Test My Notebook

Release 0.0.1

Young-Chan Park

Jul 14, 2020

TABLE OF CONTENTS:

1	Installation Guide	3
2	How To Use	5
3	Why use *testmynb* ?	7
4	API References	11
5	Index	13
6	Module Index	15
7	Search	17
Index		19

testmynb is a testing framework which runs tests written in Jupyter Notebook code cells.

CHAPTER

ONE

INSTALLATION GUIDE

1.1 Availability

Python: 3.5, 3.6, 3.7, 3.8, 3.9
Platforms: Linux, macOS

Run the following command to install *testmynb* from PyPI and check its version.

```
$ pip install testmynb
$ testmynb --version
```

CHAPTER

TWO

HOW TO USE

2.1 `%load_ext testmynb`

Load testmynb to the ipython kernel to start using testmynb.

In [1]:
```
%load_ext testmynb
```

2.2 `%%testcell` Magic

By loading `testmynb` to the ipython kernel, the `%%testcell` magic becomes available.

By adding the `%%testcell` magic to the cell, it flags the cell to be executed as a test by the `testmynb` command line tool.

In [2]:
```
%%testcell example_test_cell

assert 1==1, 'Example assert statement'
```

2.3 Test Cell

Test cells should generally have two things: a title and an assert statement.

Cell Title
The first positional argument of the cell magic line is used as the test cell's title.

When running `testmynb`, the failed/errored test cell's title is displayed to allow the user to distinguish which test failed/errored. If no title is given, the title falls back to `unnamed`.

You may still distinguish the failed/errored test cell by observing the cell body and the traceback of the test.

Assert Statement
Like `pytest`, a test cell should contain an assert statement to test whatever you're testing.

If any of the two above are missing, a message is displayed when the cell is executed.

In [3]:
```
%%testcell
test = True
# No assert statement nor cell title.
# Running this cell will print out a message
# saying there is no assert statement nor a cell title.
```

```
[testmynb] Assert statement missing.
[testmynb] Cell title missing.
```

2.4 `-n` option

The `-n` option is used to run the cell during the test, but not to treat it as a test. This can be used for cells which contain all the required import statements, or as a setup/teardown cell.

In [4]:
```
%%testcell import_statements -n
import os
import sys
```

2.5 `testmynb` commandline

Run the `testmynb` command, and the command line tool searches for all the `.ipynb` files with the `test_` prefix and runs the designated test cells.

In [5]:
```
!testmynb
```

```
========================== Test My Notebook (0.0.1) ==========================
Platform darwin
Python 3.7.1
Working Directory: ${PWD}

7 test cells across 2 notebook(s) detected.
Notebooks:
Trusted test_how_to.ipynb: ...
Untrusted test_why_use_testmynb.ipynb: ...

============== 7 test(s) passed, 0 failed, and 0 raised an error ==============
```

CHAPTER THREE

WHY USE *TESTMYNB* ?

There are times when your tests need some documentation, but text comments alone are just not sufficient enough to fully explain what the test is doing. By using `testmynb`, you may use Jupyter Notebook's features to explain your tests.

Like the example below. Say deep in your Python package, you made a function which adds a label to a given x, y coordinate. You tried your best in the docstring what the function does, and you write the tests for the function which all passes.

In [1]:
```
%load_ext testmynb
```

In [2]:
```
%%testcell imports -n
import pandas as pd
import numpy as np
```

In [3]:
```
%%testcell example_func
# Your function in your imaginary package.
def get_label(x: float, y: float) -> str:
    """
    Based on the given coordinate, returns `purple`, `red`, and/or `orange` label.

    `purple`: 0<=x<30 and 30<y<=100
    `red`: 20<=x<70 and 60<y<=100
    `orange`: 70<=x<=100 and 30<y<=100
    """
    assert 0 <= x <= 100
    assert 0 <= y <= 100

    labels = list()
    if (0 <= x < 30)\
        and (30 < y <= 100):
        labels.append('purple')
    if (20 <= x < 70)\
        and (60 < y <= 100):
        labels.append('red')
    if (70 <= x <= 100)\
        and (30 < y <= 100):
        labels.append('orange')

    return ';'.join(labels)
```

In [4]:
```
%%testcell test_get_label_original
# Tests for the above function that all passes
assert 'purple;red' == get_label(25, 70)
```

