared names are visible.

```
print("string.digits=", string.digits)
print(incrementer(10))
print(grades)
print(x, y, z)
```

expected output:

```
string.digits= 0123456789
11
['A', 'B', 'C']
77 88 99
```

7.13.4 This test case modifies grades.

The objects created by the share-names code block can be modified and blocks run afterward will see the changes.

```
grades.append("D")
```

7.13.5 This test case sees the modified grades.

```
print(grades == ["A", "B", "C", "D"])
```

expected output:

```
True
```

7.13.6 This test case shares another name.

```
hex_digits = string.hexdigits
print(hex_digits)
```

A Python block with the share-names directive can have an output block.

```
0123456789abcdefABCDEF
```

7.13.7 Use clear-names directive to un-share.

First notice that hex_digits shared by the last test case is visible. The clear-names directive un-shares any previously shared names. The names will no longer be visible to Python code blocks occurring later in the Markdown source file. The clearing does not happen until after the test case runs. This test case is the same as the previous test case to show that mylist is still visible.

```
print("Names are cleared after the code runs.")
print(grades == ["A", "B", "C", "D"])
print(hex_digits)
```

expected output:

```
Names are cleared after the code runs.
True
0123456789abcdefABCDEF
```

Here we show that grades and digits are no longer visible.

```
try:
    print(grades)
except NameError:
    pass
else:
    assert False, "expected NameError for grades"
try:
    print(hex_digits)
except NameError:
    pass
else:
    assert False, "expected NameError for hex_digits"
```

7.14 doc/directive3.md

```
# This is Markdown file directive3.md

Directives are HTML comments and are not rendered.
To see the directives press Edit on GitHub and then
the Raw button.

## share-names and clear-names directives.
```

(continues on next page)

(continued from previous page)

~~~
First a normal test case with no directives.
This generates a test case.  The name `not_shared` is local to
the function test_code_13_output_17().
```python
not_shared = "Hello World!"
print(not_shared)
```
```
Hello World!
```

This verifies `not_shared` is not visible.
<!--phmdoctest-label test_not_visible-->
```python
try:
    print(not_shared)
except NameError:
    pass
else:
    assert False, "did not get expected NameError"
```

## Share the names assigned here with later Python code blocks.
The share-names directive makes the names assigned here
global to the test module.  The names are visible to all Python code blocks
occurring later in the Markdown source file. The code assigns the
names string, x, y, z, grades, and the function incrementer().
Place the `<!--phmdoctest-share-names-->` directive in the Markdown file.

<!--phmdoctest-label test_directive_share_names-->
<!--phmdoctest-share-names-->
```python
import string

x, y, z = 77, 88, 99

def incrementer(x):
    return x + 1

grades = ["A", "B", "C"]
```

## This test case shows the shared names are visible.
```python
print("string.digits=", string.digits)
print(incrementer(10))
print(grades)
print(x, y, z)
```
expected output:
```
~~~

(continues on next page)

(continued from previous page)

~~~
string.digits= 0123456789
11
['A', 'B', 'C']
77 88 99
```

## This test case modifies grades.
The objects created by the share-names code block can be modified
and blocks run afterward will see the changes.
```python
grades.append("D")
```

## This test case sees the modified grades.
```python
print(grades == ["A", "B", "C", "D"])
```
expected output:
```
True
```

## This test case shares another name.
<!--phmdoctest-share-names-->
```python
hex_digits = string.hexdigits
print(hex_digits)
```

A Python block with the share-names directive can
have an output block.

```
0123456789abcdefABCDEF
```

## Use clear-names directive to un-share.

First notice that hex_digits shared by the last test case
is visible.
The clear-names directive un-shares any previously shared names.
The names will no longer be visible to Python code
blocks occurring later in the Markdown source file.
The clearing does not happen until after the test case runs.
This test case is the same as the previous test case to show
that mylist is still visible.
<!--phmdoctest-clear-names-->
```python
print("Names are cleared after the code runs.")
print(grades == ["A", "B", "C", "D"])
print(hex_digits)
```
~~~

(continues on next page)

(continued from previous page)

~~~
expected output:
```
Names are cleared after the code runs.
True
0123456789abcdefABCDEF
```

Here we show that grades and digits are no longer visible.
```python
try:
    print(grades)
except NameError:
    pass
else:
    assert False, "expected NameError for grades"
try:
    print(hex_digits)
except NameError:
    pass
else:
    assert False, "expected NameError for hex_digits"
```
~~~

The above fenced code block contains the contents of a Markdown file. It shows the HTML comments which are not visible in rendered Markdown. It is included in the documentation as an example raw Markdown file.

