

# Object-Oriented Programming



**Kenn H. Kim, Ph. D.**

School of Business  
Clemson University

# Object-Oriented Programming

- Defining new Python Classes
- Container Classes
- Overloaded Operators

## Recap & Learning Objectives

How to implement new Python classes

Introduce Object-Oriented Programming (OOP)

Why class?

- Make an application program more intuitive
- Make it easier to develop, debug, read, and maintain
- Encapsulate (hide) its implementation (data, methods)

OOP is a software development paradigm

That achieves modularity and code portability

By organizing programs around components (classes and objects)

3

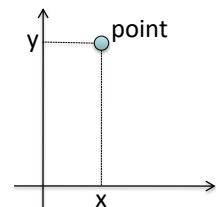
## A New Class: Point

Suppose we would like to have a class that represents points on a plane

- for a graphics app, say

Let's first informally describe how we would like to use this class

```
>>> point = Point()
>>> point.setx(3)
>>> point.sety(4)
>>> point.get()
(3, 4)
>>> point.move(1, 2)
>>> point.get()
(4, 6)
>>> point.setx(-1)
>>> point.get()
(-1, 6)
>>>
```



4

## A New Class: Point

Suppose we would like to have a class that represents points on a plane

- for a graphics app, say

Let's first informally describe how we would like to use this class

```
>>> point = Point()  
>>> point.setx(3)  
>>> point.sety(4)  
>>> point.get()  
(3, 4)  
>>> point.move(1, 2)  
>>> point.get()  
(4, 6)  
>>> point.setx(-1)  
>>> point.get()  
(-1, 6)  
>>>
```

Usage	Explanation
p.setx(xcoord)	Sets the x coordinate of point p to xcoord
p.sety(ycoord)	Sets the y coordinate of point p to ycoord
p.get()	Returns the x and y coordinates of point p as a tuple (x, y)
p.move(dx, dy)	Changes the coordinates of point p from the current (x, y) to (x+dx, y+dy)

How do we create this new class Point?

5

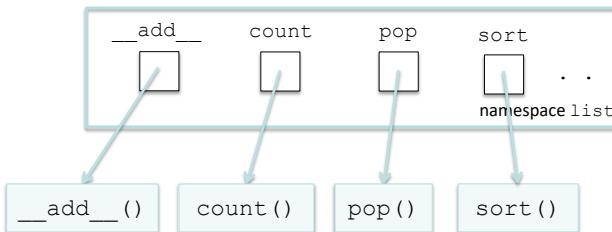
## A Class is a Namespace (REVIEW)

A class is really a namespace

- The name of this namespace is the name of the class
- The names defined in this namespace are the class attributes (e.g., class methods)
- The class attributes can be accessed using the standard namespace notation

```
>>> list.pop  
<method 'pop' of 'list' objects>  
>>> list.sort  
<method 'sort' of 'list' objects>  
>>> dir(list)  
['__add__', '__class__',  
...  
'index', 'insert', 'pop', 'remove',  
'reverse', 'sort']
```

Function `dir()` can be used to list the class attributes



6

## Class Methods (REVIEW)

A class method is really a function defined in the class namespace; when Python executes

`instance.method(arg1, arg2, ...)`

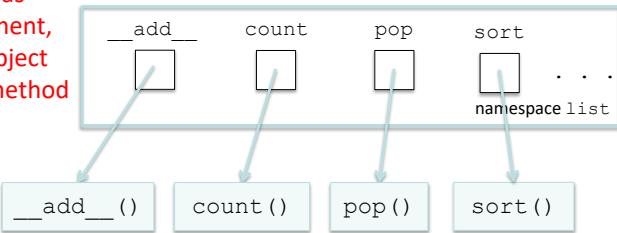
it first translates it to

`class.method(instance, arg1, arg2, ...)`

and actually executes this last statement

The function has  
an extra argument,  
which is the object  
invoking the method

```
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> lst.sort()
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> list.sort(lst)
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst.append(6)
>>> lst
[1, 2, 3, 7, 8, 9, 6]
>>> list.append(lst, 5)
>>> lst
[1, 2, 3, 7, 8, 9, 6, 5]
```



