testbook Documentation

Release 0.4.1

nteract team

May 08, 2021

CONTENTS

testbook is a unit testing framework for testing code in Jupyter Notebooks.

Previous attempts at unit testing notebooks involved writing the tests in the notebook itself. However, testbook will allow for unit tests to be run against notebooks in separate test files, hence treating `.ipynb` files as `.py` files.

Here is an example of a unit test written using testbook

Consider the following code cell in a Jupyter Notebook:

```
def func(a, b):
    return a + b
```

You would write a unit test using `testbook` in a Python file as follows:

```
from testbook import testbook


@testbook('/path/to/notebook.ipynb', execute=True)
def test_func(tb):
    func = tb.ref("func")

    assert func(1, 2) == 3
```

CHAPTER

ONE

FEATURES

- Write conventional unit tests for Jupyter Notebooks
- Execute all or some specific cells before unit test
- Share kernel context across multiple tests (using pytest fixtures)
- Support for patching objects
- Inject code into Jupyter notebooks
- Works with any unit testing library - unittest, pytest or nose

CHAPTER

TWO

DOCUMENTATION

2.1 Installation and Getting Started

`testbook` is a unit testing framework for testing code in Jupyter Notebooks.

2.1.1 Installing `testbook`

```
pip install testbook
```

2.1.2 Create your first test

Consider the following code cell in a Jupyter Notebook,

```
def foo(x):
    return x + 1
```

Here is the unit test for it which must be written in a Python module (`.py` file).

```
from testbook import testbook

@testbook('/path/to/notebook.ipynb', execute=True)
def test_foo(tb):
    foo = tb.ref("foo")

    assert foo(2) == 3
```

That's it! You can now execute the test.

2.1.3 General workflow when using testbook to write a unit test

1. Use `testbook.testbook` as a decorator or context manager to specify the path to the Jupyter Notebook. Passing `execute=True` will execute all the cells, and passing `execute=['cell-tag-1', 'cell-tag-2']` will only execute specific cells identified by cell tags.
2. Obtain references to objects under test using the `.ref` method.
3. Write the test!

2.2 Usage

The motivation behind creating testbook was to be able to write conventional unit tests for Jupyter Notebooks.

2.2.1 How it works

Testbook achieves conventional unit tests to be written by setting up references to variables/functions/classes in the Jupyter Notebook. All interactions with these reference objects are internally "pushed down" into the kernel, which is where it gets executed.

2.2.2 Set up Jupyter Notebook under test

Decorator and context manager pattern

These patterns are interchangeable in most cases. If there are nested decorators on your unit test function, consider using the context manager pattern instead.

- Decorator pattern

```
from testbook import testbook

@testbook('/path/to/notebook.ipynb', execute=True)
def test_func(tb):
    func = tb.ref("func")

    assert func(1, 2) == 3
```

- Context manager pattern

```
from testbook import testbook

def test_func():
    with testbook('/path/to/notebook.ipynb', execute=True) as tb:
        func = tb.ref("func")

        assert func(1, 2) == 3
```

Using `execute` to control which cells are executed before test

You may also choose to execute all or some cells:

- Pass `execute=True` to execute the entire notebook before the test. In this case, it might be better to set up a *module scoped pytest fixture*.
- Pass `execute=['cell1', 'cell2']` or `execute='cell1'` to only execute the specified cell(s) before the test.
- Pass `execute=slice('start-cell', 'end-cell')` or `execute=range(2, 10)` to execute all cells in the specified range.

