**Week 6 – Getting Started with Object-Oriented Programming in Python**

**Learning Outcomes**

By the end this lesson, you will:

* Identify characteristics of Python that distinguish it from other languages.
* Work with Python lists.
* Write non-object-oriented programs that use sequence, selection, and repetition to accomplish tasks.
* Write Python functions and place those functions in separate libraries.
* Define a Python class, complete with data members, functions / methods, and an initializer (constructor).
* Use Python's built-in text file objects to create and read text files.
* Use inheritance to create a hierarchy of classes that are related to each other.
* Create objects of classes and use them to carry out the work of your program.
* Deal with a list of related objects polymorphically.

You will demonstrate attainment of these skills by writing a series of Python programs that incorporate these elements.

**Introduction to Python**

Python is an ***interpreted language***. What does that mean?

One instruction at a time is converted to machine language. Compare that with a compiled language (like C++) where the entire program is converted at once. As an interpreted language, Python will be slower than a compiled one like C++.

What is an advantage of an interpreted language? What is a disadvantage?

The disadvantage is the performance penalty that you take. However, when it comes finding errors, an interpreted language has a distinct advantage. The option to “play around with the code” gives interpreted languages an easy path to developing a solution.

Indentation matters!

* each block is indented by one additional level

Python is a ***loosely typed language***. What does that mean?

You don’t declare the types of your data in a formal way.

Data types: int, float, string, list, dictionary

You declare a variable simply by naming it and assigning it a value.

age = 23

radius = 3.2

team\_name = “White Sox”

car\_model = ‘Camaro’

**Strings – a clear example of encapsulation**

Any string is an instance of a class called String. As an object of a class, it has data and methods. The data are the individual characters. And there are lots of functions.

To declare a string object:

phrase = “Happy Birthday”

phrase is now a string object. If you type “phrase” followed by a dot, you’ll see a long list of things phrase can do. These are all the functions built into the string class.

You can write out the values of variables and combine them with strings using a comma. The comma joins together strings but embeds a space.

print("Your age in months is", age\_in\_months, ".")

But the string class also supports the use of the % operator. The % operator enables us to use formatting sequences. Formatting sequences help us embed values with the string.

%5d an integer that will take up 5 spaces

%10s a string in a field of 10 spaces

%6.2f a floating point number that takes up 6 spaces with 2 after the decimal

>>> print("Your age in months is %d." % age\_in\_months)

Your age in months is 168.

>>> print("If your age in years is %d, your age in months is %d." % (age,age\_in\_months))

If your age in years is 14, your age in months is 168.

Strings also support the \* operator. The \* operator helps us build up a repetitive string.

“-“ \* 50 is a row of 50 dashes.

**Lists (arrays) are collections of data. They are also objects of a class.**

name\_of\_list = []

numbers = [3, 4, 5]

numbers.append(6)

numbers.extend([7])

In adding values to a list, there are a couple of options: append and extend

* append will add a list of items as a list. append can also take a singl value rather than a list.
* extend will add a list of items as the separate components of the list. extend can only take in a list, not a single value

>>> print(friends)

['Larry', 'Curly', 'Moe', 'Snuffleupagus', 72]

>>> friends.append(["Frank","Andrew","Ted"])

>>> print(friends)

['Larry', 'Curly', 'Moe', 'Snuffleupagus', 72, ['Frank', 'Andrew', 'Ted']]

>>> friends.extend(["Joe","Felix","Mary"])

>>> print(friends)

['Larry', 'Curly', 'Moe', 'Snuffleupagus', 72, ['Frank', 'Andrew', 'Ted'], 'Joe', 'Felix', 'Mary']

>>>

If all you want to do is add one value, use append instead of extend.

**Input**

You can input data from the user using the *input* function. input always inputs values as a string. If you need to work with a value entered by the user as another datatype, typecast it using int(), float(), str()

value = input(“prompt user for what to input”)

**Types of blocks**: if

**Types of blocks**: for

**Example: Paycheck**

Write a program that will enable a user to calculate gross and net pay for hours and payrates entered by the user.

We wrote two versions in class: payv1.py and payv2.py. The second one uses functions to calculate the gross pay and the net pay and to print a heading. You declare a function using the def keyword, followed by the name of the function, a list of parameters in parentheses, and a colon.

Here’s version 1:

# Ray Klump CS 245

# This program inputs hourly pay and hours worked and

# then computes the amount the person should be paid.

name = input("Enter your name: ")

pay\_rate = float(input("What is your hourly pay rate? "))

hours\_worked = float(input("How many hours did you work? "))

gross = pay\_rate \* hours\_worked

TAX\_RATE = 0.08

net = (1-TAX\_RATE) \* gross

print("Your gross pay was $%.2f." % gross)

print("Your net pay was $%.2f." % net)

**Functions**

A function is a named block of code.

