**Week 7 – Model-View-Controller in Python**

**Learning Outcomes**

By the end this lesson, you will:

* Design and implement a Model-View-Controller application in Python.
* Perform basic drawing operations using a popular Python drawing package.
* Implement input and output with plain text files using a controller class designed for that purpose.
* Separate code among various libraries that you can reuse in other applications by importing them.
* Implement information hiding in Python using both accessor and mutator functions and properties.
* Download documents from remote servers from your application.
* Parse data expressed in XML format.
* Separate an application’s functionality among classes having well-defined function sets and purposes.

You will demonstrate attainment of these skills by writing a series of Python programs that incorporate these elements. We will develop two applications from scratch: one that draws a variety of related shapes, and another that prints pay check information for employees communicated through an xml file.

**Now let’s add drawing to the list of things we can do in Python**

To draw the shapes, we need to place them at particular x and y coordinates. So, we need to store x and y coordinates with each Shape. Since *all* shapes need x and y coordinates, place these in the Shape class, and initialize them in the constructors. Plus, add some draw functions to draw the shapes.

To do the actual drawing, we’ll use the Turtle class from Python’s turtle library.

Here’s an overview of Turtle Graphics – it’s kinda fun.

|  |  |
| --- | --- |
| home() | moves the pen to the center |
| goto(x,y) | goes to a particular (x,y) point on the screen |
| setpos(x,y) | does the same thing as goto |
| setx(x) | sets the x coordinate |
| sety(y) | sets the y position |
| position() | returns the position – where is it? |
| heading() | the orientation of the turtle – which direction is it pointing |
| circle(r) | draws a circle of the given radius |
| pencolor(r,g,b) | sets the color of the turtle to the combination of red, green, and blue. The values should range from 0 to 1.0 |
| forward(d) | moves d units ahead |
| right(a) | turns right by a particular angle in degrees |
| left(a) | turns left by a particular angle in degrees |
| setheading(a) | sets the angle of the turtle – sets its new direction |
| pensize(s) | sets the width of the pen |
| fillcolor(r,g,b) | sets the fill color to a particular color that is a combination of red, green, and blue that are each in the range of 0 to 1.0. |
| fill(True) | sets the behavior so that the next shape is drawn filled |
| fill(False) | sets the behavior so that the next shape is drawn unfilled |
| clear() | clears the drawing |
| write(“string”,False,  align=”left”,  font=(“Arial”,8,”normal”) | writes text at a particular location with the given font and alignment. |

import math

import turtle

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 get\_draw\_params(self):

return [self.x,self.y]

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 get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend([self.rad])

return result

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 get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend([self.width,self.length])

return result

def to\_string(self):

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

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

class Turtle\_Draw\_Shape\_Controller:

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

self.turtle = turtle.Turtle()

def draw\_circle(self,rad):

self.turtle.circle(rad)

def draw\_rect(self,width,length):

self.turtle.seth(0)

self.turtle.forward(width)

self.turtle.right(90)

self.turtle.forward(length)

self.turtle.right(90)

self.turtle.forward(width)

self.turtle.right(90)

self.turtle.forward(length)

self.turtle.right(90)

def draw\_shapes(self,shapes):

for shape in shapes:

shape\_type = shape.get\_shape\_type()

params = shape.get\_draw\_params()

self.turtle.penup()

self.turtle.goto(int(params[0]),int(params[1]))

self.turtle.pendown()

if shape\_type == "c":

self.draw\_circle(int(params[2]))

elif shape\_type == "r":

self.draw\_rect(int(params[2]), int(params[3]))

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())

drawer = Turtle\_Draw\_Shape\_Controller()

drawer.draw\_shapes(shapes)

This is an MVC-compliant drawing application. The Model are the shapes. The View is the Turtle class, which we didn’t have to write. The controller is the Turtle\_Draw\_Shape\_Controller, which coordinates the activity of the model and the view. The shapes don’t have to know anything about drawing, and the turtle doesn’t have to know anything about the shapes. It is a very extensible way to do the drawing. Drawing using a different technology (instead of Turtle) will require writing a different Controller, but it won’t require changing the model at all. That is great.

One of the new additions in this code is the get\_draw\_params function. It returns a list of the drawing parameters for each class. It returns them as floats. Since the Turtle class needs ints to draw shapes, our controller has to convert them to ints before calling turtle functions. That’s kind of a pain, but data conversions are a common task for controllers, since they try to isolate intracies of the model from the programmer, the user, and the view.

