Lab 6 - Observer Pattern, Callable Objects and Comprehensions

Rahul Kukreja & Jeffrey Yim

jyim3@bcit.ca

Welcome!

In today's lab, you will:

1. Simulate an auction involving an auctioneer and bidders using the observer pattern
2. Get a chance to use dictionary comprehension
3. Experiment with callable objects (refer to the sample code 'callable_object_example.py' uploaded for this lab)

Grading

This lab and all future labs will be marked out of 10

For full marks this week, you must:

1. (4 points) Generate a correct solution and implementing the observer pattern
2. (4 point) For the correct use of callable objects and dictionary comprehensions
3. (2 point) Follow PEP-8 standards to correctly format and comment your code. This also includes best practices such as meaningful identifier names, breaking down code into re-usable functions, etc.

Requirements

Clone the project from: **https://classroom.github.com/a/a-h40-yv**

For this lab you will implement an Auction House simulation that consists of an `Auction`, an `Auctioneer` and a number of `Bidders`. All your code should be in one module that is provided for you: **`driver.py`**.

Let's break down each of our classes. I've listed the attributes I expect to see**. I have provided you with some skeleton code to start off with**. Feel free to add/remove any methods or functions or change the class design however you feel. The rest of the class design is up to you. **DO NOT** create any other classes besides these 3.

Bidder - The Observer

Bidders are the people who place the bids during an auction. These observe the auctioneer. Every time the auctioneer accepts a bid, all the bidders should be notified and given the chance to retaliate by placing a new bid. A bidder cannot retaliate to their own bid.

Each bidder has the following attributes:

- `name`
    - The name of the bidder
- `budget`
    - The amount of money that the bidder is willing to spend. The bidder will not make a bid greater than this amount.
- `bid_probability`
    - This is a floating point value between 0 and 1. This represents the percentage probability chance that this bidder will retaliate to a bid with their own bid.
- `bid_increase_perc`
    - This is a number greater than `1` . A value of `1.4` translates to `140%`. This percentage is the value of the new bid that the bidder places. For example, if the bidder has a `bid_increase_perc` value of `1.5` (that is, 150%) then if this bidder were to place a bid, it would be `1.5` times the current highest bid on the item.
- `highest_bid`
    - The highest bid amount that was bid by this bidder.

The bidder should be a callable object. A callable object is an object that can be called as a function. For example, if I have an object `my_object` which implements the `__call__(self)` protocol, then I can execute the object as a function by typing `my_object()`. This would invoke the `__call__(self)` method in the background. You can add any number of parameters to this protocol to pass arguments when invoking the object as a function. Refer to the sample code uploaded alongside this lab for an example.

The bidder should implement `__call__(self, auctioneer)`. This method should allow the bidder to place a new bid with the auctioneer (thereby causing another new bid, causing the auctioneer to notify all the observers again). Remember, a bidder should only bid if:

- The bid is against another bidder and not against themselves
- The amount that they bid (dictated by `bid_increase_perc` is not greater than their budget
- After accounting for their `bid_probability` . (Hint: use the `random.random()` method to generate a random float between 0 and 1 (exclusive))

Auctioneer - The Core

The Auctioneer is the core object being observed in our observer pattern. The auctioneer is responsible for maintaining a list of bidders and notifying them (calling them as functions) if and when it accepts a new bid.

An auctioneer (not a static class in case you were planning to make it static) contains the following attributes:

- `bidders` A list of bidder objects that the auctioneer can call as a function.
- `highest_current_bid` The value of the highest current bid.
- `highest_current_budder` A reference to the highest current bidder.

The auctioneer can accept new bids and if this bid is greater than the `highest_current_bid` then it accepts the bid and notifies all the bidders **EXCEPT** for the bidder that placed the new bid. Remember, each `Bidder` expects a reference of the auctioneer to be passed to it when it is called.

Auction - Our controller that starts the simulation

The `Auction` class represents an auction object. Again this is not a static class (I do not want to see any static methods in this lab).

The `Auction` is responsible for setting up and starting an auction for an item with a starting price. An `Auction` should implement `__init__(self, bidders)` as its initialization parameters. `Bidders` refers to a list of participating bidder objects.

#Jeff change: Clarified Auction init only needs bidders. Feel free to add extra parameters if needed

The `Auction` is responsible for registering the bidders to the `Auctioneer` (so they can be called back as part of the observer pattern). Once registered, the auction starts by placing the `item` and `starting_price` as the first bid via the `simulate_auction` function. This should cause all the observers to be notified which should lead to a chain of new bids which in turn, cause the observers to be notified again.

#Jeff change: Clarified when and how to call simulate_auction

Once the chain has come to an end (either no one bids or everyone has exhausted their budget), the Auction object should print out the result of the auction. This should be the name of the `Bidder` and the winning bid.

OPTIONAL CHALLENGE - use Dictionary Comprehensions to iterate over all the bidders, and create a dictionary where each bidder is the key and their highest bid is the associated value. Print out a summary where each bidder's name and their highest bid is displayed.

Sample Output

Your `main()` method should prompt the user for the following:

1. The name of the item being auctioned
2. The starting price
3. The number of bidders and the details of each bidder

    a. Optional: Have some test bidders preloaded. Give the user an option to add more bidders or use the existing test bidders

Once this has been done, create an `Auction` object and start the auction simulation! A possible output can look like the snipper below. Since we aren't worrying about a dedicated UI class, your print statements can be put wherever you deem them to be necessary.

```
Starting Auction!!
------------------
Auctioning Antique Vase starting at 100
Samantha bidded 120.0 in response to Starting Bid's bid of 100!
Rahul bidded 192.0 in response to Samantha's bid of 120.0!
Priya bidded 345.6 in response to Rahul's bid of 192.0!
Rahul bidded 552.96 in response to Priya's bid of 345.6!
Priya bidded 995.3280000000001 in response to Rahul's bid of 552.96!
Samantha bidded 1194.3936 in response to Priya's bid of 995.3280000000001!
Priya bidded 2149.90848 in response to Samantha's bid of 1194.3936!
Samantha bidded 2579.890176 in response to Priya's bid of 2149.90848!
Jojo bidded 3353.8572288 in response to Samantha's bid of 2579.890176!
Priya bidded 6036.9430118400005 in response to Jojo's bid of 3353.8572288!
Jojo bidded 7848.025915392001 in response to Priya's bid of 6036.9430118400005!

The winner of the auction is: Jojo at $7848.025915392001

Highest Bids Per Bidder
Bidder: Priya   Highest Bid: 6036.9430118400005
Bidder: Samantha        Highest Bid: 2579.890176
Bidder: Rahul   Highest Bid: 552.96
Bidder: Jojo    Highest Bid: 7848.025915392001
```

- Ensure you push your work to github classroom. I'd like to see sensible git commits comments, and commits must take place at logical points in development.

That’s it. Good luck, and have fun!