Homework 7

Total Points:

100 (Correctness and Style) + 3 EC (Checkpoint)

Due Dates (SD time):

- **Entire Assignment: Tuesday, Feb 27th, 11:59pm**
- **Free slip day: Wed: Feb 28th, 11:59pm**
- **Checkpoint (read below): Sunday, Feb 25th, 11:59pm**

Starter Files

Download hw07.zip. Inside the archive, you will find starter files for the questions of this homework. You cannot import anything to solve the problems.

IMPORTANT: Coding and Docstring Style

This is a reminder that your code style will be graded. Here are a few useful links:

Style Guide Document

Style Guide on Course Website

Style Guide Examples

Testing

At any point of the homework, use one of the following command to test your work:

```
>>> python3 -m doctest hw07.py
>>> py -m doctest hw07.py
>>> python -m doctest hw07.py
```

Checkpoint Submission

Due Date: Sunday, February 25th, 11:59pm (SD time)

You can earn up to **3 points extra credit** by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete **Question 1, 3** and submit the hw07.py file to gradescope.

Checkpoint submission is graded by completion, which means you can get full points if your code can pass some simple sanity check (no tests against edge cases). Note that in your final submission, you should still submit these questions, and you may modify your implementation if you notice any errors.

General Notes and Requirements

1. **DO NOT IMPORT ANY PACKAGES.**
2. Please add your own doctests (**at least three**) as the given doctests are not sufficient. **You will be graded on the doctests**. Read the special instructions before Questions 4, 5 for the docstring/doctest requirements for that OOP Question.
3. When some methods in **Question 4** require **assert statements**: follow the assert guide to write assert statements to prevent any unexpected inputs. You will be graded on this. If a question does not require assert statements, assume all inputs are valid.

Required Questions

Question 1

Let's look at an example where recursion justifies its existence. You can also think about the alternative iterative solution as well.

As you continue traveling through Pythonia, you realize how many different merchants and kingdoms there are! You see a market full of merchants selling their trinkets with signs showing prices all below $10:

```
marketplace = [
        "...............",
        ".5_1_7_x___4_7.",
        ".x__8_3__2___6.",
        "._5_2_87_35__2.",
        ".___6___5_8x2_.",
        ".2_7_9_44_3_2_.",
        "..............."
]
```

Where each integer represents the cost of that merchant's trinkets, '_' represents a walkway that costs $0, 'x' represents a scam booth, and '.' represents out of bounds.

You will be given a starting point as a tuple with two integers, a map, and a list of directions. Your function should return the amount of money you spent by following all the directions, return early if the directions cause you to walk out of the market (out of bounds), or 'scammed!' if you visited a scam booth 😱.

Assumptions:

- Each string within a list has the same size. Market maps can be of different sizes.
- Each starting point will be *within* the map.
- Each side is surrounded by out of bounds marks (dot ".")
- Directions will be a list of strings ('R', 'D', 'L', 'U' are "Right", "Down", "Left" and "Up" respectively)
- The start tuple is in the format (*row*, *column*), with the origin (0,0) as the upper-leftmost value. The row numbers increase downward, and the column numbers increase towards the right.

Requirements:

- No built-in functions (len() is fine).
- The function must be recursive.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed.

Note: For doctests, feel free to use the current marketplace map or define a new map of your own.

```
def marketplace(start, market_map, directions, spent=0):
    """
    >>> market_map = [
    ... "...............",
    ... ".5_1_7_x___4_7.",
    ... ".x__8_3__2___6.",
    ... "._5_2_87_35__2.",
    ... ".___6___5_8x2_.",
    ... ".2_7_9_44_3_2_.",
    ... "..............."]

    >>> directions = ['U','R','R','U','L','U','L','L']
>>> marketplace((5,7), market_map, directions)
15
>>> marketplace((1,7), market_map, directions)
'scammed!'

>>> directions = ['L','L','L','L','L','L','L','L','L','L','L','L']
>>> marketplace((3,10), market_map, directions)
30
"""
```

Question 2: (EC)

You've found a stall that is challenging passersby: count the number of valid palindromes in a given string.
Note: palindromes are words/phrases that reads the same backward as forward (ex: racecar, tacocat)

Write a recursive function that takes in a string and returns the number of valid palindromes within that string.

