lecture_4

March 7, 2017

1 Beginning Python Class -- Lecture 4

1.1 Expected Understanding

At this point in the class it is very important that you have a good understanding of the topics in previous lectures. The topics today will build directly on the previous topics such as:

```
- Basic Data Types
- Conditionals
- Loops
- Functions
```

Today's lecture we touch on User Defined Datatypes. This an advanced topic that usually encompasses multiple college Computer Science courses. Usually this leads to discussions of Object Oriented Programming (OOP) which we are going to try and avoid for the purposes of this class. I only want to describe objects with member attributed and methods not the use of OOP design patterns.

1.1.1 Running Python in the Lab

Not everyone has a dedicated Windows machine that they can remote into, and not everyone has their own Linux account. Therefore we are going to use an online Python Interpreter to run our Python code for this class. This way we can practice in the lab and students can practice at home without having to install python on their PC. I will run python from my machine during the lecture so that the process of running python from the command line is understood.

Here is the link to the online Python Interpreter -- https://repl.it/languages/python3

1.1.2 PEP8 Style Guide

PEP8 style guide should be used as a reference for styling decisions. This document (http://legacy.python.org/dev/peps/pep-0008/) covers topics like: - Indentation - Comments - Line Length These are a set of standards that allow python developers to easily work on each others code and avoid nasty syntax bugs caused by white space inconsistencies.

1.1.3 Syntax

Syntax in the context of computer code is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language.

Python was designed to have a very clean look to the code. The compromise is that to correctly interpret the code python elected to use white space as a syntactical character. The last language to use syntactical white space was Fortran which was created by IBM in 1957.

Languages such as Perl and C ignore white space and rely on other special characters that often mean different things in different contexts. This allowed the programmer to develop their own style independent of the format of the code.

A quick example is shown below:

```
foreach my $a (@list_of_numbers){
    print "value of a: $a\n";
}
```

The exact same Perl code could be written like:

```
foreach my $a (@list_of_numbers){ print "value of a: $a\n"; }
```

This is because one of the characters that Perl uses to group instructions are "{}" (brackets).

Python avoids this type of problem by requiring white space syntax to group sequences of instruction:

```
for a in list_of_numbers:
    print( "Value of a: " + str(a))
```

Since the 'print' statement is executed during the loop it is nested/grouped inside the loop with 4 spaces. (More on loops later in Lecture 2)

1.2 User Defined Datatypes

Lecture 1 of this class covered Basic/Built-in Datatypes such as `list` and `string`. Python allows the programmer to define their own datatypes. User defined datatypes as well as built-in datatypes, excluding `int`, `float`, and `bool`, are called classes/objects. The general form for defining a class is:

```
class CLASS_NAME:
    ATTRIBUTE = SOME_VALUE
    def __init__():
        pass
    def OTHER_MEMBER_FUNCTION():
        pass
```

For example we could define an shape datatype if we wanted to model some sort of geometry program. We will use some basic examples using squares and rectangles.

```
In [1]: import math
        class shape:
            pass

        class rectangle(shape):
            name = "Rectangle"
```

```
        def __init__(self, length, width):
            self.length = length
            self.width = width
        def get_perimeter(self):
            self.perimeter = self.length*2 + self.width*2
            return self.perimeter
        def get_area(self):
            self.area = self.length * self.width
            return self.area

    class square(rectangle):
        name = "Square"
        def __init__(self, length):
            self.length = length
            self.width = length

    x = rectangle(10,5)
    print( x.name, x.get_perimeter(), x.get_area() )

    y = square(7)
    print( y.name, y.get_perimeter(), y.get_area() )
```

```
Rectangle 30 50
Square 28 49
```

In the example above we created three classes, `shape`, `rectangle`, and `square`. I defined these classes to show basic functionality of member attributes and member functions. Member functions and attributes are accessed using the '.' operator. To access the `name` attribute of the object in `x` we do so like `x.name`. We will also talk briefly about *inheritance*.

`self.name`, `self.length`, and `self.width` are all member attributes. Member attributes can be thought of as data that pertains to the class itself. `self` is a special way to tell Python that we are talking about the data that is part of the particular object.

`get_perimeter` and `get_area` are examples of member functions. Member functions are functions that are defined for a particular object. In our case we know that each of our shapes will have a perimeter and an area associated with them. To acess a member function we use the '.' operator but also put the '()' at the end because it is a function.

