Problem 1

Write a class Room, which has the following properties:
- Building Name
- Room Number
- Size (2 dimensions)
- Maximum Occupancy (max # of people who can be inside)
- Owning Department Name

Write an __init__() method. Include at least 3 assert statements to validate the various parameters.

Suppose that you had a pair of dictionaries, which listed the names of all the buildings, and all of the departments. Use the “in” expression to confirm that the building and department names are valid names. (If you have forgotten this expression, do a Google search for “python in dictionary”.)

While in a real program, we would have getters for all of the properties, skip over those methods this time.

Thinking about setters: which of the properties are things which might change over time? We would need setters for those properties. (But it’s kind of pointless to write a setter for something that will never change.) Write down which properties you think might change, and which might not. Explain your reasoning.

class Room:
    
    def __init__(self, building, room_no, size, max_occ, owning_dept):
        # we assume that there are global variables
        # buildings
        # departments
        # which are dictionaries, linking names (strings) to more data

        assert building in buildings, "Invalid building name"
        assert owning_dept in departments, "Invalid department name"
        assert len(size) == 2, "The size must be a pair of numbers"
assert size[0] > 0 and size[1] > 0, "The size must be positive in both dimensions"

self._building = building
self._room_no = room_no
self._size = size
self._max_occ = max_occ
self._owning_dept = owning_dept

Which properties might change?
I would create setters for (at least) the owning department. Presumably, the building name and room number won't change (at least, not often), so we could skip that. (If we anticipated that building names changed over time, we could store the building object (instead of the name) as a field - we could look up using the dictionary that we are already using.) Maximum occupancy **might** change, but probably will not - since that generally is a property of the room design. (I'm assuming that the size won't change.)

Of course, any one of these **might** change over time - for instance, we might knock out a wall and change the size. But I'm willing to make the simplifying assumption that if we do that, we could just create a new Room object - instead of modifying the one we already have. Certainly, **most of the time** any attempt to change these things would indicate a bug in the program.
Problem 2

In Problem 1, we wrote a Room class - which represents a few properties about a Room which would either never change, or change very rarely. But there are other properties, which might change more rapidly.

One option is to simply add more fields to Room - and that’s not a terrible idea. But let’s explore another option: a RoomState class. Each RoomState class will contain a reference to a single Room object (this reference will never change), and many other fields which will change often.

Define a RoomState class. The __init__() method must take a reference to the Room object - and this reference must never change after __init__() finishes. Define at least 4 other properties that you might store about a room. For at least one of them, pass the initial state as a parameter to __init__() - for at least one of them, do NOT use a parameter, and instead give the parameter a default value. (You get to define what the various other properties are.)

Skip gettors for these properties, to save time. However, write at least 4 different settor methods (they could all be for the same property, or for different ones). But here’s the trick: none of the settors are allowed to simply take “the new value” as a parameter. Instead, they must change the property indirectly. For instance, if you a temperature property, then you might have a become_hotter() method.

Then, write a single method, which answers a question about the class, but is more interesting than a simple getter. For instance, you could as is_dangerously_hot(), or is_occupancy_exceeded().

class RoomState:
    def __init__(self, room, chairs, equipment):
        """Initializes a RoomState object, which store the (temporary)
        state of a Room - in contrast to the (almost never changes)
        properties stored in the Room class.

        Parameters: The room, the number of chairs (non-negative integer),
        and the current list of equipment in the room (undefined format)

        Post-conditions: sets current occupancy to 0, and current
        temperature to 76"""
        assert room is not None
        assert chairs >= 0

        self._room    = room
        self._chairs  = chairs
        self._equipment = equipment
        self._cur_occ  = 0
        self._temp_f  = 76

    def add_chairs(self, num):
        self._chairs += num
def fill_to_max_occ(self):
    self._cur_occ = self._room.get_max_occupancy()

def make_dangerous_temp(self):
    if random.randint(0, 1) == 0:
        self._temp_f = 0
    else:
        self._temp_f = 140

def steal_all_equipment(self):
    retval = self._equipment
    self._equipment = []
    return retval

def is_freezing(self):
    return self._temp_f <= 32
Problem 3

I have written some if statements below. At the beginning of each block, write all of the assertions which you know are true. (Assume that each “condA” represents some sort of expression with a Boolean result - that is, it is either True or False.)

```python
assert condA
if condB:
    # example
    assert condA
    assert condB
    return []

if condC:
    # write assertions here

    SOLUTION:
    assert condA
    assert not condB
    assert condC

elif condD:
    # write assertions here

    SOLUTION:
    assert condA
    assert not condB
    assert not condC
```
assert condD

else:
    # write assertions here

SOLUTION:
assert condA
assert not condB
assert not condC
assert not condD