Python for Engineers 0509-1820, Semester 1 2019-2020

Homework 8

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 ex1_012345678.py attached to the homework, after changing the number 012345678 with your ID number (9 digits, including check digit).
- How to write the solution: in this homework, you need to complete the code in the attached outline file.
- **You are not allowed to change the names of the classes, functions, methods and variables that already appear in the attached outline file.**
- **You are not allowed to erase the instructions that appear in the outline file.**
- Some questions are checked automatically. **You thus need to ensure that the output is exactly fits the requirements (even for spaces).**
- Check your code: in order 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 can suppose that the input received by the functions is correct.**
- Final submission date: see course web page.

In this question, you will write a program for managing rooms and guests of a hotel. The program will contain classes that represent the different types of hotel rooms, and also a class that represents the hotel.

Important additional instructions:

- It is advised to first read the **whole** assignment, to understand the homework's requirements, and to plan the different classes and the relationships between them accordingly.
- In the implementation of the classes, you need to use **inheritance** in order to avoid code duplication, as much as possible.
- In all this homework, comparison between strings should be done using lowercase only. For example, you need to treat the string "ABc" and "abC" as the same string, because in lowercase both strings are "abc".
- In **all** questions, you are allowed to add more attributes and methods than those asked in the question, if this can help 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

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

Name	Description	Type	Comments
floor	Floor number of the room	Integer (*int*) greater or equal to 0.	
number	Room number	Integer (*int*) strictly greater than 0.	
guests	List of names of the current guests of the room	List of strings. The strings contain only characters (uppercase and lowercase) and spaces.	An empty list of strings is possible (in case the room is empty)
clean_level	Level of cleanness 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	-The default value of the satisfaction level is 1.0 -A satisfaction level can be fractional, for example 4.5 -If a user enters a satisfaction level of type *int*, convert it to *float* when you save it as an attribute in the object

Start by implementing the constructor, a cording to the following header:

__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 an exception of type *TypeError*. You need to take into account the following cases:
    - The cleanness level is not of type *int*
    - The rank of the room is not of type *int*
    - The satisfaction level is not an integer of type *int* or is not a real number of type *float*. Note: the user can build a room with an integer satisfaction level. More precisely, it is possible that the satisfaction level will be given as an *int*. For example, with the value *4* , which represents a satisfaction level of 4.0.

- After checking the types, you need to check the values. If one of the values is not valid (although the type is valid), you need to raise an error of type *ValueError*. You need to handle to following cases:
    - The level of cleanness is not between 1 and 10 (including the range bounds, i.e., 1, and 10)
    - The rank of the room is not between 1 and 3 (including the range bounds)
    - The satisfaction level is not between 1 and 5 ((including the range bounds)

Notes:

- You may choose yourself the error message of each *TypeError* and *ValueError*.
- If there are several parameters that are not valid, you may choose any of these parameters for raising the exception.
- When creating a new object, all strings in the list of guest names received as input need to be converted to lowercase. This will ease the implementation of the following sections of this question.

B) Implement the method *__repr__(self)*, which returns a string that describes objects of type *Room*, according to the example hereunder.

- 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".
- The order of appearance of the attributes will conform to their order of appearance 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 *empty* (see the second example below).
- Because the attribute *satisfaction* is of type *float*, it will be printed to the screen with a precision of one digit after the comma (use the function *round*, see example below).
- There is no need to add a \n after the last attribute.

Examples of execution

```
>>> guests = ["Roni", "Danny"]
>>> r1 = Room(3, 21, guests, 5, 1)
>>> r1
floor: 3
number: 21
guests: roni, danny
clean_level: 5
rank: 1
satisfaction: 1.0
>>> r2 = Room(4, 28, [], 5, 1)
>>> r2
floor: 4
number: 28
guests: empty
clean_level: 5
rank: 1
```

```
satisfaction: 1.0
```

C) Add the following methods to the class *Room*:

Header of the 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*.
can_clean(self)	Returns a boolean value indicating whether one can clean the room. • The default behavior of this method will be to always return *True*. • Even if it can look strange that the method always returns *True*, note that its implementation of the derived classes (see following questions) will behave differently. The meaning is that objects of type *Room* always need to be cleaned (but it is not necessarily so for a class derived from *Room*).
clean(self)	This method implements the cleaning of the room… • If the room may be cleaned, the method cleans the room by raising its cleanness level using the formula $\min(10, clean\_level + rank)$, where *rank* is the rank of the room. • If the room cannot be cleaned, you need to raise an exception of type *RoomError* (see note below) with the message: “Room cannot be cleaned”.
better_than(self, other)	This method compares the level of two rooms and returns *True* if *self* is a “better” room than *other*, and returns *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* and is not of a type derived from *Room*, you need to raise an error of type *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 will check in the list of guests *guests* into the room *self* if the room is empty, and will initialize the satisfaction level of the guests 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 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. • Remark: in case of success, the method saves the guests to self.guests in lower case
check_out(self)	This function performs the check-out procedure, that is it releases the guests currently occupying the room. • If the list of guests of the room *self.guests* <u>is not</u> empty, then the check out procedure includes turning this list to an empty list. • Otherwise, if *self.guests* is an empty list, you need to 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 room self is empty, there are no guests to move, and the method will raise 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 will raise 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*. <u>The moving procedure contains the following steps:</u> 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 erase the elements in the lists of guests of *self*, so that the list will be empty. • You may assume that *other* is a valid object of type *Room* or of a type derived from *Room*. • You may assume that *self* and *other* represent different rooms.

