Homework 8

Total Points:

100 (Correctness and Style) + 3 EC (Checkpoint) + 5 EC (Question 5)

Due Dates (SD time):

- **Entire Assignment: Tuesday, March 5th, 11:59pm**
- **Checkpoint (read below): Sunday, March 3rd, 11:59pm**

Starter Files

Download hw08.zip. You will find a starter file for the questions of this homework. You cannot import anything to solve the problems except what is given in the starter code.

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 the following command to test your work:

```
>>> python3 -m doctest hw08.py
>>> py -m doctest hw08.py
>>> python -m doctest hw08.py
```

Also you can call one function at the time using:

```
>>> python3 -i hw08.py
```

Checkpoint Submission

Due Date: Sunday, March 5th, 11:59pm (SD time)

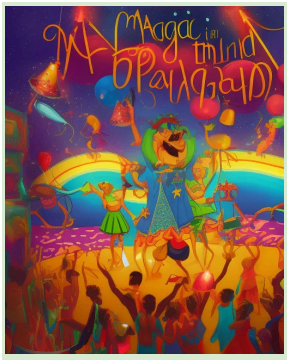
You can earn up to **3 points extra credit** by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete **Questions 1 and 2** and submit the hw08.py file to gradescope.

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 noticed any errors.

General Notes and Requirements

1. **DO NOT IMPORT ANY PACKAGES.**
2. In this homework, you only need to add doctests and docstrings for **Question 4.** For other questions, it's not required to add them, but it's still recommended to follow the full style guide.

Required Questions



It may seem that the people of Pythonia work and study all the time, but fortunately, that is not true! There is always time for a fun party to hang out with friends.

Question 1

In this question, we will design a class hierarchy for organizing different parties. Doctests for this question are in the method `q1_doctests()`.

You are strongly encouraged to expand upon the provided doctests to validate your code. **Not all cases have been tested in the doctests.**

Part 1.1: ImaginaryParty

The class `ImaginaryParty` contains general information about a generic party. This class primarily exists to define shared behavior for its subclasses, which is a specific type of party.

Class attributes:	
`theme`	its value is 'Imagination'
`max_guests`	its value is 10
Instance attributes (set via the constructor):	
`host_name`	non-empty string
`location`	non-empty string
`start_time`	non-empty string
Static method:	
`party_fact`	It returns a fact about the party; in particular, it returns a string: *"The limits of this party are up to your imagination!"*
Instance Methods:	
`generate_party_info`	It takes an unknown number of possible party details (each is a string) and returns a new string that combines the party's details and descriptions. See the doctests for more details.
`__str__`	Special method that allows us to print a string representation of an object. It returns a string combining the host's name, party location, and start time. See the doctests for more details.

Part 1.2: ArtsAndCrafts

This is a subclass of the `ImaginaryParty` class. `ArtsAndCrafts` will modify the following attributes of the parent class:

Class attributes:	
`theme`	its value is 'Arts and Crafts'
`supplies`	Initially set to 3
Class method:	
`get_remaining_spots`	Returns the number of people that can be accepted into the party by using the supplies variable. Each supply corresponds to 1 person.
Instance Methods:	
`invite`	It should first check if there are enough supplies to invite a guest (at least 1). • If there are enough, then it should subtract 1 from the supplies variable, and **print** out the current ArtsAndCrafts object by using the `self` keyword. • If there are not enough supplies, then it should return "Not enough supplies"

Part 1.3: PythoniaGastronomicAdventure

This is a subclass of the `ImaginaryParty` class. `PythoniaGastronomicAdventure` will modify the following attributes of the parent class:

Class attributes:	
`theme`	its value is 'Gastronomic Adventure'
Instance Methods:	

`generate_party_info`	Because of allergy concerns at this party, the message returned by this method will start with "Allergy Warning!", then on a new line, it should return the same message generated by the `generate_party_info` method of the parent class. **You must use the super() method to access the parent's generate_party_info() method so that we do not reuse code.** Hint: The unpacking operator `*my_list` can be useful

Question 2

No party is complete without fun activities. You want to start your party off with a bang, and decide to launch some fireworks.

