UP2 Spec: Managing a Group

Introduction

In UP2, we will be implementing another building block of the chat system: group management. We are not yet at the stage for implementing chatting, but we are going to create the system that will help us manage it. It will allow users to join a group, as well as connect and disconnect to one another.

When a peer *s* signs in to the group, she can issue commands to find out who are in the system. She may see something like this:

{a}, {b}, {c, d}

This indicates that there are two loners *a*, and *b*, whereas *c* and *d* are talking (presumably). If she connects to *a*, the management system should update accordingly:

{a, s}, {b}, {c, d}

Or she can connect to *c* (or *d*), and become part of their group:

{a}, {b}, {c, d, s}

Likewise, if *s* decides to disconnect from her group, and then *d* did the same, we will end up with *no* group -- everyone is a loner:

{a}, {b}, {c}, {d}, {s}

The constraints for group management are:

- A group has more than 1 member
- A member can be in **one and only one** group at a given time

This is already a simple state machine: a peer moves from/to state `S_TALKING` to/from `S_ALONE`, when connecting/disconnecting to/from a member.

Spec: Group class

The chat_group.py implements the `Group` class:

```
24  class Group:
25
26      def __init__(self):
27          self.members = {}
28          self.chat_grps = {}
29          self.grp_ever = 0
30
31      def join(self, name):
32          self.members[name] = S_ALONE
33          return
```

In the `Group` class, there are two dictionaries:

- *`members`* records all the users and their states. Each item is a key-value pair which records the user's current state, i.e. `S_ALONE` or state talking (`S_TALKING`). Use a user's name as the key to map to the state in this dictionary.

- *`chat_grps`* stores groups that talking to each other as key-value pairs, mapping group numbers (0...n) to the lists of users inside that group. Only when a client is in state `S_TALKING` is she in one of the chat groups. The key of the dictionary is internal to the `Group` class. To make sure keys are unique, we use a variable *grp_ever* that is incremented each time a new group is formed. Each group is a list of members.

  In the case of {a}, {b}, {c, d, s}, there is only *one* group, with a list `[c, d, s]` (not necessarily in this order, `[d, c, s]` is fine too).

The following page shows a table of the member functions, as well as the functionality we expect to have implemented. `self.join(name)` is already implemented, as shown earlier.

Member functions	Description
join(name)	A user with "name" joins. Add her to the *member* dictionary, initial state is S_ALONE
leave(name)	A user with "name" quits. Delete her from the *member* and *chat_grps* dictionary
is_member(name)	Return True if the user "name" is in the system (i.e. in the *member* dictionary)
connect(me, peer)	Connect "me" to "peer". If "peer" is already in a group, join "peer"'s group; otherwise create a new group, record the new group in *chat_grps* dictionary. In the second case, `grp_ever` should increment by one.
disconnect(me)	Remove "me" from my current chat group. If the group has only one peer left, remove that peer as well, and delete the chat group.
list_my_peers(name)	Return the chat group "name" is in, as a list. IMPORTANT: "name" is the first element in that returning list.
list_all()	Return all the information in the system; the code is given. You can make it fancy, but it NEEDS TO BE A STRING.
find_group(name)	Auxiliary function internal to the class; return two variables: whether "name" is in a group, and if true the key to its group

```
    def list_all(self):
        # a simple minded implementation
        full_list = "Users: ----------" + "\n"
        full_list += str(self.members) + "\n"
        full_list += "Groups: ----------" + "\n"
        full_list += str(self.chat_grps) + "\n"
        return full_list
```

Note that list_all() has a simple-minded implementation given.

The functions you need to implement for this project are given in **BOLD** in the table.

HINTS:

- It helps to review the syntax of dictionaries and lists first; you will need them
- Go step by step, for example implement *join* and *is_member* first to warm up
- The tricky part of *connect* and *disconnect* is to handle new group creation and delete singleton groups (group with only one member). Make sure the *member* dictionary is manipulated appropriately
- Use the console to debug interactively. The below picture shows an example, when running chat_group.py in ipython (some debug info is mine):

```
In [37]: g = Group()

In [38]: g.join("Anthony Fauci")

In [39]: g.join("Boris Johnson")

In [40]: g.join("Donald Trump")

In [41]: g.join("Donald Knuth")

In [42]: g.connect("Boris Johnson", "Donald Trump")
Donald Trump is idle as well
['Boris Johnson', 'Donald Trump']

In [43]: g.connect("Donald Knuth", "Donald Trump")
Donald Trump is talking already, connect!
['Donald Knuth', 'Boris Johnson', 'Donald Trump']

In [44]: print(g.list_all())
Users: ------------
{'Anthony Fauci': 0, 'Boris Johnson': 1, 'Donald Trump': 1, 'Donald Knuth': 1}
Groups: ------------
{1: ['Boris Johnson', 'Donald Trump', 'Donald Knuth']}


In [45]: g.connect("Anthony Fauci", "Donald Knuth")
Donald Knuth is talking already, connect!
['Anthony Fauci', 'Boris Johnson', 'Donald Trump', 'Donald Knuth']

In [46]: print(g.list_me("Donald Trump"))
['Donald Trump', 'Boris Johnson', 'Donald Knuth', 'Anthony Fauci']
```

Alternatives and Improvements

Once you have worked out the core logic, you may attempt to make some implementation improvements according to this spec.

Some challenges to think about and implement functions for:

- How many loners are in the system?
- What is the biggest group?
- Can you list all the groups with 2 members?
- ….