It takes in inputs (parameters) and may produce outputs.

You begin a function with the *def* keyword.

def name\_of\_function(param1, param2, param3):

statement1

statement2

return return\_value

Call the function like this:

value = name\_of\_function(val1, val2, val3)

Here’s version 2:

# Ray Klump CS 245

# This program inputs hourly pay and hours worked and

# then computes the amount the person should be paid.

def print\_heading():

print("\*-" \* 20)

def calc\_gross(hours, rate):

return hours \* rate

def calc\_net(gross, tax\_rate):

return (1-tax\_rate) \* gross

name = input("Enter your name: ")

pay\_rate = float(input("What is your hourly pay rate? "))

hours\_worked = float(input("How many hours did you work? "))

gross = calc\_gross(hours\_worked, pay\_rate)

TAX\_RATE = 0.08

net = calc\_net(gross, TAX\_RATE)

print\_heading()

print("Your gross pay was $%.2f." % gross)

print("Your net pay was $%.2f." % net)

print\_heading()

**Example**: Write a program that would enable someone to print a list of paychecks. Keep entering employee data until the person indicates they are done. Categorize each applicant as executive, middle manager, or plebe based on pay.

This introduces us to **if statements**. if introduces a question. elif can add other questions to the same conversation. else gives us a default action.

**Here is the source code:**

# Ray Klump CS 245

# This program will compute and print a list of employee paychecks

# Version 1: compute the pay for one employee

# things we need:

# 1. hours worked --> hours

# 2. pay rate --> rate

# 3. name --> name

tax\_rate = 0.08

name = input("Enter the employee's name: ")

rate = float(input("Enter the hourly pay: "))

hours = float(input("Enter the hours worked: "))

salary = rate \* hours

salary = (1-tax\_rate)\*salary

if salary > 2500:

category = "phat cat"

elif salary > 1000:

category = "middle class"

else:

category = "plebe"

print("%s made $%.2f. That places him in the %s category." % (name, salary, category))

Let’s do a different version now – one that keeps processing employees until we don’t want to process anymore. The user will indicate they are done by entering an “n” (or anything other than a “y”).

This allows us to learn about **while loops**.

# Ray Klump CS 245

# This program will compute and print a list of employee paychecks

# Version 1: compute the pay for one employee

# things we need:

# 1. hours worked --> hours

# 2. pay rate --> rate

# 3. name --> name

tax\_rate = 0.08

do\_again = "y"

while do\_again == "y":

name = input("Enter the employee's name: \n")

rate = float(input("Enter the hourly pay: \n"))

hours = float(input("Enter the hours worked: \n"))

salary = rate \* hours

salary = (1-tax\_rate)\*salary

if salary > 2500:

category = "phat cat"

elif salary > 1000:

category = "middle class"

else:

category = "plebe"

print("%s made $%.2f. \nThat places him in the %s category." \

% (name, salary, category))

do\_again = input("Do you have another employee to enter? (y or n) ")

do\_again = do\_again.lower()

print "Have a nice life."

This was a while loop. If instead we know how many paychecks we are going to process, we can use a counter-controlled loop. A for loop implements a **for loop**.

Here’s another version with a for loop:

# Ray Klump CS 245

# This program will compute and print a list of employee paychecks

# Version 1: compute the pay for one employee

# things we need:

# 1. hours worked --> hours

# 2. pay rate --> rate

# 3. name --> name

tax\_rate = 0.08

emp\_count = int(input("How many paychecks will you process? "))

for i in range(emp\_count):

name = input("Enter the employee's name: \n")

rate = float(input("Enter the hourly pay: \n"))

hours = float(input("Enter the hours worked: \n"))

salary = rate \* hours

salary = (1-tax\_rate)\*salary

if salary > 2500:

category = "phat cat"

elif salary > 1000:

category = "middle class"

else:

category = "plebe"

print("%s made $%.2f. \nThat places him in the %s category." \

% (name, salary, category))

print "Have a nice life."

Each Python source file can consist of multiple functions. To define a function, use def.

* Functions can return values and accept parameters.

**Example**: Convert the program you just wrote to use a function determine the category for each employee and return it and to determine taxes.

We’ll show the source code after we discuss reading data from a file.