Assumptions:

- The string will only contain alphabetical characters - i.e. no spaces, punctuation marks, numbers, etc.
- Single letters are not considered palindromes (ex: 'a' is not a valid palindrome, but 'aa' is)

Requirements:

- No built-in functions (len() is fine).
- The function must be recursive.
- No loops/list comprehension/map/filter allowed. No helper/inner function allowed except the one suggested below (it also must be recursive).
- [::-1] is not allowed, check hint below for more information about checking for valid palindromes.

Hint:

- First, we recommend you to write a recursive helper function that takes a string as a parameter and returns True if this string is a palindrome. False otherwise. Use a lecture slide for the idea here.
- Then, you can use this helper function to solve the given problem.

```
def palindromes(word):
    """
    >>> palindromes("aaa")
    3
    >>> palindromes("abcba")
    2
    >>> palindromes("ababaa")
    5
    """
```

Example 1:
```
>>> palindromes("aaa")
```
The valid palindromes for “aaa” are: [“aaa”, “aaa”, “aaa”] (highlighted and underlined). So, the answer is 3.

Example 2:
```
>>> palindromes("abcba")
```
The valid palindromes for “abcba” are: [“abcba”, “abcba”]. So, the answer is 2.

Example 3:
```
>>> palindromes("ababaa")
```
The valid palindromes for “ababaa” are: [“ababaa”, “ababaa”, “ababaa”, “ababaa”, “ababaa”]. So, the answer is 5.

OOP Questions

Question 3:

With so many deadlines in DSC20, you need an event tracker that helps you to stay on track. In order to do it, you will implement a class called EventTracker with the specifications listed below.

Requirements:

1. You **do not** need to add additional doctests.
2. You **do not** need to add docstrings.

Notes:

1. We have put the doctests in a function `doctest_q3()` to test the functionalities more thoroughly;
2. When debugging, make sure to start from the **first** failed doctest because other failures might depend on this one.
3. The `pass` keyword serves the same purpose as `return` when used as a placeholder. You should remove it and replace it with your implementation.

Class attribute:	
`event`	It should be **exactly** this string: `'midterm'`
Instance attribute:	
`event_month`	An integer representing a month of the event.
Constructor	
`__init__`	Initializes `event_month` `to a given argument`
Class Method:	
`change_event(cls, new_event):`	Changes `event` to a `new_event`
Static Method:	
`extract_month(date)`	Takes a date (datetime object) and returns a corresponding month as an integer. **Hint**: Look up attribute `month` for a datetime object.
Instance Methods:	
`change_event_month(self, new_month)`	Changes `event_month` to a `new_month`

`months_left(self, date)`	Takes a datetime object and returns a string, indicating how many months left until the event. If the number of months is negative, returns a string indicating that the event is missed. Please refer to the provided docstrings for the exact format.

Question 4:

After coming across so many different people and animals in your travels, you decide you want to keep track of them. You have a spare notebook and decide to start noting everything down in it. Implement a class called `Notebook`.

Requirements:
1. You **do not** need to add additional doctests.
2. You **do not** need to add docstrings.

Notes:
- We have put the doctests in a function `doctest_q4()` to test the functionalities more thoroughly;
- When debugging, make sure to start from the **first** failed doctest because other failures might depend on this one.
- The `pass` keyword serves the same purpose as `return` when used as a placeholder. You should remove it and replace it with your implementation.