In this question, we will write a class `Firework` with basic implementation of the common firework functionality. Doctests for this question are in the method `q2_doctests()`. You are **not** required to add any additional doctests, but it's highly recommended to add more for your own benefit. We will test your code on another set of inputs.

Part 2.1: Firework

The class `Firework` represents a general firework. This class primarily exists to define shared behavior for its subclasses.

Instance attributes:	
`brand`	non-empty string containing the firework's brand. Passed as a parameter to the constructor.
`brightness`	is the level of the brightness of the firework in lumens. The default value is 5.
`launches`	is the number of times the firework has been launched. The default value is 0.
`noise_made`	is the total noise the fireworks make in decibels. The default value is 0.
`noise_complaints`	is the number of complaints your neighbors have made. The default value is 0.
Instance Methods:	
`launch(self)`	Launch a firework. Read the description below.
`launch_several(self, amount)`	Launch fireworks the `amount` of times. You should return the outcome of all launches. **Note**: There should not be a blank line at the end of the returning string.

`launch(self):` The result of the launch should alternate, with the first launch being successful (the second one should be a dud, the third should be successful, …). Think about how you can use the `launches` attribute to implement this. If the firework properly launches, it should create 150 db of noise, if the firework is a dud it should create 30 db of noise.

- The function should return a string:

```
"The {brand} firework was launched and it was a {successful/dud} launch. It created {x} decibels of noise."
```

- Do not forget to update the `launches` and `noise_made` attributes for each launch.

Part 2: Class Firecracker

This is a subclass of the `Firework` class. `Firecracker` will have the same instance attributes and methods as `Firework`. The following are the only differences:

- `Firecracker` starts with 3 lumens for **brightness.** Do not repeat yourself, reuse a constructor from a parent class; keyword `super` is helpful here.
- `Firecracker` has a new method **calculate_complaints(self)**, which will set the number of complaints to the right amount, according to the total noise made. There should be a noise complaint for every 250 db of noise made.
    - Return "Received {x} new noise complaints." if there is enough noise to warrant a **new** complaint.
    - Return "No new complaints!" otherwise.
    - Note that `noise_made` does **not** get cleared to 0 after receiving a new complaint, and that you will need to update your `noise_complaints` attribute.

Part 3: RomanCandle

This is a subclass of the `Firecracker` class. `RomanCandle` will have the same instance attributes and methods as `Firecracker`. The following are the differences:

- `RomanCandle` has a new instance attribute **cops_called** which is an integer that should be set to `0` by default**.** Do not repeat yourself, reuse a constructor from a parent class; keyword `super` is helpful here.
- `RomanCandle` also has the method **calculate_complaints(self)**, which you should override (but the keyword super can still be used here). The method will function similarly to its parent's, except for every 4 noise complaints, the cops will be called. If the cops are called, **cops_called** should be updated.
    - If the cops are called for that calculation, you should **return** "COPS CALLED!!" and then the same output as `Firecracker` **calculate_complaints(self)**.
    - If the cops aren't called for that calculation, you should return the same output as `Firecracker` **calculate_complaints(self)**.

Check out this document for a more detailed explanation of the doctests and their outputs.

Note: We only provided the base class (`Firework`) in the starter code, you need to complete the rest of the classes (including the class definitions) yourself.

Exceptions

The noise from the fireworks attracted too much unwanted attention, so you moved the party inside and kept it quiet for the night. You decided to conduct a raffle and host dinner at your party, and you are trying to write a few functions to speed up the process.

Question 3

In this question, you need to fix the implementation of three functions so that they no longer crash. The provided implementation will not work for certain values due to exceptions, thus you need to add a try-except block in each function to deal with this. You should run the doctests to see which exceptions will be thrown. **No additional doctests are needed.**

In particular, all you need to do is add a `try-except` block to handle the potential exception. You should **not** modify the actual implementation of the given code, and you are **not** required to add input validations. The provided method descriptions summarize the intended behaviors.