**Hiding Data**

One of the concepts of object-oriented programming we discussed in Week 1 was the idea of information hiding – the notion that data should be private and accessed through only controlled pathways. Unlike other languages, Python doesn’t force us to make data accessible or inaccessible. But we can obey such guidelines through programmer’s convention.

The generally agreed-upon convention is the following: data is private and should be accessed only through functions.

What would this mean for the Circle class? The Circle class would have an x, y, and radius, but they would be private. If we wanted to learn about them, we’d have to use functions. Such functions are called *getters* or *accessors*. Likewise, if we wanted to change the values of these data members, we’d have to write another set of functions. These would be called *setters* or *mutators.*.

Let’s rewrite Shape and Circle now to feature these “private” variables and accessors and mutators to access them. Then, let’s write a main function to use these instead of accessing values directly.

Private variables in Python typically begin with **two underscores**. You should never access variables that have underscore (either single or double) prefixes outside of the class.

class Shape:

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

self.\_\_x = x

self.\_\_y = y

def get\_x(self):

return self.\_\_x

def set\_x(self,x):

self.\_\_x = x

def get\_y(self):

return self.\_\_y

def set\_y(self,y):

self.\_\_y = y

def get\_shape\_type(self):

return 's'

def get\_draw\_params(self):

return [self.get\_shape\_type(), self.\_\_x, self.\_\_y]

class Circle(Shape):

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

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

self.set\_radius(radius)

def get\_radius(self):

return self.\_\_radius

def set\_radius(self,radius):

if (radius < 0):

self.\_\_radius = 0

else:

self.\_\_radius = radius

def get\_shape\_type(self):

return 'c'

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.append(self.\_\_radius)

return result

#main

x = float(input("Enter x: "))

y = float(input("Enter y: "))

rad = float(input("Enter rad: "))

cir = Circle(x,y,rad)

cir.set\_radius(2\*cir.get\_radius())

cir.set\_x(5)

print("%.f %.f %.f" % (cir.get\_x(),cir.get\_y(),cir.get\_radius()))

print(cir.get\_draw\_params())

**Another way to keep data private: properties**

Another option for keeping data private is to use properties.

You again name the data element themselves with two leading underscores, but you also provide properties to make them accessible to the outside.

A really good resource on this is at

http://www.python-course.eu/python3\_properties.php

This approach uses things called decorators to define the publicly accessible ways of getting and setting the values of the internal variables.

The two decorators that are used in this case are

@property

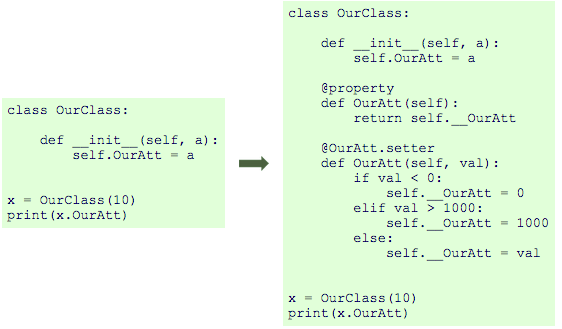
which introduces a function that returns the value of the property

and

@name\_of\_property.setter

which introduces the function that sets the value of the property.

This picture does a nice job showing how you can hide data using properties. The \_\_OurAtt is a private data. It is accessed from the outside using OurAtt, which actually calls the @property OurAtt to get the value, and @OurAtt.setter to set the attribute.



Here’s our Shape example with properties declared in this way now.

class Shape:

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

self.x = x

self.y = y

@property

def x(self):

return self.\_\_x

@x.setter

def x(self,x):

self.\_\_x = x

@property

def y(self):

return self.\_\_y

@y.setter

def y(self,y):

self.\_\_y = y

def calc\_area(self):

pass

def call\_perim(self):

pass

def get\_shape\_type(self):

return "s"

def to\_string(self):

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

def get\_draw\_params(self):

return [self.get\_shape\_type(), self.x, self.y]

class Circle(Shape):

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

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

self.radius = rad

@property

def radius(self):

return self.\_\_radius

@radius.setter

def radius(self,rad):

if rad <= 0:

self.\_\_radius = 1

else:

self.\_\_radius = rad

def calc\_area(self):

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

def calc\_perim(self):

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

def get\_shape\_type(self):

return "c"

def to\_string(self):

return "%s %d" % (super().to\_string(),self.radius)

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.append(self.radius)

return result

class Rectangle(Shape):

def \_\_init\_\_(self,x,y,width,height):

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

self.width = width

self.height = height

@property

def width(self):

return self.\_\_width

@width.setter

def width(self,wid):

self.\_\_width = int(math.fabs(wid))