Class attribute:	
`description`	It should be **exactly** this string: `'This notebook is to keep track of encountered people and animals.'`
static method:	
`edit_description()`	Return the **exact** string:

	`'This notebook only has 1 purpose - there is no need to modify its description!'`
Instance attributes:	
`title`	A string representing the notebook's name
`author`	A string representing the author's name
`tip`	An integer representing the author's **t**ime spent **i**n **P**ythonia (in months)
`encounters`	A dictionary containing all the people and animals the author has seen. The keys are `'people'` and `'animals'`, and their values are lists of strings representing people's names/animal species encountered.
Instance Methods:	
`info(self)`	Return a string with the information about the notebook. See the doctest for the specific format.
`get_people(self)`	Return a list containing the names of people met.
`get_animals(self)`	Return a list containing the species of animals seen.
`new_encounter(self, new_type, new_add)`	Update `encounters` dictionary. `new_type` references if the encounter was a person or an animal, and `new_add` is the name or species to add. If the person/animal has already been added, return `'Already encountered'`. Return `True` if the update is completed.

```
class Notebook:
    """
    >>> n1 = Notebook("My encounters", "Marina", 12, \
    {'people': [], 'animals': []})
    >>> n1.description
    'This notebook is to keep track of encountered people and
animals.'
    >>> n1.title
    'My encounters'
    >>> n1.author
    'Marina'
    >>> n1.tip
    12
    >>> n1.get_people()
    []
    >>> n1.new_encounter('people', 'tutors')
    True
    >>> n1.new_encounter('animals', 'recursive dragon')
    True
    >>> n1.new_encounter('people', 'tutors')
    'Already encountered'
    >>> n1.new_encounter('animals', 'baby pandas')
    True
    >>> n1.encounters
    {'people': ['tutors'], 'animals': ['recursive dragon', 'baby
pandas']}
    >>> n1.get_people()
    ['tutors']
    >>> n1.get_animals()
    ['recursive dragon', 'baby pandas']
    >>> n1.info()
    "Marina has spent 12 months in Pythonia and has encountered 1
people and \
2 animals. These are noted in 'My encounters'."

    >>> n2 = Notebook("Encounters", "Noto", 1, \
    {'people': ['Ben', 'Matilda', 'Charisse'], 'animals': ['cat']})
    >>> n2.get_people()
    ['Ben', 'Matilda', 'Charisse']
    >>> Notebook.edit_description()
    'This notebook only has 1 purpose - there is no need to modify its
\
description!'
    """
```

Question 5:

There are many traveling merchants in Pythonia and the surrounding kingdoms.
To better track the region's seasonal business patterns (and to get a better understanding of Object-Oriented Programming in Python), we are going to implement two classes: `Merchant` and `Kingdom`.

We recommend you to read the entire writeup before starting to code, as you will have a better understanding of how the different methods and attributes will work together to create the functionality of the classes.

When we write functions we try to make sure that every function does exactly one job. A similar approach works with classes. We want to create a class that describes the functionality of a particular object and then use these classes (objects) to send messages to each other.

Doctest Requirements:

1. Create at least 2 new objects of each class (`Merchant` and `Kingdom`) in order to test constructors and getters as a whole.
2. You do **not** need to add doctests for constructors and getters (i.e. Question 5.1.2.A, 5.1.2.B, 5.2.2.A, 5.2.2.B)
3. For every other method (including `__str__`), add additional doctests so that you **call each method at least 3 times**.
4. When adding doctests, add them to the bottom of the `doctests_q5()` docstring. There is a comment indicating where you should add your doctests. This will make it easier for graders to see your new doctests so you don't get marked off.
5. When debugging, make sure to start from the **first** failed doctest because other failures might depend on this one.

Docstring Requirements:

1. Add method descriptions to all core functionality methods (i.e. Question 5.1.2.D, 5.2.2.D). For your convenience, we have marked the questions with docstrings needed with the usual comment box;
2. Exceptions are constructors, getters, string representations (i.e. Question 5.1.2.A-C, 5.2.2 A-C). For your convenience, we have provided the docstrings for these methods.

Assert Statement Requirements: Assert statements are required in the questions **only when explicitly mentioned below** under each method, including constructors.