Important Requirements: You will **NOT** receive credit for the part if you don't follow.

1. In each question, you **must** specify the specific error type(s) in the except block(s). You cannot use
   - `except:`
   - `except Exception:`
   - `except Exception as e:`
2. Read this article for more detailed explanations of the recommended coding habit.

Part 3.1

Write a function that generates raffle tickets. The function is supposed to concatenate every element from *letters* with every letter from *numbers*, and return a list of possible raffle tickets. However, the current implementation crashes because of some bad inputs. If we catch the exception, we should just move on to the next iteration of the loop instead of terminating the entire loop. Do not use **pass** (for reason read the linked article, you don't need to log results).

```
def create_raffle_tix(letters, numbers):
    """
    >>> create_raffle_tix(["A", "B", "C"], ["1", 2, "3"])
    ['A1', 'A3', 'B1', 'B3', 'C1', 'C3']

    >>> create_raffle_tix([], [])
    []

    >>> create_raffle_tix(["X", None, "Y"], ["1", 2, [3]])
    ['X1', 'Y1']
    """
    tickets = []
    for letter in letters:
        for number in numbers:
            out.append(str1 + str2)  # add try-except block, specific
to the exception
    return tickets
```

Part 3.2

When sending out invitations to your party, you allowed your guests to upload a file in case they had any special requests, such as allergies. However, some of your guests uploaded corrupted files.

You tried writing a function that is supposed to open each filepath in **filepaths*. If we are able to open the file, we should **print** a string '*{filepath}* opened successfully'. If we are not able to open the file, we should **print** '*{filepath}* not found'. Note that *{filepath}* should be the given file path string, not the actual string '{filepath}'.

```
def special_requests(*filepaths):
    """
          >>>  special_requests('files/vegetarian.txt',  'files/b.txt',
'files/c.txt')
```

```
    files/vegetarian.txt opened successfully
    files/b.txt not found
    files/c.txt not found

    >>> special_requests('random.txt')
    random.txt not found
    """
    for filepath in filepaths:
         cur_file = open(filepath, "r") # add try-except block specific
to the exception
        cur_file.close()
```

Part 3.3

You are trying to create table placards for your party, and you asked everyone to provide their first and last name. However, some invitees only provided their first name and some invitees didn't format their names properly.

This function is trying to concatenate elements from the *first_names* list with its corresponding elements in the *last_names* list. In the current implementation, there are two potential errors that may arise. If they are thrown, you should catch them and print the error types (you can use the type() function). Otherwise, you would just append first_names[i] + last_names[i] to the output list.

```
def create_placards(first_names, last_names):
    """
     >>> create_placards(["cj", "tank", "nico", "dalton"], ["stroud",
"dell"])
    <class 'IndexError'>
    <class 'IndexError'>
    ['cj stroud', 'tank dell']

    >>> create_placards(["brock", "george", "brandon"], [None, None])
    <class 'TypeError'>
    <class 'TypeError'>
    <class 'IndexError'>
    []

    >>> create_placards([], [])
    []
    """
```

```
    names = []
    for i in range(len(first_names)):
            names.append(first_names[i] + “ ” + last_names[i]) # add
try-except specific to the exception
    return names
```

Question 4

Your guests were given a raffle ticket upon entering the party, and each ticket had a corresponding resolution that participants committed to achieving in the coming year. Unfortunately, the formatting is off and needs to be validated.

For this question, we will be validating New Year's Resolutions and Commitments using **exceptions** and NOT assertions. If a check fails, you will throw an *exception* using **raise**. You should make sure you are doing the following checks **in the specified order** to prevent issues with the doctests expecting the wrong exception.