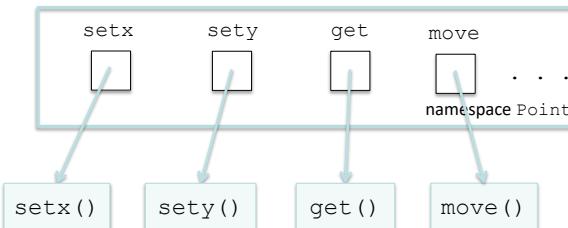
7

## Developing the Class Point

A namespace called Point needs to be defined

Namespace Point will store the names of the 4 methods (the class attributes)

Usage	Explanation
<code>p.setx(xcoord)</code>	Sets the x coordinate of point p to xcoord
<code>p.sety(ycoord)</code>	Sets the y coordinate of point p to ycoord
<code>p.get()</code>	Returns the x and y coordinates of point p as a tuple (x, y)
<code>p.move(dx, dy)</code>	Changes the coordinates of point p from the current (x, y) to (x+dx, y+dy)



8

## Defining the Class Point

A namespace called Point needs to be defined

Namespace Point will store the names of the 4 methods (the class attributes)

Each method is a function that has an extra (first) argument which refers to the object that the method is invoked on

Usage	Explanation
setx(p, xcoord)	Sets the x coordinate of point p to xcoord
sety(p, ycoord)	Sets the y coordinate of point p to ycoord
get(p)	Returns the x and y coordinates of point p as a tuple (x, y)
move(p, dx, dy)	Changes the coordinates of point p from the current (x, y) to (x+dx, y+dy)

```
>>> Point.get(point)
(-1, 6)
>>> Point.setx(point, 0)
>>> Point.get(point)
(0, 6)
>>> Point.sety(point, 0)
>>> Point.get(point)
(0, 0)
>>> Point.move(point, 2, -2)
>>> Point.get(point)
(2, -2)
```

9

## Defining the Class Point

A namespace called Point needs to be defined

Namespace Point will store the names of the 4 methods (the class attributes)

Each method is a function that has an extra (first) argument which refers to the object that the method is invoked on

variable that refers to the object on which the method is invoked

```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        # to be implemented

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        # to be implemented

    def get(self):
        'return coordinates of the point as a tuple'
        # to be implemented

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        # to be implemented
```

The Python **class** statement defines a new class (and associated namespace)

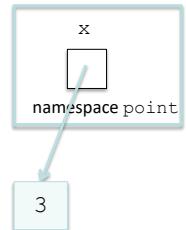
10

## The Object Namespace

We know that a namespace is associated with every class

A namespace is also associated with every object

```
>>> point = Point()
>>> Point.setx(point, 3)
>>>
```



```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        # to be implemented

    def get(self):
        'return coordinates of the point as a tuple'
        # to be implemented

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        # to be implemented
```

The Python **class** statement defines a new class (and associated namespace)

11

## Defining the Class Point

A namespace called Point needs to be defined

Namespace Point will store the names of the 4 methods (the class attributes)

Each method is a function that has an extra (first) argument which refers to the object that the method is invoked on

```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

12

## Exercise (Notebook)



Add new method `getx()`  
to class Point

```
>>> point = Point()  
>>> point.setx(3)  
>>> point.getx()  
3
```

13

## Exercise (Notebook)



Add new method `getx()`  
to class Point

```
>>> point = Point()  
>>> point.setx(3)  
>>> point.getx()  
3
```

```
class Point:  
    'class that represents a point in the plane'  
  
    def setx(self, xcoord):  
        'set x coordinate of point to xcoord'  
        self.x = xcoord  
  
    def sety(self, ycoord):  
        'set y coordinate of point to ycoord'  
        self.y = ycoord  
  
    def get(self):  
        'return coordinates of the point as a tuple'  
        return (self.x, self.y)  
  
    def move(self, dx, dy):  
        'change the x and y coordinates by dx and dy'  
        self.x += dx  
        self.y += dy  
  
    def getx(self):  
        'return x coordinate of the point'  
        return self.x
```

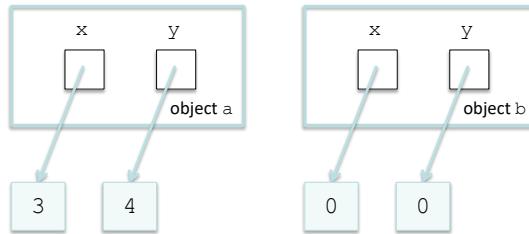
14

## The Instance Namespaces

Variables stored in the namespace of an object (instance) are called **instance variables (or instance attributes)**

Every object will have **its own namespace** and therefore **its own instance variables**