OOP Diagram

Merchant	Kingdom
Instance Fields:	**Instance Fields:**
merchant_type (str)	name (str)
stock (dict)	curr_merchants (list)
cash (int)	
kingdom_licenses (list)	
id (int)	
Class Attributes:	**Instance Attributes (not passed in):**
id_count (int)	curr_merchants (list of merchants)
Constructor:	**Constructor:**
__init__(self, merchant_type, …)	__init__(self, name)
Getters:	**Getters:**
get_type(self)	get_name(self)
get_id(self)	get_curr_merchants(self)
get_stock(self)	
get_cash(self)	
get_kingdom_licenses(self)	
String Representation:	**String Representation:**

__str__(self)	__str__(self)
Core Functionality:	**Core Functionality:**
sell(self, product, count)	new_merchant(self, merchant)
restock(self, product, count)	temp_ban(self, merchant)
travel_to_kingdom(self, kingdom)	get_total_cash(self)
	promote_market(self)

Part 5.1: Constructor & methods for Merchant class

Question 5.1.1 : Instance attributes of **Merchant** class

A. **merchant_type (str):** Given by the constructor's argument. After initialization, `merchant_type` indicates the type of merchant. The type of merchant cannot be an empty string.
B. **stock (dict):** Given by the constructor's argument. After initialization, `stock` indicates the types of products as keys, and the value as a list of length 2 containing the integer amount of the product in stock as index 0, and the integer price of the product as index 1.
C. **cash (int):** Given by the constructor's argument. After initialization, `cash` indicates the amount of money the merchant initially has.
D. **kingdom_licenses (list):** Given by the constructor's argument. After initialization, `kingdom_licenses` indicate the kingdoms that this merchant can sell in. **Note:** This is a list of the kingdom names (i.e. list of strings).
E. **id (int):** NOT provided by the constructor's argument - corresponds to the class attribute id. The first Merchant will have an ID of 1, the second will have an ID of 2, and so on.

Class attribute of **Merchant** class

A. **id_count (int)**: Class attribute that counts the number of Merchants and is used as the Merchant's ID number. This attribute starts at 1 and increments by 1 every time a new Merchant object is created.

Question 5.1.2 : Methods of **Merchant** class

A. **Constructor (__init__):**
The constructor has parameters described in Question 5.1.1 and the attributes names are the same as these parameters.

B. Getters (used to retrieve the value of an instance variable**):**

a. get_type(self)

A getter method that returns the Merchant's type. This method creates the data abstraction. (It means that the method hides the implementation details of data and exposes only the necessary information or functionality to the outside world.)

Example: return "`Shoemaker`" if the Merchant's type is "`Shoemaker`"

b. get_stock(self)

A getter method that returns the dictionary of the Merchant's stock. This method creates the data abstraction.

c. get_cash(self)

A getter method that returns the amount of cash. This method creates the data abstraction.

d. get_kingdom_licenses(self)

A getter method that returns the kingdom licenses. This method creates the data abstraction.

e. get_id(self)

A getter method that returns the Merchant's id number. This method creates the data abstraction.

Note: All of these getter methods only need a single line.

C. String representation method

a. __str__ (self)

In order to print a string representation of an object we need to write a special method (will be covered in later lectures). This method enables us to use print statements on instances of Merchant. Implement the special method using the getter methods created above as helper functions (part 1B).

<u>The format of the string should be (if printed, without parentheses, on a single line):</u>

(merchant type) has brought (products from stock). They are able to sell their wares in (kingdoms).

D. Core Functionality

a. sell(self, product, count)

Method is used when a merchant completes a sale, returning a string of the format (without parentheses)

`"(merchant type) sold (count) of (product)."`
If the merchant does not sell the product, or there is not enough of the product, this is considered an unsuccessful sale and should instead return a string of the format
`"(merchant type) does not have enough (product) to sell."`
For a successful sale, it should also adjust the product count attribute, and cash attribute for the Merchant.

b. **restock(self, product, count)**
Method is used when a merchant completes a restocks a product, returning a string of the format (without parentheses)
"(merchant type) restocked (count) of (product)."
It should also adjust the product count attribute for the Merchant.
Note: a Merchant can't restock a product they don't already have.
Requirements: **Assert Statement(s)** for this method.

c. **travel_to_kingdom(self, kingdom)**
Adds the merchant to the given `Kingdom` instance. You will see that the `Kingdom` class has a `new_merchant()` method - think of the way to use these methods together to write less repetitive code.
Return `True` if the operation was successful, `False` otherwise.

