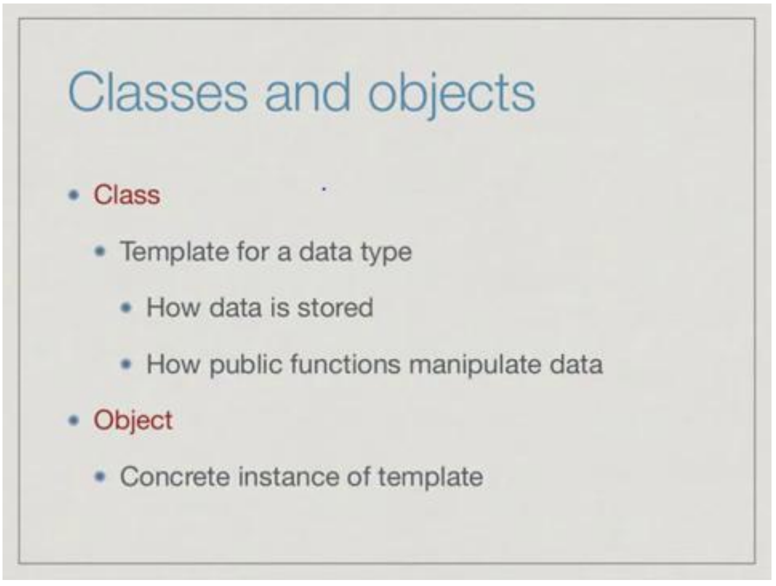
Programming Data Structures and Algorithms in Python

Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 07
Lecture - 02
Classes and Objects in Python

(Refer Slide Time: 00:02)

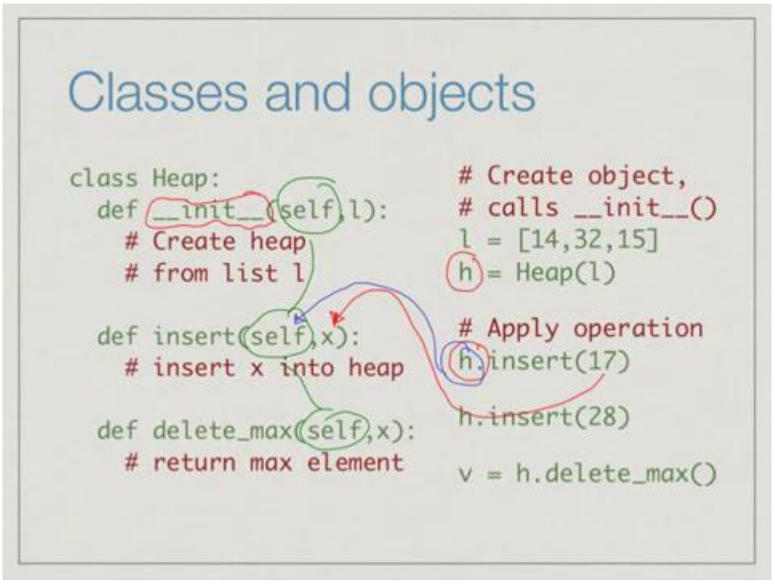


Classes and objects

- Class
  - Template for a data type
    - How data is stored
    - How public functions manipulate data
- Object
  - Concrete instance of template

In the lecture, we saw that in object oriented programming we define a data type through a template called a class, which defines the internal data implementation, and the functions that we use to manipulate the data type. And then we create instances of this data type as objects.

(Refer Slide Time: 00:22)



Classes and objects

```
class Heap:
  def __init__(self,l):
    # Create heap
    # from list l

  def insert(self,x):
    # insert x into heap

  def delete_max(self,x):
    # return max element
```

```
# Create object,
# calls __init__()
l = [14,32,15]
h = Heap(l)

# Apply operation
h.insert(17)

h.insert(28)

v = h.delete_max()
```

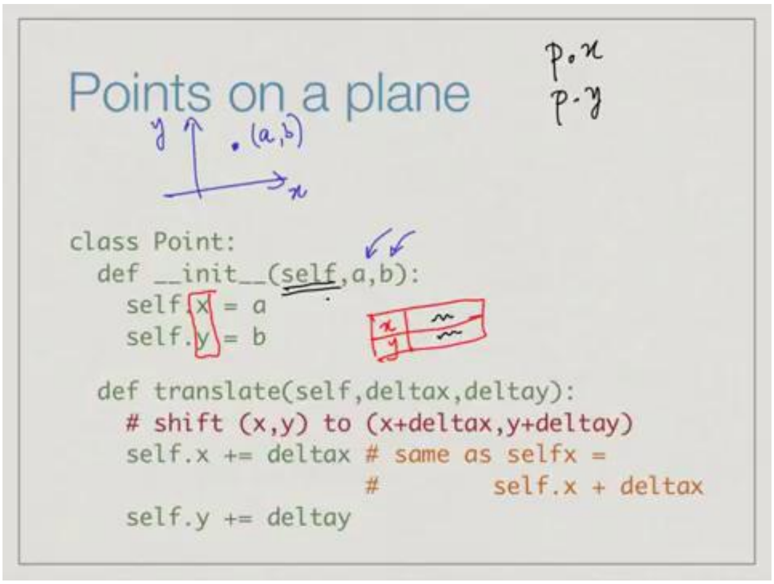
We saw a skeleton implementation of a heap. This had a special function called init, which was used to create the heap, and then we had functions insert and delete. Now one thing which we did not explain is this argument self that run through this. This is a convention which we have in python that every function defined inside a class should have as its first parameter the name self, now it need not be called self, but it is less confusing to always call it, self.

