Object Oriented Programming, Exercise 2
(1 point from each task, total 10 points)

Topics: Dictionary, classes and objects, constructors, methods, objects as elements in other data structures (e.g., lists)

Make a Git commit at least after every coding task.

Save all the tasks in separate .py files

Code in Python3 and follow the style guide.

Most of the exercises (from 4 onward) are from MOOC.fi part 8: https://programming-24.mooc.fi/part-8 (1, 2, 3) and Part 9 (1)

1. **Queueing**

   Review the definitions of the list (e.g.,

   https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-queues).

   Apply the features it offers in this task.

   ```
   # Let us define the list with names:
   queue = ["Matti", "Riikka", "Antti", "Jenni", "Anu", "Ville", "Jarno"]
   ```

   The list models the painfully slow checkout line during sales, where Matti is being served next.

   Implement (i.e. code) the following events and determine the answers to the given questions solely using list operations without directly relying on the order given above.

   Time t+ 1: The first person in the queue leaves after paying for their purchases.

   Time t+ 2: Ville recruits Anni to queue on his behalf.

   Time t+ 3: Jarno leaves, tired of the constant waiting.

   Time t+ 4: Marjo joins the end of the queue.

   Time t+ 5: As a gentleman, Antti lets the two people behind him go ahead of him.

   Is Jenni in the queue? If so, at what position? And who is the third last person in the queue?

2. **Train Carriages**

   The IT manager of the private railway company Steam & Locomotive requests your help: The company is finally transitioning to an electronic reservation management system, and they need a top-notch Python programmer. You agree to help, and your responsibility is to model the reservation status of a train carriage. The IT manager emails you the following task description:

- The carriage has a unique identifier and information about the total number of seats.
- The carriage has individually marked seats. The key point is how many reservations have been made in total and which seats are reserved.
- It must be possible to add and remove reservations one at a time. It should also be possible to reset the reservation status.
- The same seat must never be reserved twice.
- It must be possible to get a report of the reserved and unreserved seats in the form of a string that clearly presents the situation.
- It must be possible to inquire whether the carriage is already full or not.

Implement a class that models the reservation status of a train carriage according to the above specifications. Make any necessary additional assumptions. Write definitions for all the features of the class. Choose sensible names for the features.

3. Tossing a coin with unexpected results

Take a look at the coin.py below, write it down in your IDE and run it. See that coin gets tossed.

coin.py

```
# File:         coin.py
# Source:       Tony Gaddis: Starting out with Python, fourth edition
# Modified by:  Sanna Maatta & Anne Jumppanen
# Description:  Coin object and tossing

import random

# Class definition

class Coin:
    def __init__(self):
        self.sideup = 'Heads'


    def toss_the_coin(self):

        if random.randint(0,1) == 0:
            self.sideup = 'Heads'
        else:
            self.sideup = 'Tails'

    def get_sideup(self):
        return self.sideup

# Main function definition

def main():

    my_coin = Coin()

    print("This side is up:", my_coin.get_sideup())

    print("Tossing the coin...")
    my_coin.toss_the_coin()

    print("Now this side is up:", my_coin.get_sideup())


# Calling the main function
main()

```

Modify the toss_the_coin() function so that there are 3 more options:

1. Coin lands on the table upright (and not flat showing heads or tails)
2. Coin drops on the ground and disappears in a rabbit hole
3. Coin defies gravity and gets lost on a wormhole in space.

Give the different options different likelyhoods, so that the heads and tails are the most likely ones, and the rest are less likely so that the option three is the least likeliest to happen. Name the options properly and give informative and readable output of the status.

4. [Dictionary, no need for objects] Factorials

Please write a function named factorials(n: int), which returns the factorials of the numbers 1 to n in a dictionary. The number is the key, and the factorial of that number is the value mapped to it.

A reminder: the factorial of the number n is written n! and is calculated by multiplying the number by each integer smaller than itself. For example, the factorial of 4 is 4 * 3 * 2 * 1 = 24.

An example of the function in action:

```
k = factorials(5)
print(k[1])
print(k[3])
print(k[5])
```

```
1
6
120
```

5. [Dictionary, no need for objects] Smallest average

Please write a function named `smallest_average(person1: dict, person2: dict, person3: dict)`, which takes three dictionary objects as its arguments.

Each dictionary object contains values mapped to the following keys:

"name": The name of the contestant

"result1": the first result of the contestant (an integer between 1 and 10)

"result2": the second result of the contestant (as above)

"result3": the third result of the contestant (as above)

The function should calculate the average of the three results for each contestant, and then return the contestant whose average result was the smallest. The return value should be the entire dictionary object containing the contestant's information.

You may assume there will be no ties, i.e. a single contestant will have the smallest average result.

An example of the function in action:

```
person1 = {"name": "Mary", "result1": 2, "result2": 3, "result3": 3}
person2 = {"name": "Gary", "result1": 5, "result2": 1, "result3": 8}
person3 = {"name": "Larry", "result1": 3, "result2": 1, "result3": 1}

print(smallest_average(person1, person2, person3))
```

Sample output

```
{'name': 'Larry', 'result1': 3, 'result2': 1, 'result3': 1}
```

(https://programming-24.mooc.fi/part-8/1-objects-and-methods)

6. **[objects] My Favorite Book/Manga/Anime/Movie/Song**

Please write a class named Book/Manga/Anime/Movie/Song with the attributes name, author/director/writer, genre and year, along with a constructor, which assigns initial values to these attributes.