Note: the operation is successful if the merchant can be added as a new merchant to the kingdom **and** the merchant has the given kingdom in their `kingdom_licenses`
Requirements: **Assert Statement(s)** for this method.
Hint: Check the merchant instance variables to get an idea of when `travel_to_kingdom` may return False.

Part 5.2: Constructor & methods for Kingdom class

Question 5.2.1 : Instance attributes of **Kingdom** class

A. **name (str):** Given by the constructor's argument. After initialization, `name` indicates the name of the Kingdom. The name of a Kingdom cannot be an empty string.
B. **curr_merchants (list):** NOT provided by the constructor's argument. You should initialize this attribute by yourself. `curr_merchants` is a list of `Merchant` objects, which is how the merchants that are currently active will be stored. After initialization, `curr_merchants` should **not** have any Merchants in it.

Question 4.2.2 : Methods of **Kingdom** class

A. Constructor (__init__):

The constructor has parameters described in Question 5.2.1 and the attributes names are the same as these parameters.

Requirements: **Assert Statement(s)** according to each attribute description above.

B. Getters:

a. **get_name(self)**

A getter method that returns the name of the Kingdom. This method creates the data abstraction.

Example: return 'Pythonia' if the Kingdom's name is 'Pythonia'

b. **get_curr_merchants(self)**

A getter method that returns the active merchants in the Kingdom. This method creates the data abstraction.

Note: All of these getter methods only need a single line.

C. String representation method

a. **__str__ (self)**

In order to print a string representation of an object we need to write a special method (will be covered in lectures later). This method enables us to use print statements on instances of Kingdom. Implement the special method using the getter methods created above (part 2B).

The format of the string should be (if printed, without parentheses, on a single line):

```
Kingdom (name) has (num curr_merchants) active merchants.
```

D. Core Functionality

a. **new_merchant(self, merchant):**

Adds a new merchant object to the Kingdom. Return `True` if the merchant was successfully added (i.e. the merchant is not already in the kingdom - don't forget about `kingdom_licenses`). If not, return `False`.

Requirements: **Assert Statement(s)** for this method.

Note: there is no need to modify/update the merchant object for this method.

b. **temp_ban(self, merchant):**

Remove occurrence of a merchant object from the kingdom. If the merchant object exists and is successfully removed, return `True`. If not, return `False`.

Requirements: **Assert Statement(s)** for this method.

c. **get_total_cash(self):**

Return the total amount of cash the active merchants in the kingdom have.

Requirements: **One-line list comprehension.** See Lab03 Instructions. You will **NOT** get credit for this question if you don't follow Lab03 instructions.

d. **promote_market(self):**

Sell out every merchant in the kingdom. This means all of their available stock should be sold. You may want to use the methods `get_stock()` and `sell()` from the Merchant class. You should return a string that contains information on all the Merchants interacted with, with each merchant being separated by a newline. If the Kingdom does not have any merchants at the time, return an empty string.

Example output (if printed, without parentheses):

(type_1) sold (count_1) of (product_1).
(type_1) sold (count_2) of (product_2).
(type_2) sold (count_1) of (product_1).
(type_2) sold (count_2) of (product_2).
<BLANKLINE>

Note: your solution should include whitespace, including a space at the end of each line.

`type_1` and `type_2` represent different merchants in the kingdom. If there are more merchants in the kingdom, or more products in the merchant's stock, more lines should be included in the returned string. If there is a product with 0 stock, it should not be included in the output.

Notes:

1. Don't forget to update `cash` (and `stock`)!
2. Use other methods when applicable to reduce repetitive code.
3. When adding merchants, don't modify the merchant object in any way.
4. When a merchant travels to a new kingdom, there is no need to remove them from any previous kingdoms.

Submission

Please submit the homework via Gradescope. You may submit more than once before the deadline, and only the final submission will be graded. Please refer to the Gradescope FAQ if you have any issues with your submission.