```
>>> a = Point()
>>> a.setx(3)
>>> a.sety(4)
>>> b = Point()
>>> b.setx(0)
>>> b.sety(0)
>>> a.get()
(3, 4)
>>> b.get()
(0, 0)
>>> a.x
3
>>> b.x
0
>>>
```



15

## The Class and Instance Attributes

An instance of a class **inherits all the class attributes**

```
>>> dir(a)
['__class__', '__delattr__',  
 '__dict__', '__doc__',  
 '__eq__', '__format__',  
 '__ge__', '__getattribute__',  
 '__gt__', '__hash__',  
 '__init__', '__le__',  
 '__lt__', '__module__',  
 '__ne__', '__new__',  
 '__reduce__', '__reduce_ex__',  
 '__repr__', '__setattr__',  
 '__sizeof__', '__str__',  
 '__subclasshook__',  
 '__weakref__', 'get', 'move',  
 'setx', 'sety', 'x', 'y']
```

class Point attributes inherited by a



Function `dir()` returns the attributes of an object, including the inherited ones

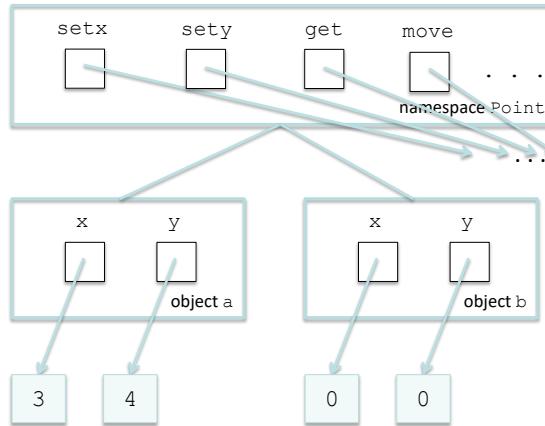
16

## The Class and Instance Attributes

Method names `setx`, `sety`, `get`, and `move` are defined in namespace `Point`  
• not in namespace `a` or `b`.

Python does the following when evaluating expression `a.setx`:

1. It first attempts to find name `setx` in object (namespace) `a`.
2. If name `setx` does not exist in namespace `a`, then it attempts to find `setx` in namespace `Point`



17

## Class Definition, in General

```
class <Class Name>:  
    <class variable 1> = <value>  
    <class variable 2> = <value>  
    ...  
    def <class method 1>(self, arg11, arg12, ...):  
        <implementation of class method 1>  
  
    def <class method 2>(self, arg21, arg22, ...):  
        <implementation of class method 2>  
  
    ...
```

18

## Class Definition, in General

```
class Point:

    def setx(self, xcoord):
        self.x = xcoord

    def sety(self, ycoord):
        self.y = ycoord

    def get(self):
        return (self.x, self.y)

    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

Note: no documentation

19

## (No) Class Documentation

```
>>> help(Point)
Help on class Point in module __main__:

class Point(builtins.object)
| Methods defined here:
|
|   get(self)
|
|   move(self, dx, dy)
|
|   setx(self, xcoord)
|
|   sety(self, ycoord)
|
| -----
|
| Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
```

20

## Class Documentation



```
class Point:  
    'class that represents a point in the plane'  
  
    def setx(self, xcoord):  
        'set x coordinate of point to xcoord'  
        self.x = xcoord  
  
    def sety(self, ycoord):  
        'set y coordinate of point to ycoord'  
        self.y = ycoord  
  
    def get(self):  
        'return coordinates of the point as a tuple'  
        return (self.x, self.y)  
  
    def move(self, dx, dy):  
        'change the x and y coordinates by dx and dy'  
        self.x += dx  
        self.y += dy
```