**Reading from a file:**

You can read from a file by opening it first and then using readline() to read a line at a time. Or you can read all the lines at once using readlines(). Or you can use the following syntax:

for line in file\_variable:

process(line)

First, of course, you need to open the file.

file\_var = open(“name\_of\_file”, “r”)

**Example**: Read salary data from a file now instead of from the command line. In the process we’ll learn how to read a text file from beginning to end and how to split a line of text into component pieces using the split() function.

Here’s the code we wrote:

# Ray Klump 2015-01-22

# New Paycheck Example

# Process a set of employees

# Ver 1 - entered by the user at the keyboard

# Ver 2 - data from text file

def calc\_gross\_pay(rate, hours):

return rate \* hours

def determine\_tax\_rate(pay\_rate):

if pay\_rate > 30:

tax\_rate = 0.25

elif pay\_rate > 20:

tax\_rate = 0.20

else:

tax\_rate = 0.15

return tax\_rate

def calc\_net\_pay(gross\_pay, rate):

tax\_rate = determine\_tax\_rate(rate)

return (1-tax\_rate) \* gross\_pay

def print\_border():

print("\*" \* 30)

def print\_pay\_check(name,gross,net):

print\_border()

print("%-10s %-10s %-10s" % ("Name","Gross","Net"))

print("%-10s $%-10.2f $%-10.2f" % (name, gross, net))

print\_border()

#main

file\_name = input("Enter the full path for the file: ")

file\_var = open(file\_name,"r")

for emp in file\_var:

emp = emp.strip() # eliminates leading and ending spaces

fields = emp.split(" ")

name = fields[0]

pay\_rate = float(fields[1])

hours\_worked = float(fields[2])

gross\_pay = calc\_gross\_pay(pay\_rate, hours\_worked)

net\_pay = calc\_net\_pay(gross\_pay,pay\_rate)

print\_pay\_check(name,gross\_pay,net\_pay)

file\_var.close()

print("Thank you for using our program.")

**Object-oriented programming in Python**

As we learned in week one, the centerpiece of programming in an object-oriented way is to create classes. We are now going to learn how to create a class in Python.

Remember that a class is a data type that has both data and functions / methods built into it.

To create a class in Python, use the keyword *class*:

class NameOfClass:

After that line, indent all the lines that make up the definition of the class. Include in that class definitions of functions. One of those functions is a special one called the *constructor*. The constructor has the special name \_\_init\_\_, and its job is to initialize the variables that make up the class’s data.

Every function, including the constructor, takes, as its first parameter, a parameter called *self*. The parameter self refers to the instance variable (i.e. object) of that type that is currently being asked to do something. Remember that you build objects by defining them as variables of classes. Well, self points to the object that is currently being asked to do something.

To create an object of a class, write a line like this:

name\_of\_object = Name\_Of\_Class(list, of, params, needed, to, initialize, it)

This links with the \_\_init\_\_ function (i.e. the constructor) for the class.

Note that you do not include the self parameter when you call a constructor or function. Even though self appears as the first parameter in each constructor and function declaration, it is kind of an invisible parameter, because you pretend it isn’t there when you call the constructor or function.

**A first object-oriented example**

In this example, we’ll create a class called Circle. A Circle has a radius. Circles also have an area and a circumference, which we can calculate through member functions. We’ll also add a to\_string function to Circle so that it can represent itself as a string.

Once we have the class definition, we can create an object of the class and use that object to call the various member functions of the Circle class.

Here is the code:

# Ray Klump

# An example of declaring a class and then using objects of that class

# Specifically, we are going to build an example that models a Circle

import math

import random

class Circle:

def \_\_init\_\_(self, rad):

self.radius = rad #self. identifies radius as a member variable

#of the class. Don't forget it!!!!

def calc\_area(self):

return math.pi \* math.pow(self.radius,2)

def calc\_circ(self):

return 2 \* math.pi \* self.radius

def to\_string(self):

return "c %.3f" % self.radius

# main

radius = random.randint(1,25)

circ = Circle(radius) # \_\_init\_\_(self,rad) is called, with

# circ being fed in for self automatically,

# and radius passed in for rad.

# At finish, circ will be an initialized circle

# object with the specified radius

area = circ.calc\_area()

circumference = circ.calc\_circ()

print("The description of this circle is\n\t%s" % circ.to\_string())

print("Its area is %.3f, and its circumference is %.3f." % (area, circumference))

There are distinct advantages to splitting the declaration of a class from its usage. This has benefits in terms of clarity and the opportunity to reuse the code in other projects.

