Overloading Operators and Inheritance
Overloaded operators

- An operator (e.g., +) is *overloaded* if it has different meanings depending on the context in which it appears.
- **Example:** $8 + 2 = 10$, “8” + “2” = “82”, $[8] + [2] = [8, 2]$.

- When we define new classes, we might want to overload operators so as to give familiar operators new meaning in the presence of instances of the defined class.
- **Example:** Suppose $p = \text{point}(3, 2)$ and $q = \text{point}(1, 4)$. We might want to interpret $p + q$ as “pointwise” addition and require $p + q$ to evaluate to a point $(4, 6)$. 
Python allows users to overload operators

- Section 3.4.8 in the Python reference manual lists a bunch of built-in methods. Some of these are:
  - `object.__add__(self, other)`
  - `object.__sub__(self, other)`
  - `object.__mul__(self, other)`
  - `object.__mod__(self, other)`

- These correspond to the familiar binary, numeric operators `+`, `-`, `*`, and `%.

- When we use one of these operators, it has the effect of calling (behind the scenes) one of the above methods.

- By redefining these methods within a user-defined class, we can overload standard Python operators.
point class revisited

- We can add the following methods to the point class:

  ```python
def __add__(self, p):
    return pointWithOperators(self.x + p.x, self.y + p.y)

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

- The class can now be used as follows:

  ```python
>>> p = pointWithOperators(1, 4)
>>> q = pointWithOperators(2, 4)
>>> p
(1, 4)
>>> q
(2, 4)
>>> p + q
(3, 8)
>>> p * q
18
```
Python documentation tells us that there are built-in Python methods corresponding to all kinds of operators including:

- comparison operators (e.g., `<`, `>`, etc.),
- indexing operator (e.g., `L[4]`),
- slicing operator (e.g., `L[3:5]`), etc.

These make the language extremely flexible and powerful.
Another powerful mechanism that is usually associated with classes is the notion of **inheritance**.