Let us just assume that this parameter is always there and it is called self. Now, what is self, self is a name that is used inside the class to refer to the object that we are currently looking at. For instance, if we are dealing with this heap h, when we say h dot insert then this insert is using the value 17. So, 17 is the value x which is being passed to insert, and h is the value on which the 17 should be added and that is a name for self. So, self in other words, tells the function which object it is operating on itself. It is a name to itself because you are telling a function an object heap h insert 17 into your 'self'.

In that sense, my values are denoted by self and inside a heap, it can may be refer to other heaps. We will see a little later that we can take one value and refer to another value. There will be my values and there will be other values. So, my values are always implicitly called self, because that is the one that is typically manipulated by a function.

To make this a little clearer, let us look at a slightly simpler example than heaps to get all the notations and the terminology correct for us.

(Refer Slide Time: 02:14)



Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    # shift (x,y) to (x+deltax,y+deltay)
    self.x += deltax # same as selfx =
                     #        self.x + deltax
    self.y += deltay
```

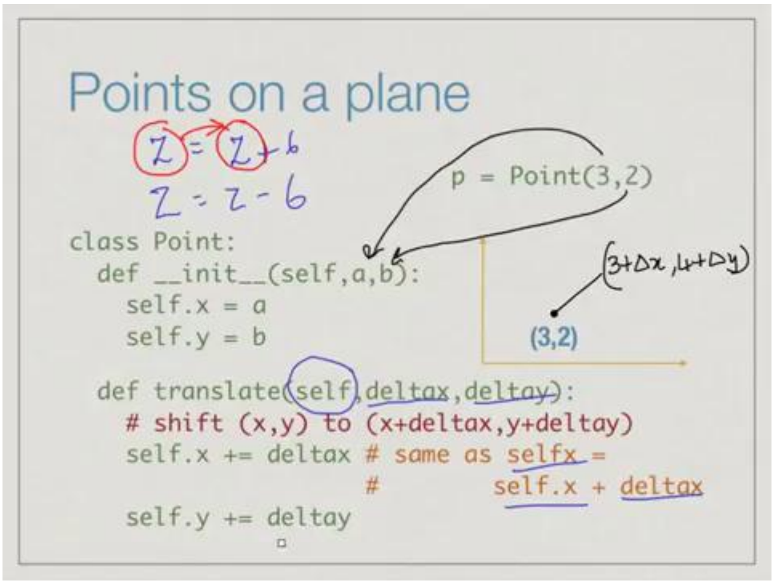
Our first example is just a representation of a point x y. So, we are just thinking about a normal coordinate system, where we have the x-axis, the y-axis. Therefore, a given point is given some coordinate like a comma b. This is a point with x-coordinate a and y-coordinate b, this is a familiar concept that all of you must have seen in mathematics somehow. So, we want to associate with such an object two quantities - the x-coordinate and the y-coordinate and this is set up by the init function by passing the values a and b that you want point to have.

And now we have within the point, we have these two attributes x and y, means every point looks like this. It has an x value and a y value, and this x value is something and the y value is something. And if we change this x and y value, then the point shifts around from one place to another.

Now, in order to designate that the x and y belong to this point and no other, we prefix it by self. So, self dot x refers to the x value within this point myself, self dot y is y value within myself. If you have a generic point p then we have p dot x, p dot y these will refer to the values x and y the names x and y associated with the point p.

Inside the class definition, self refers to the value of the attribute or the name within this particular object. Now this particular object changes as we move from one object to another, but for every object self is itself. So, for p 1 if I tell something about p 1 well in the context of p 1 self is p 1. If I have different point p 2 in the context of p 2, self is p 2. This is an important thing. Just remember that every function inside a class definition should always have the first argument as self and then the actual argument. So, init in this case takes two arguments, but we always insert a third argument.

(Refer Slide Time: 04:27)



Points on a plane

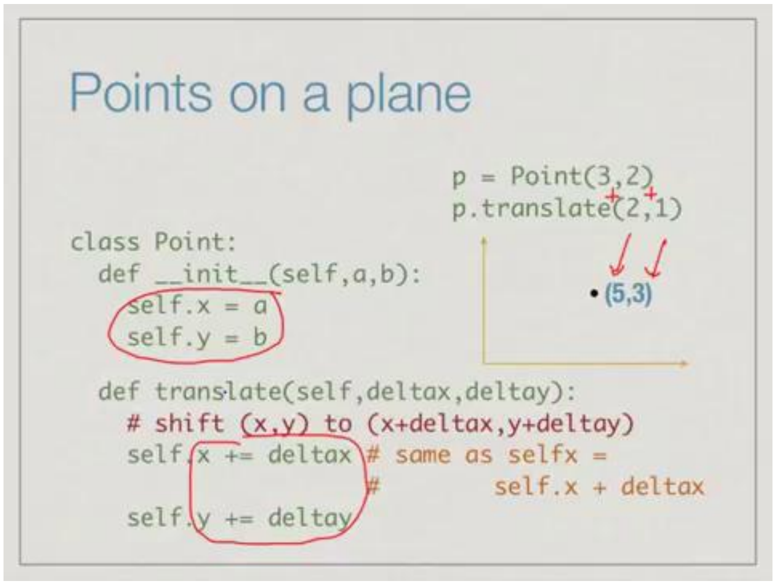
```
                                p = Point(3,2)

