CS101. A15 - Collatz Conjecture

Ref: https://en.wikipedia.org/wiki/Collatz_conjecture

The Collatz Conjecture or 3x+1 problem can be summarized as follows:

Take any positive integer n. If n is even, divide n by 2 to get n / 2. If n is odd, multiply n by 3 and add 1 to get 3n + 1. Repeat the process indefinitely. The conjecture states that no matter which number you start with, you will always reach 1 eventually.

Given a number n, return the number of steps required to reach 1. Starting with n = 12, the steps would be as follows:

1. 12
2. 6
3. 3
4. 10
5. 5
6. 16
7. 8
8. 4
9. 2
10. 1

Resulting in 9 steps. So for input n = 12, the return value would be 9.

- When argument is zero or a negative integer: `raise ValueError("Only positive numbers are allowed")`

**Tasks in the Assignment**

- Implement one function: steps(number)
- Input: Integer
- Return value: Steps required to reach 1 through the Collatz conjecture.

**Code Template (collatz_conjecture.py)**

```
"""
Collatz Conjecture
"""

def steps(number):
     pass
```

**Sample output**

```
print(steps(12))
9

print(sieve(1024))
10
```

**Test Cases**

| Function | Inputs | Output | Remarks |
|---|---|---|---|
| convert | 0 | ValueError | |
| convert | -1 | ValueError | |
| convert | 1 | 0 | |
| convert | 1024 | 10 | |
| convert | 12 | 9 | |
| convert | 77777 | 169 | |
| convert | 1000000 | 152 | |

********

CS101. A16 - Class Grades

Write a few functions to count and calculate results for a class.

**1. Rounding Scores**

Overall exam scores have to be Integers. Before assigning grades using the exam scores, change any float scores into ints. You can use Python's built-in round() function.

A score of 75.45 or 75.49 will round to 75. A score of 43.50 or 43.59 will round to 44. Input will not have any scores that have more than two places after the decimal point.

Create the function round_scores() that takes a list of student_scores. This function should *consume* the input list and return a new list with all the scores converted to ints. The order of the scores in the resulting list is not important.

```
>>> student_scores = [90.33, 40.5, 55.44, 70.05, 30.55, 25.45, 80.45, 95.3,
38.7, 40.3]
>>> round_scores(student_scores)
...
[40, 39, 95, 80, 25, 31, 70, 55, 40, 90]
```

**2. Non-Passing Students**

Write a function to count how many students failed. Create the function count_failed_students() that takes the list of student_scores. This function should count up the number of students who don't have passing scores and return that count as an integer. A student needs a score greater than **40** to achieve a passing grade on the exam.

```
>>> count_failed_students(student_scores=[90,40,55,70,30,25,80,95,38,40])
5
```

**3. The "Best"**

Find the student scores that are **greater than or equal to** a given threshold. Create the function above_threshold() taking student_scores, and threshold (the "top score" threshold) as parameters. This function should return a list of all scores that are >= to threshold.

```
>>> above_threshold(student_scores=[90,40,55,70,30,68,70,75,83,96],
threshold=75)
[90,75,83,96]
```

**4. Calculating Letter Grades**

The "letter grade" lower thresholds are calculated based on the highest score achieved, and increment evenly between the high score and the failing threshold of **<= 40**.

Create the function letter_grades() that takes the "highest" score on the exam as a parameter, and returns a list of lower score thresholds for each grade interval: ["D", "C", "B", "A"].

```
"""Where the highest score is 100, and failing is <= 40.
       "F" <= 40
 41 <= "D" <= 55
 56 <= "C" <= 70
```

```
 71 <= "B" <= 85
 86 <= "A" <= 100
"""

>>> letter_grades(highest=100)
[41, 56, 71, 86]


"""Where the highest score is 88, and failing is <= 40.
       "F" <= 40
 41 <= "D" <= 52
 53 <= "C" <= 64
 65 <= "B" <= 76
 77 <= "A" <= 88
"""

>>> letter_grades(highest=88)
[41, 53, 65, 77]
```

**5. Matching Names to Scores**

Two lists are available as input: a) list of exam scores in descending order (student_scores), and b) list of student names also sorted in descending order by their exam scores (student_names). Your task is to match each student name with their exam score and print out an overall class ranking. Match each student name on the student_names list with their score from the student_scores list. Your function should return a list of strings with the format <rank>. <student name>: <student score>.

```
>>> student_scores = [100, 99, 90, 84, 66, 53, 47]
>>> student_names =  ['Aditya', 'Leslie', 'Vinayak', 'Shivam', 'Abhilash',
'Karan', 'Vishnu']
>>> student_ranking(student_scores, student_names)
...
['1. Aditya: 100', '2. Leslie: 99', '3. Vinayak: 90', '4. Shivam: 84', '5.
Abhilash: 66', '6. Karan: 53', '7. Vishnu: 47']
```

**6. A "Perfect" Score**

Although a "perfect" score of 100 is rare on an exam, it is interesting to know if at least one student has achieved it.

