CS101. A18 - Card Games

Ref: Card Games - Exercism

Elyse is really looking forward to playing some poker (and other card games) during her upcoming trip to Vegas. Being a big fan of "self-tracking" she wants to put together some small functions that will help her with tracking tasks and has asked for your help thinking them through.

1. Tracking Poker Rounds

Elyse is especially fond of poker, and wants to track how many rounds she plays - and *which rounds* those are. Every round has its own number, and every table shows the round number currently being played. Elyse chooses a table and sits down to play her first round. She plans on playing three rounds.

Implement a function get_rounds(<round_number>) that takes the current round number and returns a single list with that round and the *next two* that are coming up:

```
>>> get_rounds(27)
[27, 28, 29]
```

2. Keeping all Rounds in the Same Place

Elyse played a few rounds at the first table, then took a break and played some more rounds at a second table ... but ended up with a different list for each table! She wants to put the two lists together, so she can track all of the poker rounds in the same place.

Implement a function concatenate_rounds(<rounds_1>, <rounds_2>) that takes two lists and returns a single list consisting of all the rounds in the first list, followed by all the rounds in the second list:

```
>>> concatenate_rounds([27, 28, 29], [35, 36])
[27, 28, 29, 35, 36]
```

3. Finding Prior Rounds

Talking about some of the prior Poker rounds, another player remarks how similarly two of them played out. Elyse is not sure if she played those rounds or not.

Implement a function list_contains_round(<rounds>, <round_number>) that takes two arguments, a list of rounds played and a round number. The function will return True if the round is in the list of rounds played, False if not:

```
>>> list_contains_round([27, 28, 29, 35, 36], 29)
True
>>> list_contains_round([27, 28, 29, 35, 36], 30)
False
```

4. Averaging Card Values

Elyse wants to try out a new game called Black Joe. It's similar to Black Jack - where your goal is to have the cards in your hand add up to a target value - but in Black Joe the goal is to get the *average* of the card values to be 7. The average can be found by summing up all the card values and then dividing that sum by the number of cards in the hand.

Implement a function card_average(<hand>) that will return the average value of a hand of Black Joe.

```
>>> card_average([5, 6, 7])
6.0
```

5. Alternate Averages

In Black Joe, speed is important. Elyse is going to try and find a faster way of finding the average.

She has thought of two ways of getting an *average-like* number:

- Take the average of the *first* and *last* number in the hand.
- Using the median (middle card) of the hand.

Implement the function approx_average_is_average(<hand>), given hand, a list containing the values of the cards in your hand.

Return True if either *one* or *both* of the, above named, strategies result in a number *equal* to the *actual average*.

Note: The length of all hands are odd, to make finding a median easier.

```
>>> approx_average_is_average([1, 2, 3])
True
>>> approx_average_is_average([2, 3, 4, 8, 8])
True
>>> approx_average_is_average([1, 2, 3, 5, 9])
False
```

6. More Averaging Techniques

Intrigued by the results of her averaging experiment, Elyse is wondering if taking the average of the cards at the *even* positions versus the average of the cards at the *odd* positions would give the same results. Time for another test function!

Implement a function average_even_is_average_odd(<hand>) that returns a Boolean indicating if the average of the cards at even indexes is the same as the average of the cards at odd indexes.

```
>>> average_even_is_average_odd([1, 2, 3])
True
>>> average_even_is_average_odd([1, 2, 3, 4])
False
```

7. Bonus Round Rules

Every 11th hand in Black Joe is a bonus hand with a bonus rule: if the last card you draw is a Jack, you double its value.

Implement a function maybe_double_last(<hand>) that takes a hand and checks if the last card is a Jack (11). If the the last card **is** a Jack (11), double its value before returning the hand.

```
>>> hand = [5, 9, 11]
>>> maybe_double_last(hand)
[5, 9, 22]
>>> hand = [5, 9, 10]
```

```
>>> maybe_double_last(hand)
[5, 9, 10]
```

Code Template (lists_card_games.py)

```
"""
Exercise with lists in Python
"""

def get_rounds(number):
    """

    :param number: int - current round number.
    :return: list - current round and the two that follow.
    """
    pass

def concatenate_rounds(rounds_1, rounds_2):
    """

    :param rounds_1: list - first rounds played.
    :param rounds_2: list - second set of rounds played.
    :return: list - all rounds played.
    """
    pass

def list_contains_round(rounds, number):
    """

    :param rounds: list - rounds played.
    :param number: int - round number.
    :return:  bool - was the round played?
    """
    pass

def card_average(hand):
    """

    :param hand: list - cards in hand.
    :return:  float - average value of the cards in the hand.
    """
    pass

def approx_average_is_average(hand):
    """

    :param hand: list - cards in hand.
    :return: bool - is approximate average the same as true average?
    """
    pass

def average_even_is_average_odd(hand):
    """

    :param hand: list - cards in hand.
    :return: bool - are even and odd averages equal?
    """
    pass
```

```
def maybe_double_last(hand):
      """

      :param hand: list - cards in hand.
      :return: list - hand with Jacks (if present) value doubled.
      """
      pass
```