2.2.3 Obtain references to objects present in notebook

Testing functions in Jupyter Notebook

Consider the following code cell in a Jupyter Notebook:

```
def foo(name):
    return f"You passed {name}!"

my_list = ['spam', 'eggs']
```

Reference objects to functions can be called with,

- explicit JSON serializable values (like `dict`, `list`, `int`, `float`, `str`, `bool`, etc)
- other reference objects

```
@testbook.testbook('/path/to/notebook.ipynb', execute=True)
def test_foo(tb):
    foo = tb.ref("foo")

    # passing in explicitly
    assert foo(['spam', 'eggs']) == "You passed ['spam', 'eggs']!"

    # passing in reference object as arg
    my_list = tb.ref("my_list")
    assert foo(my_list) == "You passed ['spam', 'eggs']!"
```

Testing function/class returning a non-serializable value

Consider the following code cell in a Jupyter Notebook:

```
class Foo:
    def __init__(self):
        self.name = name

    def say_hello(self):
        return f"Hello {self.name}!"
```

When `Foo` is instantiated from the test, the return value will be a reference object which stores a reference to the non-serializable `Foo` object.

```
@testbook.testbook('/path/to/notebook.ipynb', execute=True)
def test_say_hello(tb):
    Foo = tb.ref("Foo")
    bar = Foo("bar")

    assert bar.say_hello() == "Hello bar!"
```

2.2.4 Share kernel context across multiple tests

If your use case requires you to execute many cells (or all cells) of a Jupyter Notebook, before a test can be executed, then it would make sense to share the kernel context with multiple tests.

It can be done by setting up a module or package scoped pytest fixture.

Consider the code cells below,

```
def foo(a, b):
    return a + b
```

```
def bar(a):
    return [x*2 for x in a]
```

The unit tests can be written as follows,

```
import pytest
from testbook import testbook


@pytest.fixture(scope='module')
def tb():
    with testbook('/path/to/notebook.ipynb', execute=True) as tb:
        yield tb

def test_foo(tb):
    foo = tb.ref("foo")
    assert foo(1, 2) == 3


def test_bar(tb):
    bar = tb.ref("bar")

    tb.inject("""
        data = [1, 2, 3]
    """)
    data = tb.ref("data")

    assert bar(data) == [2, 4, 6]
```

Warning: Note that since the kernel is being shared in case of module scoped fixtures, you might run into weird state issues. Please keep in mind that changes made to an object in one test will reflect in other tests too. This will likely be fixed in future versions of testbook.

2.2.5 Support for patching objects

Use the `patch` and `patch_dict` contextmanager to patch out objects during unit test. Learn more about how to use `patch` here.

Example usage of `patch`:

```
def foo():
    bar()
```

```
@testbook('/path/to/notebook.ipynb', execute=True)
def test_method(tb):
    with tb.patch('__main__.bar') as mock_bar:
        foo = tb.ref("foo")
        foo()

        mock_bar.assert_called_once()
```

Example usage of `patch_dict`:

```
my_dict = {'hello': 'world'}
```

```
@testbook('/path/to/notebook.ipynb', execute=True)
def test_my_dict(tb):
    with tb.patch('__main__.my_dict', {'hello' : 'new world'}) as mock_my_dict:
        my_dict = tb.ref("my_dict")
        assert my_dict == {'hello' : 'new world'}
```

2.3 Examples

Here are some common testing patterns where testbook can help.

2.3.1 Mocking requests library

Notebook:

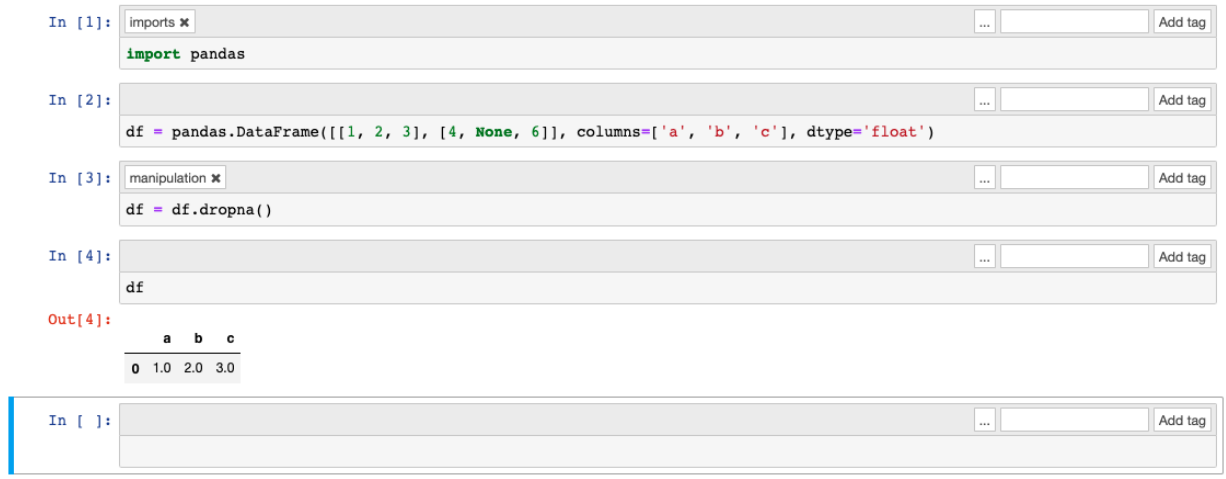
```
In [1]: import requests

In [2]: def get_details(url):
            return requests.get(url).content

In [ ]:
```

