CS151 Fall 2012 Lecture 26

Stephanie R Taylor

November 5, 2012

1 Administrative Topics

- Return the quizzes
- Project grades will be sent on Wed. Bruce it traveling and that has interrupted his grading schedule.

2 Object-Oriented Design

One of the most striking aspects of the code we have been stepping through recently is number of symbol tables and the number of arrows on the board. This is indicative of a truly object-oriented design in other words a design that relies on objects with data and responsibilities. We see lots of symbol tables in the beginning because objects are created at the beginning of the code we are setting up the objects so they can do all the work.

OO design is built around certain design principles. There are four chief principles we will talk about his this class (see below). We have already covered two - modularity and encapsulation, although we haven't necessarily used those terms. Today, we are going to talk about the third - a powerful concept called inheritance. The fourth, polymorphism, is related to inheritance, and we will talk about that later in the week.

- Modularity: making functional units that can be re-used in many contexts (we see this in procedural programming as well – we should break

down the problem into parts that are logically self-contained)

- Encapsulation: information hiding – ensuring code that uses a class object does not need to know any of its implementation details.
- Inheritance: capturing commonality in a base class that can be extended to handle special cases
- Polymorphism: the ability to treat different objects the same way

3 Inheritance

We would like to avoid duplication of fields and code, if possible.

- Duplication of code creates more potential for mistakes
- Consolidating code reduces time spent programming
- Consolidated code is easier to debug

The model developed to represent this kind of relationship is called inheritance. A parent class contains all of the common code and data fields, while the child, or derived class contains the unique data or methods required for implementation.

The concept is similar to a taxonomic tree in biology. Mammals all share a set of characteristics that separate them from other creatures, but individual mammal species each have unique attributes. The concept of inheritance is extremely powerful, because it permits you to leverage code many times. Code written for a parent class is reused for each child class.

Note that inheritance is more than simply writing functions outside of a class structure that assume a particular design for the fields of a number of different classes. Inheritance, by incorporating the methods inside the parent class, follows the principle of encapsulation: only the programmer writing the parent class needs to know the particulars of implementation.

3.1 Student Class Example

Let's redesign the Student class to make it more realistic. In particular, let's design two classes – one for matriculated students (those in a degree program) and one for non-matriculated students.

Here are the lists of data fields I want for each of them.

- Student Class
  - name
  - id
  - grades
- MatriculatedStudent Class
  - name
  - id
  - grades
  - year

We will derive the MatriculatedStudent class from the Student class. This means the MatriculatedStudent class will have everything the Student class has and more.

We begin by defining the Student class:

```
class Student:
    def __init__(self, name, id):
        self.name   = name
        self.id     =   id
        self.grades = []

    # accessor methods

    def getName(self):
        return self.name

    def getId(self):
        return self.id
```

```
    # return a list of just the number parts
    # of the grade duples
    def getNumericGrades( self ):
        nums = []
        for duple in self.grades:
            nums.append( duple[1] )
        return nums

    def getGPA( self ):
        ns = self.getNumericGrades()
        if len(ns) == 0:
            print "We can't compute a GPA with no grades!"
            return None
        gpa = 0.0
        for num in ns:
            gpa += num # equivalent to gpa = gpa + num
        return gpa/len(ns)

    # return a copy of the list of grades
    def getGrades( self ):
        copy = []
        for grade in self.grades:
            copy.append( grade )
        return copy

    # mutator methods
    def setYear( self, newyear ):
        self.year = newyear

    # add the grade duple
    # (course name <str>, grade <int> or <flt>)
    # to the list of grades
    def addGrade( self, grade ):
        self.grades.append( grade )
```

and testing it

```
if __name__ == '__main__':
    s = Student( "Holly Highschool", 210291 )
    print s.getName()
    s.addGrade( ('MA121',3.7) )
    print s.getGPA()
```

Then, we define the MatriculatedStudent class:

```
1  class MatriculatedStudent(Student):
2      def __init__( self, name, id, year ):
3          Student.__init__(self, name, id )
4          self.year = year
5
6      def getYear():
7          return self.year
8
9      def setYear( self, year ):
10         self.year = year
```

Notice that in line 1, we specify that MatriculatedStudent is derived from the Student class using the parentheses.

Notice that in lines 2–4, we write an __init__ method that overrides that in the Student class. But note also that we actually call the Student.__init__ explicitly (we are using modularity and avoiding code duplication). Then, we add the year field.

Lines 6–10 simply add the accessor and mutator for the year.

And we test this code:

```
if __name__ == '__main__':
    s = MatriculatedStudent( "Fred Firstyear", 897987, 2013 )
    s.addGrade( ('CS151',2.3) )
    s.addGrade( ('BI163', 2.5) )
    s.addGrade( ('HI140', 3.3) )
    s.addGrade( ('MA121',3.1) )
    print s.getName()
    print s.getGPA()
```

The line `s.addGrade( ('CS151',2.3) )` works because the MatriculatedStudent inherited the addGrade method from Student.

Here are a few notes on terminology:

- Class Student
  - is a parent class
  - is a base class
- Class Matriculated Student

– is a child class
– is a derived class
– inherits methods from its parent class
– can override any of those methods

4 Proj 8

The are a couple of rules to follow.

1. Use drawString for all drawing.

2. Do not temporarily change the turtle state (e.g. color or pen width) in drawString. Use symbols. In the last project, you may have had 'L' mean save the color, change the color to green, draw a leaf, and then restore the color. In this project, you should instead use something like '$< aL >$' where '$<$' means save the color, 'a' means set the turtle to a specific color (e.g. 'green'), 'L' means draw the leaf, and '$>$' means restore the color. In other words, in this project, more detail and more control are placed in the L-system and its strings.

3. You may want to follow a stronger version of rule 2, i.e. Don't add any parameters to drawString. Instead, think of the Interpreter as an object that stores and sets info about the turtle's state. For example, add a field named textToDraw to the Interpreter. Then add support in drawString for a symbol 'T' to call turtle.write(self.textToDraw).