Homework 3

CS 173: Intermediate Computer Science — Spring 2014
Instructor: Thomas Bressoud — Due: 2014-02-18

Some of the goals of this homework assignment:

- To practice with container classes.
- To implement operator overloading.
- Continue learning to unit test.
- Build part of the foundation needed for our Discrete Event Simulation series.

Each of the following sections will specify the ADTs needed for this assignment. For each, you should go from the description to an interface specification written as docstrings in a Python class. There should be a docstring for the class itself and one for each method with preconditions and postconditions. From the command line, I should be able to run `pydoc <ADTfile>` on each Python file and get a complete description of the ADT interface.
Once the interface specification is written, you should implement both the unittest and the method implementations concurrently, writing tests that thoroughly test both normal cases usage of the methods and tests that verify correct operation of any exceptional conditions.
Note that, in this first specification of an Event ADT and Simulator ADT, we specify a specific behavior for the Event and Simulator. In a later assignment, we will make generic superclass versions of Event and Simulator and use Inheritance to obtain more specific behaviors.

OrderedSet ADT

The OrderedSet is an ordered collection of *unique* elements. This container class ADT should support the following operations:

- `s.insert(x)` inserts an element `x` into the ordered set `s`. The position of the insert is based on using relational operators on the elements of the ordered set to find the position such that any subsequent elements are $>$ the inserted element. Thus, if there are elements that compare equal to the inserted element, the inserted element will be the last of the equal elements. If the object referred to by `x` is already in the collection, it should be repositioned to its proper ordered position, and should not be present in the collection more than once.
- `s.removeFirst()` removes and returns the first element from the ordered set `s`. `None` is returned if there are no elements in the ordered set.

- `s.remove(x)` removes element `x` from the ordered set `s`. To match `x`, the id of `x` and the id of an element should have the same value, ensuring that they refer to the same object, as opposed to having the same comparison value. We should not use `==`, comparing values, to determine a match for the remove. An IndexError should be raised if `x` is not present in the ordered set.
- `len(s)` returns the number of elements in the ordered set `s`
- `str(s)` returns a string representation of the ordered set `s`. The string should begin with '[', the first element e, if it exists, should be the str(e) immediately followed by '*', and remaining elements should be comma-space separated string versions of each element. The string should terminate with ']'.
- `x in s` indicates whether an element `x` is contained in the set `s`. The detection of whether or not `x` is in the collection is not (necessarily) the same as the result from using relational operators, which check equal (or relative) value, not common object identity.

Operator overloading should be used to implement the `in` operator (with method `__contains__(self, item)`), and to get invocation on `s` from the built-in `len()` and `str()` functions. Implement your OrderedSet class in a file called `OrderedSet.py` using a Python list. Maintain a unit testing program for your class in a file called `test_OrderedSet.py`, using the unittest module. Since you will be graded on the quality and completeness of your test suite, think carefully about your tests.

CounterEvent ADT

A `CounterEvent` maintains the `time` information associated with a discrete event simulation event, and whose `execute` semantics, the set of steps to perform when the DES time arrives, are given below. Note that multiple (different) CounterEvents might have the same time and still be different events. Since we are overloading relational operators, this means that references to two CounterEvents `a` and `b` might compare equal by the `==` operator, but may or may not yield `True` for the boolean expression `a is b`.

- `CounterEvent(t)` Creates a new CounterEvent object whose `time` value is given by `t`. The data type of t may be an int or a float, but should always be stored as a float.
- `str(e)` yields a string of the form `<Event:  2.5>` if the event has time value 2.5. So the string begins with a less-than sign, the word 'Event' followed be a colon and a space, the string representation of the time float, and then a greater-than sign.
- `==`, `<`, `!=`, and `<=`: The relational operators return a boolean `True` or `False` reflecting the comparison of the two CounterEvents instance `time` variable. Note that `!=` and `<=` can be implemented by simple use of the first two, and that Python will automatically handle the symmetric relational operators `>` and `>=`.

- `e.time()` returns the time value from CounterEvent `e`.
- `e.execute(sim)` The parameter, `sim`, is a reference to a CounterSim specified below. The execute method both prints and returns the string 'The event time is', followed by a single space and then the string version of the time of the event. In addition, the execute method will "reschedule" itself with the simulator by adjusting its time to increase by 2.0 and (re)insert itself into the simulators' collection of events. This reschedule only happens if the (current) event time is < 10.

Operator overloading should be used to implement the relational operators. Implement your CounterEvent class in a file called `CounterEvent.py`. Maintain a unit testing program for your class in a file called `test_CounterEvent.py`, using the unittest module.

CounterSim ADT

A `CounterSim` is the real driver for this simulation. It's information/data is an OrderedSet whose elements are CounterEvents, along with a float that holds simulated time.

- `CounterSim()` Creates a new CounterSim object with instance variables for an empty OrderedSet event collection and time, initially 0.0.
- `now()` returns the current time from the CounterSim.
- `insert(e)` inserts the event given by `e` in time order into its collection of events.
- `setup(t = 0.0)` Initializes the CounterSim simulation by creating a first CounterEvent whose desired execution time is given by `t`, and with a default value for t of time 0.0. The CounterSim insert is invoked to initialize the simulation with this newly created event.
- `doAllEvents()` is the simulation engine. Its operation is a simple loop that, as long as there are events available in the event collection, repeats the following:
  - Remove the first (smallest event time) event from the collection.
  - Set the virtual time to the time of the event.
  - Invoke the execute method of the event, passing itself as the needed simulator argument.

Implement your CounterSim class in a file called `CounterSim.py`. Maintain a unit testing program for your class in a file called `test_CounterSim.py`, using the unittest module. We will use this unittest in lieu of a main.py to test just a small set of simple executions of the counter simulator. For instance, one part of the unittest in my implementation is:

```
class CounterSimTest(unittest.TestClass):

  def testStartZero(self):
```

```
sim = CounterSim()
sim.setup()
sim.doAllEvents()
```

The printed output from this test yields the following:

```
The Event time is 0.0
The Event time is 2.0
The Event time is 4.0
The Event time is 6.0
The Event time is 8.0
The Event time is 10.0
```

Please submit your homework (`OrderedSet.py, test_OrderedSet.py`,`CounterEvent.py, test_CounterEvent.py`, `CounterSim.py, test_CounterSim.py`) through submitbox. Post any questions or requests for clarification via Piazza so that all students may benefit. I will answer questions there instead of by email, and any email queries I will redirect to be asked on Piazza.