Python for Engineers 0509-1820, Fall 2020-2021

Home Assignment 8

OOP

General instructions

- Read the questions **carefully** and make sure your programs work according to the requirements.
- The homework needs to be done individually!
- Read the submission rules on the course web page. All the questions need to be submitted together in the file `ex8_012345678.py` attached to the homework, after changing the number 012345678 with your ID number (Teudat Zehut if you have one, otherwise it's typically a number beginning with 9).
- How to write the solution: in this homework, you need to complete the code in the attached outline file.
- Do not change the names of the classes, functions, methods and variables that already appear in the attached outline file.
- Do not erase the instructions that appear in the outline file.
- Some questions are checked automatically. You thus need to ensure that the output exactly matches the requirements (even for spaces).
- Check your code: to ensure correctness of your programs and their robustness in the presence of faulty input, for each question run your program with a variety of different inputs, those that are given as examples in the question and additional ones of your choice (check that the output is correct and that the program does not crash).
- Unless stated otherwise, you may assume that the input given to methods is correct.
- Submission is due by: see course web page.

In this assignment, you will write a program for managing rooms and guests of a hotel. The program consists of classes that represent mini-bars, hotel rooms, and the hotel.

Important additional instructions:

- **It is advised to first read the whole assignment, to understand the requirements, and to plan the different classes and the relationships between them accordingly.**
- Try to avoid code duplication as much as possible in your implementation!
- In the entire homework, string comparison is case insensitive, e.g., you need to treat the string "ABc" and "abC" as equal, since when converted to lowercase, both are "abc".
- In **all** questions, you may add attributes and methods (other than the ones specified by the question) if this helps you to implement the solution.
- Complete the code of the classes in the outline file in accordance with the requirements expressed in the following questions.

Question 1

Implement the class **`Minibar`**, with the following attributes:

Name	*Description*	*Type*	*Comments*
`drinks`	Drinks available in the mini-bar	Dictionary whose keys are drink names and values are the respective prices	An empty dictionary is valid
`snacks`	Snacks available in the mini-bar	Dictionary whose keys are snack names and values are the respective prices	An empty dictionary is valid
`bill`	Current bill	A real number (`float`)	Should be initialized to zero

A) Start by implementing a constructor:

```
__init__(self, drinks, snacks)
```

Notes:

- You may assume inputs are correct; no need to check types or validate the input.
- Do <u>not</u> modify the strings given by the user (e.g., do not convert them to lowercase)
- Prices can be ints or floats.

B) Next implement the methods `eat_a_snack` and `drink_a_drink`.

`eat_a_snack(self, snack)` gets a snack name (a string), removes it from the available snacks in `self` and adds its price to the current bill. The method returns no value.

Similarly, `drink_a_drink(self, drink)` gets a drink name (a string), removes it from the available drink in `self` and adds its price to the current bill. The method returns no value.

If the requested item name is not available (in the respective dictionary), the method should raise a ValueError with the error message `'The snack is not in the minibar'` for a snack or `'The drink is not in the minibar'` for a drink.

C) Then implement the method __repr__ that returns a string. Infer the format from the following example.

Notes:

- The output string should have three lines, separated by '\n'.
- No '\n' at the end.
- Do not forget the space between the colon (:) and the list of items.
- Snacks and drink names should appear in a list (see above).

Example:

```
>>> drinks1 = {'coke': 12, 'rum': 25}
>>> snacks1 = {'m&m': 10, 'cake': 30}
>>> m = Minibar(drinks1, snacks1)
>>> print(m)
The minibar contains the drinks: ['coke', 'rum']
And the snacks: ['m&m', 'cake']
The bill for the minibar is: 0.0
>>> m.eat_a_snack('m&m')
>>> print(m)
The minibar contains the drinks: ['coke', 'rum']
And the snacks: ['cake']
The bill for the minibar is: 10.0
>>> m.drink_a_drink('coke')
>>> print(m)
The minibar contains the drinks: ['rum']
And the snacks: ['cake']
The bill for the minibar is: 22.0
>>> m.drink_a_drink('beer')
Traceback (most recent call last):
…
ValueError: The drink is not in the minibar
```

Question 2

Implement the class **`Room`**, which represents a room in a hotel. Each room has the following attributes:

Name	*Description*	*Type*	*Comments*
`minibar`	Mini-bar associated with the room	`Minibar` (from Q1)	Has snacks and drinks
`floor`	Floor number of the room	Non-negative integer (`int`)	
`number`	Room number	Positive integer (`int`)	
`guests`	Names of the current guests of the room	List of strings. Each name consists only of letters (uppercase and lowercase) and spaces.	An empty list is valid (vacant room)
`clean_level`	Level of cleanliness of the room from 1 (dirty) to 10 (clean)	Integer between 1 and 10.	
`rank`	Luxury level of the room. There are 3 levels: 1 (Basic), 2 (Standard), 3 (Luxury)	Integer between 1 and 3	
`satisfaction`	Level of satisfaction of the room guests, between 1.0 (lowest satisfaction) and 5.0 (highest satisfaction)	Real number (`float`) between 1.0 and 5.0	Defaults to 1.0; Can be non-integral (e.g., 4.5); The attribute stored in the object should be a `float` even if a user provided a satisfaction level of type `int`