Test:

```
from testbook import testbook

@testbook('/path/to/notebook.ipynb', execute=True)
def test_get_details(tb):
    with tb.patch('requests.get') as mock_get:
        get_details = tb.ref('get_details') # get reference to function
        get_details('https://my-api.com')

        mock_get.assert_called_with('https://my-api.com')
```

2.3.2 Asserting dataframe manipulations

Notebook:

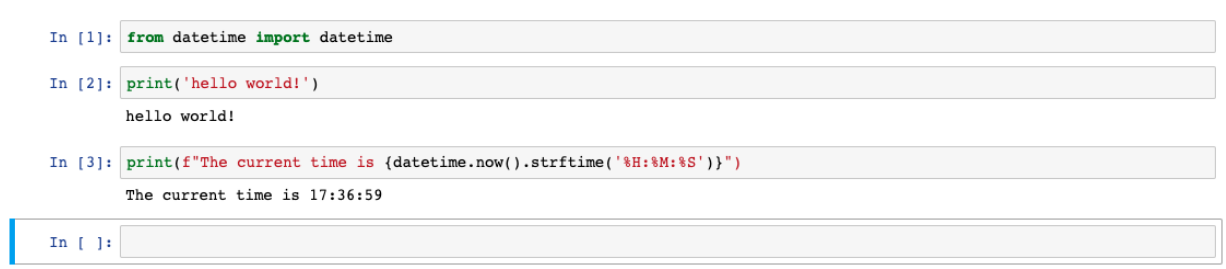


Test:

```
from testbook import testbook

@testbook('/path/to/notebook.ipynb')
def test_dataframe_manipulation(tb):
    tb.execute_cell('imports')

    # Inject a dataframe with code
    tb.inject(
        """
        df = pandas.DataFrame([[1, None, 3], [4, 5, 6]], columns=['a', 'b', 'c'],␣
↪dtype='float')
        """
    )

    # Perform manipulation
    tb.execute_cell('manipulation')

    # Inject assertion into notebook
    tb.inject("assert len(df) == 1")
```

2.3.3 Asserting STDOUT of a cell

Notebook:



Test:

```
from testbook import testbook

@testbook('stdout.ipynb', execute=True)
def test_stdout(tb):
    assert tb.cell_output_text(1) == 'hello world!'

    assert 'The current time is' in tb.cell_output_text(2)
```

2.4 Reference

This part of the documentation lists the full API reference of all public classes and functions.

2.4.1 testbook.client module

class testbook.client.**TestbookNotebookClient**(*nb*, *km=None*, ***kw*)

Bases: nbclient.client.NotebookClient

cell_execute_result(*cell: Union[int, str]*) → List[Dict[str, Any]]

Return the execute results of cell at a given index or with a given tag.

Each result is expressed with a dictionary for which the key is the mimetype of the data. A same result can have different representation corresponding to different mimetype.

