M08-HW-KEY

October 19, 2022

1 Metadata

```
Course:  DS 5100
Module:  08 Python Testing
Topic:   HW 08 Unit Testing a Book Lover Class
Author:  R.C. Alvarado (adapted)
Date:    30 June 2022
```

2 Student Info

- Name:

- Net UD:
- URL of this file in GitHub:

3 Instructions

In your **private course repo on Rivanna**, use this Jupyter notebook and the data file described to write code that performs the tasks below.

Save your notebook in the `M08` directory.

Remember to add and commit these files to your repo.

Then push your commits to your repo on GitHib.

Be sure to fill out the **Student Info** block above.

To submit your homework, save your results as a PDF and upload it to GradeScope. More information about how to create the PDF for this assignment are included at the end of this document.

TOTAL POINTS: 20

4 Overview

In this assessment, you will write and test a simple class using Python's `unittest` module.

It is designed to develop your ability to create a unit test suite as well as your ability to write classes and use Pandas.

You will create 3 files: - `booklover.py` that will contain the class `BookLover`. - `booklover_test.py` that will contain the class `BookLoverTestSuite`. - `booklover_results.txt` that will contain the results of applying the second class to the first.

You will create them all in the same directory.

Note that these are not Jupyter notebooks; you will need to use a text editor to create them. You can use the text editor that comes with Jupyter Lab if you'd like, or use VSCode, or whatever suits you.

5 Tasks

5.1 Task 1

(5 points; .5 pt per attribute and method)

Define a `BookLover` class in a file named `booklover.py`.

5.1.1 Attributes

Attribute	Value
`name`	The name of the person (type:string)
`email`	The person's email, serving as a unique identifier (type:string)
`fav_genre`	The person's favorite book genre (e.g., mystery, fantasy, or historical fiction). (type:string)
`num_books`	Keeps track of the number of books the person has read (type:int)
`book_list`	a dataframe with the columns `['book_name', 'book_rating']`

The columns in `book_list` have the following meanings: - `book_name` is the title of the book the person has read. - `book_rating` is the person's rating of that book on a scale of 1 to 5, where 1 means the person did not like the book at all, and 5 means the person loved the book.

Some example book entries are:

```
("Jane Eyre", 4)
("Fight Club", 3)
("The Divine Comedy", 5)
("The Popol Vuh", 5)
```

5.1.2 Methods

Initializer:

`__init__()`: - `name`, `email`, and `fav_genre` (in this order) are required. - `num_books` and `book_list` are optional. - Use these default parameters:

```
  num_books = 0
  book_list = pd.DataFrame({'book_name':[], 'book_rating':[]})
```

Do not add any additional fields of your own.

Method 1:

`add_book(book_name, rating)`: - This function takes a `book name` (string) and `rating` (integer from 0 to 5) - It tries to add the book to `book_list`. See hint below on how to pass a new book to the dataframe. - Only add a book to the person's `book_list` if that book doesn't already exist. - It is sufficient to match on `book_name`. - If it does exist, tell the user.

Hint: To add a new book to the book list (which is a dataframe), do this in your method, where `book_name` and `book_rating` are the arguments passed to the method.:

```
new_book = pd.DataFrame({
    'book_name': [book_name],
    'book_rating': [book_rating]
})

self.book_list = pd.concat([self.book_list, new_book], ignore_index=True)
```

Of course, be sure to see if `book_name` is not in the book list.

Method 2:

`has_read(book_name)` - This function takes `book_name` (string) as input and determines if the person has read the book. - That is, if that `book name` is in `book_list`. - Again, it is sufficient to match on `book_name`. - The method should return `True` if the person has read the book, `False` otherwise.

Method 3:

`num_books_read()` - This function takes no parameters and just returns the total number of books the person has read.

Method 4:

`fav_books()`: - This function takes no parameters and returns the filtered dataframe of the person's most favorite books. - Books in this list should have a rating > 3.

Once you have created your class

Be sure to instantiate your class to see if everything is working. You can do this by prototyping your class in a notebook, where you can run code that uses it there, and then save the class to the `.py` file when you are done.

Or you can create a another file, say `demo.py` that imports and uses the class.

A final option – which the test file will use – is to put this code at the bottom of your `.py` file, after and outside of your class definition:

```
if __name__ == '__main__':

    test_object = BookLover("Han Solo", "hsolo@millenniumfalcon.com", "scifi")
```

```
test_object.add_book("War of the Worlds", 4)
# And so forth
```

NOTE: The methods listed above do not have `self` as their first argument, but they should in your class.

