PythonSharing

In this exercise, we will implement an ultra-simplified free-for-all scooter sharing service. We will have three classes, `Scooter`, `SharingService` and `SSUser` (“sharing-service user”). The general idea is as follows:

- Each scooter has a position (a `Vec2D`), and keeps track of which user is currently using it, if any. It can be queried to know whether it’s free or not.
- The sharing service has a list of scooters. It can be queried to get a list of free scooters, if there are any, and to return the closest free scooter from a given position.
- Each user is registered to the service. It has a position (again, a `Vec2D`). It knows which scooter it’s currently using, if any. It can take the closest available scooter (provided there is one). It can use it to get to a new position (after which the scooter will be available again, but at the new position).

Throughout the exercise, *you are not required to perform any argument checking*. The constructors are already mostly (not fully) implemented. An auxiliary function called `argmin` is also provided. A test script called `"runtest.py"` is already written to help you when you’re writing your code.

Here are the details of the tasks that you need to do to complete the exercise:

1. **[PARTIALLY-IMPLEMENTED]** The constructor for the class `Scooter` is supposed to take two arguments, the coordinates of the position. You should use them to create a `Vec2D` object and store it in an attribute of the scooter called `pos`. The constructor also adds a `user` attribute to the scooter, initially set to `None`.

   Example:

   ```
   In [1]: s = Scooter(1.0, 0.0)

   In [2]: s.pos
   Out[2]: V[1.0, 0.0]

   In [3]: s.user is None
   Out[3]: True
   ```

2. Add a method to the `Scooter` class called `isfree` that takes no arguments and just returns `True` is the `user` field is `None` and `False` otherwise.

   Example (continued from above):

   ```
   In [4]: s.isfree()        # after initialization the scooter is free
   Out[4]: True

   In [5]: s.user = "FAKE!"  # let's just put something in there for testing

   In [6]: s.isfree()
   Out[6]: False
   ```

3. When printed, a `Scooter` object should print `"Scooter"` followed by its position and its status (`True` if free, `False` otherwise), in parentheses, see the example. The status should be obtained from the `isfree` method.

   Example (continued from above):

   ```
   In [7]: s # s.pos is V[1.0,0.0], and s.isfree() returns False
   Out[7]: Scooter(V[1.0, 0.0], False)
   ```

4. **[PARTIALLY-IMPLEMENTED]** The constructor for the class `SharingService`, takes a `scooter_poslist` as its argument. This is a list of positions, each one provided as a tuple with two coordinates. Currently, the constructor sets its field `scooters` to an empty list. Change it to become a list of `Scooter` objects, each one initialized from a position in the `scooter_poslist` argument. Tip: the shortest way to write this is using a comprehension syntax.

Example (**not** continued from above):

```
In [1]: ss = SharingService([(1,0), (0,2), (-2,1)]) # three scooters

In [2]: ss.scooters # we created three scooters, all of them currently free
Out[2]: [Scooter(V[1.0, 0.0], True), Scooter(V[0.0, 2.0], True), Scooter(V[-2.0, 1.0]) True)]
```

5. When printed, a `SharingService` object should just print `"SS"` followed by its `scooters` list in parentheses, as in this example (continued from above):

```
In [3]: ss
Out[3]: SS([Scooter(V[1.0, 0.0], True), Scooter(V[0.0, 2.0], True), Scooter(V[-2.0, 1.0]) True)])
```

6. **[PARTIALLY IMPLEMENTED]** The constructor for the class `SSUser` takes two arguments, a `SharingService` object called `service` and a tuple of two numbers `tpos` denoting its position. It initializes three internal attributes: `service`, `pos` and `scooter`. The `service` attribute should just use the argument, but it should check that it is a `SharingService`. The `pos` attribute should be a `Vec2D` initialized from the given `tpos`. The `scooter` attribute should be initialized to `None` (meaning that the user is not currently using any scooter; it will eventually contain `Scooter` objects when the user takes a scooter).