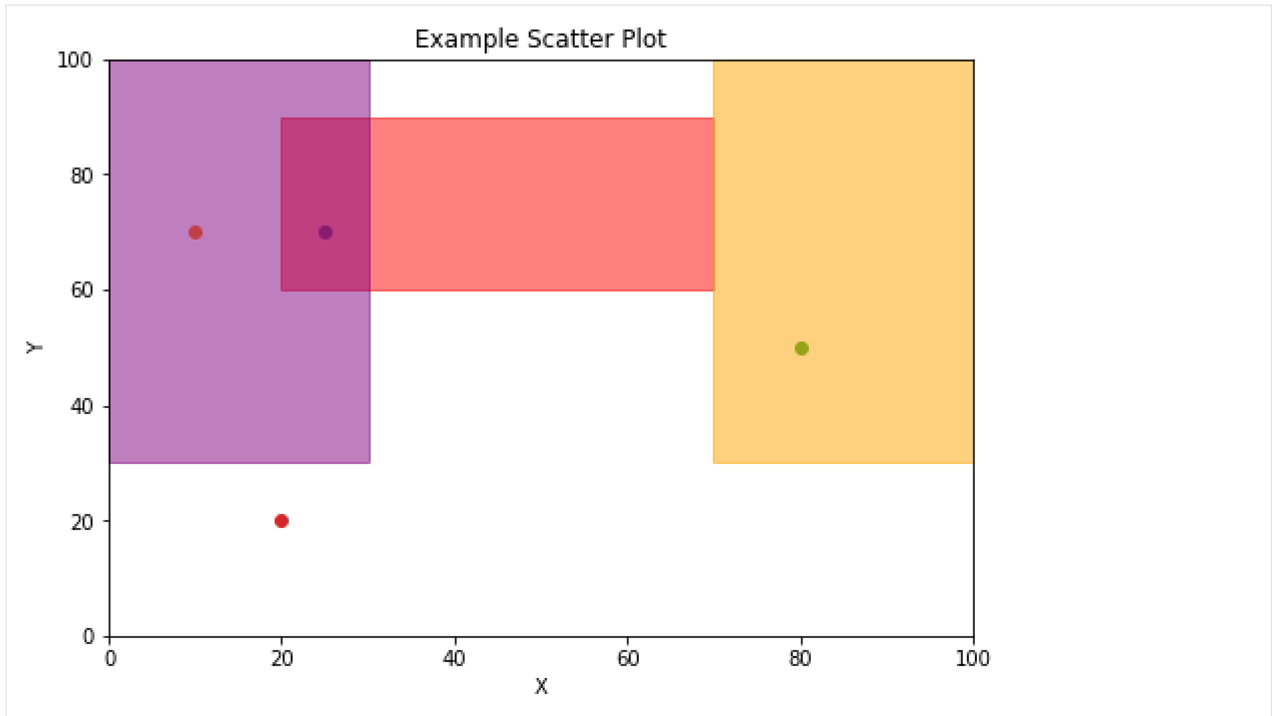
(continues on next page)

(continued from previous page)

```
assert 'purple' == get_label(10, 70)
assert 'orange' == get_label(80, 50)
assert '' == get_label(20, 20)
```

Technically, this all passes and the function works as intended. But text only explanation of the function, and looking at the source itself just doesn't intuitively tell the reader what the function does. In this situation, a graphical explanation may aid in the reader to understand what the function/tests are doing.

In [5]:
```
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure()

ax = fig.add_axes([0,0,1,1])
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Example Scatter Plot')

ax.set_ylim(0, 100)
ax.set_xlim(0, 100)

ax.axvspan(20, 70, ymin=.6, ymax=.9, alpha=0.5, color='red')
ax.axvspan(0, 30, ymin=.3, ymax=1.0, alpha=0.5, color='purple')
ax.axvspan(70, 100, ymin=.3, ymax=1., alpha=0.5, color='orange')

ax.scatter(25, 70) # This should output `purple;red`
assert 'purple;red' == get_label(25, 70)
ax.scatter(10, 70) # This should output `purple`
assert 'purple' == get_label(10, 70)
ax.scatter(80, 50) # This should output `orange`
assert 'orange' == get_label(80, 50)
ax.scatter(20, 20) # This should output ``
assert '' == get_label(20, 20)
plt.show()
```



The above graphically shows what the function does and intuitively shows what the tests are testing!

Also, now we can graphically see what the function does, we can improve the tests with something like the `hypothesis` package!

In [6]:
```
%%testcell test_get_label_improved

from hypothesis import given, strategies

@given(
    x = strategies.integers(min_value = 0, max_value = 19)
    , y = strategies.integers(min_value = 31, max_value = 100)
)
def test_get_label_purple(x, y):
    assert 'purple' == get_label(x, y)


@given(
    x = strategies.integers(min_value = 30, max_value = 69)
    , y = strategies.integers(min_value = 61, max_value = 90)
)
def test_get_label_red(x, y):
    assert 'red' == get_label(x, y)

@given(
    x = strategies.integers(min_value = 70, max_value = 100)
    , y = strategies.integers(min_value = 31, max_value = 100)
)
def test_get_label_orange(x, y):
    assert 'orange' == get_label(x, y)
```

CHAPTER

FOUR

API REFERENCES

4.1 Classes

class testmynb.handler.**TestHandler**(**notebooks*)

class testmynb.notebook.**TestCell**(*data*, *notebook*)

Bases: `collections.UserString`

A class for Jupyter Notebook code cell.

Variables

- **ignore** (*bool*) – Whether the cell magic line contained the *-t* option.
- **name** (*str*) – The user defined name of the test cell block.

class testmynb.notebook.**Notebook**(*ipynb: TextIO*)

Bases: `nbformat.notebooknode.NotebookNode`

A class used to read the Jupyter Notebook

Parameters **ipynb** (*TextIO*) – Path to the *.ipynb* file.

Variables

- **ipynb** (*TextIO*) – Absolute path to the *.ipynb* file that was given to instantiate the instance.
- **name** (*str*) – Name of the *.ipynb* file.
- **trusted** (*bool*) – Whether the Notebook is *Trusted* or not for the user.
- **nbformat** (*str*) – The Jupyter Notebook format number.

extract_codes()

Returns a list of code cells with the *%%testcell* cell magic.

4.2 Functions

testmynb.handler.**find_notebooks**(**args*)

CHAPTER FIVE

INDEX

CHAPTER

SIX

MODULE INDEX

CHAPTER

SEVEN

SEARCH

INDEX

E

`extract_codes()` (*testmynb.notebook.Notebook method*), 11

F

`find_notebooks()` (*in module testmynb.handler*), 11

N

`Notebook` (*class in testmynb.notebook*), 11

T

`TestCell` (*class in testmynb.notebook*), 11
`TestHandler` (*class in testmynb.handler*), 11