@property

def height(self):

return self.\_\_height

@height.setter

def height(self,ht):

self.\_\_height = int(math.fabs(ht))

def calc\_area(self):

return self.width \* self.height

def calc\_perim(self):

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

def get\_shape\_type(self):

return "r"

def to\_string(self):

return "%s %d %d" % (super().to\_string(),self.width,self.height)

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend([self.width, self.height])

return result

#main

x = float(input("Enter x: "))

y = float(input("Enter y: "))

rad = float(input("Enter rad: "))

cir = Circle(x,y,rad)

cir.radius = 2 \* cir.radius

cir.x = 5

print("%.f %.f %.f" % (cir.x,cir.y,cir.radius))

print(cir.get\_draw\_params())

Notice how much cleaner the main part of the code is now. No more weird and wordy get and set calls.

**Shape Drawer with Text File Input and Output**

Let’s tie this all together. We will now use these Shape Classes and our Turtle\_Draw\_Shape\_Controler in conjunction with a new class called Shape\_IO\_Controller. The Shape\_IO\_Controller will read and write text files of shapes. They will use the Shape classes’ toString functions to help write a representation of each shape to a text file. Shape\_IO\_Controller will read those fields back in in exactly the same order to reconstruct each Shape.

The new Shape\_IO\_Controller class is here. It includes functions writing Shape objects to a file, writing them to the console window and reading them back in from a file. Notice how the read\_file function works: if it detects that a line begins with an r or a c, it will read and interpret the rest of the columns on the line as describing attributes of the rectangle or circle and create the rectangle or circle, finally adding it to the result list. If the line doesn’t start with an r or c, it won’t append anything onto the result list for that particular line of the file.

''' The following class will write and read shape data

to and from text files and the console.

'''

class Shape\_IO\_Controller:

def read\_file(self,fname):

fin = open(fname,"r")

result = []

for line in fin:

parts = line.split(" ")

if (parts[0] == "c"):

x = int(parts[1])

y = int(parts[2])

rad = int(parts[3])

shp = Circle(x,y,rad)

elif (parts[0] == "r"):

x = int(parts[1])

y = int(parts[2])

width = int(parts[3])

height= int(parts[4])

shp = Rectangle(x,y,width,height)

else:

shp = None

if shp != None:

result.append(shp)

fin.close()

return result

def write\_file(self,fname,shapes):

fout = open(fname,"w")

for s in shapes:

fout.write("%s\n" % s.to\_string())

fout.close()

def write\_console(self,shapes):

for s in shapes:

print(s.to\_string())

Here is the full source code:

import math

import turtle

class Shape:

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

self.x = x

self.y = y

@property

def x(self):

return self.\_\_x

@x.setter

def x(self,x):

self.\_\_x = x

@property

def y(self):

return self.\_\_y

@y.setter

def y(self,y):

self.\_\_y = y

def calc\_area(self):

pass

def call\_perim(self):

pass

def get\_shape\_type(self):

return "s"

def to\_string(self):

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

def get\_draw\_params(self):

return [self.get\_shape\_type(), self.x, self.y]

class Circle(Shape):

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

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

self.radius = rad

@property

def radius(self):

return self.\_\_radius

@radius.setter

def radius(self,rad):

if rad <= 0:

self.\_\_radius = 1

else:

self.\_\_radius = rad

def calc\_area(self):

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

def calc\_perim(self):

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

def get\_shape\_type(self):

return "c"

def to\_string(self):

return "%s %d" % (super().to\_string(),self.radius)

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.append(self.radius)

return result

class Rectangle(Shape):

def \_\_init\_\_(self,x,y,width,height):

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

self.width = width

self.height = height

@property

def width(self):

return self.\_\_width

@width.setter

def width(self,wid):

self.\_\_width = int(math.fabs(wid))

@property

def height(self):

return self.\_\_height

@height.setter

def height(self,ht):

self.\_\_height = int(math.fabs(ht))

def calc\_area(self):

return self.width \* self.height

def calc\_perim(self):

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

def get\_shape\_type(self):

return "r"

def to\_string(self):

return "%s %d %d" % (super().to\_string(),self.width,self.height)

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend([self.width, self.height])

return result

''' The following class will write and read shape data

to and from text files and the console.