   Example (continued from above):

```
In [4]: u1 = SSUser(ss, (0,0))

In [5]: u1.service is ss
Out[5]: True

In [6]: u1.scooter is None
Out[6]: True
```

7. When printed, a `SSUser` should just print `"SSUser"` followed by its position and current scooter, in parentheses and with some additional text, as in this example (continued from above):

```
In [7]: u1
Out[7]: SSUser(pos=V[0.0, 0.0], scooter=None)
```

8. Add a method called `free_scooters` to the `SharingService` class that takes no arguments and returns the list of free scooters. It should go through its internal `scooters` list, use the `isfree` method from the `Scooter` class, and accumulate the free scooters in a new list. The list may end up being empty if there are no free scooters. Tip: the shortest way to do this is using a filtered comprehension; as a challenge, you may try using the `filter` function with a `lambda`.

   Example (continued from above):

```
# when all scooters are free, it returns a list equal (but not identical)
# to its internal scooters list
In [8]: ss.free_scooters()
Out[8]: [Scooter(V[1.0, 0.0], True), Scooter(V[0.0, 2.0], True), Scooter(V[-2.0, 1.0]) True)]

In [9]: ss.scooters[1].user = u1 # let's force a scooter to be non-free

In [10]: ss.free_scooters() # the scooter with index 1 is now skipped
Out[10]: [Scooter(V[1.0, 0.0], True), Scooter(V[-2.0, 1.0]) True)]

In [11]: ss.scooters # the internal scooters list is left untouched
Out[11]: [Scooter(V[1.0, 0.0], True), Scooter(V[0.0, 2.0], True), Scooter(V[-2.0, 1.0]) True)]
```

9. Add a method called `closest_scooter` to the `SharingService` class that takes a `Vec2D` argument `pos` and does the following:

   - gets a list of free scooters from the `free_scooters` method
   - if there are no free scooters, it returns `None`

- otherwise it creates a list of distances between `pos` and each free scooter, using the `dist` method of `Vec2D` and the `pos` attribute of the scooter object. Just do this for all scooter objects (e.g. with a comprehension).
- it uses the `argmin` function (provided) on the distances list to get the index of the closest free scooter, and finally uses that to return the corresponding `Scooter` object

Example (continued from above):

```
# the scooter at (0,2) is not free, we get the one at (1,0)
In [12]: ss.closest_scooter(Vec2D(0, 1.5))
Out[12]: Scooter(V[1.0, 0.0], True)

# let's make the other scooters non-free, just for testing
In [13]: ss.scooters[0].user = "FAKE!"
In [13]: ss.scooters[2].user = "FAKE!"

In [14]: ss.closest_scooter(Vec2D(1, 0)) is None # no free scooters, return None
Out[14]: True

# let's revert the scooters status
In [15]: ss.scooters[0] = None
In [16]: ss.scooters[1] = None
In [17]: ss.scooters[2] = None
```

10. Add a method called `take_scooter` to the `SSUser` class that takes no arguments and does the following:

    - if the user's `scooter` attribute is already taken (not `None`), raise an exception
    - call the `closest_scooter` method on the user's `service` attribute, with the user's `pos` attribute as its argument. If the result is `None`, raise an exception. Otherwise we'll get a `Scooter` object, call it `s`
    - the user goes to the scooter: set its `pos` attribute to be equal to that of the scooter
    - the user takes the scooter: set the user's attribute `scooter` to the found object `s`, and the scooter's attribute `user` to the user that is calling the method.

