CS313E: Elements of Software Design

Turtle Graphics

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: September 18, 2012 at 12:44

Python and OO

Python is an *object-oriented* language. That implies a certain approach to thinking about problems.

Conceptualize any problem in terms of a collection of "objects"—data structures consisting of data fields and methods together with their interactions.

Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. *We'll talk about some of these later.*

Object Orientation

The basic idea of object oriented programming (OOP) is to view the world as a *collection of objects*, each of which has certain state and can perform certain actions.

Each object has:

- some *data* that it maintains characterizing its current state;
- a set of actions (*methods*) that it can perform.

A user interacts with an object by calling its methods; this is called *method invocation*. That should be the *only way* that a user interacts with an object.

OO Paradigm: An Example

Think of the world—from a problem solving perspective—as a collection of objects each maintaining some private data and offering methods (services) that you can call upon.

Example: A Coke machine has:

Data:	products inside, change available, amount previously deposited, etc.
Methods:	accept a coin, select a product, dispense a soda, provide change after purchase, return money deposited, re-stock the machine, ask how many sodas are left, log purchases, change prices, etc.

For this example do a conceptual design where you specify the data and the methods. I.e., what interface does the machine present to the world and how does accessing the interface change the data (state)?

Object Orientation

The programmer interacts with objects by invoking their methods, which may:

- update the state of the object,
- query the object about its current state,
- compute some function of the state and externally provided values,
- some combination of these.

Name potential instances of each of these for our Coke Machine example.

Another OO Example: A Simple Calculator

Imagine that you're trying to do some simple arithmetic. You need a Calculator application, programmed in an OO manner. It will have:

Some data: the current value of its *accumulator* (the value stored and displayed on the screen).

Some methods: things that you can ask it to do (add a number to the accumulator, subtract a number, multiply by a number, divide by a number, display the current accumulator value, etc.).

In Python, you implement this with a `class`.

Some Standard Methods

The “type” of an object in an OO language is called its *class*. Many common types in Python (stings, lists) are classes. You can tell because they have methods defined on them (e.g., `lst.append()` is a call to the append method on an instance of the list class.

For any class, you need to be able to create a new *instance* of that class. Python use the `__init__` method, called via the class name.

For *most* classes, it’s nice to be able to *print* a representation of the instance, though what that means is very specific to the class. In Python, use the `__str__` method.

Methods with names that start and end with two underscores are treated specially in Python, by convention. They are sometimes called *magic methods*.

Calculator Specification

Recall that earlier we wrote a program to solve the following problem:

```
Write a simple interactive calculator that takes commands from the
user.  The calculator maintains and displays an Accumulator, which is
the current stored value.  Commands can be any of the following:

     clear      ; zero the accumulator
     show       : display the accumulator value
     add k      ; add k to the accumulator
     sub k      ; subtract k from the accumulator
     mult k     ; multiply accumulator by k
     div k      ; divide accumulator by k
     help       ; show commands
     exit       ; terminate the computation

This version need only accept non-negative integer arguments, though it
can yield negative and float results.
```

A Calculator Class

Below is a Python implementation of the Calculator class:

```
class Calc:
    """This is a simple calculator class. It stores and displays
    a single number in the accumulator.  To that number, you can
    add, subtract, multiply or divide.
    """

    def __init__(self):
        "Constructor for new Calc objects, with display 0."
        self._accumulator = 0

    def getAccumulator(self):
        """Accessor for the accumulator."""
        return self._accumulator

    def __str__(self):
        """Printing displays the accumulator value."""
        return "Value is: " + str(self._accumulator)

    def clear(self):
        """Zero the accumulator."""
        self._accumulator = 0
        print(self)
```

A Calculator Class (2)

```
    def add(self, num):
        """Add num to accumuator."""
        self._accumulator += num
        print(self)
    def sub(self, num):
        """Subtract num from accumuator."""
        ...
    def mult(self, num):
        """Multiply accumuator by num."""
        ...
    def div(self, num):
        """Divide accumuator by num (unless num is 0)."""
        ...

c = Calc()                 # create a new calculator object
c.add(5)                   # add 5 to the accumulator
c.mult(3)                  # multiply by 3
c.div(2)                   # divide by 2
c.sub(1)                   # subtract 1
print(c)                   # print the result
```

What do you think is printed?

A Calculator Class (3)

We now have a Calculator class that models the calculator. You might still want to build a “driver” routine to provide the interactive interface as we did in our non-OO version of the Calculator.

What functionality is provided by the driver, rather than by the calculator instance?

A Calculator Class (3)

We now have a Calculator class that models the calculator. You might still want to build a "driver" routine to provide the interactive interface as we did in our non-OO version of the Calculator.

What functionality is provided by the driver, rather than by the calculator instance?

1. Implementing the top level loop to permit accessing the calculator interactively.
2. Reading commands from the user, parsing them, and calling the appropriate class method, if appropriate.
3. Implementing some additional commands not supplied by the class, like `help` and `exit`.

Objects as Abstract

The class definition provides a description of the object “type.” A particular object of that type is called a *class instance*.

The *only* way you should ever interact with an object (class instance) is via method invocation. We say that the class is an *abstract data type.*

This means that the class must be written so it does everything needed, but no more.

Writing vs. Using Classes

Try rewriting our previous CokeMachine example where the machine functionality is defined as a class.