'''

class Shape\_IO\_Controller:

def read\_file(self,fname):

fin = open(fname,"r")

result = []

for line in fin:

parts = line.split(" ")

if (parts[0] == "c"):

x = int(parts[1])

y = int(parts[2])

rad = int(parts[3])

shp = Circle(x,y,rad)

elif (parts[0] == "r"):

x = int(parts[1])

y = int(parts[2])

width = int(parts[3])

height= int(parts[4])

shp = Rectangle(x,y,width,height)

else:

shp = None

if shp != None:

result.append(shp)

fin.close()

return result

def write\_file(self,fname,shapes):

fout = open(fname,"w")

for s in shapes:

fout.write("%s\n" % s.to\_string())

fout.close()

def write\_console(self,shapes):

for s in shapes:

print(s.to\_string())

class Turtle\_Draw\_Shape\_Controller:

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

turtle.setup(width,height)

self.window = turtle.Screen()

self.artist = turtle.Turtle()

def draw\_rectangle(self,width,height):

self.artist.seth(0)

for i in range(2):

self.artist.forward(width)

self.artist.right(90)

self.artist.forward(height)

self.artist.right(90)

def draw(self, shapes):

for s in shapes:

self.artist.penup()

params = s.get\_draw\_params()

# params[0] is the shapetype

# params[1] is x

# params[2] is y

# then for circles:

# params[3] is radius

# whereas for rectangles:

# params[3] is width

# params[4] is height

self.artist.goto(params[1],params[2])

self.artist.pendown()

if (params[0] == "c"):

self.artist.circle(params[3])

elif (params[0] == "r"):

self.draw\_rectangle(params[3],params[4])

def close(self):

self.window.bye()

def main():

sio = Shape\_IO\_Controller()

shapes = sio.read\_file("shapes.txt")

sio.write\_console(shapes)

fname = input("Enter name of file: ")

sio.write\_file(fname,shapes)

tdsc = Turtle\_Draw\_Shape\_Controller(400,400)

tdsc.draw(shapes)

input("Press enter to continue.")

tdsc.close()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Adding a Face Class**

Just to get more practice with inheritance and composition, let’s consider creating a Face class.

* Let’s assume Face’s are rectangular in shape. That is, they have a width and a height. Then, it makes sense to descend Face from Rectangle. Face is the subclass, and Rectangle is the superclass.
* The eyes of a Face are circular in shape. Therefore, it makes sense for Face to have two Circle objects built into it. The eyes cannot exist separate from the face, because that would be gross. This is an example of composition (exclusive ownership, in which the owner is responsible for creating the things that are owned).

Here’s a version of the Face class that might fit into our program really well:

class Face(Rectangle):

def \_\_init\_\_(self,x,y,width,height):

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

self.left\_eye = Circle(x+int(0.2\*width),y-

int(0.2\*height),int(0.1\*width))

self.right\_eye = Circle(x+int(0.8\*width),y-

int(0.2\*height),int(0.1\*width))

def get\_shape\_type(self):

return "f"

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend(self.left\_eye.get\_draw\_params())

result.extend(self.right\_eye.get\_draw\_params())

return result

def to\_string(self):

return "%s %s %s" % (super().to\_string(), self.left\_eye.to\_string(),

self.right\_eye.to\_string())

We could modify Turtle\_Draw\_Shape\_Controller as follows to draw Face objects, too.

class Turtle\_Draw\_Shape\_Controller:

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

turtle.setup(width,height)

self.window = turtle.Screen()

self.artist = turtle.Turtle()

def draw\_rectangle(self,width,height):

self.artist.seth(0)

for i in range(2):

self.artist.forward(width)

self.artist.right(90)

self.artist.forward(height)

self.artist.right(90)

def draw(self, shapes):

for s in shapes:

self.artist.penup()

params = s.get\_draw\_params()

# params[0] is the shapetype

# params[1] is x

# params[2] is y

# then for circles:

# params[3] is radius

# whereas for rectangles:

# params[3] is width

# params[4] is height

self.artist.goto(params[1],params[2])

self.artist.pendown()

if (params[0] == "c"):

self.artist.circle(params[3])

elif (params[0] == "r"):

self.draw\_rectangle(params[3],params[4])

**elif (params[0] == "f"):**

**'''**

**0 - shapetype**

**1-2 - x,y location**

**3-4 - width, length**

**5 - shape type for the left eye (c)**

**6 - 7 - x,y location of left eye**

**8 - radius of the left eye**

**9 - shape type for the right eye (c)**

**10 - 11 - x,y location for right eye**

**12 - radius of the right eye**

**'''**

**self.draw\_rectangle(params[3],params[4])**

**# draw the left eye**

**self.artist.penup()**

**self.artist.goto(params[6],params[7])**

**self.artist.pendown()**

**self.artist.circle(params[8])**

**self.artist.penup()**

**self.artist.goto(params[10],params[11])**

**self.artist.pendown()**

**self.artist.circle(params[12])**

def close(self):

self.window.bye()

Of course, we could also modify Shape\_IO\_Controller to read and write Face objects with plain text files, too.

