CS 120: Intro to Computer Programming II

In-Class Activity - 04 Classes and Objects - Day 3

In this ICA, you'll practice with another class. This class will rely heavily on encapsulation.

Activity 1 - Turn in this one

Write a class `TODO_List`, which will keep track of the things that you need to get done. Eventually (if you have time during this ICA), we'll support urgent items, appointments with specific times, and even repeating events.

For now, we'll implement a very simple version. The first version should only keep track of "urgent" events; these are events that have to be done ASAP, and don't have any time associated with them. Your class should provide the following methods:

- `add_urgent(msg)`

  Adds a new urgent item to the TODO list.

- `get_first_ten()`

  Returns to the user the first ten items on the TODO list, as an array. (Return a shorter array if there are fewer than ten.) Each urgent item should be listed as an `(id,msg)` pair - the `id` is an auto-generated identifier for that task. (For this version of the class, it's fine if this is just the index into a private array.)

  Do not use Python's `id()` function to assign ID values! The IDs you use should be small numbers, like 1,2,3,4 ...

- `complete(id)`

  Completes the task, given by the ID value. The task should be removed from the TODO list, and never reported to the user again.

Activity 2 - Turn in this one

Experiment with the class you wrote in the previous step. What happens when you complete a task? If you took my suggestion, and just listed the items in an array, then completing a task shifts everything over in the array - meaning that it changes the IDs of all the other tasks. This isn't a nice interface for our users!

Modify your class, to make IDs unchanging. In this new version, instead of the ID simply being an index into an array, assign the ID whenever a new TODO item is created. Store the ID with the TODO item - the easiest way to do this, internally, is to store the `(id,msg)` pair in your internal array.

NOTE:
A simple and classic strategy for assigning IDs is to simply use a counter. Create a field (initialized to 1, probably) when the TODO list is created; each time that you add a new item, you use the counter as your current ID, and then increment the counter by one.

(activity continues on the next page)

Activity 3 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Now, we're going to add appointments to the TODO list. Appointments are less urgent than the "urgent" items, but appointments also have times associated with them.

Since we don't want to worry about the complexities of real-world time, we're going to simulate time with an integer. Time "begins" at $t = 0$, and you can advance it whenever you want (but no going backward!)

Add two new methods:

- `add_appointment(msg, time)`

  This adds an appointment to the TODO list. The time is given as an integer. Assign an ID to this appointment, just like you would an urgent item; make sure that IDs for the urgent items and the appointments don't overlap. (A simple way to do this is to use the same counter variable for assigning IDs to both.)

- `update_time(time)`

  This "moves the clock forward" to a new time. (Don't let the user ever move time backward.) Auto-delete any appointments which are now in the past.

In addition, you'll need to change some existing methods:

- `complete()` will change, because now there are two types of items. I would suggest keeping the urgent items and the appointments in separate arrays; if you do that, then `complete()` now needs to be updated to search both ones.

- When the user asks for the first ten items, always list the urgent items first - and if there are 10 or more urgent items, then that's **all** that they will get. But if there's space, return your appointments as well, sorted by time (of course).

  When you report an appointment in this array, use **three** entries in the tuple, not two: `(id,msg,time)` .

(activity continues on the next page)

Challenge Activity - Do not turn in this one

Add support for repeating events. Add a new method:

- `add_repeating_event(msg, first_time, how_often)`

  This is a lot like `add_appointment()`, except that a repeating event happens over and over, forever. `first_time` gives you the time when the first occurrence happens; `how_often` gives you how long it is between repititions.

Now, you have some design decisions to make - I'm not going to tell you all the details. Here are some things to think about:

- How to store a repeating event? Since (in theory) there are an **infinite** number of copies of each event, you certianly can't store "all" of them!
- How to report a repeating event to the user? Will it look exactly like an appointment, or different somehow?
- When an event repeats, will each iteration of the event have the same ID, or will they all be different?
- When a user calls `complete()` on an event, this probably only completes the first one on the current list. How do you keep track of what events have been completed at any given moment?
- Do you want to add an interface which would allow a user to delete an event entirely?