What's in the Box?

This exercise reinforces concepts of Classes, class instances and methods.

**Part (a):** The program below defines a **Box** class. Each **Box** instance contains four data fields (all floats)—a data field named *x*, a data field named *y*, a data field named *w*, and a data field named *h*. The values for these fields are passed to the *constructor* as arguments.

```python
class Box(object):
    def __init__(self, x_coord=0, y_coord=0, width=100, height=100):
        '''box at (x_coord, y_coord) with given width and height'''
        # requires: all arguments are numeric, width > 0, height > 0
        # assumes: measurements are in pixels
        self.x = x_coord    # x coordinate of lower left corner
        self.y = y_coord    # y coordinate of lower left corner
        self.w = width      # width of the box
        self.h = height     # height of the box

    def add(self, other):
        '''sum of self and other'''
        new_x = min(self.x, other.x)
        new_y = min(self.y, other.y)
        new_w = max(self.x + self.w, other.x + other.w) - new_x
        new_h = max(self.y + self.h, other.y + other.h) - new_y
        return Box(new_x, new_y, new_w, new_h)

    def move(self, amt_x=0, amt_y=0):
        '''move self by amt_x horizontally and amt_y vertically'''
        self.x += amt_x
        self.y += amt_y

    def getStr(self):
        '''string representation of self'''
        templ = 'box@({:.1f}, {:.1f}), {:.1f} pxl X {:.1f} pxl'
        return templ.format(self.x, self.y, self.w, self.h)

b1 = Box()
b1_str = b1.getStr()
print('b1:', b1_str)
b2 = Box(-10, 50, 75, 75)
print('b2:', b2.getStr())
b3 = b1.add(b2)
print('b3:', b3.getStr())
b4 = b2.move(-150, -100)
print('b2:', b2.getStr())
print('b4:', b4.getStr())
```
Form groups of 2-3 students. Bring up the Visualization. We will explore this visualization together as a class, with different groups offering their explanations of answers to the following questions.

1. **At step 2:** Why was the name **Box** entered into the global namespace (frame)? In what sense is executing a class definition like executing a function definition or an assignment statement?

2. **At step 3:**
   a. Why was the **__init__** name space (frame) created?
   
   b. Why does this name space contain the names **self, x_coord, y_coord, width** and **height** in it, and how were the values for these names determined?

3. **In steps 4—7:** What namespace does Python add the names **x, y, w** and **h** to and why? How does it find the value associated with each name?

4. **At step 8:** What will happen next? (Discuss with your partner before pressing Forward.)

5. **At step 9:**
   a. Why was the name **b1** entered into the global namespace (frame)?
   
   b. What is the type of **b1**?
   
   c. What is the value of **b1**?

6. **At step 10:**
   a. Why is the **getStr** name space created?
   
   b. Why does this namespace contain the name **self**?
   
   c. Why does **self** reference the same value as the name **b1** in the global name space?

7. **At step 11:** What will happen next? (Discuss before pressing Forward.)

8. **At step 12:** How does Python find the values for **self.x, self.y, self.w**, and **self.h**?
9. **At step 13:** What will happen next? (Discuss before pressing Forward.)

10. **At step 15:** Discuss answers to a—c below before pressing Forward. Then advance through steps 14—20 to check your answers.
   
   a. How does Python find the method to call to evaluate the right side of the assignment?
   
   b. What will be in the new namespace that Python creates for this method?
   
   c. What will this method return?

11. **At step 22:** Discuss answers to a—c below before pressing Forward. Then advance through steps 22—25 to check your answers.
   
   a. How does Python find the method to call to evaluate the second argument of the `print` statement?
   
   b. What will be in the new namespace that Python creates for this method?
   
   c. What will this method return and what will be displayed?

12. **At step 27:** What will happen next? (Discuss before pressing Forward.)

13. **In steps 29-32:** How does Python determine the values of the expressions on the right sides of the assignments; what namespace does it enter the new names into and why?

14. **At step 33:** What will happen next? (Discuss before pressing Forward.)
15. **At step 39:** What will happen next? (Discuss before pressing.)