Note: in Python, we can raise our own exceptions that are defined according to the needs of a specific program. An example of this is the Exception of type **RoomError, which is defined in the outline file**, and this type of exception can be raised with the keyword *raise*, exactly as we did up to now for any standard exception.

Example of execution (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 2

Let us now implement the classes ***BudgetRoom*** and ***LegacyRoom***, assuming we already have the class *Room* at our disposal. These classes will represent types of rooms that contain all the attributes and operations of a room of type *Room*, but with changes and additions relevant for each type of room, as described below.

A) Constructors (see example in section B below)

A.1) Implement the constructor of the class BudgetRoom (which represents a budget (i.e. "cheap" room) according to the following header:

__init__(self, floor, number, guests, clean_level, rank=1, satisfaction=1.0, clean_stock=0)

- Note that *rank* receives a default value of 1.
- A budget room has the additional attribute **clean_stock:**

- This attribute represents the total number of cleaning operations to be done in the room during the guest stay in the room
- Its type is an integer (*int*) greater or equal to 0
- It receives a default value of 0
- <u>You can assume</u> that the type and value of the attribute are valid, so you don't need to check their validity.

<u>A.2) Implement the constructor of the class *LegacyRoom* (which represents a room with "full service") according to the following header:</u>

__init__(self, floor, number, guests, clean_level, rank=2, satisfaction=1.0, minibar_drinks=2, minibar_snacks=2)

- Note that *rank* gets a default value of 2.
- The class has additional attributes *minibar_drinks* and *minibar_snacks*:
  - They represent the number of drinks and snacks that are available at the minibar when the guests enter the room.
  - Both are integers (*int*) greater or equal to 0.
  - Both get a default value of 2.
  - <u>You can assume</u> that the type and value of the attribute are valid, so you don't need to check their validity.

<u>Note:</u>

- As part of the implementation of the two constructors described above, you need to call the constructor or class *Room* with the right parameters.

B) <u>The *repr* method:</u> this method returns a string that describes the object in a way that resembles that described above for class *Room*, but with additional descriptions for the new attributes that are specific to the new types of rooms. The representation should conform to the examples below:

```
>>> br1 = BudgetRoom(1, 12, ["Loren", "Or"], 5)
>>> br1
floor: 1
number: 12
guests: loren, or
clean_level: 5
rank: 1
satisfaction: 1.0
type: BudgetRoom
clean_stock: 0
>>> lr1 = LegacyRoom(5, 94, ["Ronen", "Dror", "Liat", "Smadar"], 5)
>>> lr1
floor: 5
number: 94
guests: ronen, dror, liat, smadar
clean_level: 5
rank: 2
satisfaction: 1.0
type: LegacyRoom
```

```
minibar_drinks: 2
minibar_snacks: 2
```

Instructions:

- Before describing the attributes specific to one of the new classes, the class should print the line "type<one space>name of the class".
- The descriptions of the new fields need to conform to the order in which they appear in section A above.
- The implementation of this method should use the *__repr__* method of class *Room*.

C) Operations: implement the methods supported by the classes *BudgetRoom* and *LegacyRoom*, respecting the following rules:

- *Efficient code-writing*: You should not write twice the same implementation of a method.
- You may need to implement the same method more than once in order to adapt it to each class, according to the given requirements. If so, you need to make sure you are not duplicating code. More precisely, you need to check if you could call a method that you already implemented in the base class, when you are implementing it in the derived classes.

C.1) The class *BudgetRoom* should implement the following methods:

Header of the method	**Description**
is_occupied(self)	Same description as for class *Room*
can_clean(self)	Returns a boolean value indicating whether the room can be cleaned • A budget room can be cleaned if the number of cleaning operations to be done in the room (*clean_stock*) is strictly greater than 0.
clean(self)	This method should work as defined in class *Room*, with the addition that, if the room can be cleaned, the method diminishes by 1 the number of cleaning operations that can be done for the room. • In the implementation of the method, you need to call the corresponding method of the base class (*Room*).
better_than(self, other)	Same description as for class *Room*
check_in(self, guests)	This method should work as defined in class *Room*, with the addition that, if the room *self* is empty, it initializes *self.clean_stock* to 0. • In the implementation of the method, you need to call the corresponding method of the base class (*Room*).

