جامعة الانبار
كلية علوم الحاسوب وتكنولوجيا المعلومات
قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP
المرحلة الثانية
الفصل الدراسي الأول والثاني

مدرس المادة
م.د. سميه عبدالله حمد

Passing Objects (Classes) to Functions

Objects as Arguments and Parameters

You can pass an object as an argument to a function, in the usual way.

Here is a simple function called distance involving our new Point objects.

The job of this function is to figure out the distance between two points.

```
import math
class Point:
    """ Point class for representing and manipulating x,y
coordinates. """
    def __init__(self, initX, initY):
        self.x = initX
        self.y = initY
    def getX(self):
        return self.x
    def getY(self):
        return self.y
def distance(point1, point2):
    xdiff = point2.getX()-point1.getX()
    ydiff = point2.getY()-point1.getY()
    dist = math.sqrt(xdiff**2 + ydiff**2)
    return dist
p = Point(4,3)
q = Point(0,0)
print(distance(p,q))
```

Output:

5.0

distance takes two points and returns the distance between them. Note that distance is **not** a method of the Point class. You can see this by looking at the indentation pattern. It is not inside the class definition. The other way we can know that distance is not a method of Point is that self is not included as a formal parameter. In addition, we do not invoke distance using the dot notation.

We *could have* made distance be a method of the Point class. Then, we would have called the first parameter self, and would have invoked it using the dot notation, as in the following code. Which way to implement it is a matter of coding style. Both work correctly. Most programmers choose whether to make functions be stand-alone or methods of a class based on whether the function semantically seems to be an operation that is performed on instances of the class. In this case, because distance is really a property of a pair of points and is symmetric (the distance from a to b is the same as that from b to a) it makes more sense to have it be a standalone function and not a method.

```
import math
class Point:
    """ Point class for representing and manipulating x,y
coordinates. """
    def __init__(self, initX, initY):
        self.x = initX
        self.y = initY
    def getX(self):
        return self.x
    def getY(self):
        return self.y
```

```
    def distance(self, point2):
        xdiff = point2.getX()-self.getX()
        ydiff = point2.getY()-self.getY()
        dist = math.sqrt(xdiff**2 + ydiff**2)
        return dist
p = Point(4,3)
q = Point(0,0)
print(p.distance(q))
```

Output:

5.0

Returning Objects (classes) From Functions

- **Converting an Object to a String**

When we're working with classes and objects, it is often necessary to print an object (that is to print the state of an object). Consider the example below.

```
class Point:
    """ Point class for representing and manipulating x,y
coordinates. """
    def __init__(self, initX, initY):
        """ Create a new point at the given coordinates. """
        self.x = initX
        self.y = initY
    def getX(self):
        return self.x
    def getY(self):
        return self.y
```

```
p = Point(7, 6)
print(p)
```

Output:

```
<__main__.Point object>
```

The print function shown above produces a string representation of the Point p. The default functionality provided by Python tells you that p is an object of type Point. However, it does not tell you anything about the specific state of the point.

We can improve on this representation if we include a special method call __str__. Notice that this method uses the same naming convention as the constructor, that is two underscores before and after the name. It is common that Python uses this naming technique for special methods.

The __str__ method is responsible for returning a string representation as defined by the class creator. In other words, you as the programmer, get to choose what a Point should look like when it gets printed. In this case, we have decided that the string representation will include the values of x and y as well as some identifying text. It is required that the __str__ method create and *return* a string.

```
class Point:
    """ Point class for representing and manipulating x,y
coordinates. """
    def __init__(self, initX, initY):
        """ Create a new point at the given coordinates. """
        self.x = initX
        self.y = initY
    def getX(self):
        return self.x
    def getY(self):
```

```
        return self.y
    def __str__(self):
        return "x=" + str(self.x) + ", y=" + str(self.y)
p = Point(7, 6)
print(p)
```

Output:

x=7, y=6

When we run the program above you can see that the print function now shows the string that we chose.

As we saw earlier, these automatic mechanisms do not do exactly what we want. Python provides many default implementations for methods that we as programmers will probably want to change. When a programmer changes the meaning of a special method we say that we **override** the method. Note also that the str type converter function uses whatever __str__ method we provide.

Also we can write __str__ method as:

```
def __str__(self):
    return "x = {}, y = {}".format(self.x, self.y)
```

- **Instances as Return Values**

Functions and methods can return objects. This is actually nothing new since everything in Python is an object and we have been returning values for quite some time. The difference here is that we want to have the method create an object using the constructor and then return it as the value of the method.

Suppose you have a point object and wish to find the midpoint halfway between it and some other target point. We would like to write a method, call it halfway that takes another Point as a parameter and returns the Point that is halfway between the point and the target.

```
class Point:
    def __init__(self, initX, initY):
        """ Create a new point at the given coordinates. """
        self.x = initX
        self.y = initY
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def __str__(self):
        return "x=" + str(self.x) + ", y=" + str(self.y)
    def halfway(self, target):
         mx = (self.x + target.x) / 2
         my = (self.y + target.y) / 2
         return Point(mx, my)
p = Point(3, 4)
q = Point(5, 12)
mid = p.halfway(q)
print(mid)
print(mid.getX())
print(mid.getY())
```

Output:

```
x=4.0, y=8.0
4.0
8.0
```

The resulting Point, mid, has an x value of 4 and a y value of 8. We can also use any other methods since mid is a Point object.

In the definition of the method halfway see how the requirement to always use dot notation with attributes disambiguates the meaning of the attributes x and y: We can always see whether the coordinates of Point self or target are being referred to.