16. **At step 40:** What will happen next? (Discuss before pressing Forward.)

17. **At step 41:** Before pressing forward, discuss what this print statement will display (consult the snapshot showing the value of `b3` and the code for `getStr`). Then advance through steps 41—45 to check your answer.

18. **At step 46:** What will happen next? (Try to predict before pressing Forward.)

19. **At step 50:** What will happen next? (Try to predict before pressing Forward.)

20. **At step 51:** Before pressing forward, try to predict what this print statement will display (consult the snapshot showing the value of `b2` and the code for `getStr`). Then advance through steps 51—55 to check your answer.

21. **At step 56:** Before pressing forward, try to predict what this print statement will display (consult the snapshot showing the value of `b2` and the code for `getStr`). Then press forward to check your answer.

22. Which of the method(s) in the **Box** class *mutates* a box?
Part (b): This part illustrates one way to extend the Box class for use in a drawing program. We use the Python turtle module, which implements a simple drawing tool based on the LOGO programming language (see the handout on Turtle Graphics).

First look over the copy of the program handed out with this exercise. The program in this handout is similar to the one just visualized. Differences include:

• The program imports the turtle module.
• The Box class contains a draw method, which receives the turtle module (by default) as an input to use in drawing a Box instance.
• The program defines a test_draw function, which draws the boxes in a list of Box instances.
• The erroneous assignment to b4 is removed.
• The program creates a list of Box instances and assigns the list to a new variable (blst).
• The program calls test_draw with three different lists.

To see how this program works, download box.py and open it in Spyder. Then run it as follows:

• When the screen comes up, move it around and resize the Spyder window and panes as necessary so you can see both the console and the Turtle Graphics screen. (If you see a prompt in the console and do not see a screen, look for the screen under the Spyder window.)

Discuss with your partner:

-- Which statements in the program produced the output in the console that you are seeing?

-- Which rectangle in the Turtle Graphics window is a drawing of b1? b2? b3?

-- Where/why is the execution ‘paused’?

• To continue execution, after the prompt in the console, press enter. Discuss with your partner: Why does b2 draw at a different position than before?

• Continue the execution and just enjoy it for a minute.

• In the console window, call getStr on b1, b2, and b3 to check that boxes still store the expected values.

• In the console window, evaluate [b.getStr() for b in blst]
Part (c): Download the file `shapes.py` and open it in Spyder. It contains the definitions for the `Box` class and the `test_draw` function from the previous exercise. It also contains several triple-quoted strings at the end of the file labeled with ‘deletion numbers.’ These are included to save you time typing code to test new methods. At various points in the remainder of this exercise, you will be instructed to delete the triple quotes surrounding one of these sections and run the program to test some new behavior you have coded.

In this part of this exercise, you will extend the `Box` class with a `mult` method.

Add a method named `mult` to calculate the “product” of a box and a positive scalar (a numeric value, with a default value of 1). Define `mult` so that `b.mult(n)` returns a new box with the same lower left corner as `b`, but whose dimensions are the product of `n` and the dimensions of `b`. It must not modify `b`. For example:

```python
In [95]: b1 = Box()
In [96]: b2 = b1.mult(1.5)
In [97]: b3 = b1.mult(1/3)
In [98]: b1.getStr(), b2.getStr(), b3.getStr()
Out[98]:
('box@(0.0, 0.0), 100.0 pxl X 100.0 pxl',
 'box@(0.0, 0.0), 150.0 pxl X 150.0 pxl',
 'box@(0.0, 0.0), 33.3 pxl X 33.3 pxl')
```

Delete the line marked `Deletion #1` and the matching set of triple quotes. Run the program and check that it is producing correct output. Fix any errors before proceeding.

Part (d): In this part, you will extend the file with a class called `Circle` for representing circles. You will define it so that `Circle` instances behave in much the same way as `Box` instances.