21

## Class Documentation



```
>>> help(Point)  
Help on class Point in module __main__:  
  
class Point(builtins.object)  
|   class that represents a point in the plane  
|  
| Methods defined here:  
|  
|   get(self)  
|       return a tuple with x and y coordinates of the point  
|  
|   move(self, dx, dy)  
|       change the x and y coordinates by dx and dy  
|  
|   setx(self, xcoord)  
|       set x coordinate of point to xcoord  
|  
|   sety(self, ycoord)  
|       set y coordinate of point to ycoord  
|  
|   ...  
-  
| Data descriptors defined here:  
...
```

22

## Exercise (Notebook)

Develop class Animal that supports methods:

- setSpecies(species)
- setLanguage(language)
- speak()

```
>>> snoopy = Animal()
>>> snoopy.setSpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
I am a dog and I bark.
```

23

## Exercise (Notebook)

Develop class Animal that supports methods:

- setSpecies(species)
- setLanguage(language)
- speak()

```
>>> snoopy = Animal()
>>> snoopy.setSpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
I am a dog and I bark.
```

```
class Animal:
    'represents an animal'

    def setSpecies(self, species):
        'sets the animal species'
        self.spec = species

    def setLanguage(self, language):
        'sets the animal language'
        self.lang = language

    def speak(self):
        'prints a sentence by the animal'
        print('I am a {} and I {}'.format(self.spec, self.lang))
```

24

## Overloaded Constructor (Notebook)

It takes 3 steps to create a Point object at specific x and y coordinates

It would be better if we could do it in one step

```
>>> a = Point()
>>> a.setx(3)
>>> a.sety(4)
>>> a.get()
(3, 4)
>>>
```

```
>>> a = Point(3, 4)
>>> a.get()
(3, 4)
>>>
```

called by Python each time a Point object is created

```
class Point:
    'class that represents a point in the plane'

    def __init__(self, xcoord, ycoord):
        'initialize coordinates to (xcoord, ycoord)'
        self.x = xcoord
        self.y = ycoord

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

25

## Default Constructor



Problem: Now we can't create an uninitialized point

```
>>> a = Point()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a = Point()
TypeError: __init__() takes exactly 3 arguments (1 given)
>>>
```

Built-in types support default constructors

```
>>> n = int(3)
>>> n
3
>>> n = int()
>>> n
0
>>>
```

```
class Point:
    'class that represents a point in the plane'

    def __init__(self, xcoord, ycoord):
        'initialize coordinates to (xcoord, ycoord)'
        self.x = xcoord
        self.y = ycoord

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

26

## Default Constructor

Want to augment class Point so it supports a default constructor

```
>>> a = Point()
>>> a.get()
(0, 0)
>>>
```

xcoord is set to 0 if the argument is missing  
ycoord is set to 0 if the argument is missing

```
class Point:
    'class that represents a point in the plane'

    def __init__(self, xcoord=0, ycoord=0):
        'initialize coordinates to (xcoord, ycoord)'
        self.x = xcoord
        self.y = ycoord

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

27

## Exercise (Notebook)

Modify the class Animal we developed in the previous section so it supports a two, one, or no input argument constructor

```
>>> snoopy = Animal('dog', 'bark')
>>> snoopy.speak()
I am a dog and I bark.
>>> tweety = Animal('canary')
>>> tweety.speak()
I am a canary and I make sounds.
>>> animal = Animal()
>>> animal.speak()
I am a animal and I make sounds.
```

28

## Exercise (Notebook)



Modify the class Animal we developed in the previous section so it supports a two, one, or no input argument constructor

```
class Animal:  
    'represents an animal'  
  
    def __init__(self, species='animal', language='make sounds'):  
        self.species = species  
        self.language = language  
  
    def setSpecies(self, species):  
        'sets the animal species'  
        self.spec = species  
  
    def setLanguage(self, language):  
        'sets the animal language'  
        self.lang = language  
  
    def speak(self):  
        'prints a sentence by the animal'  
        print('I am a {} and I {}'.format(self.spec, self.lang))
```

29

## Our Classes Are Not User-friendly



What would we prefer?