Create the function perfect_score() with parameter student_info. student_info is a list of lists containing the name and score of each student: [["Don", 90], ["Harry", 80]]. The function should return *the first* [<name>, <score>] pair of the student who scored 100 on the exam.

If no 100 scores are found in student_info, an empty list [] should be returned.

```
>>> perfect_score(student_info=[["Charles", 90], ["Tony", 80], ["Alex",
100]])
["Alex", 100]

>>> perfect_score(student_info=[["Charles", 90], ["Tony", 80]])
```

[]

**Code Template (class_grade.py)**

```
"""
Working with loops, lists
"""

def round_scores(student_scores):
      """
      :param student_scores: list of student exam scores as float or int.
      :return: list of student scores *rounded* to nearest integer value.
      """
      pass

def count_failed_students(student_scores):
      """
      :param student_scores: list of integer student scores.
      :return: integer count of student scores at or below 40.
      """
      pass

def above_threshold(student_scores, threshold):
      """
      :param student_scores: list of integer scores
      :param threshold :  integer
      :return: list of integer scores that are at or above the "best"
threshold.
      """
      pass

def letter_grades(highest):
      """
      :param highest: integer of highest exam score.
      :return: list of integer lower threshold scores for each D-A letter
grade interval.
            For example, where the highest score is 100, and failing is <=
40,
            The result would be [41, 56, 71, 86]:

            41 <= "D" <= 55
            56 <= "C" <= 70
            71 <= "B" <= 85
            86 <= "A" <= 100
      """
      pass

def student_ranking(student_scores, student_names):
      """
      :param student_scores: list of scores in descending order.
      :param student_names: list of names in descending order by exam score.
      :return: list of strings in format ["<rank>. <student name>:
<score>"].
      """
      pass

def perfect_score(student_info):
      """
      :param student_info: list of [<student name>, <score>] lists
```

```
    :return: first `[<student name>, 100]` or `[]` if no student score of
100 is found.
    """
    pass
```

**Sample output**

```
print(round_scores([90.33, 40.5, 55.44, 70.05, 30.55, 25.45, 80.45, 95.3,
38.7, 40.3]))
[90, 40, 55, 70, 31, 25, 80, 95, 39, 40]

print(count_failed_students([40, 40, 35, 70, 30, 41, 90]))
4

print(above_threshold([40, 39, 95, 80, 25, 31, 70, 55, 40, 98], 98))
[98]
```

**Test Cases**

| Function | Inputs | Output | Remarks |
|---|---|---|---|
| round_scores | [ ] | [ ] | |
| | [.5] | [0] | |
| | [1.5] | [2] | |
| | [90.33, 40.5, 55.44, 70.05, 30.55, 25.45, 80.45, 95.3, 38.7, 40.3] | [90, 40, 55, 70, 31, 25, 80, 95, 39, 40] | |
| | [50, 36.03, 76.92, 40.7, 43, 78.29, 63.58, 91, 28.6, 88.0] | [50, 36, 77, 41, 43, 78, 64, 91, 29, 88] | |
| count_failed_students | [89, 85, 42, 57, 90, 100, 95, 48, 70, 96] | 0 | |
| | [40, 40, 35, 70, 30, 41, 90] | 4 | |
| above_threshold | [40, 39, 95, 80, 25, 31, 70, 55, 40, 90], 98 | [ ] | |
| | [88, 29, 91, 64, 78, 43, 41, 77, 36, 50], 80 | [88, 91] | |
| | [100, 89], 100 | [100] | |
| | [88, 29, 91, 64, 78, 43, 41, 77, 36, 50], 78 | [88, 91, 78] | |
| | [], 80 | [ ] | |
| letter_grades | 100 | [41, 56, 71, 86] | |
| | 97 | [41, 55, 69, 83] | |
| | 85 | [41, 52, 63, 74] | |