Parameters **cell** (*int or str*) – The index or tag to look for

Returns The execute results

Return type List[Dict[str, Any]]

Raises

- **IndexError** – If index is invalid
- ***TestbookCellTagNotFoundError*** – If tag is not found

cell_output_text(*cell*) → str

Return cell text output

property cells

execute() → None

Executes all cells

execute_cell(*cell*, ***kwargs*) → Union[Dict, List[Dict]]

Executes a cell or list of cells

get(*item*)

inject(*code: str*, *args: List = None*, *kwargs: Dict = None*, *run: bool = True*, *before: Union[str, int, None] = None*, *after: Union[str, int, None] = None*, *pop: bool = False*) → testbook.testbooknode.TestbookNode

Injects and executes given code block

Parameters

- **code** (*str*) – Code or function to be injected
- **args** (*iterable, optional*) – tuple of arguments to be passed to the function

- **kwargs** (*dict, optional*) – dict of keyword arguments to be passed to the function
- **run** (*bool, optional*) – Control immediate execution after injection (default is True)
- **after** (*before,*) – Inject code before or after cell
- **pop** (*bool*) – Pop cell after execution (default is False)

Returns Injected cell

Return type TestbookNode

patch(*target*, ***kwargs*)
Used as contextmanager to patch objects in the kernel

patch_dict(*in_dict*, *values=()*, *clear=False*, ***kwargs*)
Used as contextmanager to patch dictionaries in the kernel

ref(*name: str*) → Union[testbook.reference.TestbookObjectReference, Any]
Return a reference to an object in the kernel

value(*code: str*) → Any
Execute given code in the kernel and return JSON serializeable result.

If the result is not JSON serializeable, it raises *TestbookAttributeError*. This error object will also contain an attribute called *save_varname* which can be used to create a reference object with *ref()*.

Parameters **code** (*str*) – This can be any executable code that returns a value. It can be used the return the value of an object, or the output of a function call.

Returns

Return type The output of the executed code

Raises ***TestbookSerializeError*** –

2.4.2 testbook.exceptions module

exception testbook.exceptions.**TestbookAttributeError**
Bases: AttributeError

exception testbook.exceptions.**TestbookCellTagNotFoundError**
Bases: *testbook.exceptions.TestbookError*

Raised when cell tag is not declared in notebook

exception testbook.exceptions.**TestbookError**
Bases: Exception

Generic Testbook exception class

exception testbook.exceptions.**TestbookExecuteResultNotFoundError**
Bases: *testbook.exceptions.TestbookError*

Raised when there is no execute_result

exception testbook.exceptions.**TestbookRuntimeError**(*evalue*, *traceback*, *eclass=None*)
Bases: RuntimeError

exception testbook.exceptions.**TestbookSerializeError**
Bases: *testbook.exceptions.TestbookError*

Raised when output cannot be JSON serialized

2.5 Changelog

2.5.1 0.4.1

- check for errors when `allow_errors` is true

2.5.2 0.4.0

- Testbook now returns actual object for JSON serializable objects instead of reference objects. Please note that this may break tests written with prior versions.

2.5.3 0.3.0

- Implemented container methods – **len** – **iter** – **next** – **getitem** – **setitem** – **contains**
- Fixed testbook to work with ipykernel 5.5

2.5.4 0.2.6

- Fixed Python underscore (_) issue

2.5.5 0.2.5

- Fixed testbook decorator.

2.5.6 0.2.4

- Add `cell_execute_result` to `TestbookNotebookClient`
- Use testbook decorator with pytest fixture and marker

2.5.7 0.2.3

- Accept notebook node as argument to testbook
- Added support for specifying kernel with `kernel_name` kwarg

2.5.8 0.2.2

- Added support for passing notebook as file-like object or path as str

2.5.9 0.2.1

- Added support for `allow_errors`

2.5.10 0.2.0

- Changed to new package name `testbook`
- Supports for patch and patch_dict
- Slices now supported for execute patterns
- Raises TestbookRuntimeError for all exceptions that occur during cell execution

2.5.11 0.1.3

- Added warning about package name change

2.5.12 0.1.2

- Updated docs link in setup.py

2.5.13 0.1.1

- Unpin dependencies

2.5.14 0.1.0

- Initial release with basic features

PYTHON MODULE INDEX

t

INDEX