```
>>> a = Point(3, 4)  
>>> a  
<__main__.Point object at 0x10278a690>
```

```
>>> a = Point(3, 4)  
>>> a  
Point(3, 4)
```

```
>>> a = Point(3, 4)  
>>> b = Point(1, 2)  
>>> a + b  
Traceback (most recent call last):  
  File "<pyshell#44>", line 1, in <module>  
    a+b  
TypeError: unsupported operand type(s) for  
+: 'Point' and 'Point'
```

```
>>> a = Point(3, 4)  
>>> b = Point(1, 2)  
>>> a+b  
Point(4, 6)
```

```
>>> appts = Queue()  
>>> len(appts)  
Traceback (most recent call last):  
  File "<pyshell#40>", line 1, in <module>  
    len(appts)  
TypeError: object of type 'Queue' has no len()
```

```
>>> appts = Queue()  
>>> len(appts)  
0
```

30

15

## Python Operators

```
>>> 'he' + 'llo'  
'hello'  
>>> [1,2] + [3,4]  
[1, 2, 3, 4]  
>>> 2+4  
6
```

```
>>> 'he'.__add__('llo')  
'hello'  
>>> [1,2].__add__([3,4])  
[1, 2, 3, 4]  
>>> int(2).__add__(4)  
6
```

Operator + is defined for multiple classes; it is an **overloaded operator**.

- For each class, the definition—and thus the meaning—of the operator is different.
  - integer addition for class `int`
  - list concatenation for class `list`
  - string concatenation for class `str`
- How is the behavior of operator + defined for a particular class?

Class method `__add__()` implements the behavior of operator + for the class

When Python evaluates

`object1 + object 2`

... it first translates it to  
method invocation ...

`object1.__add__(object2)`

... and then evaluates  
the method invocation

31

## Python Operators

In Python, all expressions involving operators are translated into method calls

- (Recall that method invocations are then further translated to function calls in a namespace)

```
>>> '!'*10  
'!!!!!!!!!!'  
>>> [1,2,3] == [2,3,4]  
False  
>>> 2 < 5  
True  
>>> 'a' <= 'a'  
True  

```

Operator	Method
<code>x + y</code>	<code>x.__add__(y)</code>
<code>x - y</code>	<code>x.__sub__(y)</code>
<code>x * y</code>	<code>x.__mul__(y)</code>
<code>x / y</code>	<code>x.__truediv__(y)</code>
<code>x // y</code>	<code>x.__floordiv__(y)</code>
<code>x % y</code>	<code>x.__mod__(y)</code>
<code>x == y</code>	<code>x.__eq__(y)</code>
<code>x != y</code>	<code>x.__ne__(y)</code>
<code>x &gt; y</code>	<code>x.__gt__(y)</code>
<code>x &gt;= y</code>	<code>x.__ge__(y)</code>
<code>x &lt; y</code>	<code>x.__lt__(y)</code>
<code>x &lt;= y</code>	<code>x.__le__(y)</code>
<code>repr(x)</code>	<code>x.__repr__()</code>
<code>str(x)</code>	<code>x.__str__()</code>
<code>len(x)</code>	<code>x.__len__()</code>
<code>&lt;type&gt;(x)</code>	<code>&lt;type&gt;.__init__(x)</code>

32

## Python Operators

In Python, all expressions involving operators are translated into method calls

- (Recall that method invocations are then further translated to function calls in a namespace)

```
>> '!'.__mul__(10)
'!!!!!!!!!!!!'
>>> [1,2,3].__eq__([2,3,4])
False
>>> int(2).__lt__(5)
True
>>> 'a'.__le__('a')
True
>>> [1,1,2,3,5,8].__len__()
6
```

Operator	Method
x + y	x.__add__(y)
x - y	x.__sub__(y)
x * y	x.__mul__(y)
x / y	x.__truediv__(y)
x // y	x.__floordiv__(y)
x % y	x.__mod__(y)
x == y	x.__eq__(y)
x != y	x.__ne__(y)
x > y	x.__gt__(y)
x >= y	x.__ge__(y)
x < y	x.__lt__(y)
x <= y	x.__le__(y)
repr(x)	x.__repr__()
str(x)	x.__str__()
len(x)	x.__len__()
<type>(x)	<type>.__init__(x)

33

## Python Operators

Built-in function `repr()` returns the **canonical string representation** of an object

- This is the representation printed by the shell when evaluating the object

```
>>> repr([1,2,3])
'[1, 2, 3]'
>>> repr(193)
'193'
>>> repr(set())
'set()'
```