7.15 doc/directive3_report.txt

```
             doc/directive3.md fenced blocks
---------------------------------------------------------
block     line  test    TEXT or directive
type    number  role    quoted and one per line
---------------------------------------------------------
python      13  code
            17  output
python      23  code    -label test_not_visible
python      41  code    -label test_directive_share_names
                        -share-names
python      53  code
            60  output
python      70  code
python      75  code
            79  output
python      85  code    -share-names
            93  output
python     108  code    -clear-names
           114  output
python     121  code
---------------------------------------------------------
9 test cases.
```

(continues on next page)

(continued from previous page)

```
4 code blocks with no output block.
```

The above fenced code block contains the contents of a plain text file. It is included in the documentation as an example text file.

7.16 doc/test_directive3.py

```python
"""pytest file built from doc/directive3.md"""
import pytest

from phmdoctest.fixture import managenamespace
from phmdoctest.functions import _phm_compare_exact


def test_code_13_output_17(capsys):
    not_shared = "Hello World!"
    print(not_shared)

    _phm_expected_str = """\
Hello World!
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_not_visible():
    try:
        print(not_shared)
    except NameError:
        pass
    else:
        assert False, "did not get expected NameError"

    # Caution- no assertions.


def test_directive_share_names(managenamespace):
    import string

    x, y, z = 77, 88, 99

    def incrementer(x):
        return x + 1

    grades = ["A", "B", "C"]

    # Caution- no assertions.
    managenamespace(operation="update", additions=locals())


def test_code_53_output_60(capsys):
```

(continues on next page)

(continued from previous page)

```
    print("string.digits=", string.digits)
    print(incrementer(10))
    print(grades)
    print(x, y, z)

    _phm_expected_str = """\
string.digits= 0123456789
11
['A', 'B', 'C']
77 88 99
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_70():
    grades.append("D")

    # Caution- no assertions.


def test_code_75_output_79(capsys):
    print(grades == ["A", "B", "C", "D"])

    _phm_expected_str = """\
True
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_85_output_93(capsys, managenamespace):
    hex_digits = string.hexdigits
    print(hex_digits)

    _phm_expected_str = """\
0123456789abcdefABCDEF
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
    managenamespace(operation="update", additions=locals())


def test_code_108_output_114(capsys, managenamespace):
    print("Names are cleared after the code runs.")
    print(grades == ["A", "B", "C", "D"])
    print(hex_digits)

    _phm_expected_str = """\
Names are cleared after the code runs.
True
0123456789abcdefABCDEF
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
    managenamespace(operation="clear")
```

(continues on next page)

(continued from previous page)

```python


def test_code_121():
    try:
        print(grades)
    except NameError:
        pass
    else:
        assert False, "expected NameError for grades"
    try:
        print(hex_digits)
    except NameError:
        pass
    else:
        assert False, "expected NameError for hex_digits"

    # Caution- no assertions.
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.17 This is file doc/my_markdown_file.md

```
The label directive can be placed on any fenced code block.
```

7.18 Examples of code and session blocks

This file (project.md) has some example code and session blocks including a doctest directive example.

7.18.1 An example with a blank line in the output

Note no directive in the output block of a Python code block output block pair.

```python
def greeting(name: str) -> str:
    return 'Hello' + '\n\n' + name
print(greeting('World!'))
```

Here is the output it produces.

```
Hello

World!
```

7.18.2 Interactive Python session requires <BLANKINE> in the expected output

Blank lines in the expected output must be replaced with <BLANKLINE>. To see the <BLANKLINE> navigate to project.md unrendered.

```
>>> print('Hello\n\nWorld!')
Hello

World!
```

7.18.3 Interactive Python session with doctest directive

Here is an interactive Python session showing an expected exception and use of the doctest directive IGNORE_EXCEPTION_DETAIL. To see the doctest directive navigate to project.md unrendered.