class Point:
  def __init__(self,a,b):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    # shift (x,y) to (x+deltax,y+deltay)
    self.x += deltax # same as self.x =
                     #     self.x + deltax
    self.y += deltay
```

Let us look at how this works. For instance, if we say p is equal to point 3, 2; then 3 will be passed as a, and 2 will be passed as b. And this will set up a point that we have drawn here, which internally is represented as self dot x is 3 and self dot y is 2. Now here is a slightly different function, it takes a point and shifts it. So, you want to say shift this to 3 plus delta x and 4 plus delta y. This function we called translate. So, it takes the amount of shift as the argument, delta x and delta y. And as usual we are always providing self as the default first argument.

It just keep this in mind every python function, every python class, if you want to use a function in the object oriented style, the first argument must necessarily be self and then the real arguments. So, what do we want to do when we want to translate a point, we want to take self dot x and move it to self dot x plus the value delta x. So, you want self dot x plus delta x. Now, this is a very common paradigm in python and other programming languages where you want to take a name say z, and then you want to shift it by some amount, say z plus 6 or z is equal to z minus 6.

Whenever we have this kind of a thing where the same name is being updated, there is a short form where we can combine the operation with the assignment. So, self dot x plus equal to delta x is just a short cut in python for self dot x equal to self dot x plus delta x; it means that implicitly the name on the left is the first argument to the operation mentioned along with the assignment operation. This is a very convenient shortcut which allows us to save some typing. Instead of writing self dot x equal to self dot x plus delta x we just say self dot x plus equal to delta x. This shifts the x coordinate of the current point by the argument provided to translate. Similarly, self dot y plus equal to delta y will shift the argument by the amount provided by delta y.

(Refer Slide Time: 06:37)



Points on a plane

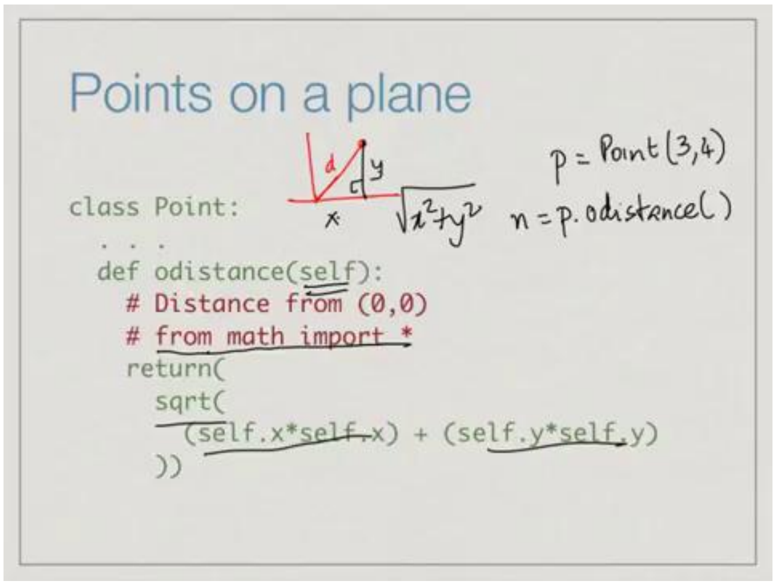
```
p = Point(3,2)
p.translate(2,1)
```