The last item that I would like to bring to your intention from the code above is that a `rectangle` is a `shape` and *inherits* from the `shape` class. In our case I didn't define anything for the `shape` class but I could. Next since every `square` is a `rectangle` we *inherit* the `square` class from the `rectangle`. We redefine the `__init__` function for the `square` class because we only need the length of a single side for a `square`. The `get_perimeter` and `get_area` functions are still available for use in the `square` class because they are *inherited* from the `rectangle` class.

1.2.1 Terminology

We are going to continue talking about our shapes and rectangles to introduce some terminology that you will find when learning about class/objects.

Sub-class/Child

- `rectangle` is a *sub-class* of `shape`.
- `square` is a *sub-class* of `rectangle`.
- `rectangle` is a *child* of `shape`.
- `square` is a *child* of `rectangle`.

Super-class/Parent

- `shape` is a *super-class* of `rectangle`.
- `rectangle` is a *super-class* of `square`.
- `shape` is a *parent* of `rectangle`.
- `rectangle` is a *parent* of `square`.

Inherit

- `square` *inherits* from `rectangle` which in turn *inherits* from `shape`.

1.2.2 One More Shape

Let's add one more type of `shape` to our collection of classes.

```
In [2]: class triangle(shape):
            name = "Triangle"
            def __init__(self, s1, s2, s3):
                self.side1 = s1
                self.side2 = s2
                self.side3 = s3
                self.get_perimeter()
                self.get_area()
            def get_perimeter(self):
                self.perimeter = self.side1 + self.side2 + self.side3
                return self.perimeter
            def get_area(self):
                self.semiperimeter = self.perimeter / 2
                s = self.semiperimeter
                a = self.side1
                b = self.side2
                c = self.side3
                # Heron's Formula
                self.area = 0.25 * math.sqrt( 4*pow(a,2)*pow(b,2) - pow( (pow(a,2) + pow(b,2) -

        z = triangle( 3,4,5 )
        print( z.name, z.perimeter, z.area )
```

```
Triangle 12 6.0
```

In the example above we created another *subclass* of `shape` called `triangle`. `triangle` has been set up to give us the same information about perimeter and area as we setup for the `rectangle` class.

NOTE: I didn't do any error checking to make sure that the triangle was actually a valid triangle. I leave it to the user to pass real valid line lengths to the object otherwise it will crash Python.

2 Motivation for this Lecture

Inheritance and *Classes* can quickly become quite complex but they allow the programmer a flexible way of modeling a wide variety of problems. The reason that I wanted to cover a brief overview of classes/objects is because built-in datatypes like `list` and `string` are objects. Both `list` and `string` have their own member functions that can be used when working on these built-in datatypes. See lecture_1 for more information of the available members for other built-in datatypes.

2.1 String

Let me go over a quick explanation of the available methods and attributes for strings. Strings have the following attributes and methods associated with them.

```
In [3]: x = str
        [method for method in dir(x) if callable(getattr(x, method))
                and not method.startswith( '__' )]

Out[3]: ['capitalize',
         'casefold',
         'center',
         'count',
         'encode',
         'endswith',
         'expandtabs',
         'find',
         'format',
         'format_map',
         'index',
         'isalnum',
         'isalpha',
         'isdecimal',
         'isdigit',
         'isidentifier',
         'islower',
         'isnumeric',
         'isprintable',
         'isspace',
```

```
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

We can take a look at some of these to see how they work. More information can be found about these methods at the following link https://docs.python.org/3.5/library/stdtypes.html#string-methods.

```
In [4]: first = "dallin"
        last = "marshall"

        print( first.capitalize() )
        print( last.upper() )
        print( last.title().swapcase() )

        print( '|' + first.center( 20 ) + '|' )
        print( last.isupper() )
        print( first.count('a'), last.count('l') )
```

```
Dallin
MARSHALL
mARSHALL
|       dallin       |
False
1 2
```

2.2 Practice Problem 1

In a while loop continue asking the user for strings until the user enters an empty string. Each time they give a string add it to the end of a list of strings and print out the list. Make use of the `append` method of the `list` object.

2.3 Practice Problem 2

Create a `class` for one more shape that describes a circle. Allow the user access to the same functions of `get_perimeter` and `get_area`.

The perimeter of a circle is `2 * pi * r`, where r is the radius.

The area of a circle is `pi * r * r`.

By using the math library we can easily get the value for pi.

```
In [5]: import math
        print( math.pi )
```

```
3.141592653589793
```

2.4 Challenge Problem

I would like you to create a class that could be used to help us model a card game. Create a `class` card that has the attributes: `suit`, `symbol`, and `value`. `value` being the numerical value of the card in Black Jack. `symbol` would be the character in the corner of the card. 'K' would be the symbol for a card representing a King. Create a `class` deck that contains all 52 cards of a standard deck.