Sample output

```
Shown in the description
```

Test Cases

Function	Inputs	Output	
get_rounds	0	[0, 1, 2]	
get_rounds	1	[1, 2, 3]	
get_rounds	10	[10, 11, 12]	
get_rounds	27	[27, 28, 29]	
get_rounds	99	[99, 100, 101]	
get_rounds	666	[666, 667, 668]	
concatenate_rounds	[ ], [ ]	[ ]	
concatenate_rounds	[0, 1], [ ]	[0, 1]	
concatenate_rounds	[ ], [1, 2]	[1, 2]	
concatenate_rounds	[1], [2]	[1, 2]	
concatenate_rounds	[27, 28, 29], [35, 36]	[27, 28, 29, 35, 36]	
concatenate_rounds	[1, 2, 3], [4, 5, 6]	[1, 2, 3, 4, 5, 6]	
list_contains_round	[ ], 1	False	
list_contains_round	[1, 2, 3], 0	False	
list_contains_round	[27, 28, 29, 35, 36], 30	False	
list_contains_round	[1], 1	True	
list_contains_round	[1, 2, 3], 1	True	
list_contains_round	[27, 28, 29, 35, 36], 29	True	
card_average	[1]	1.0	
card_average	[5, 6, 7]	6.0	
card_average	[1, 2, 3, 4]	2.5	

card_average	[1, 10, 100]	37.0	
approx_average_is_average	[0, 1, 5]	False	
approx_average_is_average	[3, 6, 9, 12, 150]	False	
approx_average_is_average	[1, 2, 3, 5, 9]	False	
approx_average_is_average	[2, 3, 4, 7, 8]	False	
approx_average_is_average	[1, 2, 3]	True	
approx_average_is_average	[2, 3, 4]	True	
approx_average_is_average	[2, 3, 4, 8, 8]	True	
approx_average_is_average	[1, 2, 4, 5, 8]	True	
average_even_is_average_odd	[5, 6, 8]	False	
average_even_is_average_odd	[1, 2, 3, 4]	False	
average_even_is_average_odd	[1, 2, 3]	True	
average_even_is_average_odd	[5, 6, 7]	True	
maybe_double_last	[1, 2, 11]	[1, 2, 22]	
maybe_double_last	[5, 9, 11]	[5, 9, 22]	
maybe_double_last	[5, 9, 10]	[5, 9 , 10]	
maybe_double_last	[1, 2, 3]	[1, 2, 3]	

****

CS101. A19 - Basic Robot Simulator

Write a robot simulator. A robot factory's test facility needs a program to verify robot movements. The robots have 4 possible movements (by one step):

- move right
- move left
- move up
- move down

Robots are placed on a hypothetical infinite grid in the standard quadrant system. A robot's location is identified with a set of {x,y} coordinates, e.g., position {3,8}, is in the (+,+) quadrant.

The robot then receives a number of instructions, at which point the testing facility verifies the robot's new position, and in which direction it is pointing.

- The letter-string "RRDDDL" means:
  - Move right 2 steps
  - Move down 3 steps
  - Move left one step
- Say a robot starts at {3, 8}. Then running this stream of instructions should leave it at {4, 5}.

Tasks to complete

Implement a `move` function. Input parameter is a tuple with 2 elements. Element 0 is a another tuple with 2 elements (x_pos, y_pos). Element 1 is the route string.

Return value: a Tuple. The new position of the Robot.

Code Template (robot_move.py)

```
def move(input_list):
  pass
```

Sample output

```
move(((3,8), "RRDDDL"))
(4,5)
```

Test Cases

Function	Inputs	Return Value	
move	((0,0), “R”)	(1, 0)	
move	((0,0), “L”)	(-1, 0)	
move	((0,0), “U”)	(0, 1)	
move	((0,0), “D”)	(0, -1)	
move	((0,0), “RL”)	(0, 0)	
move	((0,0), “LR”)	(0, 0)	
move	((0,0), “UD”)	(0, 0)	

move	((0,0), “DU”)	(0, 0)	
move	((0,0), “RDLU”)	(0, 0)	
move	((0,0), “LURD”)	(0, 0)	
move	((0,0), “URDL”)	(0, 0)	
move	((0,0), “DRUL”)	(0, 0)	
move	((3, 8), “RRDDDL”)	(4, 5)	
move	((3, 8), “RURULDLD”)	(3, 8)	
move	((-3, 8), "RRDDDL")	(-2, 5)	
move	((3,-8), "RURULDLD")	(3, -8)	

****