| | | | |
|---|---|---|---|
| | 92 | [41, 54, 67, 80] | |
| | 81 | [41, 51, 61, 71] | |
| student_ranking | [100, 98, 92, 86, 70, 68, 67, 60], ['Rui', 'Betty', 'Joci', 'Yoshi', 'Kora', 'Bern', 'Jan', 'Rose'] | ['1. Rui: 100', '2. Betty: 98', '3. Joci: 92', '4. Yoshi: 86', '5. Kora: 70', '6. Bern: 68', '7. Jan: 67', '8. Rose: 60'] | |
| | [82], ['Betty'] | ['1. Betty: 82'] | |
| | [88, 73], ['Paul', 'Ernest'] | ['1. Paul: 88', '2. Ernest: 73'] | |
| perfect_score | [ ['Joci', 100], ['Vlad', 100], ['Raiana', 100], ['Alessandro', 100] ] | ['Joci', 100] | |
| | [ ['Jill', 30], ['Paul', 73] ] | [ ] | |
| | [ ['Rui', 60], ['Joci', 58], ['Sara', 91], ['Kora', 93], ['Alex', 42], ['Jan', 81], ['Lilliana', 40], ['John', 60], ['Bern', 28], ['Vlad', 55] ] | [ ] | |
| | [ ['Yoshi', 52], ['Jan', 86], ['Raiana', 100], ['Betty', 60],['Joci', 100], ['Kora', 81], ['Bern', 41], ['Rose', 94] ] | ['Raiana', 100] | |
| | [ ] | [ ] | |

********

## CS101. A17 - Queue at the Theme Park

A very popular theme park has a world famous rollercoaster ride. There are two queues for this ride, each represented as a list:

1. Normal Queue
2. Express Queue (*also known as the Fast-track*) - where people pay extra for priority access.

Your task is to write Python code to better manage the crowd in the queues.

**1. Add me to the queue**

Add one person to one of the two queues based on their ticket type. Define the **add_me_to_the_queue()** function that takes 4 parameters <express_queue>, <normal_queue>, <ticket_type>, <person_name> and returns the appropriate queue updated with the person's name.

1. <ticket_type> is an int with 1 == express_queue and 0 == normal_queue.
2. <person_name> is the name (as a str) of the person to be added to the respective queue.

```
>>> add_me_to_the_queue(express_queue=["Shambhu", "Suppandi"],
normal_queue=["Tantri", "Kalia"], ticket_type=1, person_name="Doob doob")
...
["Shambhu", "Suppandi", "Doob doob"]

>>> add_me_to_the_queue(express_queue=["Tony", "Bruce"],
normal_queue=["RobotGuy", "WW"], ticket_type=0, person_name="HawkEye")
....
["RobotGuy", "WW", "HawkEye"]
```

**2. Where are my friends?**

One person arrived late at the park but wants to join the queue where their friends have been waiting. But they have no idea where their friends are standing and there isn't any phone reception to call them.

Define the find_my_friend() function that takes 2 parameters: `queue` and `friend_name`.

1. <queue> is the list of people standing in the queue.
2. <friend_name> is the name of the friend whose index (place in the queue) you need to find.

Remember: Indexing starts at 0 from the left, and -1 from the right.

```
>>> find_my_friend(queue=["Natasha", "Steve", "T'challa", "Wanda", "Rocket"],
friend_name="Steve")
...
1
```

**3. Can I please join them?**

Now that their friends have been found (in task #2 above), the late arriver would like to join them at their place in the queue. Define the add_me_with_my_friends() function that takes 3 parameters: queue, index, and person_name.

1. <queue> is the list of people standing in the queue.
2. <index> is the position at which the new person should be added.

3. <person_name> is the name of the person to add at the index position.

Return the queue updated with the late arrivals name.

```
>>> add_me_with_my_friends(queue=["Natasha", "Steve", "T'challa", "Wanda",
"Rocket"], index=1, person_name="Bucky")
...
["Natasha", "Bucky", "Steve", "T'challa", "Wanda", "Rocket"]
```

**4. Mean a person in the queue**

Define the remove_the_mean_person() function that takes 2 parameters: queue and person_name.

1. <queue> is the list of people standing in the queue.
2. <person_name> is the name of the person that needs to be kicked out.

```
>>> remove_the_mean_person(queue=["Natasha", "Steve", "Eltran", "Wanda",
"Rocket"], person_name="Eltran")
...
["Natasha", "Steve", "Wanda", "Rocket"]
```

**5. Namefellows**

You may not have seen two unrelated people who look exactly the same, but you have *definitely* seen unrelated people with the exact same name (*namefellows*)! Today, it looks like there are a lot of them in attendance. You want to know how many times a particular name occurs in the queue.

Define the how_many_namefellows() function that takes 2 parameters queue and person_name.

1. <queue> is the list of people standing in the queue.
2. <person_name> is the name you think might occur more than once in the queue.

Return the number of occurrences of person_name, as an int.

```
>>> how_many_namefellows(queue=["Natasha", "Steve", "Eltran", "Natasha",
"Rocket"], person_name="Natasha")
...
2
```

**6. Remove the last person**

Define the function remove_the_last_person() that takes 1 parameter queue, which is the list of people standing in the queue.

You should update the list and also return the name of the person who was removed, so you can write them a refund.

```
>>> remove_the_last_person(queue=["Natasha", "Steve", "Eltran", "Natasha",
"Rocket"])
...
'Rocket'
```

## 7. Sort the Queue List

For administrative purposes, you need to get all the names in a given queue in alphabetical order.

