OOP in Python

Classes

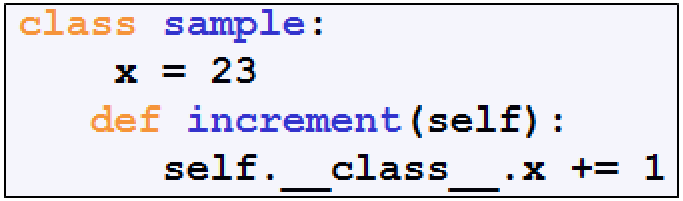
- Everything in Python is an object
    - mylist.append()
    - “string”.upper()
- Method calls from objects

Defining a class

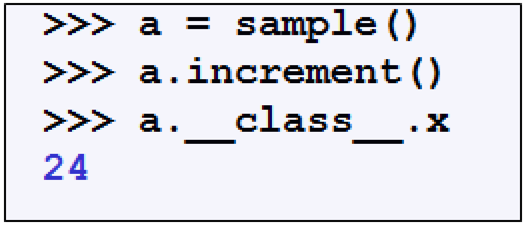
- Class - special data type which defines how to build a certain kind of object
- Class - user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
- Attributes
  - Data members (class and instance variables)
  - Methods

Defining a Class

- Class Variable vs Instance Variable
  - Shared by all instances
  - Declared outside all methods
- Instance Variables
  - Defined inside a method
  - Belongs to only current instance of a class



```
class sample:
     x = 23
    def increment(self):
       self.__class__.x += 1
```

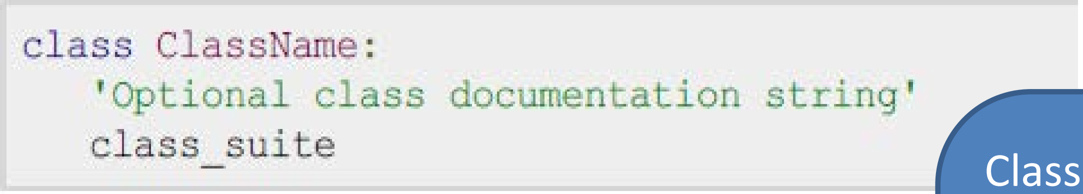


```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

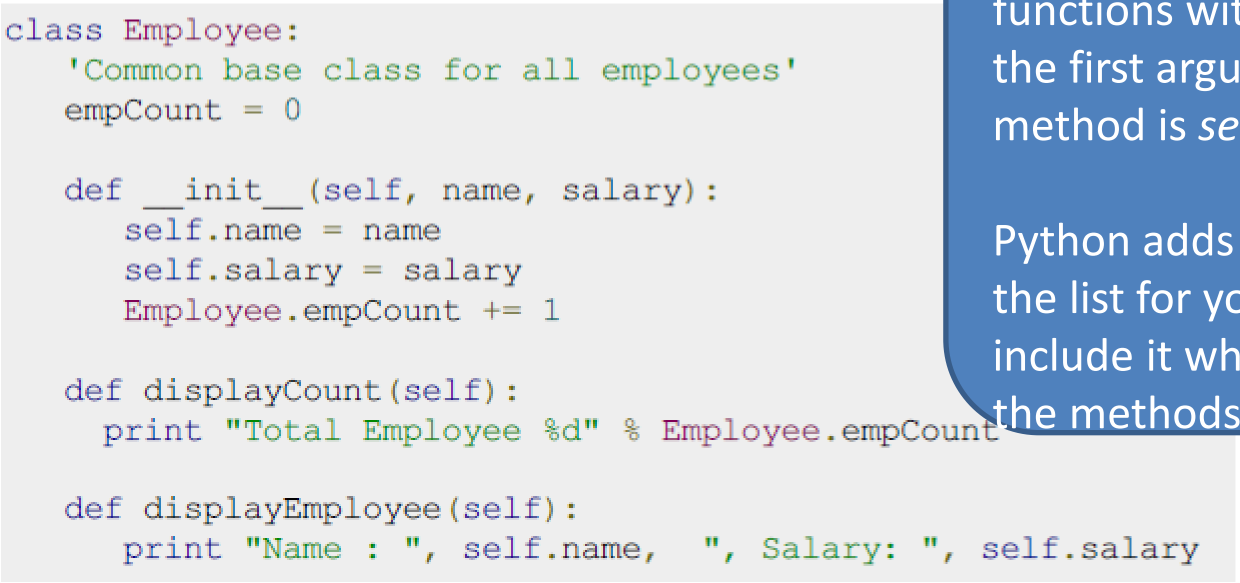
- Ins
  - Individual object of a certain class
  - objects that follow the definition given inside of the class

Creating a Class

- Python doesn’t use separate class interface definitions as in some languages