A) Start by implementing a constructor:

```
__init__(self, floor, number, guests, clean_level,
         rank, satisfaction=1.0)
```

You may assume that the types and the values of the parameters are correct, according to the information provided in the table above, except for the cases described hereunder, for which you need to check the validity of the parameter:

- You need to start by checking the types. In case one of the types is not valid, you need to raise a `TypeError`. Consider the following cases:
  - The cleanliness level is not of type `int`;
  - The rank of the room is not of type `int`;

    - The satisfaction level type is neither of `int` nor `float`. Note: it is perfectly fine for the user to construct a Room and pass an `int` for the satisfaction level (e.g., 4, which represents a satisfaction level of 4.0).
- After checking types, you need to check the values. If one of the values is invalid (but the type is valid), raise a `ValueError`. You need to handle to following cases:
    - The level of cleanliness is not in the range 1…10 (including 1 and 10);
    - The rank of the room is not in the range 1…3 (including 1 and 3);
    - The satisfaction level is not in the range 1…5 (including 1 and 5).

Notes:

- You may choose the error messages for each `TypeError` and `ValueError`.
- If multiple parameters are invalid, you may choose which one raises the exception.
- When creating a new `Room`, store its guest names in lowercase. This would make the implementation of the next parts easier.

B) Then, implement the method __repr__ that returns a string. Infer the format from the following example.

- The string will contain a separate line for each attribute of the room, in the format "name of the attribute:<one space>value of the attribute", except the mini-bar, for which we use the representation from Q1.
- The attributes will appear in the same order as in the table above.
- For the attribute `guests`, the value of the attribute will be the list of guest names in lowercase (in any order), with the names separated by <<comma><one space>>. If the list of guests is empty, the value of the attribute will be the word `empty`.
- The attribute `satisfaction` (of type `float`) will be represented with a precision of one decimal digit (use the function `round`, see example below).
- Do not add a newline (`'\n'`) after the last attribute.

Example

```
>>> m = Minibar({'coke': 10, 'lemonade': 7}, {'bamba': 8, 'mars': 12})
>>> print(Room(m, 12, 101, ["Ronen", "Shir"], 6, 2))
The minibar contains the drinks: ['coke', 'lemonade']
And the snacks: ['bamba', 'mars']
The bill for the minibar is: 0.0
floor: 12
number: 101
guests: ronen, shir
clean_level: 6
rank: 2
satisfaction: 1.0
```

C) Add the following methods to the class Room:

Method	*Description*
`is_occupied(self)`	Returns `True` if the room is occupied, that is, if there are guests in the room, and otherwise returns `False`.
`clean(self)`	This method implements a cleaning of the room. More luxurious rooms get a better cleaning, so the cleanliness level is changed to $\min(10, clean\_level + rank)$, where `rank` is the rank of the room and `clean_level` is the current cleanliness level.
`better_than(self, other)`	This method compares the level of two rooms and returns `True` if `self` is a "better" room than `other` (and `False` otherwise). • The room `self` is considered "better" than the room `other` if $(self.rank, self.floor, self.clean\_level) > (other.rank, other.floor, other.clean\_level)$, where the order of the comparison operator ">" is according to the standard Python **ordering of tuples**. • If `other` is not of type `Room`, you need to raise a `TypeError` with the message `"Other must be an instance of Room"`.
`check_in(self, guests)`	This method performs the check-in of guests into the room. • The method checks-in `guests` (a list of guests) into the room `self` if the room is empty, and initializes the satisfaction level to 1.0. • If the room is occupied, one cannot perform the check-in, and the method will raise an exception of type `RoomError` with the message: `"Cannot check-in new guests to an occupied room"`. • You may assume that `guests` is a *non-empty* list of strings with valid names (i.e., only English uppercase/lowercase letters and spaces). • Recall that the attribute `self.guests` should be a list of lowercase strings.
`check_out(self)`	This method performs the check-out procedure, i.e., removes guests currently occupying the room. • If the list of guests of the room `self.guests` <u>is not</u> empty, then the method empties that list. • Otherwise (`self.guests` is already empty), raise an exception of type `RoomError` with the message `"Cannot check-out an empty room"`.

`move_to(self, other)`	This method moves the guests of room `self` to room `other` (if the latter is empty). • If `self` is empty, there are no guests to move, and the method raises an exception of type `RoomError` with the message "`Cannot move guests from an empty room`". • If room `other` is occupied, one cannot move the guests, and the method raises an exception of type `RoomError` with the message "`Cannot move guests into an occupied room`". • If `self` is empty and `other` is occupied, then you should raise the exception for `self`. The moving procedure contains the following steps: 1. Moving the list of guests from `self` to the relevant list in `other`. 2. If `other` is a "better" room than `self` (as defined above in the method `better_than`), then the level of satisfaction of the guests gets raised, according to the following formula: $other.satisfaction = \min(5.0, self.satisfaction + 1.0)$ Otherwise, the level of satisfaction in `other` is set to that of `self` when performing the move. 3. Finally, you need to remove the guests from the guest list of `self`, so `self` becomes empty. • You may assume that `other` is a valid `Room`. • You may assume that `self` and `other` represent different rooms.