5.2 Solution

Here is the code that should appear in a Python file.

Unlike the test code, it is also executable here in the notebook.

[1]:
```
import pandas as pd

class BookLover():

    def __init__(self, name, email, fav_genre):
        self.name = name
        self.email = email
        self.fav_genre = fav_genre
        self.num_books = 0
        self.book_list = pd.DataFrame({'book_name':[], 'book_rating':[]})

    def add_book(self, book_name, book_rating):
        new_book = pd.DataFrame({
            'book_name': [book_name],
            'book_rating': [book_rating]
        })
        if self.has_read(book_name):
            return False
        else:
            self.book_list = pd.concat([self.book_list, new_book],␣
 ↪ignore_index=True)

    def has_read(self, book_name):
        return any(self.book_list.book_name == book_name)

    def num_books_read(self):
        return(len(self.book_list))

    def fav_books(self):
        return self.book_list[self.book_list.book_rating > 3]
```

[2]:
```
lover = BookLover("Fred", "a@b.com", "scifi")
```

[3]:
```
lover.book_list
```

[3]:
```
Empty DataFrame
Columns: [book_name, book_rating]
```

```
Index: []
```

[4]:
```
books = [
    ("Jane Eyre", 4),
    ("Fight Club", 3),
    ("The Divine Comedy", 5),
    ("The Popol Vuh", 5)
]
```

[5]:
```
for book in books: lover.add_book(*book)
```

[6]:
```
lover.add_book(*books[2])
```

[6]:
```
False
```

[7]:
```
lover.has_read(books[0][0])
```

[7]:
```
True
```

[8]:
```
lover.num_books_read()
```

[8]:
```
4
```

[9]:
```
lover.book_list
```

[9]:
```
            book_name  book_rating
0          Jane Eyre          4.0
1         Fight Club          3.0
2  The Divine Comedy          5.0
3      The Popol Vuh          5.0
```

[10]:
```
lover.fav_books()
```

[10]:
```
            book_name  book_rating
0          Jane Eyre          4.0
2  The Divine Comedy          5.0
3      The Popol Vuh          5.0
```

5.3 Task 2

(6 points; 1 pt per method)

Create a test suite for the previous class in a file named `booklover_test.py`.

In the file, write a class called `BookLoverTestSuite`, being sure to import the `unittest` library and the BookLover class from the first file.

5.3.1 Unit Tests

In this class, include the unit tests below:

- `test_1_add_book()`: Add a book and test if it is in `book_list`.
- `test_2_add_book()`: Add the same book twice. Test if it's in `book_list` only once.
- `test_3_has_read()`: Pass a book in the list and test the answer is `True`.
- `test_4_has_read()`: Pass a book NOT in the list and use `assert False` to test if the answer is `True`
- `test_5_num_books_read()`: Add some books to the list, and test num_books matches expected.
- `test_6_fav_books()`: Add some books with ratings to the list, making sure some of them have rating > 3.
  - Your test should check that the returned books have rating > 3.

Note that you do not need to create an `__init__()` method in this class, nor do you have to define any class variables.

Instead, treat every method as a small, stand-alone program in which you create a new object for your test. This is not the best practice in a production environment, but it works and it will enable you to get the gist of unit testing.

5.3.2 Template

Here is a template of the file you will create:

```
import unittest
from booklover import BookLover

class BookLoverTestSuite(unittest.TestCase):

    def test_1_add_book(self):
        # add a book and test if it is in `book_list`.

    def test_2_add_book(self):
        # add the same book twice. Test if it's in `book_list` only once.

    def test_3_has_read(self):
        # pass a book in the list and test if the answer is `True`.

    def test_4_has_read(self):
        # pass a book NOT in the list and use `assert False` to test the answer is `True`

    def test_5_num_books_read(self):
        # add some books to the list, and test num_books matches expected.

    def test_6_fav_books(self):
        # add some books with ratings to the list, making sure some of them have rating > 3.
        # Your test should check that the returned books have rating  > 3

if __name__ == '__main__':

    unittest.main(verbosity=3)
```

The last part of the file is **crucial**: It tells the Python interpreter to run the bit of code at the end if the file is being run directly (and not being imported into another file).