```
>>> int('def')
Traceback (most recent call last):
    ...
ValueError:
```

7.18.4 Session with py as the fenced code block info_string

```
>>> coffee = 5
>>> coding = 5
>>> enjoyment = 10
>>> print(coffee + coding)
10
>>> coffee + coding == enjoyment
True
```

7.19 This is Markdown file setup.md

7.19.1 This will be the setup code.

The setup logic makes the names assigned here global to the test module. The code assigns the **names** math, mylist, a, b, and the function doubler(). Use phmdoctest –setup FIRST to select it. Setup code does not have an output block.

```
import math

mylist = [1, 2, 3]
a, b = 10, 11

def doubler(x):
    return x * 2
```

7.19.2 This test case shows the setup names are visible

```
print("math.pi=", round(math.pi, 3))
print(mylist)
print(a, b)
print("doubler(16)=", doubler(16))
```

expected output:

```
math.pi= 3.142
[1, 2, 3]
10 11
doubler(16)= 32
```

7.19.3 This test case modifies mylist.

The objects created by the –setup code can be modified and blocks run afterward will see the changes.

```
mylist.append(4)
print(mylist)
```

expected output:

```
[1, 2, 3, 4]
```

7.19.4 This test case sees the modified mylist.

```
print(mylist == [1, 2, 3, 4])
```

expected output:

```
True
```

7.19.5 This will be specified as the teardown code.

Use phmdoctest –teardown LAST to select it. Teardown code does not have an output block.

```
mylist.clear()
assert not mylist, "mylist was not emptied"
```

7.20 doc/test_setup.py

```
"""pytest file built from doc/setup.md"""
import pytest

from phmdoctest.fixture import managenamespace
from phmdoctest.functions import _phm_compare_exact


@pytest.fixture(scope="module")
def _phm_setup_teardown(managenamespace):
    # setup code line 9.
    import math

    mylist = [1, 2, 3]
    a, b = 10, 11

    def doubler(x):
        return x * 2

    managenamespace(operation="update", additions=locals())
    yield
    # teardown code line 58.
    mylist.clear()
    assert not mylist, "mylist was not emptied"

    managenamespace(operation="clear")


pytestmark = pytest.mark.usefixtures("_phm_setup_teardown")


def test_code_20_output_27(capsys):
    print("math.pi=", round(math.pi, 3))
    print(mylist)
    print(a, b)
    print("doubler(16)=", doubler(16))

    _phm_expected_str = """\
math.pi= 3.142
[1, 2, 3]
10 11
doubler(16)= 32
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_37_output_42(capsys):
    mylist.append(4)
    print(mylist)

    _phm_expected_str = """\
```

(continues on next page)

(continued from previous page)

```
[1, 2, 3, 4]
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_47_output_51(capsys):
    print(mylist == [1, 2, 3, 4])

    _phm_expected_str = """\
True
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.21 This is Markdown file setup_doctest.md

7.21.1 This will be the setup code.

The setup logic makes the names assigned here global to the test module. The code assigns the **names** math, mylist, a, b, and the function doubler(). Use phmdoctest –setup FIRST to select it. Setup code does not have an output block.

```
import math

mylist = [1, 2, 3]
a, b = 10, 11

def doubler(x):
    return x * 2
```

7.21.2 This test case shows the setup names are visible

```
print("math.pi=", round(math.pi, 3))
print(mylist)
print(a, b)
print("doubler(16)=", doubler(16))
```

expected output:

```
math.pi= 3.142
[1, 2, 3]
10 11
doubler(16)= 32
```

7.21.3 This test case modifies mylist.

The objects created by the –setup code can be modified and blocks run afterward will see the changes.

```
mylist.append(4)
print(mylist)
```

expected output:

```
[1, 2, 3, 4]
```

7.21.4 The next test case sees the modified mylist.

```
print(mylist == [1, 2, 3, 4])
```

expected output:

```
True
```

7.21.5 The names created by the setup code are optionally visible to sessions.

When running phmdoctest setup names become visible to sessions by using these options:

- –setup specifies a code block that initializes variables.
- –setup-doctest injects the setup variables into the doctest namespace.

Run the generated test file with pytest.

- Specify –doctest-modules to run the sessions.
- Sessions run in a separate context from the Python code/output block pairs. The setup and teardown get repeated.