```
class ClassName:
   'Optional class documentation string'
   class_suite
```

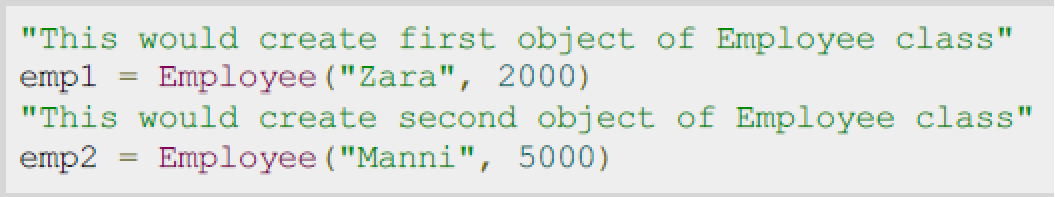


```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

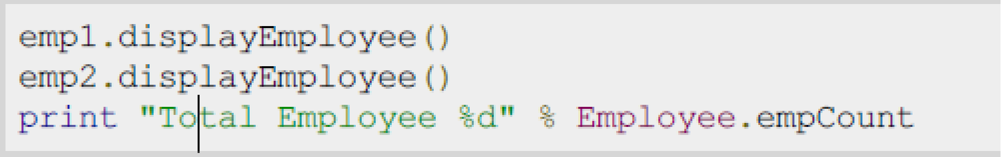
Class methods are like normal functions with the exception that the first argument to each method is *self*.

Python adds the *self* argument to the list for you; you don't need to include it when you call the methods

Creating instance of objects



```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```



```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

- No “new” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- __init__ method for Employee is passed “Zara” and 2000 and the new class instance is bound to emp1

Constructor

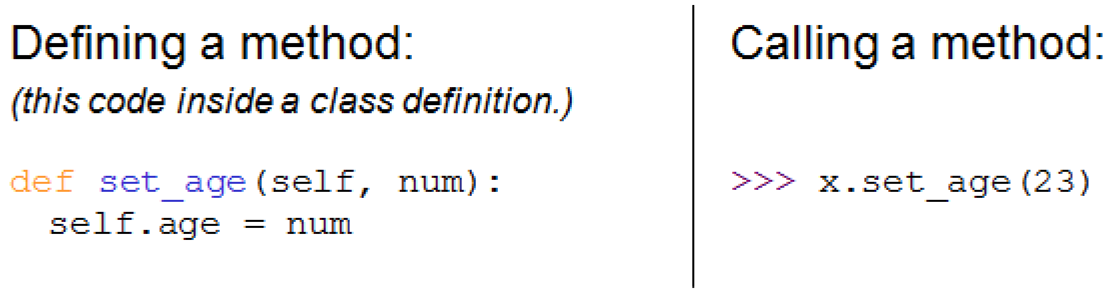
- The __init__ method is run as soon as an object of a class is instantiated. Its aim is to initialize the object.
- can take any number of arguments.
- the first argument `self` in the definition of __init__ is special

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument **`self`**
- In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword *this* in Java or C++
- But Python uses *self* more often than Java uses *this*

Self

- Although you must specify `self` explicitly when *defining* the method, you don't include it when *calling* the method.
- Python passes it for you automatically



Defining a method:

