The **point** class

- By creating the **point** class, we are essentially adding a new data type called **point** to Python.

- We can then define objects belonging to the **point** class (i.e., we can define variables of type **point**).

- A typical class specifies
  - a collection of data and
  - a collection of methods (functions).

- In the case of the point class, the data is simply an $x$-coordinate and the $y$-coordinate.

- The methods are what we might want to use to manipulate a point.

- Thus a class can be viewed as a way of packaging a collection of data and providing ways to modify the package.
# Definition of the point class

class point():

    # This is the initializing method or constructor for the class.  
    # Most classes will have one or more constructor methods.  
    # Examples: p = point(5, 7) will call this method to construct  
    # an instance p of the point class.
    def __init__(self, a, b):
        self.x = a
        self.y = b
The initialization method

- Most classes will have a special method (function) `__init__` called the *initialization method* that will be called whenever we want to create a `point` object.

- The function header is:
  ```python
  __init__(self, a, b):
  ```

- This method is called as `p = point(10, 12)`. The argument 10 corresponds to parameter `a`, the argument 12 corresponds to parameter `b`.

- There is no argument corresponding to `self`. `self` is a Python keyword that refers to the object being created.

- We use two pieces of data, a variable `x` and a variable `y`, in the `point` class.

- In side the method, these two pieces of data are assigned values `a` and `b` respectively.

- Initialization methods are also called *constructors*. 
Methods in the point class

Here are function headers for some of the methods in the point class.

- def translateX(self, a):
- def translateY(self, a):
- def distance(self, p):

These are called using the “dot” syntax such as p.translateX(10)

Here p corresponds to self in the parameter list and 10 corresponds to a.
Operator overloading refers to situations in which the same operator has different meanings.

We have already seen operator overloading for “+” because this refers to numeric addition as well as string concatenation.

Python provides names for operators that we can use to overload them: `__add__`, `__sub__`, `__mul__`, etc.

These names can be used instead of the actual operators. Try:

```
p = 10
p.__add__(2)
```

Look at Section 3.4.8 in Python 2 documentation for the complete list.
def __add__(self, other):
    return point(self.x + other.x, self.y + other.y)

def __mul__(self, other):
    return self.x*other.x + self.y*other.y

• In the definition of __add__, we call the initialization method to construct a point object before returning it.

• These methods are called as:

    p = point(10, 12)
    q = point(-1, 10)
    r = p + q
    print p * q
The `__repr__()` method

- This returns the “official” string representation of an object of the class.
- Try deleting this method from the point class and then try:
  ```python
  p = point(10, 20)
  p
  ```
- There is a related method called `__str__()` that behaves similarly. We will not discuss this here.
A second example: the *queue* class

- Our goal is to define a class called *queue* that can be used to represent a collection of items.
- We want to support two methods:
  - *join*: which is for an item to join the queue
  - *leave*: which is for an item that has been longest in the queue to leave.
- Here is how we want to use this class:
  ```python
  >>> Q = queue()
  >>> Q.join(10)
  >>> Q.join(20)
  >>> Q.leave()
  >>> 10
  ```