Suppose your program needs to maintain millions of polygons.

This is something that graphics programs might have to do because complicated scenes are often constructed using polygons.

Each polygon has a number of attributes:
- Number of points (vertices) in the polygon,
- List of the vertices in the polygon in, say clockwise, order,
- Colors of the vertices and colors of the line segments (edges) connecting consecutive vertices,
- Whether the interior is transparent or not....
An object-oriented programming language allows us to package all of these attributes of a polygon together into an object.

We could then also define functions (or methods) that operate on the polygon object.

For example:
- deleteVertex, addVertex
- rotatePolygon, translatePolygon,
- ...
Built-in Objects in Python

- We have already seen examples of built-in objects in Python: strings, lists, etc.
- **Example:**
  
  ```python
  L = [3, 2, 9]
  L.append(10)
  ```
- This defines an *object* called L of *class* list. Then it applies the *method* append to L.
- L is a “package” consisting of the list items along with other information about the list (e.g., its length).
Is this just new jargon for stuff you already know?

- To some extent, the answer is yes.
- Specifically:
  - class = data type,
  - object = variable,
  - method = function
- So by defining a class, you are essentially extending the language by defining a new data type.
- **Example:** By defining a class called `polygon` you have created a new data type called `polygon`. You can then objects (variables) of class (type) `polygon`. 
Motivation

- Efficiency, with respect to running time and memory usage is one important focus of programmers.
- Another important focus is *maintainability*.
- As software sizes grow into millions of lines (e.g., Microsoft Windows OS) of code we want to ensure:
  - Smooth transition from one version to the next
  - Smooth transition when software engineers leave the project and new engineers join the project
- Object-oriented programming is one approach to programming in a disciplined manner.
Motivation

- By defining the class polygon and methods that operate on instances of the polygon class, you are making a commitment that:
  - Objects of the polygon class can be accessed using a certain syntax (e.g., `P.deleteVertex(q)`).
  - The methods have certain specified behaviors.
- The internal implementation of the class might change a lot over time, but the interface and external behavior remains largely static.
- This means that other code that depends on the polygon class will not suddenly stop working because the internals of the polygon class have changed.
A Brief History

- Objects, classes, etc., as a formal notion in programming we introduced in the 60s in a programming language called Simula 67.

- SmallTalk was designed in the 70s at Xerox Parc and it refined notions introduced in Simula 67.

- In the 90s, object-oriented programming reached a wide audience with the introduction of C++ and then Java.

- Object-oriented programming is nicely suited for programming Graphics User Interfaces (GUIs). With the rise of GUIs, object-oriented programming languages have stayed popular.

- Now we have “hybrid” programming languages such as Python, that allow different styles of programming (e.g., procedural, functional, object-oriented, etc.)
Example: `point` class

- We want to define a class called `point`.
- Each object of this class represents a point in 2-dimensional Euclidean space.
- We want to be able to write code such as:

```python
p = point(10, 20)
q = point(20, 30)
r = p * q
p.translateX(30)
print p
print p.distance(q)
```
Review of this code

```
p = point(10, 20)
q = point(20, 30)
```

- Here we define two objects (variables) of class (type) `point`.
  (This is similar to assignment `x = 10` or `L = [3, 4, 1, 7].`)
- We need code inside the `point` class to allow this type of initialization.
We need code in the **point** class to define the “*” for point objects.

Suppose that we want the “*” operator to mean dot-product of two points; thus, this evaluates to a number (scalar).

When we define a class, we will often **overload** operators to work for objects in the new class.
p.translateX(30)
print p
print p.distance(q)

- We need code for two methods (functions) in the `point` class, namely `translateX` and `distance`.

- We also need code that specifies how we want a point to appear when it is printed.
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.
The initialization method

# 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:
```python
p = 10
p.__add__(2)
```

Look at Section 3.4.8 in Python 2 documentation for the complete list.