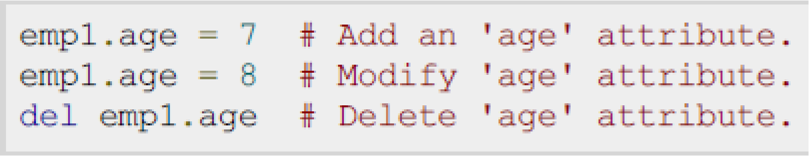
(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

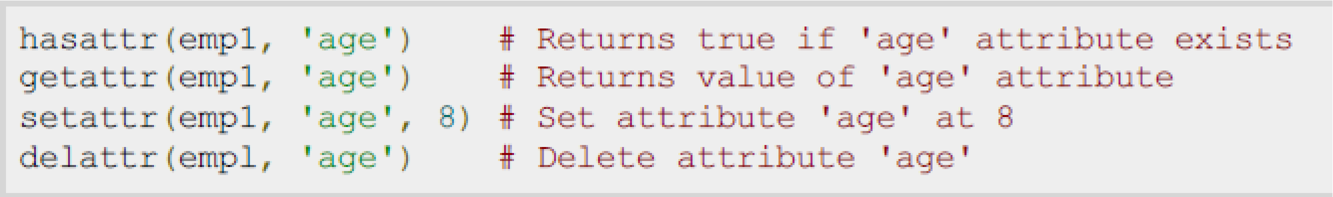
Calling a method:

```
>>> x.set_age(23)
```

Add or remove data members in a Class

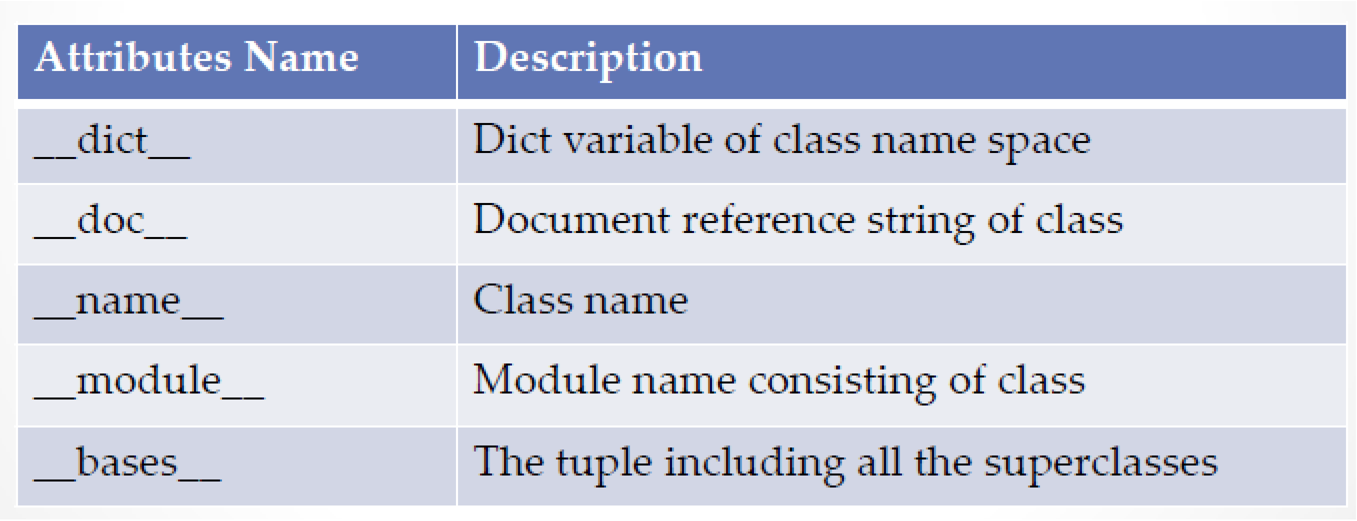


```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```