	• You may assume that *guests* is a non-empty list of strings • 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)	Same description as for class *Room*
move_to(self, other)	This method should work as defined in class *Room*, with the addition that, if the move can be done (that is, if the room *self* is not empty and the room *other* is empty), and if the object *other* is an object of class *BudgetRoom*, the method will change the value of *other.clean_stock* to make it equal to the value of *self.clean_stock*. • In the implementation of the method, you need to call the corresponding method of the base class (*Room*).
grant_clean(self)	This method grants additional cleaning operations as a gift to the guests by increasing *clean_stock* by 1. • As a result, the degree of satisfaction of the guests (*satisfaction*) is increased to: $\min(5.0, satisfaction + 0.5)$ • If the room is empty, the gift is not granted, and an exception of type *RoomError* should be raised, with the message "Cannot grant an empty room".
grant_snack(self)	This method grants a free snack to the room's guests. • As a result, the degree of satisfaction of the guests (*satisfaction*) is increased to: $\min(5.0, satisfaction + 0.8)$ • But the snack makes the room more dirty, and hence lowers the cleanness level *clean_level* to: $\min(5.0, satisfaction + 0.8)$ • If the room is empty, the gift is not granted and an exception of type *RoomError* should be raised, with the message "Cannot grant an empty room".

Example of execution (make sure you understand all the results):

```
>>> br1 = BudgetRoom(1, 12, ["loren", "or"], 5)
>>> br1.clean_stock
0
>>> br1.satisfaction
      1.0

>>> br1.clean()
```

```
Traceback (most recent call last):
  …
RoomError: Room cannot be cleaned
>>> br1.grant_clean()
>>> br1.clean_stock
1
>>> br1.satisfaction
1.5
>>> br1.clean()
>>> br1.clean_stock
0
>>> br1.clean_level
6
>>> br1.grant_snack()
>>> br1.clean_level
5
>>> br1.satisfaction
2.3
>>> br2 = BudgetRoom(2, 23, [], 6)
>>> br2.better_than(br1)
True
>>> br1.grant_clean()
>>> br1.move_to(br2)
>>> br2
floor: 2
number: 23
guests: loren, or
clean_level: 6
rank: 1
satisfaction: 3.8
type: BudgetRoom
clean_stock: 1
```

C.2) The class *LegacyRoom* should implement the following methods:

Header of the method	**Description**
is_occupied(self)	Same description as for class *Room*
can_clean(self)	Returns a boolean value indicating whether the room can be cleaned • A *LegacyRoom* can always be cleaned
clean(self)	Same description as for class *Room*
better_than(self, other)	Same description as for class *Room*
check_in(self, guests)	This method should work as defined in class *Room*, with the addition that, if the room *self* is empty, the method initializes *self.minibar_drinks* and *self.minibar_snacks* to 2. • In the implementation of the method, you need to call the corresponding method of the base class (*Room*). • 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)	Same description as for class *Room*
move_to(self, other)	Same description as for class *Room*
add_drinks(self, quantity)	Increases the number of drinks in the minibar by *quantity* (a strictly positive integer (*int*)) • As a result, the degree of satisfaction of the guests (*satisfaction*) is increased to: $\min(5.0, satisfaction + 0.2 * quantity)$ • You may assume that the type and the value of *quantity* are valid • You may assume that there are guests in the room when the method is called
add_snacks(self, quantity)	Increases the number of snacks in the minibar by *quantity* (a strictly positive integer (*int*)) • As a result, the degree of satisfaction of the guests (*satisfaction*) is increased to: $\min(5.0, satisfaction + 0.3 * quantity)$ • You may assume that there are guests in the room when the method is called • But the snacks make the room more dirty, and hence lower the cleanness level *clean_level* to: $\max(1, clean\_level - 1)$ • You may assume that the type and the value of *quantity* are valid

Example of execution (make sure you understand all the results):

```
>>> lr1 = LegacyRoom(5, 94, ["Ronen", "Dror", "Liat", "Smadar"], 5)
>>> lr1.satisfaction
1.0
>>> lr1.add_drinks(3)
>>> lr1.minibar_drinks
5
>>> lr1.satisfaction
1.6
>>> lr1.clean()
>>> lr1.clean_level
7
>>> lr1.add_snacks(2)
>>> lr1.minibar_snacks
4
>>> lr1.satisfaction
2.2
>>> lr1.clean_level
6
>>> lr1.check_out() ## note: None is returned, and so nothing is printed
>>> lr1.is_occupied()
False
>>> lr1.check_out()
Traceback (most recent call last):
  …
RoomError: Cannot check-out an empty room
```

Question 3

We will now implement the class *Hotel* which represents the Hotel.