**XML**

Plain text files are fine, but they require you to know their format before you can read them. If you are going to use a text file created by someone else, the two of you have to communicate ahead of time to decide what the file format is. XML, a self-describing file format, offers an alternative.

The next assignment will give you an opportunity to grab an XML file of Shapes from the web, parse it, and then draw them.

XML stands for eXtensible Markup Language. It is a way of expressing organized data in a self-describing way, using <tags> to identify what each piece of data represents.

Here’s an xml file that is similar to the one you’ll work with:

<shapes>

<circle>

<radius>25</radius>

<x>15</x>

<y>100</y>

</circle>

<rectangle>

<length>40</length>

<width>70</width>

<x>-20</x>

<y>120</y>

</rectangle>

<circle>

<radius>50</radius>

<x>-50</x>

<y>-50</y>

</circle>

</shapes>

You can see that it defines a list of shapes. Our job is to extract the shapes from this file.

The first step is to grab the file from some place on the web. Python makes it pretty easy to grab content from the web. Here’s a demonstration.

# illustrate grabbing a file from a website

import urllib.request

try:

url = "http://cs.lewisu.edu/~klumpra/2016Summer/shapes.xml"

local\_file\_name = "test\_shapes.xml"

retriever = urllib.request.URLopener()

retriever.retrieve(url,local\_file\_name)

fin = open(local\_file\_name,"r")

for line in fin:

print(line)

fin.close()

except:

print("Something went wrong.")

At this point, if it was successful, the local file will be a copy of what is online. Now comes the challenging part. Let’s parse it to grab the shapes that are in there. We’ll run through the contents of the file looking for Circles first. Here’s a start:

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

text = ""

for line in file\_var:

text = text + line.strip()

shapes\_list = []

pos = text.find("<circle>")

while pos >= 0:

endpos = text.find("</circle>",pos+1)

tagbeg = text.find("<x>",pos+1)

tagend = text.find("</x>",tagbeg+1)

x = float(text[tagbeg+3:tagend])

tagbeg = text.find("<y>",pos+1)

tagend = text.find("</y>",tagbeg+1)

y = float(text[tagbeg+3:tagend])

tagbeg = text.find("<radius>",pos+1)

tagend = text.find("</radius>",tagbeg+1)

radius = float(text[tagbeg+8:tagend])

cir = Circle(radius,x,y)

shapes\_list.append(cir)

pos = text.find("<circle>",endpos+9)

For the assignment, we need to put this into a class and add code to look for other shape types, like Rectangles (<rectangle> … </rectangle>), Regular Polygons (<regpoly> … </regpoly>), Lines (<line> … </line>), Text (<text> … </text>, Faces (<face> … </face>), and Houses (<house> … </house>).

**A Beginning-To-End Example: XML-Driven Paycheck Calculator**

In this example, we build a paycheck calculator and printer that reads data about employees and their hours worked from two xml files, one containing employee data, and the other containing timecard data.

The classes this project contains are divided into model and controller:

Model:

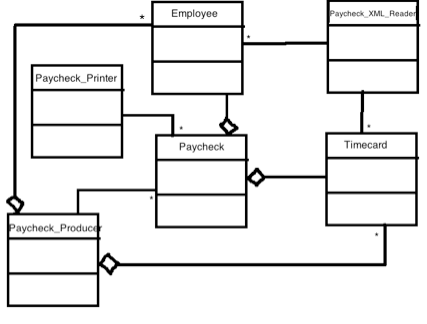
* Employee, with properties empid, name, address, and pay\_rate
* Timecard, with properties empid and hours (representing hours worked)
* Paycheck, a joining together of Employee and Timecard, which represents the check we will pay that employee for the work represented on the timecard

Controller:

* Paycheck\_XML\_Reader, which has read\_employees and read\_timecards functions which read xml files describing employees and timecards
* Paycheck\_Producer, which builds the Paycheck objects corresponding to each Employee and Timecard combination. It has a produce\_checks function that does this.
* Paycheck\_Printer, which actually prints the paycheck objects using its print\_checks function

The views are not classes in this example: they are the xml files we’re reading from and the screen we’re writing to. Sometimes that is the case in object-oriented software: the source or destination of our model data are not objects of classes but features of the operating system. In that case, we don’t have to declare classes that represent them, because they already exist as features of the language.