```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(emp1, 'age')    # Delete attribute 'age'
```

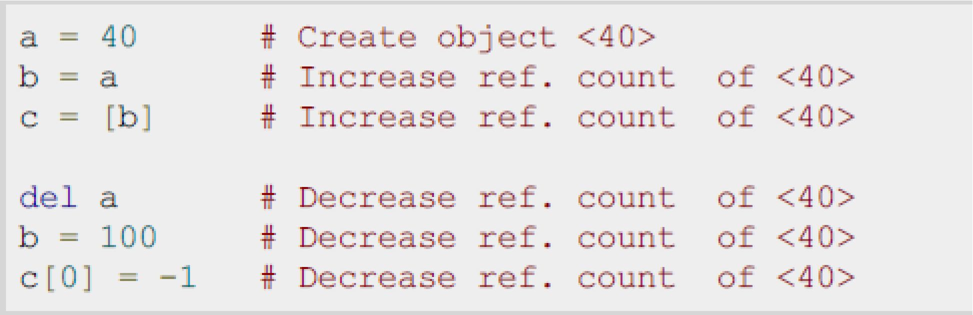
Built in Class Attributes



Attributes Name	Description
__dict__	Dict variable of class name space
__doc__	Document reference string of class
__name__	Class name
__module__	Module name consisting of class
__bases__	The tuple including all the superclasses

Destroying Objects

- deletes unneeded objects automatically to free memory space
- triggered when an object's reference count reaches zero



```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

Destroying Objects

- a class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed

Class Inheritance

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*
- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
    Class Cs_student(student):
```

  - Python has no 'extends' keyword like Java.
  - Multiple inheritance is supported.

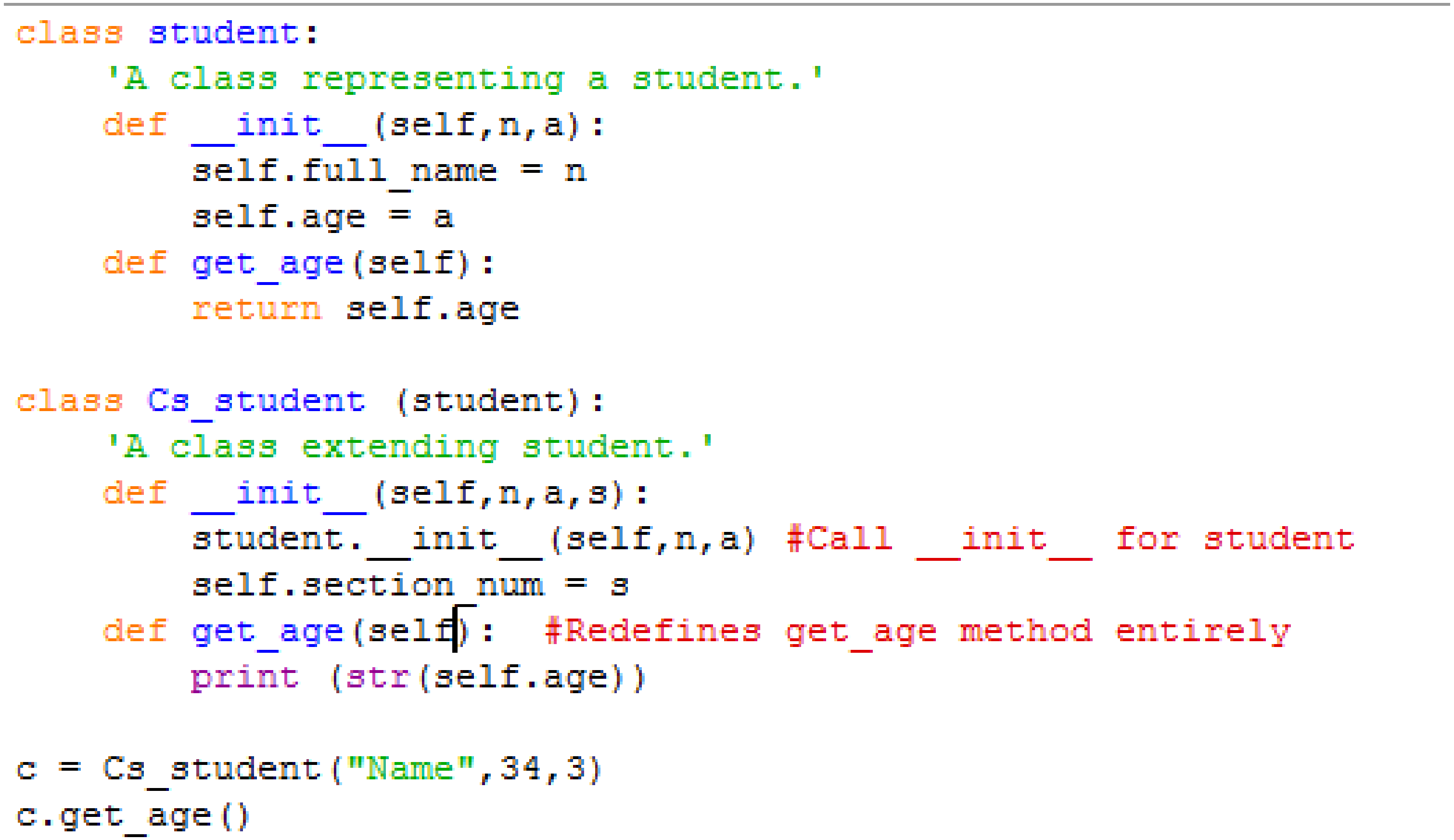
Class Inheritance - Overriding

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won’t get executed.
- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent’s version of the method.

