CSci 1133
Pre-read for Lab 4

Repetition

Primer on Object Methods

We've learned that object-oriented languages like Python store values as *objects*, and that all objects possess both *type* and *value*. The *type* of an object derives from its object *class name*. In other words, integer objects are all instances of a class of objects named `int`. The process of creating an object is called *instantiation*; we *instantiate* an object. Recall that a variable is created by first *instantiating* a single instance of an object *class*, and then *binding* a variable name to that object:

```
>>> month = 11
```

The digit string '11' is interpreted by Python as a "shortcut" for an explicit call to the class *constructor*. A *constructor* is a function with the same name as the object class that *constructs* (creates) a new object and returns it to the caller. The preceding line is equivalent to the following, using an explicit call to the integer class constructor:

```
>>> month = int(11)
```

We employed this earlier in order to "convert" values of one type to another. For example:

```
>>> month = int(input("Enter a month value: "))
```

"converts" a string object (the value returned by the input function) by constructing an integer object from the string value returned by the input function.

In addition to *type* and *value*, all objects also possess *methods*. A *method* is a special function that is bound to an object and implements specific *operations* that can be applied to, or on behalf of, a specific object.

Let's look at our integer example. Let's say that we would like to add one to the month value:

```
>>> nextmonth = month + 1
```

The infix ' + ' symbol is just "syntactical sugar". What is *really* happening is this:

```
>>> nextmonth = month.__add__(1)
```

Addition is an *operation* that is accomplished by calling the object's __add__ *method* with a single argument. The object *method* is a function with the special property that it acts upon the object that it is bound to. In this case, the __add__ *method* (function) adds the argument value to the value in the month object, resulting in the creation of an entirely new object.

We've already been using lots of methods. Recall various module functions:

```
>>> math.sqrt(8)
>>> random.random()
>>> turtle.forward()
```

Each of these "function calls" actually invokes the *methods* of the `math`, `random` and `turtle` objects to which they belong. We invoke (call) a *method* using the so-called "dot" operator. The "dot" operator distinguishes a *stand-alone function* call such as:

```
>>> print('howdy')
```

from a *method* call such as:

```
>>> random.uniform(1,2)
```

Turtle Methods

All of the turtle graphics code we've written so far has used a single "turtle" for drawing. Although we've used a number of "methods" to accomplish various tasks (turtle.forward, turtle.goto. turtle.left, etc.), these are thinly disguised functions that are bound to the *module* object created when we `import turtle`.

But we can actually draw with any number of "turtles" independently on the same window! In order to do so, we have to explicitly *instantiate* individual turtle objects and then invoke various operations on behalf of the turtle we wish to affect using *method* calls. We *construct* or *instantiate* a turtle object by calling the `Turtle()` constructor:

```
>>> firstturtle = turtle.Turtle()
>>> secondturtle = turtle.Turtle()
```

Note that the turtle constructor is a method of the turtle *module* object, so it requires the "dot" operator. The `Turtle()` constructor will create a new turtle object that is subsequently bound to a variable name using the assignment operator.

At this point, we can change the color and heading of the first turtle independently of the second one by applying the relevant methods to the `firstturtle` object:

```
>>> firstturtle.color('blue')
>>> firstturtle.setheading(90)
```

and we can move the second turtle independently of the first by applying the forward method to the second turtle object:

```
>>> secondturtle.forward(100)
```

Turtle objects can be passed to user defined functions as *arguments* in the same way as integer, floating-point and string objects:

```
def somefunc( someturtle ):
   someturtle.goto(0,0)
   someturtle.setheading(90)
   someturtle.forward(100)
```

Primer on Working With Colors

So far, we've changed the color of turtle objects using color *strings* (e.g., 'red', 'blue', 'yellow', etc.). You can also specify specific colors using the "RGB" specification. Because our eyes contain retinal cells that perceive three different spectral bands roughly corresponding to red, blue and green, combining varying intensities of red, green and blue light can create most perceivable colors.

The RGB specification consists of three floating-point values, each in the range from 0.0 to 1.0, indicating the intensity (as a fraction or percent) of each color. For instance, (1.0, 0.0, 0.0) produces pure red, (0.0, 1.0, 0.0) green, and (0.0, 0.0, 1.0) blue Colors can be combined to produce other "additive" colors: (1.0, 1.0, 0.0) will add red + green, producing (surprisingly!) yellow. Changing intensity will change the color value. For example, (0.5, 0.5, 0.0) will produce a greenish-brown.

There are a number of color charts on the www that will show you how to obtain any color using an RGB description.

The `begin_fill and end_fill` turtle methods provide the ability to "color" any region of the window. Using them is quite simple:

```
>>> t1 = turtle.Turtle()
>>> t1.fillcolor(1,0,.5)
>>> t1.penup()
>>> t1.begin_fill()
>>> t1.goto(100,0)
>>> t1.goto(100,100)
>>> t1.end_fill()
```

The `fillcolor` method sets the fill color (bright "fuscia" in this case). The `begin_fill` method indicates the start of a "fill region". Every movement of the turtle following this command will define the boundary of an area that will be subsequently filled with the *fillcolor*. After the region has been drawn, the `end_fill` method will terminate the fill definition and cause the enclosed region to be colored.