Here is the UML Diagram:



Here are the xml files. First, staff.xml:

<staff>

<employee>

<empid>123</empid>

<name>Joe Smith</name>

<address>472 Gary Busey Lane</address>

<payrate>17.52</payrate>

</employee>

<employee>

<empid>456</empid>

<name>Jane Doe</name>

<address>1323 North Primrose Path</address>

<payrate>21.50</payrate>

</employee>

<employee>

<empid>789</empid>

<name>Floyd Milton</name>

<address>5559 Winsdor Way</address>

<payrate>18.80</payrate>

</employee>

</staff>

Next, timecards.xml:

<worklog>

<timecard>

<empid>123</empid><hours>17.5</hours>

</timecard>

<timecard>

<empid>456</empid><hours>21.75</hours>

</timecard>

<timecard>

<empid>789</empid><hours>15.5</hours>

</timecard>

</worklog>

Here is the code we’ll end up with:

class Employee:

@property

def name(self):

return self.\_\_name

@name.setter

def name(self,val):

self.\_\_name = val

@property

def empid(self):

return self.\_\_empid

@empid.setter

def empid(self,val):

self.\_\_empid = val

@property

def address(self):

return self.\_\_address

@address.setter

def address(self,val):

self.\_\_address = val

@property

def pay\_rate(self):

return self.\_\_pay\_rate

@pay\_rate.setter

def pay\_rate(self,val):

self.\_\_pay\_rate = val

def \_\_init\_\_(self,empid,name,address,pay\_rate):

self.empid = empid

self.name = name

self.address = address

self.pay\_rate = pay\_rate

def to\_string(self):

return "%s\n%s\n%s\n$%.2f" % (self.empid,self.name,self.address,self.pay\_rate)

class Timecard:

@property

def empid(self):

return self.\_\_empid

@empid.setter

def empid(self,val):

self.\_\_empid = val

@property

def hours(self):

return self.\_\_hours

@hours.setter

def hours(self,val):

if (val < 0):

self.\_\_hours = 0

else:

self.\_\_hours = val

def \_\_init\_\_(self,empid,hours):

self.empid=empid

self.hours=hours

class Paycheck:

@property

def employee(self):

return self.\_\_employee

@employee.setter

def employee(self,val):

self.\_\_employee = val

@property

def timecard(self):

return self.\_\_timecard

@timecard.setter

def timecard(self,val):

self.\_\_timecard = val

def \_\_init\_\_(self,emp,tc):

self.employee = emp

self.timecard = tc

def calc\_pay(self):

return self.employee.pay\_rate \* self.timecard.hours

def to\_string(self):

return "%s\n$%.2f" % (self.employee.to\_string(),self.calc\_pay())

class Paycheck\_Producer:

def \_\_init\_\_(self, employees, cards):

self.employees = employees

self.timecards = cards

def find\_employee(self,empid):

for emp in self.employees:

if emp.empid == empid:

return emp

return None

def produce\_checks(self):

checks = []

for tc in self.timecards:

emp = self.find\_employee(tc.empid)

if emp != None:

check = Paycheck(emp,tc)

checks.append(check)

return checks

class Paycheck\_Printer:

def print\_separator(self,count):

print("\*"\*count)

def print\_checks(self,checks):

for check in checks:

self.print\_separator(30)

print(check.to\_string())

self.print\_separator(30)

print()

class Paycheck\_XML\_Reader:

def read\_employees(self,fname):

try:

fin = open(fname,"r")

employees = []

text = ""

for line in fin:

text = text + line

fin.close()

begpos = text.find("<staff>")

endpos = text.find("</staff>")

emp\_begpos = text.find("<employee>",begpos+1)

while (emp\_begpos >= 0):

emp\_endpos = text.find("</employee>",emp\_begpos+1)

empid\_begpos = text.find("<empid>",emp\_begpos+1,emp\_endpos)

empid\_endpos = text.find("</empid>",empid\_begpos+1)

empid = text[empid\_begpos+7:empid\_endpos]

empname\_begpos = text.find("<name>",emp\_begpos+1,emp\_endpos)

empname\_endpos = text.find("</name>",empname\_begpos+1)

empname = text[empname\_begpos+6:empname\_endpos]

empaddr\_begpos = text.find("<address>",emp\_begpos+1,emp\_endpos)

empaddr\_endpos = text.find("</address>",empaddr\_begpos+1)

empaddr = text[empaddr\_begpos+9:empaddr\_endpos]

emprate\_begpos = text.find("<payrate>",emp\_begpos+1,emp\_endpos)

emprate\_endpos = text.find("</payrate>",emprate\_begpos+1)

emprate = float(text[emprate\_begpos+9:emprate\_endpos])