```
>>> [1,2,3]
[1, 2, 3]
>>> 193
193
>>> set()
set()
```

```
>>> [1,2,3].__repr__()
'[1, 2, 3]'
>>> int(193).__repr__()
'193'
>>> set().__repr__()
'set()'
```

Operator	Method
x + y	x.__add__(y)
x - y	x.__sub__(y)
x * y	x.__mul__(y)
x / y	x.__truediv__(y)
x // y	x.__floordiv__(y)
x % y	x.__mod__(y)
x == y	x.__eq__(y)
x != y	x.__ne__(y)
x > y	x.__gt__(y)
x >= y	x.__ge__(y)
x < y	x.__lt__(y)
x <= y	x.__le__(y)
repr(x)	x.__repr__()
str(x)	x.__str__()
len(x)	x.__len__()
<type>(x)	<type>.__init__(x)

34

## Overloading `repr()`



In Python, operators are translated into method calls

To add an overloaded operator to a user-defined class, the corresponding method must be implemented

To get this behavior

```
>>> a = Point(3, 4)
>>> a
Point(3, 4)
```

```
>>> a = Point(3, 4)
>>> a.__repr__()
Point(3, 4)
```

method `__repr__()` must be implemented and added to class `Point`

`__repr__()` should return the (canonical) string representation of the point

```
class Point:
    # other Point methods here

    def __repr__(self):
        'canonical string representation Point(x, y)'
        return 'Point({}, {})'.format(self.x, self.y)
```

35

## Overloading Operator +



To get this behavior

```
>>> a = Point(3,4)
>>> b = Point(1,2)
>>> a+b
Point(4, 6)
```

```
>>> a = Point(3,4)
>>> b = Point(1,2)
>>> a.__add__(b)
Point(4, 6)
```

method `__add__()` must be implemented and added to class `Point`

`__add__()` should return a new `Point` object whose coordinates are the sum of the coordinates of `a` and `b`

Also, method `__repr__()` should be implemented to achieve the desired display of the result in the shell

```
class Point:
    # other Point methods here

    def __add__(self, point):
        return Point(self.x+point.x, self.y+point.y)

    def __repr__(self):
        'canonical string representation Point(x, y)'
        return 'Point({}, {})'.format(self.x, self.y)
```

36

## Overloading len()

To get this behavior

```
>>> appts = Queue()
>>> len(appts)
0
```

```
>>> appts = Queue()
>>> appts.__len__()
0
```

method `__len__()` must be implemented and added to class `Queue`

`__len__()` should return the number of objects in the queue

- i.e., the size of `list` `self.q`

We use the fact that `len()` is implemented for class `list`

```
class Queue:
    def __init__(self):
        self.q = []

    def isEmpty(self):
        return (len(self.q) == 0)

    def enqueue (self, item):
        return self.q.append(item)

    def dequeue(self):
        return self.q.pop(0)

    def __len__(self):
        return len(self.q)
```

37

## Overloading Operator ==

For user-defined classes, the default behavior for operator `==` is to return `True` only when the two objects are the same object.

```
>>> a = Point(3,5)
>>> b = Point(2,5)
>>> a == b
False
>>> a == a
True
```

Usually, that is not the desired behavior

- It also gets in the way of satisfying the contract between constructor and `repr()`

For class `Point`, operator `==` should return `True` if the two points have the same coordinates

contract between constructor and `repr()` is now satisfied

```
class Point:

    # other Point methods here

    def __eq__(self, other):
        'self == other if they have the same coordinates'
        return self.x == other.x and self.y == other.y
    def __repr__(self):
        'return canonical string representation Point(x, y)'
        return 'Point({}, {})'.format(self.x, self.y)
```

38

# Inheritance (Notebook)

Code reuse is a key software engineering goal

- One benefit of functions is they make it easier to reuse code
- Similarly, organizing code into user-defined classes makes it easier to later reuse the code

A class can also be reused by extending it through **inheritance**

Example: Suppose that we find it convenient to have a class that behaves just like the built-in class `list` but also supports a method called `choice()` that returns an item from the list, chosen uniformly at random.