```
class Point:
  def __init__(self,a,b):
    self.x = a
    self.y = b

  def translate(self,deltax,deltay):
    # shift (x,y) to (x+deltax,y+deltay)
    self.x += deltax # same as selfx =
                     #     self.x + deltax
    self.y += deltay
```

For instance, now if we say p dot translate 2 1 then we get a new point which 3 plus 2 5 for the x coordinate and 2 plus 1 3, so this 3 plus 2 gives us 5, and 2 plus 1 gives us 3.

This shifts the point from 3, 2 to 5, 3. This is how we define these internal. The internal implementation is defined inside the init function; this is the function that is called when the point is set up and this associates these internal names x and y with the objects. This is where the implementation really gets set up, and then the functions that we define externally like translate manipulate this internal representation in an appropriate way, so that it changes consistently with what you expect the functions to be.

(Refer Slide Time: 07:24)



Points on a plane

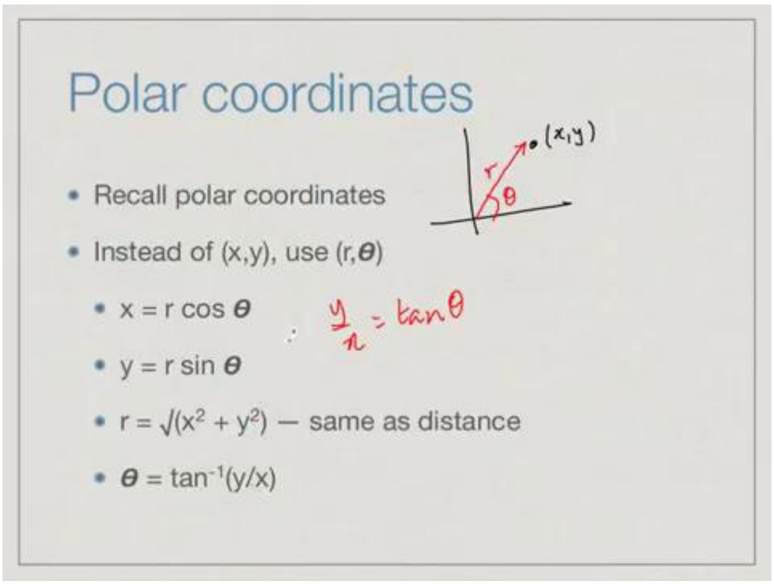
```
class Point:
  . . .
  def odistance(self):
    # Distance from (0,0)
    # from math import *
    return(
      sqrt(
        (self.x*self.x) + (self.y*self.y)
      ))
```

Let us look at different functions. So supposing we want to compute the distance of a point from the origin. So, we want to know what is this distance. This distance by Pythagoras' theorem is nothing but the square root of x square plus y square. So, remember this is like a hypotenuse of a right angled triangle. So, you take a x square plus y square root and you get d.

If you want the distance of a point, we do not give it any arguments, but we always have this default argument self. So, we want to know what is the distance from 0, 0 to the current point. So, we would say something like in our earlier case p is equal to point say 3 comma 4 and then we will say p dot o distance to get its distance. Maybe we would assign this to a name, let us not call it. So, we have might assign this to a name like n right.

So, when we do this, it will look at the current value of self dot x, the current value of self dot y, square them, add them and take the square root. Now one thing to remember is that actually square root is not a function available by default in python. So, you actually have to import the math library. At the top of your class definition, you should have remembered to write from math import star. Assuming that we are done that then square root is defined. This is a typical function which returns some information about the point; the earlier function just translated the point, did not tell us anything; this is the function that returns information about the point.

(Refer Slide Time: 09:01)



Polar coordinates

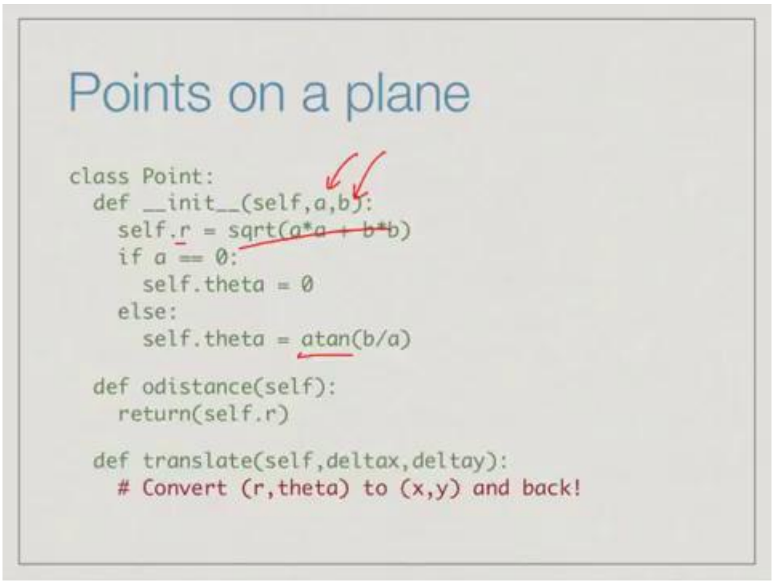
- Recall polar coordinates
- Instead of (x,y), use (r,θ)
  - $x = r \cos \theta$
  - $y = r \sin \theta$
  - $r = \sqrt{(x^2 + y^2)}$ — same as distance
  - $\theta = \tan^{-1}(y/x)$

$\frac{y}{x} = \tan\theta$

Now if o distance is something that we need to do often, then may be it's useful to just keep the point in a different representation. So, you may remember or you may have seen somewhere in high school mathematics that if you take the point x, y then an alternative way of representing where this point is to actually use polar coordinates. So, you can keep this distance and you can keep this angle. So, if I have r and theta it's the same information as keeping x and y.

The connection between the two is that x is r cos theta where cos is the trigonometric cosine function; y is equal to r sin theta. And on the other hand, if I have x and y then I can recover r as we just did for o distance it's the square root of x square plus y square, and theta so y by x is actually if you if you divided y by x you get tan of theta that is because it's sin divided by cos and the r cancels. So, y by x is tan theta, so theta is the tan inverse of y by x. Now speaking of changing implementation, we could change our implementation, so that we do not keep the internal representation in terms of x and y, we actually keep it in terms of r and theta, but the functions remain the same.

(Refer Slide Time: 10:22)



Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.r = sqrt(a*a + b*b)
    if a == 0:
      self.theta = 0
    else:
      self.theta = atan(b/a)

  def odistance(self):
    return(self.r)

  def translate(self,deltax,deltay):
    # Convert (r,theta) to (x,y) and back!
```