The value 55 is appended to mylist. Note that the 4 appended by the test case above is not there. This is because the sessions run in a separate context.

```
>>> mylist.append(55)
>>> mylist
[1, 2, 3, 55]
```

The change to mylist made in the session above is visible.

```
>>> mylist
[1, 2, 3, 55]
>>> round(math.pi, 3)
3.142
```

7.21.6 This will be specified as the teardown code.

Use phmdoctest –teardown LAST to select it. Teardown code does not have an output block.

```
mylist.clear()
assert not mylist, "mylist was not emptied"
```

7.22 doc/test_setup_doctest.py

```
"""pytest file built from doc/setup_doctest.md"""
import pytest

from phmdoctest.fixture import managenamespace
from phmdoctest.functions import _phm_compare_exact


@pytest.fixture(scope="module")
def _phm_setup_doctest_teardown(doctest_namespace, managenamespace):
    # setup code line 9.
    import math

    mylist = [1, 2, 3]
    a, b = 10, 11

    def doubler(x):
        return x * 2

    managenamespace(operation="update", additions=locals())
    # update doctest namespace
    additions = managenamespace(operation="copy")
    for k, v in additions.items():
        doctest_namespace[k] = v
    yield
    # teardown code line 86.
    mylist.clear()
    assert not mylist, "mylist was not emptied"

    managenamespace(operation="clear")


pytestmark = pytest.mark.usefixtures("_phm_setup_doctest_teardown")


@pytest.fixture()
def populate_doctest_namespace(doctest_namespace, managenamespace):
    additions = managenamespace(operation="copy")
    for k, v in additions.items():
        doctest_namespace[k] = v


def session_00000():
```

(continues on next page)

(continued from previous page)

```
    r"""
    >>> getfixture('populate_doctest_namespace')
    """


def test_code_20_output_27(capsys):
    print("math.pi=", round(math.pi, 3))
    print(mylist)
    print(a, b)
    print("doubler(16)=", doubler(16))

    _phm_expected_str = """\
math.pi= 3.142
[1, 2, 3]
10 11
doubler(16)= 32
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_37_output_42(capsys):
    mylist.append(4)
    print(mylist)

    _phm_expected_str = """\
[1, 2, 3, 4]
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_47_output_51(capsys):
    print(mylist == [1, 2, 3, 4])

    _phm_expected_str = """\
True
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def session_00001_line_69():
    r"""
    >>> mylist.append(55)
    >>> mylist
    [1, 2, 3, 55]
    """


def session_00002_line_76():
    r"""
    >>> mylist
    [1, 2, 3, 55]
    >>> round(math.pi, 3)
```

(continues on next page)

(continued from previous page)

```
    3.142
    """
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.23 This is Markdown file mark_example.md

7.23.1 Fenced code block expected output block pair.

Example code adapted from the Python Tutorial:

```
squares = [1, 4, 9, 16, 25]
print(squares)
```

expected output:

```
[1, 4, 9, 16, 25]
```

7.23.2 The code block has 2 directives:

- phmdoctest-label test_datetime
- phmdoctest-mark.slow

The first directive names the generated test function.

The second directive add @pytest.mark.slow decorator. slow is a pytest user defined marker that is used to select/deselect test cases using the pytest –marker command line option.

```
from datetime import date

d = date.fromordinal(730920)  # 730920th day after 1. 1. 0001
print(d)
```

```
2002-03-11
```

7.23.3 A doctest session

Example borrowed from Python Standard Library fractions documentation.

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
```

7.24 doc/mark_example.md

~~~
# This is Markdown file mark_example.md
## Fenced code block expected output block pair.

Example code adapted from the Python Tutorial:
```python
squares = [1, 4, 9, 16, 25]
print(squares)
```
expected output:
```
[1, 4, 9, 16, 25]
```

## The code block has 2 directives:

- phmdoctest-label test_datetime
- phmdoctest-mark.slow

The first directive names the generated test function.

The second directive add @pytest.mark.slow decorator. slow is
a pytest user defined marker that is used to select/deselect
test cases using the pytest --marker command line option.

<!--phmdoctest-label test_datetime-->
<!--phmdoctest-mark.slow-->
```python
from datetime import date

d = date.fromordinal(730920)  # 730920th day after 1. 1. 0001
print(d)
```

```
2002-03-11
```

## A doctest session

Example borrowed from Python Standard Library
fractions documentation.

```py
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
~~~

(continues on next page)

(continued from previous page)

~~~
>>> Fraction('3/7')
Fraction(3, 7)
```
~~~

The above fenced code block contains the contents of a Markdown file. It shows the HTML comments which are not visible in rendered Markdown. It is included in the documentation as an example raw Markdown file.