```
>>> mylst = MyList()
>>> mylst.append(2)
>>> mylst.append(3)
>>> mylst.append(5)
>>> mylst.append(7)
>>> len(mylst)
4
>>> mylst.index(5)
2
>>> mylst.choice()
7
>>> mylst.choice()
3
>>>
```

39

# Implementing Class `MyList`

Example: Suppose that we find it convenient to have a class that behaves just like the built-in class `list` but also supports a method called `choice()` that returns an item from the list, chosen uniformly at random.

Approach 1: Develop class `MyList` from scratch

- Just like class `Queue`

```
import random
class MyList:
    def __init__(self, initial = []):
        self.lst = initial

    def __len__(self):
        return len(self.lst)

    def append(self, item):
        self.lst.append(item)

    # implementations of remaining "list" methods

    def choice(self):
        return random.choice(self.lst)
```

Huge amount of work!

40

## Implementing Class MyList

Example: Suppose that we find it convenient to have a class that behaves just like the built-in class `list` but also supports a method called `choice()` that returns an item from the list, chosen uniformly at random.

Approach 2: Develop class `MyList` by **inheritance** from class `list`

```
Class MyList inherits all the attributes of class list
import random
class MyList(list):
    'a subclass of list that implements method choice'

    def choice(self):
        'return item from list chosen uniformly at random'
        return random.choice(self)
```

41

## Implementing Class MyList

```
import random
class MyList:
    def __init__(self, initial = []):
        self.lst = initial

    def __len__():
        return len(self.lst)

    def append(self, item):
        self.lst.append(item)

    # implementations of remaining "list" methods

    def choice(self):
        return random.choice(self.lst)
```

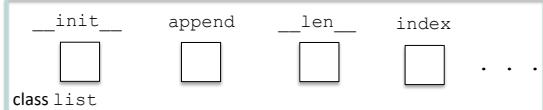
```
import random
class MyList(list):
    'a subclass of list that implements method choice'

    def choice(self):
        'return item from list chosen uniformly at random'
        return random.choice(self)
```

42

## Class MyList by Inheritance

```
>>> dir(MyList)
['__add__', '__class__',
...
'choice', 'count', 'extend',
'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>>
```



Class MyList inherits all the attributes of class list  
Object mylst inherits all the attributes of class MyList (which inherits all the attributes of class list)

```
import random
class MyList(list):
    'a subclass of list that implements method choice'

    def choice(self):
        'return item from list chosen uniformly at random'
        return random.choice(self)
```

```
[2,3,5,7]
object mylst
```

43

## Class Definition, in General

A class can be defined “from scratch” using:

```
class <Class Name>:
```

A class can also be **derived** from another class, through **inheritance**

```
class <Class Name>(<Super Class>):
```

```
class <Class Name>:
```

is a shorthand for

```
class <Class Name>(object):
```

object is a built-in class with no attributes; it is the class that all classes inherit from, directly or indirectly

```
>>> help(object)
Help on class object in module builtins:

class object
|   The most base type
```

A class can also inherit attributes from more than one superclass

```
class <Class Name>(<Super Class 1>, <Super Class 2>, ...):
```

44

## Overriding Superclass Methods (Notebook)

Sometimes we need to develop a new class that can almost inherit attributes from an existing class... but not quite.

For example, a class `Bird` that supports the same methods class `Animal` supports (`setSpecies()`, `setLanguage()`, and `speak()`) but with a different behavior for method `speak()`

```
>>> snoopy = Animal()
>>> snoopy.setSpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
I am a dog and I bark.
>>> tweety = Bird()
>>> tweety.setSpecies('canary')
>>> tweety.setLanguage('tweet')
>>> tweety.speak()
tweet! tweet! tweet!
```

45

## Overriding Superclass Methods (Notebook)

Sometimes we need to develop a new class that can almost inherit attributes from an existing class... but not quite.

For example, a class `Bird` that supports the same methods class `Animal` supports (`setSpecies()`, `setLanguage()`, and `speak()`) but with a different behavior for method `speak()`

```
class Animal:
    'represents an animal'

    def setSpecies(self, species):
        'sets the animal species'
        self.spec = species

    def setLanguage(self, language):
        'sets the animal language'
        self.lang = language