```
parentClass.methodName(self, a, b, c)
```

  - **The only time you ever explicitly pass ‘self’ as an argument is when calling a method of an ancestor.**

Class Inheritance

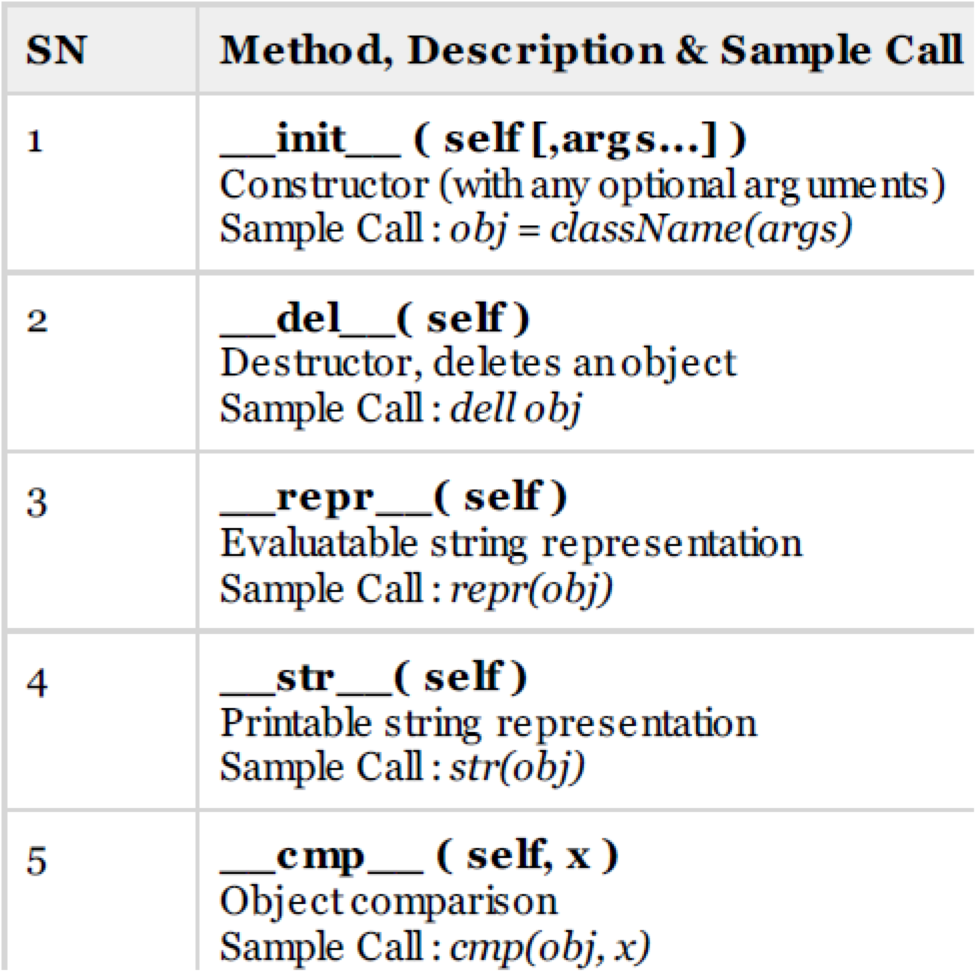