For instance, we could take the earlier definition and change it. So, we again pass it x and y. So, from the user's prospective, the user believes that the point is defined in terms of the x and y coordinate, but instead of using a and b as the argument directly to set up the point, we first set up the r - the radius by taking square root of a square plus b square.

And then depending, so we want to divide b by a, but if a is 0, then we have a special case b by a will give us an error. So, if a is 0, we set the angle to be 0; otherwise, this is the python function in the math library for tan inverse, arc tan, we set theta to be the arc tan b minus a b divided a. So, we internally manipulate the x y version to r theta using the same formula that we had shown before which is that r is square root of x square plus y square and theta is tan inverse of y by x. Only thing we have to take care is when x is equal to 0, we have to manually set theta to 0.

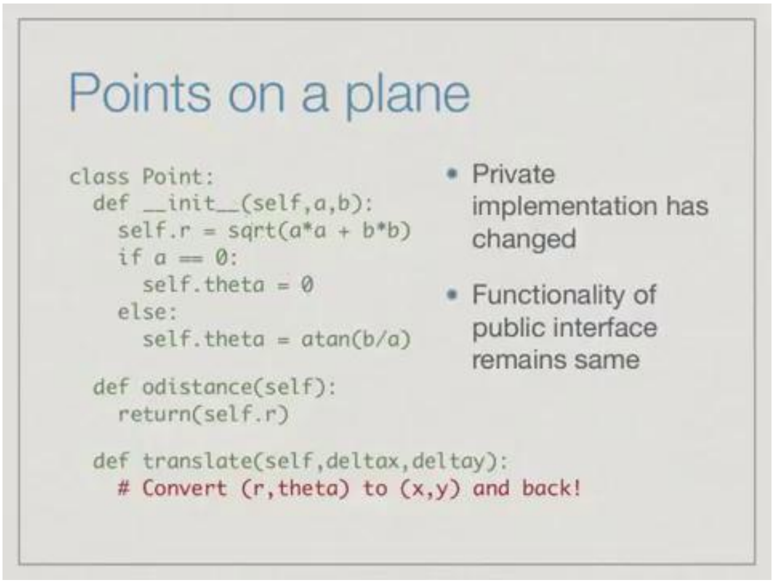
Now internally we are now keeping self dot r and self dot theta. We are not keeping self dot x and self dot y. This is useful because if you want the o distance - the origin distance we just have to return the r value we do not do any computation. So, in other words if we are going to use o distance very often then it is better to use the calculation square root a square plus b square once at the beginning when we setup the point, and just return the r value without any calculation whenever you want the distance from the origin.

This might be a requirement depending on how you are using it and one implementation may be better than the other, but from the user's perspective the same function is there there is self there is o distance. So, if I take a point and I ask for o distance I get the distance from the origin whether or not the point is represented using x y or r theta.