A) Implement the constructor *__init__(self*, *name*, *rooms)* which receives the name of the hotel (parameter *name*, of type string) and a list of rooms (parameter *rooms*). You do not need to check the validity of the rooms.

Instructions:

- You may assume that *name* is a valid string that represents the name of the hotel, and contains only spaces, numbers and English letters (in lowercase and uppercase)
- You may assume that the list *rooms* is not empty and that each element of the list is a valid room (of type *Room* or a type derived from *Room*), and that each element represents a different room (i.e. no room object appears twice in the list, and two different rooms never have the same room number and 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* can contain both occupied rooms and empty rooms.
- You may keep the rooms as an attribute of the hotel object, using any Python data structure that you like. More precisely, you are not obliged to use a list for the representation of the rooms in internal implementation of the object.
- You are allowed to 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 BudgetRoom objects><one space>BudgetRooms\n
<number of LegacyRooms objects><one space>LegacyRooms\n
<number of Room objects that are not instances of BudgetRoom or
LegacyRooms><one space>other room types\n
<number of occupied rooms><one space>occupied rooms"
```

Example of execution:

```
>>> h = Hotel("Best",[BudgetRoom(15, 140, [], 5), BudgetRoom(1, 2,
["Liat"], 7)])
>>> h
Best hotel has:
2 BudgetRooms
0 LegacyRooms
0 other room types
1  occupied rooms
```

Note: when writing the code of *__repr__*, the computation of the number of different rooms can be done in any way you decide. In particular, you are allowed to use helper methods.

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

Header of the method	**Description**
check_in(self, guests, rank)	The method tries to perform the check-in for the list of guest names *guests* (a list of strings) to *one* room (any room) having the rank *rank* (an integer number). • If an empty room with the required rank is found, the method will perform the check-in to this room for all the guests in *guests*. Then it will return the room object of the room that was found. If no suitable room was found, the method will return *None*. • You may assume that the names in the list *guests* do not 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 perform a check-out for the guest named *guest* (string) together with all the other guests who are staying with him 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*. • When searching for the room where *guest* is staying, you should disregard the letter case. For example, *UZI* is considered the same as *uzi*.
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 him to. • The upgrade operation includes moving the guest, with the other guests that are staying with him in the room (if any), to another room in the hotel which is vacant and "better" (as defined in question 1) 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*.

	• When searching for the room where *guest* is staying, you should disregard the letter case. • If there are several available rooms to which one can upgrade the guest, you may choose any of them.

Notes:

- Every method should always end its 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 lower case and/or upper case.

Example of execution:

- A code resembling that of the example hereunder is implemented in the function ***test_hotel*** in the outline file**.**
- The file ***test_hotel_output.txt*** (attached to the homework) contains the printed result of the execution of the above function, and is intended for you to check that the printed values of your program are the same.
- 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).

```
>>> rooms = [BudgetRoom(15, 140, [], 5), LegacyRoom(12, 101, ["Ronen",
"Shir"], 6), BudgetRoom(1, 2, ["Liat"], 7), Room(2, 23, [], 6, 3)]
>>> h = Hotel("Dan",rooms)
>>> h.upgrade("Liat")
floor: 15
number: 140
guests: liat
clean_level: 5
rank: 1
satisfaction: 2.0
type: BudgetRoom
clean_stock: 0
>>> h.check_out("Ronen")
floor: 12
number: 101
guests: empty
clean_level: 6
rank: 2
satisfaction: 1.0
type: LegacyRoom
minibar_drinks: 2
minibar_snacks: 2
>>> h.check_in(["Alice", "Wonder"], 2)
floor: 12
number: 101
guests: alice, wonder
```

```
clean_level: 6
rank: 2
satisfaction: 1.0
type: LegacyRoom
minibar_drinks: 2
minibar_snacks: 2
>>> h.check_in(["Alex"], 3)
floor: 2
number: 23
guests: alex
clean_level: 6
rank: 3
satisfaction: 1.0
>>> h
Dan hotel has:
2 BudgetRooms
1 LegacyRooms
1 other room types
3 occupied rooms
>>> h.check_in(["Oded", "Shani"], 3)
>>> h.check_in(["Oded", "Shani"], 1)
floor: 1
number: 2
guests: oded, shani
clean_level: 7
rank: 1
satisfaction: 1.0
type: BudgetRoom
clean_stock: 0
>>> h.check_out("Liat")
floor: 15
number: 140
guests: empty
clean_level: 5
rank: 1
satisfaction: 2.0
type: BudgetRoom
clean_stock: 0
>>> h.check_out("Liat")
>>> h
Dan hotel has:
2 BudgetRooms
1 LegacyRooms
1 other room types
3 occupied rooms
```

Good luck!