Here is an example of what it means to throw an exception. Consider the following function that takes in a string:

```
def duplicate_and_flip(input_string):
    return input_string + input_string[::-1]
```

If we wanted to raise an error whenever the function receives a non-string input, we could modify the code to the following:

```
def duplicate_and_flip(input_string):
    if not (what goes here?):
        raise TypeError("Invalid input detected")
    return input_string + input_string[::-1]
```

Note that when you are writing doctests, you must write them in the following format:

```
    """
    >>> duplicate_and_flip(100)
    Traceback (most recent call last):
    ...
    TypeError: Invalid input detected
    """
```

You do not need to print or return the output message. This "expected output" tells the doctest module to expect a `TypeError` exception with message "Invalid input detected".

Checks (in this order):

1. `resolution` should be a non-empty string.
2. All characters in `resolution` should be only alphabetic characters or spaces (no numbers/symbols).
3. `commitment` should be a non-empty list of strings.
4. Each string in `commitment` should start with the phrase 'I commit to'.

If all of the checks pass, you should **return** the string 'New Years resolution validated'.

Refer to the doctests to see which exceptions are thrown for each of the checks and how the corresponding messages should be formatted.

```
def validate_resolution(resolution, commitments):
    """
    >>> resolution = 'Exercise regularly'
    >>> commitments = ['I commit to run twice a week', \
                       'I commit to go to the gym every day']
    >>> validate_resolution(resolution, commitments)
    'New Years resolution validated'
    >>> validate_resolution(resolution, ['Funny joke'])
    Traceback (most recent call last):
    ...
    ValueError: commitment does not start with 'I commit to'
    >>> validate_resolution(resolution, 1)
    Traceback (most recent call last):
    ...
    TypeError: commitments is not a list
```

```
>>> validate_resolution('', [1, 2, 'hi'])
Traceback (most recent call last):
...
TypeError: resolution is empty
>>> validate_resolution('Hello', ['hi', 2, 'hi'])
Traceback (most recent call last):
...
TypeError: non-string item in commitments at index 1
>>> validate_resolution('hi', [])
Traceback (most recent call last):
...
TypeError: commitments is empty
>>> validate_resolution('1: idk', [])
Traceback (most recent call last):
...
ValueError: non-alpha character in resolution
>>> validate_resolution(False, 0)
Traceback (most recent call last):
...
TypeError: resolution is not a string
"""
```

Question 5 (Extra Credit)

Write a recursive function that flattens nested lists and tuples (no dictionaries or sets) (`nested_input`) and returns a flattened list. In the final flattened list, it should contain every **integer** that was present within the original list/tuple. There may be values of all types: strings, floats, etc (but only integers should be included in the final list).

Requirement: Recursion. No explicit for loops, while loops, map, filter, or list comprehensions. Do not modify the function header.

Example:

Input (*nested_input*):	Output:
[1, (2, "Hi"), [4, (5.1, 6)], 7, [(8, 9), 10]]	[1, 2, 4, 6, 7, 8, 9, 10]

```
def flatten(nested_input):
    """
    >>> flatten([1, (2, "Hi"), [4, (5.1, 6)], 7, [(8, 9), 10]])
    [1, 2, 4, 6, 7, 8, 9, 10]
     >>> flatten([3.5, ("Hello", 2), [5, (6.2, "World")], 8, [(9, 10),
"11"]])
    [2, 5, 8, 9, 10]
      >>> flatten([(1.1, "A"), [2, (3.3, "B", 4)], 5, ("C", 6, [7,
"D"]), 8.8])
    [2, 4, 5, 6, 7]
    """
```

Submission

When submitting to Gradescope, you can include the files folder by compressing it with your hw08.py file, creating a .zip file you can submit that zip file to Gradescope. If you have your hw08.py and files folder inside of another folder, make sure that you select hw08.py and the files folder directly to compress it, rather than compressing the encompassing folder, as it will cause issues with your submission.

You can do this by going into File Explorer or Finder and selecting **both** the files folder and the hw08.py file. Right-click and then select either

Windows: Send to -> Compressed (zipped) folder
Mac: Compress 2 items

Your .zip file should have the following files if you open it:

```
files/
--- file1.txt
--- file2.txt
--- ... (other files)
hw08.py
```

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.