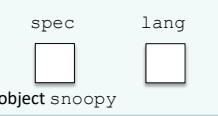
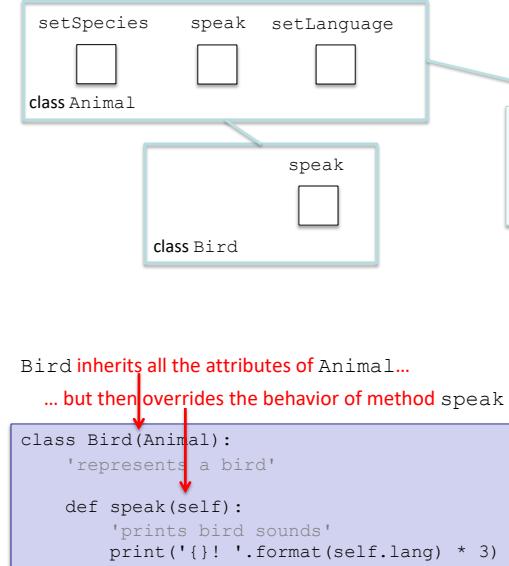
    def speak(self):
        'prints a sentence by the animal'
        print('I am a {} and I {}'.format(self.spec, self.lang))

class Bird(Animal):
    'represents a bird'

    def speak(self):
        'prints bird sounds'
        print('{}! '.format(self.lang) * 3)
```

46

## Overriding Superclass Methods (Notebook)



method speak() defined in Animal is used

Bird inherits all the attributes of Animal...

... but then overrides the behavior of method speak()

```

class Bird(Animal):
    'represents a bird'

    def speak(self):
        'prints bird sounds'
        print('{}! '.format(self.lang) * 3)

```

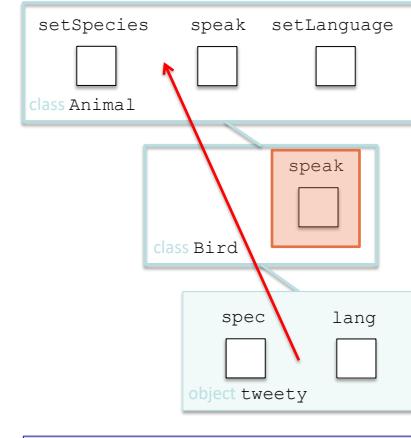
```

>>> snoopy = Animal()
>>> snoopy.setSpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
I am a dog and I bark.
>>>

```

47

## Overriding Superclass Methods (Notebook)



Python looks for the definition of an attribute by starting with the name-space associated with object and continuing up the class hierarchy.

```

class Bird(Animal):
    'represents a bird'

    def speak(self):
        'prints bird sounds'
        print('{}! '.format(self.lang) * 3)

```

method speak() defined in Bird is used

```

>>> snoopy = Animal()
>>> snoopy.setSpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
I am a dog and I bark.
>>> tweety = Bird()
>>> tweety.setSpecies('canary')
>>> tweety.setLanguage('tweet')
>>> tweety.speak()
tweet! tweet! tweet!

```

48

## Extending Superclass Methods



A superclass method can be inherited as-is, overridden, or extended.

```
class Super:  
    'a generic class with one method'  
    def method(self):  
        print('in Super.method')  
  
class Inheritor(Super):  
    'class that inherits method'  
    pass  
  
class Replacer(Super):  
    'class that overrides method'  
    def method(self):  
        print('in Replacer.method')  
  
class Extender(Super):  
    'class that extends method'  
    def method(self):  
        print('starting Extender.method')  
        Super.method(self)           # calling Super method  
        print('ending Extender.method')
```

49

## Object-Oriented Programming (OOP)

Code reuse is a key benefit of organizing code into new classes; it is made possible through **abstraction** and **encapsulation**.

**Abstraction:** The idea that a class object can be manipulated by users through method invocations alone and without knowledge of the implementation of these methods.

- Abstraction facilitates software development because the programmer works with objects abstractly (i.e., through “abstract”, meaningful method names rather than “concrete”, technical code).

**Encapsulation:** In order for abstraction to be beneficial, the “concrete” code and data associated with objects must be encapsulated (i.e., made “invisible” to the program using the object).

- Encapsulation is achieved thanks to the fact that (1) every class defines a namespace in which class attributes live, and (2) every object has a namespace, that inherits the class attributes, in which instance attributes live.

**OOP** is an approach to programming that achieves modular code through the use of objects and by structuring code into user-defined classes.

50

25