<u>Note:</u> Python supports raising user-defined exceptions, so each program can define its own exceptions. An example of this is `RoomError`**, which is defined in the outline file**. It can be raised with the keyword `raise`, just like any standard exception.

Example (make sure you understand all the results):

```
>>> r1 = Room(2, 23, ["Dana", "Ron"], 5, 2)
>>> r_better = Room(6, 57, [], 4, 3)
>>> r_better.better_than(r1)
True
>>> r_better.check_in(["Amir"])
>>> r_better.clean()
>>> r_better.clean_level
7
>>> r1.check_in(["Avi", "Hadar"])
Traceback (most recent call last):
…
RoomError: Cannot check-in new guests to an occupied room
>>> r1.is_occupied()
True
>>> r1.check_out() ## note: None is returned, and so nothing is printed
>>> r1.is_occupied()
False
>>> r_better.move_to(r1)
>>> r1.satisfaction
1.0
>>> r1.guests
['amir']
>>> r1.move_to(r_better)
>>> r1.is_occupied()
False
>>> r_better.satisfaction
2.0
>>> r_better.guests
['amir']
```

Question 3

Finally, we implement the class `Hotel` which represents the hotel.

A) Implement the constructor `__init__(self, name, rooms)` which gets the name of the hotel (a string) and a list `rooms` of `Room` instances. You do not need to check the validity of the rooms.

Notes:

- You may assume that `name` is a valid string that represents the name of the hotel, and contains only spaces, digits and English lowercase/uppercase letters`
- You may assume that the list `rooms` is not empty and that each element of the list is a valid room (of type `Room`), and that each element represents a different room (i.e., no room object appears twice in the list, and two different rooms have a different room number or a different floor number). You may assume that the names of the guests are different from each other, both within the same room and in different rooms.
- The list `rooms` may contain occupied rooms and/or empty rooms.
- You may keep the rooms as an attribute of the hotel object, using any Python data structure that you like. In other words, the internal representation of a Hotel object doesn't have to use a list to hold the rooms.
- You may add additional attributes and methods that could help you in implementing the class.

B) Implement the method `__repr__(self)` which returns a string that represents the hotel according to the following format:

```
"<self.name><one space>hotel has:\n
<number of Room objects><one space>rooms\n
<number of occupied rooms><one space>occupied rooms"
```

Example

```
>>> m = Minibar(…)
>>> h = Hotel("Best",[Room(m, 15, 140, [], 5), Room(m, 1, 2, ["Liat"],
7)])
>>> h
Best hotel has:
2 rooms
1 occupied rooms
```

Note: when writing the code of `__repr__`, counting rooms can be done any way you decide. In particular, you may to use helper methods.

C) The `Hotel` class should implement the following methods:

Method	*Description*
`check_in(self, guests, rank)`	The method tries to check-in `guests` (a list of names) to any *one* room of rank `rank` (an integer). • If an <u>empty room with the required rank</u> is found, the method checks-in `guests` to this room, and returns the room object. If no suitable room was found, the method returns None. • You may assume that the names in the list `guests` do <u>not</u> conflict with the names of the guests that are already currently staying at the hotel. • You can assume that the list `guests` is a *non-empty* list of strings with valid names (that is, the strings contain only English letters and spaces) and letters that can be in uppercase or lowercase.
`check_out(self, guest)`	Tries to check-out the guest named `guest` (string) and all the other guests staying with her in the same room (if any). • If the room where the guest is staying is found, the check-out procedure needs to be performed successfully, and the method should return the room where the guest was staying. • Otherwise, it is impossible to perform the check-out, and the method should return None. • Recall that guest names are case insensitive.
`upgrade(self, guest)`	Tries to perform an "upgrade" for the guest named `guest` (string), if `guest` indeed is currently staying at a room in the hotel and if there is an available room to upgrade her to. • The upgrade operation includes moving the guest, with the other guests that are staying with her in the room (if any), to another room in the hotel which is vacant and "better" (as defined in question 2) than his current room. • If the upgrade succeeded, the method should return the new room assigned to the guest. • Otherwise, if the guest is not staying at the hotel or if the upgrade operation could not be done, the method should return None. • Recall that guest names are case insensitive. • If several upgraded rooms are available, you may choose any of them.

Notes:

- All methods should always end execution without raising any exception of type `RoomError`.
- You may assume the validity (type and value) of the parameters of each method. In particular, you may assume that each string in the input represents a valid name of a guest in lowercase and/or uppercase.

Example

- See the function `test_hotel` in the outline file.
- The file `test_hotel_output.txt` contains the output generated by running the test function above, so you could check your program prints the same values.
- Note that some operations have more than one valid output (for example, there are several possibilities for upgrading the room of Liat). In such a case, outputs different from those of the example will be accepted (depending on the implementation).

Good luck!