For many programming problems, you will be *writing* classes to model some system.

For others, you will be *using* classes that someone else has written. It is important to understand the data (state) maintained by the class and also the methods available on the class.

Turtle Graphics

Turtle graphics was first developed as part of the children's programming language Logo in the late 1960's. It exemplifies OOP extremely well. *You will be using classes already defined for you.*

There are various versions of Turtle Graphics defined for Python. The version we're going to use is defined in Chapter 7 of your textbook. *It is not the version from Lambert's earlier book.*

Objects are turtles that move about on a screen (window). The turtle's tail can be up or down. When it is down, the turtle draws on the screen as it moves.

The Turtle's State

A turtle has a given position designated by its (x, y) coordinates, with $(0, 0)$ being in the middle of the window. The state of the turtle consists of:

Position: denoted by its current x and y coordinates

Heading: denoted by an angle in degrees. East is 0 degrees; north is 90 degrees; west is 180 degrees; south is 270 degrees.

Color: initially blue, the color can be set to 16 million colors

Width: the width of the line drawn as the turtle moves (initially 2 pixels)

Down: an attribute indicating that the turtle's tail is down.

Turtle Methods

Some of the methods are listing on p. 250 of your textbook.

t = Turtle()	create a new Turtle object and open its window
t.home()	move the turtle to $(0, 0)$, pointing east
t.down()	lower the tail
t.up()	raise the tail
t.setheading(d)	change heading to direction d
t.left(d)	turn left d degrees
t.right(d)	turn right d degrees

Turtle Methods

t.forward(n)	move in the current direction n pixels
t.goto(x, y)	move to coordinates (x, y)
t.position()	return the current position
t.heading()	return the current direction
t.isdown()	return True if the pen is down
t.pencolor(r, g, b)	change the color to the specified RGB value
t.width(width)	change the linewidth to width pixels
t.width()	return the width of t's window in pixels

Keeping it On Screen

Because the window goes away immediately after the program terminates, it may be hard to see the result unless you delay things. Use the `time` module to do that.

The following will cause the program to pause for 10 seconds. You can set the time to any amount you like.

```
import time
time.sleep(10) # wait 10 seconds
```

A Turtle Function: Draw Squares

```
from turtle import *
import math
import time

def drawSquare(turtle, x, y, length=100):
    """Draws a square with the given turtle, an
    upper-right corner point (x, y), and a side’s length"""
    turtle.up()
    turtle.goto(x, y)
    turtle.setheading(270)
    turtle.down()
    for count in range(4):
        turtle.forward(length)
        turtle.right(90)

ttl = Turtle()
ttl.pencolor(’red’)
drawSquare(ttl, -50, -50)
ttl.pencolor(’blue’)
drawSquare(ttl, 50, 50, 50)
time.sleep(20)
```

A Turtle Function: Draw Circles

```
import math
def drawCircle(turtle, x, y, radius=100):
    """Draw a circle centered on (x, y), with
    the specified radius."""
    turtle.up()
    turtle.goto(x + radius, y)
    turtle.setheading(90)
    arc = (2 * math.pi * radius) / 360
    turtle.down()
    for i in range(360):
        turtle.forward(arc)
        turtle.left(1)
    turtle.up()
    turtle.goto(x, y)
```

Using Other Functions: Draw a Donut

```
def drawDonut(turtle, x, y):
    """Draw 36 circles in a donut shape. Each
    circle has radius 45.  Figure is centered
    on (x, y)."""
    turtle.up()
    turtle.setheading(0)
    direction = turtle.heading()
    turtle.goto(x, y)
    for i in range(36):
        turtle.setheading(direction)
        turtle.forward(55)
        x1, y1 = turtle.position()
        turtle.down()
        drawCircle(turtle, x1, y1, 45)
        turtle.up()
        turtle.goto(x, y)
        direction += 10
```

Colors

What's on this slide may not quite work on all versions of turle graphics.

Colors are in the RGB system, using a triple: (R, G, B). Each element in the triple is an intensity from 0 to 255, indicating the contribution of R (red), G (green), and B (blue). For example:

black	(0,0,0)
red	(255,0, 0)
green	(0, 255, 0)
blue	(0, 0, 255)
gray	(127, 127, 127)
white	(255, 255, 255)
burnt orange	(255, 125, 25)

This is a nice website that allows you to find the RGB values for various colors: `www.colorschemer.com/online.html`.

Colors

Turtles have two “colormodes” and you’ll get an error if you try to do some things in the wrong mode. The modes are 1 and 255. In mode 255, use triples of range $0 \leq c \leq 255$. In mode 1, use percentages (range $0 \ldots 1$).

```
>>> t = Turtle()
>>> t.pencolor(127, 127, 127)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
   ....
    raise TurtleGraphicsError("bad color sequence: %s" % str(color))
turtle.TurtleGraphicsError: bad color sequence: (127, 127, 127)

>>> t.pencolor(0.5, 0.5, 0.5)
>>> t.screen.colormode(255)
>>> print (t.screen.colormode())
255
>>> t.pencolor(127, 127, 127)
>>> t.screen.colormode(1)
```

Some Sample Projects

Write Turtle graphics functions that will do the following:

1. draw a cube;
2. draw a regular polygon with k sides and radius r (distance from center to one of the vertices);
3. draw m concentric circles;
4. draw the UT Tower.