Now, of course, using o distance is r theta is good for o distance not very good for translate. If I want to translate the point by delta x delta y, I have to convert the point back from r theta to x, y; using x equal to r cos theta and y equal to r sin theta then do x plus delta x, y to plus delta y and convert it back to r theta right. So, you pay a price in one function or the other; with the x y representation translate is better; with the r theta representation o distance is better. And this is a very typical case of the kind of compromise that you have to deal with and you have to decide which of these operations is slightly to be more common and more useful for you to implement directly.

If you think translate happens more often it's probably better to use x and y; if you think origin from the distance is more important, so probably better to use r and theta. So, often there is no one good answer. It is not like saying that a heap implementation is always better than a sorted list implementation for a priority queue. There may be tradeoffs which depend on the type of use they are going to put a data structure to as to which internal implementation works worst best, but what you have to always keep in mind is that the implementation should not change the way the functions behave. To the external user, function must behave exactly the same way.

(Refer Slide Time: 13:31)



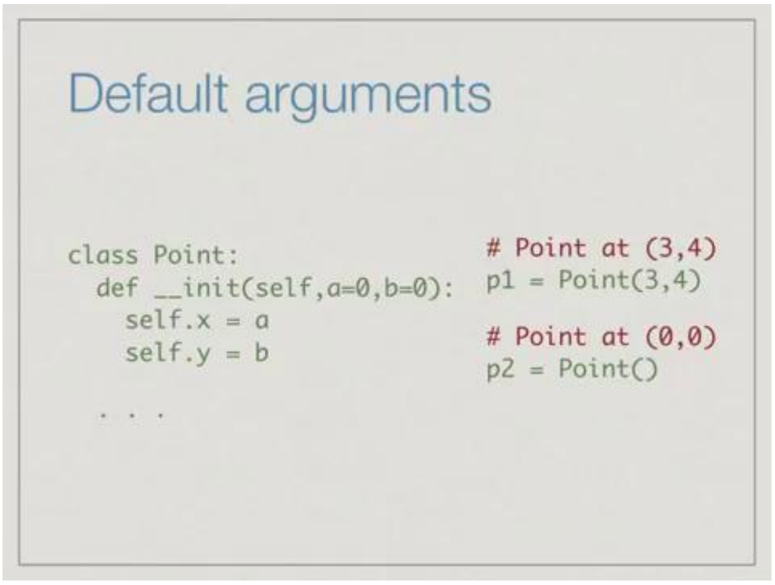
Points on a plane

```
class Point:
  def __init__(self,a,b):
    self.r = sqrt(a*a + b*b)
    if a == 0:
      self.theta = 0
    else:
      self.theta = atan(b/a)

  def odistance(self):
    return(self.r)

  def translate(self,deltax,deltay):
    # Convert (r,theta) to (x,y) and back!
```

- Private implementation has changed
- Functionality of public interface remains same

In this particular example just to illustrate what we have seen, again. We have changed the private implementation, namely we have moved from x, y to r theta, but the functionality of the public interface the functions o distance translate etcetera remain exactly the same.

(Refer Slide Time: 13:48)



Default arguments

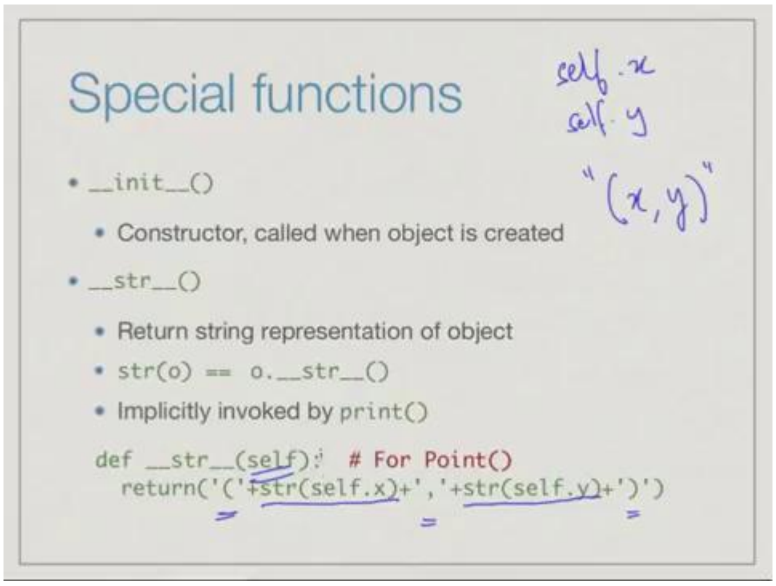
```
class Point:
  def __init(self,a=0,b=0):
    self.x = a
    self.y = b

  . . .
```

```
# Point at (3,4)
p1 = Point(3,4)

# Point at (0,0)
p2 = Point()
```

Now we have seen earlier that in python functions, we can provide default arguments which make sometimes the argument optional. So, for instance, if you want to say that if we do not specify the x and y coordinates over a point then by default the point will be created at the origin. Then we can use a equal to 0, and b equal to 0 as default arguments, so that if the user does not provide values for a and b, then x will be set to 0 and y will be set to 0.