emp = Employee(empid,empname,empaddr,emprate)

employees.append(emp)

emp\_begpos = text.find("<employee>",emp\_endpos+1)

return employees

except:

return None

def read\_timecards(self,fname):

try:

fin = open(fname,"r")

timecards = []

text = ""

for line in fin:

text = text + line

begpos = text.find("<worklog");

endpos = text.find("</worklog");

tc\_begpos = text.find("<timecard>",begpos+1)

while (tc\_begpos >= 0):

tc\_endpos = text.find("</timecard>",tc\_begpos+1)

empid\_begpos = text.find("<empid>",tc\_begpos+1,tc\_endpos)

empid\_endpos = text.find("</empid>",empid\_begpos+1)

empid = text[empid\_begpos+7:empid\_endpos]

hours\_begpos = text.find("<hours>",tc\_begpos+1,tc\_endpos)

hours\_endpos = text.find("</hours>",hours\_begpos+1)

hours = float(text[hours\_begpos+7:hours\_endpos])

tc = Timecard(empid,hours)

timecards.append(tc)

tc\_begpos = text.find("<timecard>",tc\_endpos+1)

fin.close()

return timecards

except:

return None

def main():

print("Welcome to Paycheck Printer")

emp\_file\_name = input("Enter name of employee file: ")

time\_file\_name = input("Enter name of timecard file: ")

reader = Paycheck\_XML\_Reader()

employees = reader.read\_employees(emp\_file\_name)

timecards = reader.read\_timecards(time\_file\_name)

if employees != None and timecards != None:

producer = Paycheck\_Producer(employees, timecards)

checks = producer.produce\_checks()

printer = Paycheck\_Printer()

printer.print\_checks(checks)

else:

print("I'm sorry, but we couldn't read the input files.")

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

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Summary of this example**

The purpose of this example was to guide you through the process of creating an object-oriented solution to a problem.

For beginning programmers, it is often tricky to design things well like this. However, the following rules of thumb might help you:

Identify your model classes

Define them

Include properties rather than public variables

Include an \_\_init\_\_ (if anything needs to passed in for

initialization, such as for aggregation or

composition)

Include to\_string for easier printout

For each kind of activity that needs to be done, create a controller

Name it clearly

Identify the view classes that controllers interact with to communicate and change model data

Could be files on the disk, or the command prompt. It’s any

medium that enables your program to interact with its environment

Make main your hub!!!

It coordinates the activities

It should create the controllers and perhaps the model objects

main should be readable. You should be able to read it from top to

bottom and make sense of what your program is doing

**Supplement: Help with Webshapes Homework**

Here is how to structure the webshapes homework

Create a Web\_Retriever class. The Web\_Retriever class needs to have just one function – download – that takes in the url you want to retrieve the file from and the name of the file to which you want to save the file locally. (Of course, it will also need a self parameter, as all functions that belong to classes have to have a self parameter). It will return True if it successfully downloads the File and False otherwise. You’ll know if it successfully downloads the file because the try … except you have to wrap up the code you include in the function will fail if you can’t download the file successfully.

Create Shape, Circle, Rectangle, and Face classes that are identical to what we did in the notes. Here they are:

class Shape:

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

self.x = x

self.y = y

@property

def x(self):

return self.\_\_x

@x.setter

def x(self,x):

self.\_\_x = x

@property

def y(self):

return self.\_\_y

@y.setter

def y(self,y):

self.\_\_y = y

def get\_shape\_type(self):

return "shape"

def to\_string(self):

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

def get\_draw\_params(self):

return [self.get\_shape\_type(), self.x, self.y]

class Circle(Shape):

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

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

self.radius = rad

@property

def radius(self):

return self.\_\_radius

@radius.setter

def radius(self,rad):

if rad <= 0:

self.\_\_radius = 1

else:

self.\_\_radius = rad

def get\_shape\_type(self):

return "circle"

def to\_string(self):

return "%s %d" % (super().to\_string(),self.radius)

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.append(self.radius)

return result

class Rectangle(Shape):

def \_\_init\_\_(self,x,y,width,height):

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

self.width = width

self.height = height

@property

def width(self):

return self.\_\_width

@width.setter

def width(self,wid):

self.\_\_width = int(math.fabs(wid))

@property

def height(self):

return self.\_\_height

@height.setter

def height(self,ht):

self.\_\_height = int(math.fabs(ht))

def get\_shape\_type(self):

return "rect"

def to\_string(self):

return "%s %d %d" % (super().to\_string(),self.width,self.height)