Add your class definition incrementally as follows:

1. Define just a constructor for the class. The constructor will need four parameters: the first, named `self`, will refer to the instance being constructed; the second and third will receive (numeric) values for the x-coordinate and y-coordinate, respectively, of the center of the circle; and the fourth will receive a (positive numeric) value for the radius. Define default values for the last three parameters: use 0 for default coordinates and 50 for the default radius. Define the constructor so that an instance of a circle has data attributes named `x`, `y`, and `r` which store the values passed in for the x-coordinate, y-coordinate, and radius, respectively.

To test the constructor, delete the line marked `Deletion #2` and the matching set of triple quotes. Run the program. Then run some experiments in the console to check that the circles were created and that they store the expected information. For example:
2. Add a `getStr` method similar to that defined in the `Box` class. To test the new method, delete the line marked **Deletion #3** and the matching set of triple quotes. Run the program and check that it displays the appropriate output.

3. Add an `add` method to implement the following definition for addition:

   Given two circles, $C_1$, with center $(x_1, y_1)$ and radius $r_1$, and $C_2$, with center $(x_2, y_2)$ and radius $r_2$, if either “contains” the other, then their sum has the same center and radius as the containing circle; otherwise, their sum is the circle $C_3$ with center $(x_3, y_3)$ and radius $r_3$ where

   \[
   x_3 = \frac{x_1 + x_2}{2} + \frac{(x_2-x_1)(r_2-r_1)}{2d},
   \]

   \[
   y_3 = \frac{y_1 + y_2}{2} + \frac{(y_2-y_1)(r_2-r_1)}{2d},
   \]

   \[
   r_3 = (d + r_1 + r_2)/2, \text{ and}
   \]

   \[
   d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.
   \]

Define `add` so that it constructs and returns a new `Circle` instance without modifying its arguments. For example:

```python
In [26]: c1.getStr()
Out[26]: 'circle@(0.0, 0.0), 50.0 pxl'

In [27]: c3.getStr()
Out[27]: 'circle@(0.0, 80.0), 20.0 pxl'

In [28]: c5 = c1.add(c3)

In [29]: c5.getStr()
Out[29]: 'circle@(0.0, 25.0), 75.0 pxl'
```
Test the **add** method by deleting the line marked **Deletion #4** and the matching set of triple quotes, running the program, and checking that it displays the appropriate output.

4. Add a **move** method similar to the one defined in the **Box** class—it just moves the center of the circle. (Later, we'll see that inheritance can and should be used to avoid duplicating methods in different, but related classes, like these.) Test the **move** method by deleting the line marked **Deletion #5** and the matching set of triple quotes, running the program, and checking that it displays the appropriate output.

5. Add a **mult** to calculate the "product" of a circle and a positive scalar (a numeric value; use a default value of 1). Define it so that `c.mult(n)` returns a new circle with the same center as `c`, but whose radius is the product of `n` and the radius of `c`, and does not modify `c`. Test the **mult** method by deleting the line marked **Deletion #6** and the matching set of triple quotes, running the program, and checking that it displays the appropriate output.

**Part (e):** This part will illustrate polymorphism at work. First, follow the link under “A Draw Method for Circles” in this week’s Artifacts section on the course website. It takes you to a page containing code for a **draw** method to use with **Circle** instances. If you followed the instructions in Step 1 of Part (d) in defining the **Circle** constructor, you can add a correct draw method for **Circle** instances by copying and pasting this code to the end of your definition for class **Circle**. Do this, and then check that it works by running the program and then calling **test_draw** in the console with a list of **Circle** instances. Fix any errors.

Finally, delete the line marked **Deletion #7** and the matching set of triple quotes. Run the program and take a minute to admire the circles and boxes it draws.

The final call to **test_draw** in this program illustrates the concept of **polymorphism**: When evaluating `next_shape.draw(next_color)` in the body of **test_draw**, the **draw** method that Python invokes depends on the type of the object referenced by `next_shape`—on **Box** instances, it invokes the **draw** method defined in class **Box**, and on **Circle** instances, it invokes the **draw** method defined in class **Circle**. Polymorphism is an important characteristic of object-oriented programming.