5.4 Solution

Here is the code that should appear in a Python file:

```
import unittest
from booklover import BookLover

class BookLoverTestSuite(unittest.TestCase):

    def test_1_add_book(self):
        # add a book and test if it is in `book_list`.

        book_lover = BookLover("RCA", "a@b.com", "scifi")
        test_name = "Test Book"
        test_rating = 5
        book_lover.add_book(test_name, test_rating)
        self.assertTrue(book_lover.has_read(test_name))

    def test_2_add_book(self):
        # add the same book twice. Test it's in `book_list` only once.

        book_lover = BookLover("RCA", "a@b.com",  "scifi")
        test_name = "Test Book"
        test_rating = 5
        book_lover.add_book(test_name, test_rating)
        book_lover.add_book(test_name, test_rating)
        expected = 1
        actual = len(book_lover.book_list[book_lover.book_list.book_name == test_name])
        self.assertEqual(expected, actual)

    def test_3_has_read(self):
        # pass a book in the list and test the answer is `True`.

        book_lover = BookLover("RCA", "a@b.com",  "scifi")
        test_name = "Test Book"
        test_rating = 5
        book_lover.add_book(test_name, test_rating)
        self.assertTrue(book_lover.has_read(test_name))

    def test_4_has_read(self):
        # pass a book NOT in the list and use `assert False` to test if the answer is `True`

        book_lover = BookLover("RCA", "a@b.com",  "scifi")
        test_name = "Test Book"
```

```
        self.assertFalse(book_lover.has_read(test_name))

    def test_5_num_books_read(self):
        # add some books to the list, and test num_books matches expected.

        book_lover = BookLover("RCA", "a@b.com",  "scifi")
        test_books = [
            ("Jane Eyre", 4),
            ("Fight Club", 3),
            ("The Divine Comedy", 5),
            ("The Popol Vuh", 5)
        ]
        for book in test_books:
            book_lover.add_book(*book)

        self.assertEqual(len(test_books), book_lover.num_books_read())

    def test_6_fav_books(self):
        # add some books with ratings to the list, making sure some of them have rating $> 3$.
        # Your test should check that the returned books have rating $ > 3

        book_lover = BookLover("RCA", "a@b.com",  "scifi")
        test_books = [
            ("Jane Eyre", 4),
            ("Fight Club", 3),
            ("The Divine Comedy", 5),
            ("The Popol Vuh", 5)
        ]
        for book in test_books:
            book_lover.add_book(*book)

        actual = len(book_lover.fav_books())
        expected = len([x for x, y in test_books if y > 3])
        self.assertEqual(actual, expected)


if __name__ == '__main__':

    unittest.main(verbosity=3)
```

[12]:
```
!python booklover_test.py
```

```
test_1_add_book (__main__.BookLoverTestSuite) … ok
test_2_add_book (__main__.BookLoverTestSuite) … ok
test_3_has_read (__main__.BookLoverTestSuite) … ok
test_4_has_read (__main__.BookLoverTestSuite) … ok
test_5_num_books_read (__main__.BookLoverTestSuite) … ok
test_6_fav_books (__main__.BookLoverTestSuite) … ok
```

```
----------------------------------------------------------------------
Ran 6 tests in 0.013s

OK
```

5.5 Task 3

(6 points; 1 pt per test)

Run the tests and save results in a file named `booklover_results.txt`. All six tests must return positive.

Test your class at the command line as follows:

```
rivanna$ python booklover_test.py
```

Look at the output and make sure your tests are working and your code passes the test. When you are ready, output the test results to the third file as follows:

```
rivanna$ python booklover_test.py 2> booklover_results.txt
```

5.6 Task 4

(3 points; 1 point per file)

All three files created and named properly.

6 Submission Instructions

Once you are done with the above, combine the three files into one and save it to a PDF file and upload it to Gradescope.

There are at least two ways to do this: * Cut and paste the files back into a Jupyter notebook and export to PDF. * If you do this, put the files in three separate blocks. * Choose Raw as the format for each of the blocks. * Follow the recipe below on Rivanna (for those who want to hone their command-line skills):

Combine and Convert to PDF:

Concatenate your files into one with the following from the command line:

```
rivanna$ cat -n booklover.py booklover_test.py booklover_results.txt > HW08.txt
```

Then convert `HW09.txt` to a PDF file as follows:

```
rivanna$ module load ghostscript
rivanna$ pdfroff --pdf-output=HW08.pdf HW08.txt
```

Go to Rivanna's web-based File Explorer (from the UVA OpenOnDemand Dashboard), locate the file, and then download it.

Then upload the PDF to Gradescope.

Mischief managed.