Define the sorted_names() function that takes 1 argument, queue, (the list of people standing in the queue), and returns a sorted copy of the list.

```
>>> sorted_names(queue=["Natasha", "Steve", "Eltran", "Natasha", "Rocket"])
...
['Eltran', 'Natasha', 'Natasha', 'Rocket', 'Steve']
```

**Code Template (list_methods_theme_park.py)**

```
"""
Working with lists with the Hypercoaster example
"""

def add_me_to_the_queue(express_queue, normal_queue, ticket_type,
person_name):
      """

      :param express_queue: list - names in the Fast-track queue.
      :param normal_queue:  list - names in the normal queue.
      :param ticket_type:  int - type of ticket. 1 = express, 0 = normal.
      :param person_name: str - name of person to add to a queue.
      :return: list - the (updated) queue the name was added to.
      """
      pass

def find_my_friend(queue, friend_name):
      """

      :param queue: list - names in the queue.
      :param friend_name: str - name of friend to find.
      :return: int - index at which the friends name was found.
      """
      pass

def add_me_with_my_friends(queue, index, person_name):
      """

      :param queue: list - names in the queue.
      :param index: int - the index at which to add the new name.
      :param person_name: str - the name to add.
      :return: list - queue updated with new name.
      """
      pass

def remove_the_mean_person(queue, person_name):
      """

      :param queue: list - names in the queue.
      :param person_name: str - name of mean person.
      :return:  list - queue update with the mean persons name removed.
      """
```

```
    pass

def how_many_namefellows(queue, person_name):
    """

    :param queue: list - names in the queue.
    :param person_name: str - name you wish to count or track.
    :return:  int - the number of times the name appears in the queue.
    """
    pass

def remove_the_last_person(queue):
    """

    :param queue: list - names in the queue.
    :return: str - name that has been removed from the end of the queue.
    """
    pass


def sorted_names(queue):
    """

    :param queue: list - names in the queue.
    :return: list - copy of the queue in alphabetical order.
    """
    pass
```

**Sample output**

```
Shown in the description.
```

**Test Cases**

| Function | Inputs | Output | |
|---|---|---|---|
| add_me_to_the_queue | ['Tony', 'Bruce'], ['RobotGuy', 'WW'], 0, 'HawkEye' | ['RobotGuy', 'WW', 'HawkEye'] | |
| | ['Tony', 'Bruce'], ['RobotGuy', 'WW'], 1, 'RichieRich' | ['Tony', 'Bruce', 'RichieRich'] | |
| | [‘Suppandi’], [‘Kalia’],0,’Shambhu’ | [‘Kalia’, ‘Shambhu’] | |
| find_my_friend | ((['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'], ‘Suppandi' | 0 | |
| | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'], ‘Kalia' | 1 | |
| | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'], ‘Hodja’ | 4 | |
| add_me_with_my_friends | ['Natasha', 'Steve', 'Tchalla', 'Wanda', 'Rocket'], 0, 'Bucky' | ['Bucky', 'Natasha', 'Steve', 'Tchalla', 'Wanda', 'Rocket'] | |
| | ['Natasha', 'Steve', 'Tchalla', 'Wanda', 'Rocket'], 1, 'Bucky' | ['Natasha', 'Bucky', 'Steve', 'Tchalla', 'Wanda', 'Rocket'] | |

| | | | |
|---|---|---|---|
| | ['Natasha', 'Steve', 'Tchalla', 'Wanda', 'Rocket'], 5, 'Bucky' | ['Natasha', 'Steve', 'Tchalla', 'Wanda', 'Rocket', 'Bucky'] | |
| remove_the_mean_person | ['Natasha', 'Steve', 'Ultron', 'Wanda', 'Rocket'], 'Ultron' | ['Natasha', 'Steve', 'Wanda', 'Rocket'] | |
| | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja','Doob doob'], 'Doob doob' | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'] | |
| | ['Ultron', 'Natasha', 'Steve', 'Wanda', 'Rocket'], 'Ultron' | ['Natasha', 'Steve', 'Wanda', 'Rocket'] | |
| how_many_namefellows | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'], 'Hodja' | 1 | |
| | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'], 'Rajah' | 0 | |
| | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'], 'Tantri' | 1 | |
| remove_the_last_person | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'] | Hodja | |
| sorted_names | ['Steve', 'Ultron', 'Natasha', 'Rocket'] | ['Natasha', 'Rocket', 'Steve', 'Ultron'] | |
| | ['Suppandi', 'Kalia', 'Shambhu', 'Tantri', 'Hodja'] | ['Hodja', 'Kalia', 'Shambhu', 'Suppandi', 'Tantri'] | |

********