Your class should work something like this:

```
python = Book("Fluent Python", "Luciano Ramalho", "programming", 2015)
everest = Book("High Adventure", "Edmund Hillary", "autobiography", 1956)

print(f"{python.author}: {python.name} ({python.year})")
print(f"The genre of the book {everest.name} is {everest.genre}")
```

```
Luciano Ramalho: Fluent Python (2015)
The genre of the book High Adventure is autobiography
```

7. [object] Pet

Please define the class Pet. The class should include a constructor, which takes the initial values of the attributes name, species and year_of_birth as its arguments, in that specific order.

Please also write a function named new_pet(name: str, species: str, year_of_birth: int) outside the class definition. The function should create and return a new object of type Pet, as defined in the class Pet.

An example of how the function is used:

```
fluffy = new_pet("Fluffy", "dog", 2017)
print(fluffy.name)
print(fluffy.species)
print(fluffy.year_of_birth)
```

```
Fluffy
dog
2017
```

8. [object] movies of a genre (can also be books, tv series, anime, manga…)

Please write a function named movies_of_genre(movies: list, genre: str) which takes a list of objects of type Movie and a string representing a genre as its arguments.

The function should return a new list, which contains the books with the desired genre from the original list. Please see the examples below.

```
john_woo = Movie("A Better Tomorrow", "John Woo", "action", 1986)
kungfu = Movie("Chinese Odyssey", "Stephen Chow", "comedy", 1993)
jet_li = Movie("The Legend", "Corey Yuen", "comedy", 1993)

movies = [john_woo, kungfu, jet_li, Movie("Hero", "Yimou Zhang", "action",
2002)]

print("Movies in the action genre:")
for movie in movies_of_genre(books, "action"):
    print(f"{movie.director}: {movie.name}")
```

In terminal the program prints the following:

```
Movies in the action genre:
John Woo: A Better Tomorrow
Yimou Zhang: Hero
```

9. Passing submissions

The exercise template contains a class named ExamSubmission which, as the name implies, models an examinee's submission in an exam. The class has two attributes defined: examinee (str) and points (int).

Please write a function named passed(submissions: list, lowest_passing: int) which takes a list of exam submissions and an integer number representing the lowest passing grade as its arguments.

The function should create and return a new list, which contains only the passed submissions from the original list. Please do not change the list given as an argument or make any changes to the ExamSubmission class definition.

You can find the template from Itslearning passing_submissions.py

10. [object] Comparing properties

Code all the parts in the same file, no need to differentiate the parts in the code.

The database of a real estate agency keeps records of available properties with objects defined by the following class:

```
class RealProperty:
    def __init__(self, rooms: int, square_metres: int, price_per_sqm:
int):
        self.rooms = rooms
        self.square_metres = square_metres
        self.price_per_sqm = price_per_sqm
```

Your task is to implement methods which allow for comparison between available properties.

Part 1:

Please write a method named bigger(self, compared_to) which returns True if the RealProperty object itself is bigger than the one it is compared to.

An example of how the function should work:

```
central_studio = RealProperty(1, 16, 5500)
downtown_two_bedroom = RealProperty(2, 38, 4200)
suburbs_three_bedroom = RealProperty(3, 78, 2500)

print(central_studio.bigger(downtown_two_bedroom))
print(suburbs_three_bedroom.bigger(downtown_two_bedroom))
```

The program would then print the following:

```
False

True
```

Part 2:

Please write a method named price_difference(self, compared_to) which returns the difference in price between the RealProperty object itself and the one it is compared to. The price difference is the

absolute value of the difference between the total prices of the two properties. The total price of a property is its price per square metre multiplied by the amount of square metres in the property.

An example of how the function should work:

```
central_studio = RealProperty(1, 16, 5500)
downtown_two_bedroom = RealProperty(2, 38, 4200)
suburbs_three_bedroom = RealProperty(3, 78, 2500)

print(central_studio.price_difference(downtown_two_bedroom))
print(suburbs_three_bedroom.price_difference(downtown_two_bedroom))
```

The program would then print the following:

```
71600

35400
```

Part 3:

Please write a method named more_expensive(self, compared_to) which returns True if the RealProperty object itself is more expensive that the one it is compared to.

An example of how the function should work:

```
central_studio = RealProperty(1, 16, 5500)
downtown_two_bedroom = RealProperty(2, 38, 4200)
suburbs_three_bedroom = RealProperty(3, 78, 2500)

print(central_studio.more_expensive(downtown_two_bedroom))
print(suburbs_three_bedroom.more_expensive(downtown_two_bedroom))
```

The program would then print the following:

```
False

True
```

Return and demonstrate

1. Return your report to Itslearning by **deadline 6 Feb 2025**.

2. Demonstrate your code and pseudocode to teacher on-site 7 Feb 2025.