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend([self.width, self.height])

return result

class Face(Rectangle):

def \_\_init\_\_(self,x,y,width,height):

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

self.left\_eye = Circle(x+int(0.2\*width),y-int(0.2\*height),int(0.1\*width))

self.right\_eye = Circle(x+int(0.8\*width),y-int(0.2\*height),int(0.1\*width))

def get\_shape\_type(self):

return "face"

def get\_draw\_params(self):

result = super().get\_draw\_params()

result.extend(self.left\_eye.get\_draw\_params())

result.extend(self.right\_eye.get\_draw\_params())

return result

Create the Line, Banner, House, and Rectangular Polygon classes. Line, Banner, and Regular\_Polygon should descend from Shape, and House should descend from Rectangle. You have to figure out what are the important data – i.e. characteristics – that make each different kind of shape distinct.

Create your Turtle\_Draw\_Shape\_Controller class like we did in class. You will have to expand the draw function to be able to handle the wider variety of shapes your program will now support. Here is the code we did to get you started:

class Turtle\_Draw\_Shape\_Controller:

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

turtle.setup(width,height)

self.window = turtle.Screen()

self.artist = turtle.Turtle()

def draw\_rectangle(self,width,height):

self.artist.seth(0)

for i in range(2):

self.artist.forward(width)

self.artist.right(90)

self.artist.forward(height)

self.artist.right(90)

def draw(self, shapes):

for s in shapes:

self.artist.penup()

params = s.get\_draw\_params()

self.artist.goto(params[1],params[2])

self.artist.pendown()

if (params[0] == "circle"):

self.artist.circle(params[3])

elif (params[0] == "rect"):

self.draw\_rectangle(params[3],params[4])

elif (params[0] == "face"):

self.draw\_rectangle(params[3],params[4])

self.artist.penup()

self.artist.goto(params[6],params[7])

self.artist.pendown()

self.artist.circle(params[8])

self.artist.penup()

self.artist.goto(params[10],params[11])

self.artist.pendown()

self.artist.circle(params[12])

def close(self):

self.window.bye()

Create a Shape\_XML\_Parser class. This one will be a little tedious. You will have to write XML parse routines for each of the various kinds of shapes, based on what we did for circle in class. Here is code to get you started:

class Shape\_XML\_Parser:

def parse(self,fname):

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

text = ""

for line in file\_var:

text = text + line.strip()

shapes\_list = []

pos = text.find("<circle>")

while pos >= 0:

endpos = text.find("</circle>",pos+1)

tagbeg = text.find("<x>",pos+1)

tagend = text.find("</x>",tagbeg+1)

x = int(text[tagbeg+3:tagend])

tagbeg = text.find("<y>",pos+1)

tagend = text.find("</y>",tagbeg+1)

y = int(text[tagbeg+3:tagend])

tagbeg = text.find("<radius>",pos+1)

tagend = text.find("</radius>",tagbeg+1)

radius = int(text[tagbeg+8:tagend])

cir = Circle(x,y,radius)

shapes\_list.append(cir)

pos = text.find("<circle>",endpos+9)

pos = text.find("<rectangle>")

while pos >= 0:

endpos = text.find("</rectangle>",pos+1)

tagbeg = text.find("<x>",pos+1)

tagend = text.find("</x>",tagbeg+1)

x = int(text[tagbeg+3:tagend])

tagbeg = text.find("<y>",pos+1)

tagend = text.find("</y>",tagbeg+1)

y = int(text[tagbeg+3:tagend])

tagbeg = text.find("<length>",pos+1)

tagend = text.find("</length>",tagbeg+1)

length = int(text[tagbeg+8:tagend])

tagbeg = text.find("<width>",pos+1)

tagend = text.find("</width>",tagbeg+1)

width = int(text[tagbeg+7:tagend])

rec = Rectangle(x,y,width,length)

shapes\_list.append(rec)

pos = text.find("<rectangle>",endpos+12)

file\_var.close()

return shapes\_list

So, basically, for each different kind of shape, you start off trying to find the corresponding starting tag (like <circle> or <rectangle> or <banner>, etc.), and then you have a while loop that keeps on parsing out sections of the file that have that format.

In main, the hub, this is what you do:

Create the Web\_Retriever

Ask the user for the name of the url and local file name.

Tell the Web\_Retriever object to download the file.

If that was successful, create the Turtle\_Draw\_Shape\_Controller and Shapes\_XML\_Parser, the XML parse to execute its parse function to return a list of shapes, feed that list of shapes to the Turtle\_Draw\_Shape\_Controller, and also print those shapes to the screen using their to\_string functions.