For instance, if we want to point at a specific place 3 comma 4, we would invoke this function this class, we create an object by passing the argument 3 comma 4, but if we do not pass any argument like p 2 then we get a point at the origin.

(Refer Slide Time: 14:35)



Special functions

- `__init__()`
  - Constructor, called when object is created
- `__str__()`
  - Return string representation of object
  - `str(o) == o.__str__()`
  - Implicitly invoked by `print()`

```
def __str__(self):   # For Point()
  return('('+str(self.x)+','+str(self.y)+')')
```

self.x
self.y
"(x, y)"

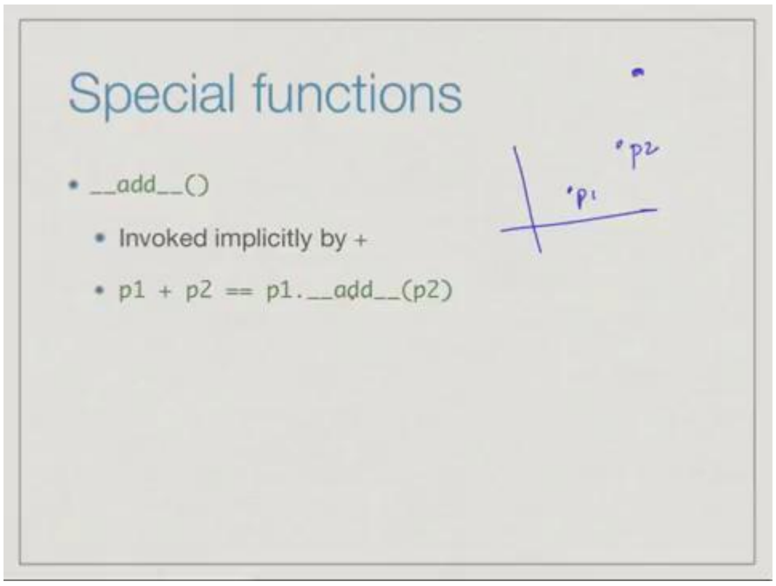
The function init clearly looks like a special function because of these underscore underscore on either side which we normally do not see when we or normally do not thing of using to write a python function. As we said before python interprets init as a constructor, so when we call a object like p equal to 0.54, then this implicitly calls init and init is used to set up self dot x self dot y. The internal representation of the point is set up in the correct way by init. So, init is a special function. Now python has other special functions.

For instance, one of the common things that we might want to do is to print out the value of an object, what does an object contain. And for this the most convenient way is to convert the object to a string. The function str normally converts an object to a string, but how do we describe how str should behave for an object, well there is special function that we can write called underscore underscore str. So, underscore underscore str is implicitly invoke when we write str of o. So, str of an object o is nothing but o dot underscore underscore str.

And for instance, print - the function print implicitly takes any name you pass to print and converts it to a string represent, when I say print x and x is an integer implicitly str of x is what is displayed. So, str is invoked and str in turn internally invokes this special function underscore underscore str. Let us see how this would work for instance for our point thing. So, if we want to represent the points so internally we are self dot x and self dot y, we want to print this out in this form value x and the value y.

So, what we do? We set up str, so remember that self is always a parameter. So, what it does is, it first creates a string with the open bracket and the close bracket either end and a comma in the middle; and in between the open bracket and the comma it puts the string representation of self dot x and in between the comma and the close bracket it produces the string representation of self dot y. This creates a string from the value, it internally invokes str on the values themselves self dot x and self dot y and then constructs these extra things the open close bracket and the comma to make it look like a point as we would expect.

(Refer Slide Time: 16:55)



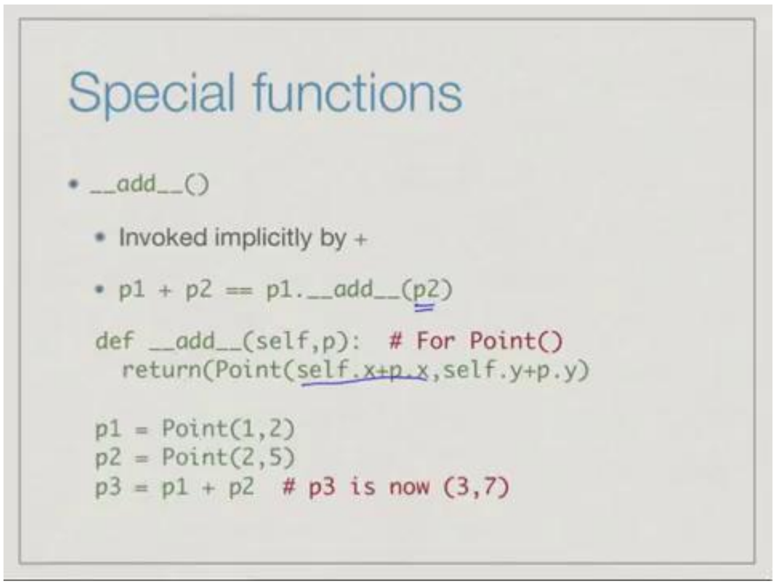
Special functions

- __add__()
    - Invoked implicitly by +
    - p1 + p2 == p1.__add__(p2)