In this example, we create a file called shapes.py. That file defines the Circle class. We’ll also add the definition of the Rectangle class to it:

# Ray Klump

# This library introduces a bunch of shapes

import math

class Circle:

def \_\_init\_\_(self, rad):

self.radius = rad #self. identifies radius as a member variable

#of the class. Don't forget it!!!!

def calc\_area(self):

return math.pi \* math.pow(self.radius,2)

def calc\_circ(self):

return 2 \* math.pi \* self.radius

def to\_string(self):

return "c %.3f" % self.radius

class Rectangle:

def \_\_init\_\_(self, width, length):

self.width = width

self.length = length

def calc\_area(self):

return self.width \* self.length

def calc\_perim(self):

return 2 \* (self.width + self.length)

def to\_string(self):

return "r %.3f %.3f" % (self.width, self.length)

And here is the code that imports shapes so that it can access Circle and Rectangle:

# Ray Klump

# An example of declaring a class and then using objects of that class

# Specifically, we are going to build an example that models a Circle

import random

#from shapes import Circle

import shapes

# main

radius = random.randint(1,25)

circ = shapes.Circle(radius) # \_\_init\_\_(self,rad) is called, with

# circ being fed in for self automatically,

# and radius passed in for rad.

# At finish, circ will be an initialized circle

# object with the specified radius

area = circ.calc\_area()

circumference = circ.calc\_circ()

print("The description of this circle is\n\t%s" % circ.to\_string())

print("Its area is %.3f, and its circumference is %.3f." % (area, circumference))

length = random.randint(1,25)

width = random.randint(1,25)

rec = shapes.Rectangle(length, width)

rec\_area = rec.calc\_area()

rec\_perim = rec.calc\_perim()

print("The description of this rectangle is\n\t%s" % rec.to\_string())

print("Its area is %.3f, and its perimeter is %.3f." % (rec\_area, rec\_perim))

**Obviously, Circle and Rectangle are related, right?**

So, it would be good if we could treat circles and rectangles as the related kinds of things they are. Thanks to the concepts of abstraction, inheritance and polymorphism, which we learned about in the first week, you can.

First, let’s declare the mother of all classes for this example – shape. Shape is really generic, so generic, in fact, that we can’t define the guts of some of the things it can do. Function definitions that simply declare the existence of functionality without actually specifying how that functionality should be done are called *abstract*. Classes that have one or more abstract functions are called *abstract classes*. In this Shape class, we spell out two abstract functions that all shapes must support: calc\_area and calc\_perim. We’ll also equip it with a get\_shape\_type function that will return “s” but will be redefined in descendant classes, as well as a to\_string function that will also be redefined in descendant classes.

import math

class Shape:

def \_\_init\_\_(self,x=0,y=0):

self.x = x

self.y = y

def calc\_area():

pass

def calc\_perim():

pass

def get\_shape\_type(self):

return "s"

def to\_string(self):

return "%s %f %f" % (self.get\_shape\_type(),self.x,self.y)

class Circle(Shape):

def \_\_init\_\_(self,x=0,y=0,rad=0):

super().\_\_init\_\_(x,y)

self.rad = rad

def calc\_area(self):

return self.rad \* self.rad \* math.pi

def calc\_perim(self):

return 2 \* self.rad \* math.pi

def get\_shape\_type(self):

return "c"

def to\_string(self):

return "%s %f %f %f" % (super().to\_string(), self.rad, \

self.calc\_area(), self.calc\_perim())

class Rectangle(Shape):

def \_\_init\_\_(self,x=0,y=0,width=0,length=0):

super().\_\_init\_\_(x,y)

self.width = width

self.length = length

def calc\_area(self):

return self.length \* self.width

def calc\_perim(self):

return 2 \* self.length + 2 \* self.width

def get\_shape\_type(self):

return "r"

def to\_string(self):

return "%s %f %f %f %f" % (super().to\_string(), self.width, self.length, \

self.calc\_area(),self.calc\_perim())

r1 = Rectangle(10,10,15,20)

c1 = Circle(20,5,15)

print(r1.to\_string())

print(c1.to\_string())

Notice the use of the *pass* instruction. “pass” simply means “nothing to see here. Move on.”

To declare that a class descends from another class, use the following syntax:

class descendant(parent):

The technical term for *parent* is *superclass*. The technical term for *descendant* is *subclass*. In other words, subclasses descend from superclasses. Superclasses are the parents; subclasses are the children.

We defined the Circle and Rectangle classes as descendants of Shape. Both Circle and Rectangle will fill in the definitions for the functions we declared the definition for in Shape but didn’t spell out.

Notice how we make use of the super() keyword to access superclass versions of the get\_shape\_type and to\_string functions. We also make use of them in the descendant classes’ \_\_init\_\_ functions. This gives us access to how the super class did something.