7.25 doc/test_mark_example.py

```python
"""pytest file built from doc/mark_example.md"""
import pytest

from phmdoctest.functions import _phm_compare_exact


def test_code_6_output_11(capsys):
    squares = [1, 4, 9, 16, 25]
    print(squares)

    _phm_expected_str = """\
[1, 4, 9, 16, 25]
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


@pytest.mark.slow
def test_datetime(capsys):
    from datetime import date

    d = date.fromordinal(730920)  # 730920th day after 1. 1. 0001
    print(d)

    _phm_expected_str = """\
2002-03-11
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def session_00001_line_44():
    r"""
    >>> from fractions import Fraction
    >>> Fraction(16, -10)
    Fraction(-8, 5)
    >>> Fraction(123)
    Fraction(123, 1)
    >>> Fraction()
    Fraction(0, 1)
    >>> Fraction('3/7')
    Fraction(3, 7)
    """
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.26 This is Markdown file inline_example.md

To comment out sections of Python code blocks use inline annotations.

- phmdoctest:pass
- phmdoctest:omit

This example shows use of phmdoctest:omit to comment out one line at a time in two places.

```
def cause_assertion():
    print("before assert...")
    assert False                  # phmdoctest:omit
    print("after assert.")
    print("bye")  # phmdoctest:omit

cause_assertion()
```

Expected output:

```
before assert...
after assert.
```

This example shows use of phmdoctest:omit to comment out an indented section.

```
def prints_too_much(condition):
    print("called with", condition)
    if condition:                 # phmdoctest:omit
        print("-" * 50)
        # note the section continues across blank lines

        print("=" * 50)
        print("*" * 50)

    # Can't use phmdoctest:omit on the next line because
    # the else: line would get a Python SyntaxError.
    if condition:
        # So use phmdoctest:pass on the next line.
        print("condition is true")  # phmdoctest:pass
    else:
        print("condition is false")
    print("done")

prints_too_much(True)
prints_too_much(False)
```

Expected output:

```
called with True
done
called with False
```

(continues on next page)

(continued from previous page)

```
condition is false
done
```

7.27 doc/test_inline_example.py

```
"""pytest file built from doc/inline_example.md"""
from phmdoctest.functions import _phm_compare_exact


def test_code_11_output_21_2(capsys):
    def cause_assertion():
        print("before assert...")
        # assert False                   # phmdoctest:omit
        print("after assert.")
        # print("bye")  # phmdoctest:omit

    cause_assertion()

    _phm_expected_str = """\
before assert...
after assert.
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)


def test_code_29_output_52_2(capsys):
    def prints_too_much(condition):
        print("called with", condition)
        # if condition:              # phmdoctest:omit
        #     print("-" * 50)
        #     # note the section continues across blank lines
        #
        #     print("=" * 50)
        #     print("*" * 50)

        # Can't use phmdoctest:omit on the next line because
        # the else: line would get a Python SyntaxError.
        if condition:
            # So use phmdoctest:pass on the next line.
            pass  # print("condition is true")  # phmdoctest:pass
        else:
            print("condition is false")
        print("done")

    prints_too_much(True)
    prints_too_much(False)

    _phm_expected_str = """\
called with True
done
```

(continues on next page)

(continued from previous page)

```
called with False
condition is false
done
"""
    _phm_compare_exact(a=_phm_expected_str, b=capsys.readouterr().out)
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

7.28 tests/project_test.py

```
"""Example pytest usage of testfile_creator and testfile_tester fixtures.

pytester requires conftest.py in tests folder with pytest_plugins = [“pytester”]
Requires pytest >= 6.2.
"""

from phmdoctest.tester import testfile_creator
from phmdoctest.tester import testfile_tester


def test_generate_run_project(testfile_creator, testfile_tester):
    """Generate pytest file from project.md and run it with pytester."""
    testfile = testfile_creator("project.md")
    result = testfile_tester(
        contents=testfile, pytest_options=["-v", "--doctest-modules"]
    )
    result.assert_outcomes(passed=4)
```

The above syntax highlighted fenced code block contains the contents of a python source file. It is included in the documentation as an example python file.

CHAPTER EIGHT

SEARCH

- search

PYTHON MODULE INDEX

INDEX