    Example (continued from above):

```
In [18]: u1  # before
Out[18]: SSUser(pos=V[0.0, 0.0], scooter=None)

In [19]: u1.take_scooter() # going to take the scooter at (1, 0)

In [20]: u1  # after: changed position, took the scooter
Out[20]: SSUser(pos=V[1.0, 0.0], scooter=Scooter(V[1.0, 0.0], False))

In [21]: u1.take_scooter() # must give an error, u1 is already on a scooter
### SOME ERROR MESSAGE HERE...

In [22]: u1.scooter.user is u1 # self-consistency check
Out[22]: True

In [23]: u2 = SSUser(ss, (0,3)) # create another user

In [24]: u2.take_scooter() # takes the scooter located at position (0,2)

In [25]: u3 = SSUser(ss, (0,0)) # create one more user

In [26]: u3.take_scooter() # takes the scooter located at position (-2,1)

In [27]: ss.free_scooters() # no more free scooters
Out[27]: []

In [28]: u4 = SSUser(ss, (3,5)) # create one more user

In [29]: u4.take_scooter() # this must give an error, no free scooters left
### SOME ERROR MESSAGE HERE...
```

11. Add a method called `use_scooter` to the `SSUser` class that takes a `new_pos` argument (a `Vec2D`) and does the

following:

- looks into the `scooter` attribute of the user, and raises an exception if it is `None`
- complete the journey: sets both the user's and scooter's positions to `new_pos`
- dismount: set both the user's attribute `scooter` and the scooter's attribute `user` to `None`

Example (continued from above):

```
In [30]: u4 # this user is not on a scooter, its attempt to take one had failed
Out[30]: SSUser(pos=V[3.0, 5.0], scooter=False)

In [31]: u4.use_scooter(Vec2D(10,10)) # must give an error, not on a scooter
### SOME ERROR MESSAGE HERE...

In [32]: u1 # this user had taken the scooter at (1,0)
Out[32]: SSUser(pos=V[1.0, 0.0], scooter=Scooter(pos=V[1.0, 0.0], False))

In [33]: u1.use_scooter(Vec2D(5,-2)) # travel to position (5,-2)

In [34]: u1 # user is now at (5,-2) and off the scooter
Out[34]: SSUser(pos=V[5.0, -2.0], scooter=None)

In [35]: ss # the scooter that was at (1,0) is now at (5,-2) and free
Out[35]: SS([Scooter(V[5.0, -2.0], True), Scooter(V[0.0, 2.0], False), Scooter(V[-2.0, 1.0], False)])
```

12. The `take_scooter` and `use_scooter` methods above are supposed to raise exceptions in some cases. Write your own exception types and use them in those methods; call them `NonFreeUserError`, `NoFreeScootersError` and `UserNotOnScooterError`.

Extra challenges (if you have time)

13. As currently written, the `argmin` function goes through the data twice: the first time around to determine the minimum value, and the second time to determine its index. This is concise but inefficient. Rewrite it so that it only reads the input list once.

    For an extra challenge, do it in a single line of code with the `min` function, using a `lambda`. If you do this, you may want to also do the following.

    The `argmin` function is only used in the `SharingService.closest_scooter` method, described in task 9. Modify the end of that method: do not explicitly build a list of distances, and don't use `argmin` at all; instead, use the `min` method with an appropriate `key` to find the closest scooter.

14. We could easily make our model a little more realistic. For example:
    - Each `SSUser` could have a `credit` (and a method to increase it by some amount)
    - Each `Scooter` could have a usage cost, maybe proportional to the distance traveled.
    - So in order for a user to take a scooter, their credit should be non-zero; and after `use_scooter` you should detract the cost from the user's credit. The credit might then become negative.
    - Alternatively, you can decide that negative credits are not allowed, and that if the user credit gets to zero before reaching their destination, the scooter stops and the user is dismounted (we might assume that the users travel in straight lines).

15. What if at some point we wanted to take all of the free scooters back to some place, maybe the point `(0,0)`? Say that there is a worker with a van, and we wanted to design the shortest route that they could take, starting from `(0,0)`, collecting all the free scooters one by one, bringing them all back to `(0,0)`. How would you do that? (Hint: after you thought a while about this, you may want to google the acronym "TSP".)

16. Maybe scooters going around in straight lines on an empty plane is not the most realistic scenario. Maybe a `Map` object containing a graph of possible destinations with their distances would be better. Maybe users could set their destination and follow the shortest path from their origin to their destination. (Maybe one could look into gathering OpenStreetMaps data?)