In them main part of the code, we created two shape objects and had them do their thing.

This example illustrated encapsulation and inheritance. Let’s now illustrate polymorphism.

**Polymorphism**

Now here’s where the power of polymorphism come to bear. Here’s a main program that creates a Circle and a Shape, stores them in a list of shapes, and then marches through the list of shapes to display what has been created.

shapes = []

r1 = Rectangle(10,10,15,20)

c1 = Circle(20,5,15)

shapes.append(r1)

shapes.append(c1)

for shape in shapes:

print(shape.to\_string())

What’s amazing (and cool) about this is that each of the objects are stored in the same list, but we can invoke their specific ways of calculating area, perimeter, and the string representation without having to do anything special. This is an example of polymorphism.

**New Homework Assignment**

You are to create a tool that computes best running or cycling times based on gps data. We’ll spend a few minutes now going over some of the more challenging aspects of the project. These include the following:

* computing the distance traveled each interval
* computing the pace for each interval
* determining the fastest time
* converting the fastest time to minutes and seconds

*Classes you’ll need to create*

Create a Report\_Printer class that has at least a print\_result function. The print\_result function should print an individual line of the table of results. To do that, Report\_Printer will have to take in the time string, the latitude, the longitude, the distance, and the pace. In main, you will need to create a Report\_Printer object, and then you’ll have to tell the Report\_Printer object to print\_result for each row of the table you need to write. If you pass negative values or 0 for the distance and pace, the print\_result function should take that as a cue that this must be the first interval and that, therefore, it should print \*’s for the pace and distance for that row. You might also want to include a print\_heading function and print\_separator function to print the heading and the line that separates the heading from the data.

Create a GPS\_Calculuator class. The GPS\_Calculator class will have a member function called calc\_distance that will implement the formula found at http://andrew.hedges.name/experiments/haversine/. It will need to take in the latitude and longitude of two points. It will need to convert those latitude and longitude values to radians using math.radians before you do any calculations. It will return the distance computed from the inputted latitudes and longitudes. You might also want to include in GPS\_Calculator a function that will take the time string (such as “11:03:56”) and return the seconds since midnight that that time string corresponds to. That will enable you to determine the difference in seconds between two points in time easily, so that you can compute the running pace (time / distance) for each interval. Note that, to compute the running pace in minutes / mile given the distance in miles and the time in seconds, you’re going to have to divide by 60 to convert the seconds to minutes.

*The main part calls the shots*

In the main part,

* create the Report\_Printer and GPS\_Calculator objects
* ask the user for the name of the file (note that you will have to create the file by copying the data from the homework sheet into a new text file you create. Store that file in a convenient location, which likely is just the same folder where you have saved your program’s source code.)
* open the file the user identified
* tell the Report\_Printer object to print the heading and separator.
* Initialize a variable that keeps track of the best pace (perhaps called best\_pace) to an impossibly slow value, like 1000, so that you can update as you find faster paces.
* Initialize a Boolean variable (perhaps called *first*) to True to indicate that you’re about to read the very first line.
* For every line in the file:
  + Split the line into parts
  + Convert the time string that is the first part of the line into seconds, perhaps using a function of GPS\_Calculator as I discussed above.
  + Parse out and convert the next two parts of the line into floats that represent the lat and lon
  + If this is the first line you’ve read, set variables time1, lat1 and lon1, which store your reference against which you will measure running progress for the next line your input, to the values of lat, lon, and the time in seconds that you have read from the current line, and then tell the Report\_Printer object to print\_result, passing it whatever information is necessary to inform print\_result that this is the very first line and that, therefore, it should print asterisks for the pace and distance. Then set first to False so that subsequent lines aren’t treated as the first line.
  + Otherwise, if this is not the first line, set lat2, lon2, and time2 equal to the values that were read from this line, ask GPS\_Calculator to calc\_distance from lat1, lon1, time1, lat2, lon2, and time2, and then report that distance that calc\_distance and the corresponding pace to the screen, using the Report\_Printer’s print\_result function. Also, update the best\_pace if the new pace is quicker than the previous one. Finally, make sure that you set lat1, lon1, and time1 equal to lat2, lon2, and time2 so that you are prepared to use the as the basis of comparison for the next pace and distance calculations.
* After all the lines have been processed, print the best pace in minutes and seconds. You’ve kept track of the best pace in the variable best\_pace, so this part should be straightforward, except you’ll have to determine the formula for separating the minutes from the seconds so that you can print them as shown on the homework sheet.