Another interesting function that python has as a special function is add. So, when we write plus the function add is invoked. In other words p 1 plus p 2, if these are two points would invoke p 1 underscore underscore add p 2. So, what we would expect that if I had p 1 and if I had p 2 then I would get something which gives me a point where I combine these two.

So, I get the x coordinate as x 1 p 1 plus p 2 and y coordinate p 1 plus p 2. It's up to us. I mean it does not mean it has to be this way, but if I say p 1 p 2 the function that is invoked is add. And it is up to us define what add means. Let us assume that we want to construct a new point whose x coordinate and y coordinate is the sum of the two points given to us.

(Refer Slide Time: 17:42)



Special functions

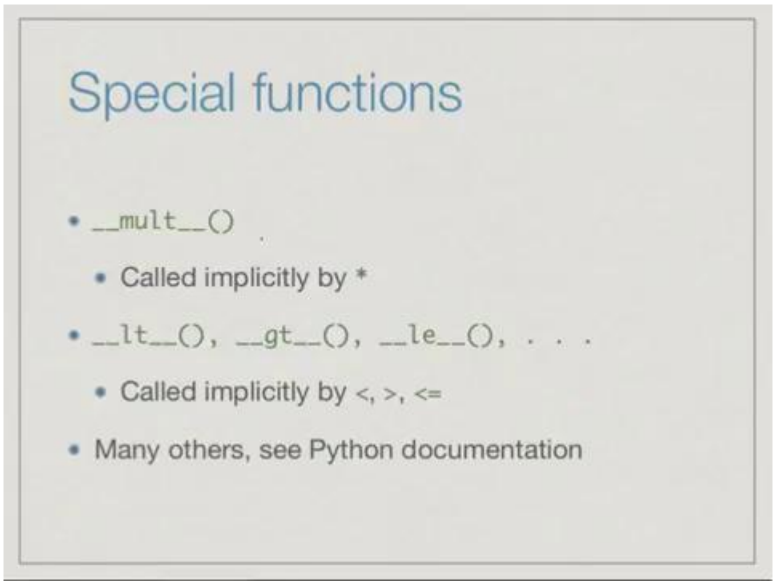
- __add__()
  - Invoked implicitly by +
  - p1 + p2 == p1.__add__(p2)

```
def __add__(self,p):  # For Point()
  return(Point(self.x+p.x,self.y+p.y))

p1 = Point(1,2)
p2 = Point(2,5)
p3 = p1 + p2  # p3 is now (3,7)
```

Here is way we would do it; we would create a new point whose x coordinate is self dot x plus p dot x. Now, notice that, self is the function associated with p 1 in this case, add; so self refers to p 1. When I say p 1 plus p 2 and the other argument p 2 is the point p. So, I can look at the values of p and say p dot x p dot y, I can look at my value and say self dot x self dot y, and now I can combine these by creating a new point self dot x plus p dot x and self dot x plus p dot y.

For instance, if we have two points at 1, 2 and 2, 5 then this will return a point at 3, 7 and I must store it in new point, so p 3 now becomes a new point whose x coordinate is 3 and y coordinate is 7.

(Refer Slide Time: 18:30)



Special functions

- `__mult__()`
  - Called implicitly by *
- `__lt__(), __gt__(), __le__(), . . .`
  - Called implicitly by <, >, <=
- Many others, see Python documentation

In the same way, we could have a special way of defining multiplication, and in python the underscore underscore mult function is the one that is implicitly called by multiply. Similarly, we can define comparisons; we might say what is to be done when we compare whether p 1 is less than p 2, do we check both coordinates are less, we check the distance from the origin is less; we have complete freedom how to define this.

We just write p 1 less then p 2 for readability enough in our program, and internally it will call a function less than lt, similarly greater than will call the function gt, and there is something for less than equal to greater then equal to and so on. These are all very convenient functions that python allows us to call implicitly, and allows us to use conventional operators and conventional functions in our code. So, we do not really have to think of these objects in a different way when we are coding our programs.

There are several other such functions; it is impossible to list all of them and it is not useful to list all of them at this introductory stage when you are learning python, but you can always look up the python documentation, and see all the special functions that are defined for objects and classes.