Modeling Motion Computing Final
Winter Quarter

Turn in your Python code defining the following two classes:

1) Write a module `aviary` containing a class `Bird` that will make the following short program print the output indicated. The `Bird` class should implement the following specification:

    1. A `Bird` should get a name when it is created and print out "Hi, I'm " followed by the bird's name as shown in the sample output.
    2. At any time `Bird` objects should be either flying or not flying.
    3. When a new `Bird` object is created, it should not be flying.
    4. Each `Bird` should have a method `fly()` which takes no arguments
    5. If the `Bird` was not flying when `fly()` was called, it should print its name followed by the string, "is now flying".
    6. If the `Bird` was already flying when `fly()` was called, it should print its name followed by the string "is already flying".
    7. Each `Bird` should have a method `land()` which takes no arguments.
    8. if a `Bird` is flying when `land()` is called, it should print its name followed by the string "landed".
    9. if a `Bird` is not flying when `land()` is called, it should print its name followed by the string "already perched".

    Test program:

    ```
    from aviary import Bird
    feathers = Bird(name="Feathers")
    feathers.land()
    feathers.fly()
    feathers.fly()
    feathers.land()
    hawk = Bird(name="Talon")
    hawk.fly()
    ```

    Expected output from the test program:

    ```
    Hi, I'm Feathers
    Feathers already perched
    Feathers is now flying
    Feathers is already flying
    Feathers landed
    Hi, I'm Talon
    Talon is now flying
    ```

2) Create a class called `LogisticEquation` which provides an abstraction for the following equation:

$$\frac{dP}{dt} = kP\left(1 - \frac{P}{C}\right)$$

`LogisticEquation` should have the following capabilities which should allow the test program below to run successfully drawing two graphs in separate windows and printing out the (estimated) value of the function at two points:

1. Its `__init__()` method should allow the user to specify values for k, C, and an initial value (the value of the solution at time $t = 0$).
2. It should have attributes called `k` and `C` and `intialvalue` representing the corresponding values.
3. It should have an `evaluate()` method which takes a values for time as an argument and returns the value of the solution to the logistic equation at that time. This value can be computed directly from a solution formula or estimated (e.g. with Euler's method).
4. Its should have a `drawGraph()` method which draws a graph of the solution over a period of time beginning at $t = 0$.
5. The `drawGraph()` method should take two arguments. The first should be the maximum time shown on the right side of the graph. The second optional argument should be the spacing in time between points on the graph (Δt).
6. If the second argument to `drawGraph()` is not specified when calling the method, a default value of .1 units should be used.
7. Successive calls to the `drawGraph()` method should create a new window with a new graph leaving previous graphs visible.

Test program:

```
logeq = LogisticEquation(k=1,C=100,initialvalue=10)
print  "With C=100, Population at t=5  is",logeq.evaluate(t=5)
logeq.drawGraph(tfinal=10)
logeq.C=200
print  "With C=200, Population at t=5 is",logeq.evaluate(t=5)
logeq.drawGraph(tfinal=10,dt=1)
```

Note: The printed output of the test program is:

```
With C=100, Population at t=5  is 95.0049209969
With C=200, Population at t=5 is 178.677124681
```

Since the two graphs use different values for C (also called the *carry capacity* and the logistic equation approaches this value over time the final values at $t = 10$ shown on the graph should be different.