```
class student:
    'A class representing a student.'
    def __init__(self,n,a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age

class Cs_student (student):
    'A class extending student.'
    def __init__(self,n,a,s):
        student.__init__(self,n,a) #Call __init__ for student
        self.section num = s
    def get_age(self):  #Redefines get_age method entirely
        print (str(self.age))

c = Cs_student("Name",34,3)
c.get_age()
```

Class Inheritance

- **issubclass(sub, sup)** - returns true if **sub** is indeed a subclass of the superclass **sup**.
- **isinstance(obj, Class)** - returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Built in methods

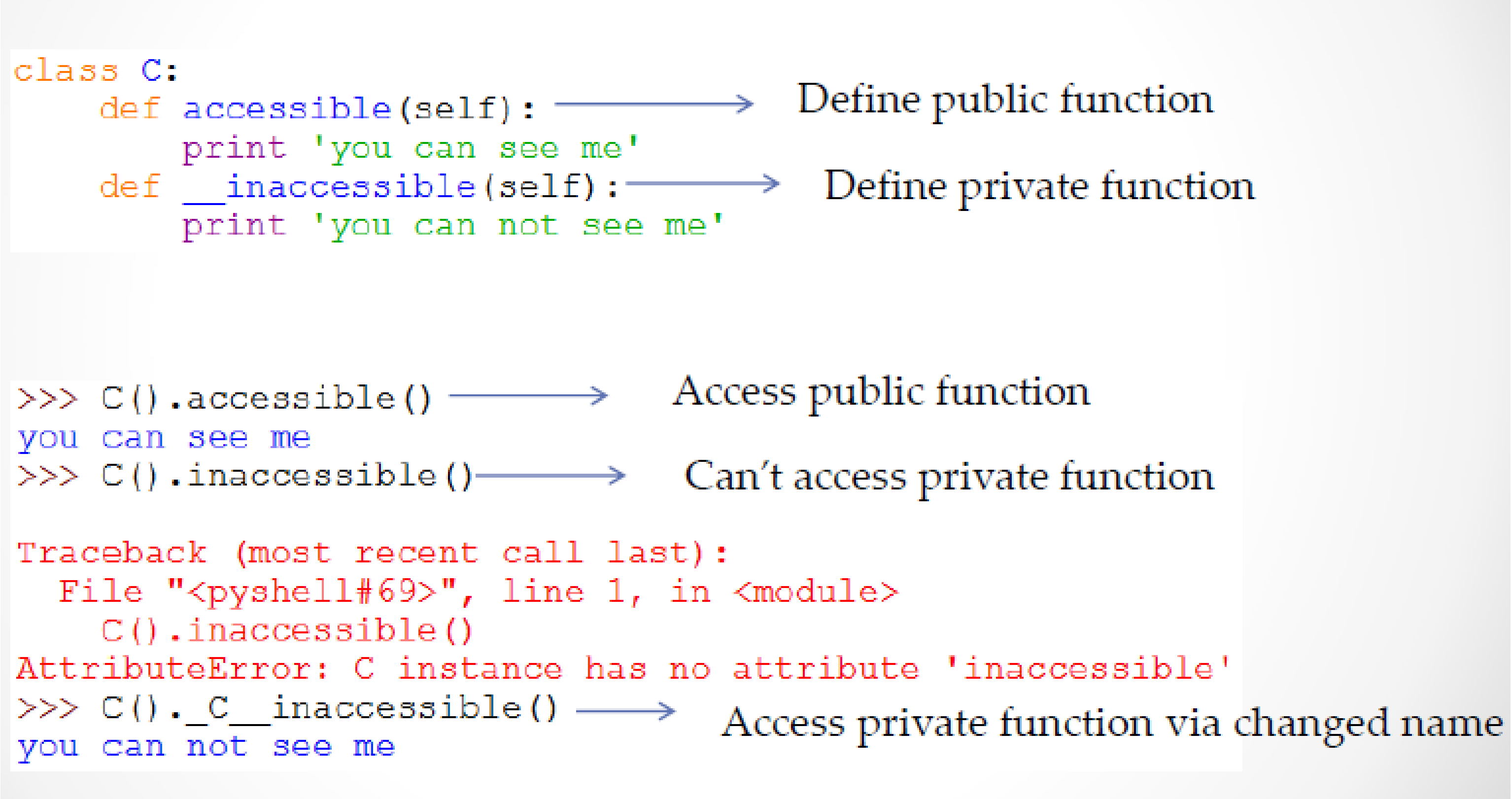


SN	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *dell obj*
3	**__repr__(self)** Evaluatable string representation Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

Accessibility/Encapsulation

- Any attribute/method with 2 leading underscores in its name (but none at the end) is **private** and can’t be accessed outside of class
- Names with two underscores at the beginning ***and the end*** are for built-in methods or attributes for the class
- There is no ‘protected’ status in Python; so, subclasses would be unable to access these private data either.

Encapsulation



```
class C:
    def accessible(self):
        print 'you can see me'
    def __inaccessible(self):
        print 'you can not see me'
```

Define public function

Define private function

```
>>> C().accessible()
you can see me
>>> C().inaccessible()

Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    C().inaccessible()
AttributeError: C instance has no attribute 'inaccessible'
>>> C()._C__inaccessible()
you can not see me
```

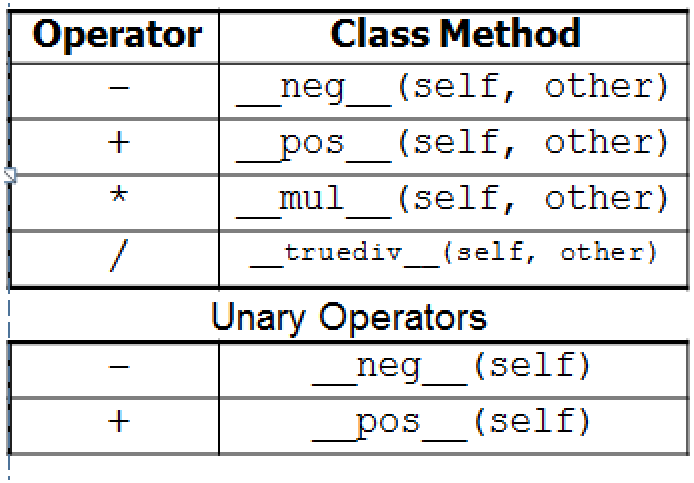
Access public function

Can't access private function

Access private function via changed name

Operator Overloading

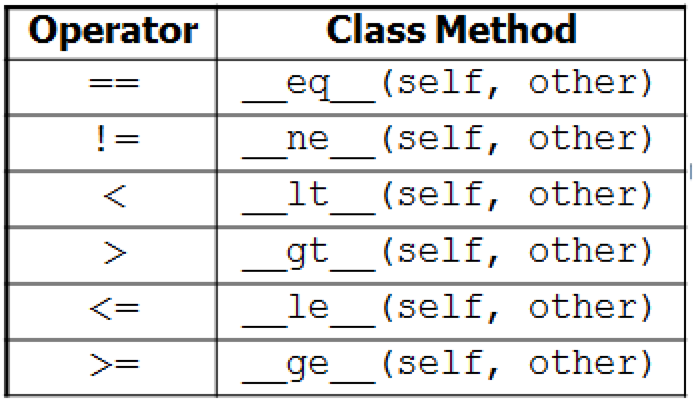
- You can define functions so that Python's built-in operators can be used with your class



Operator	Class Method
–	__neg__(self, other)
+	__pos__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)

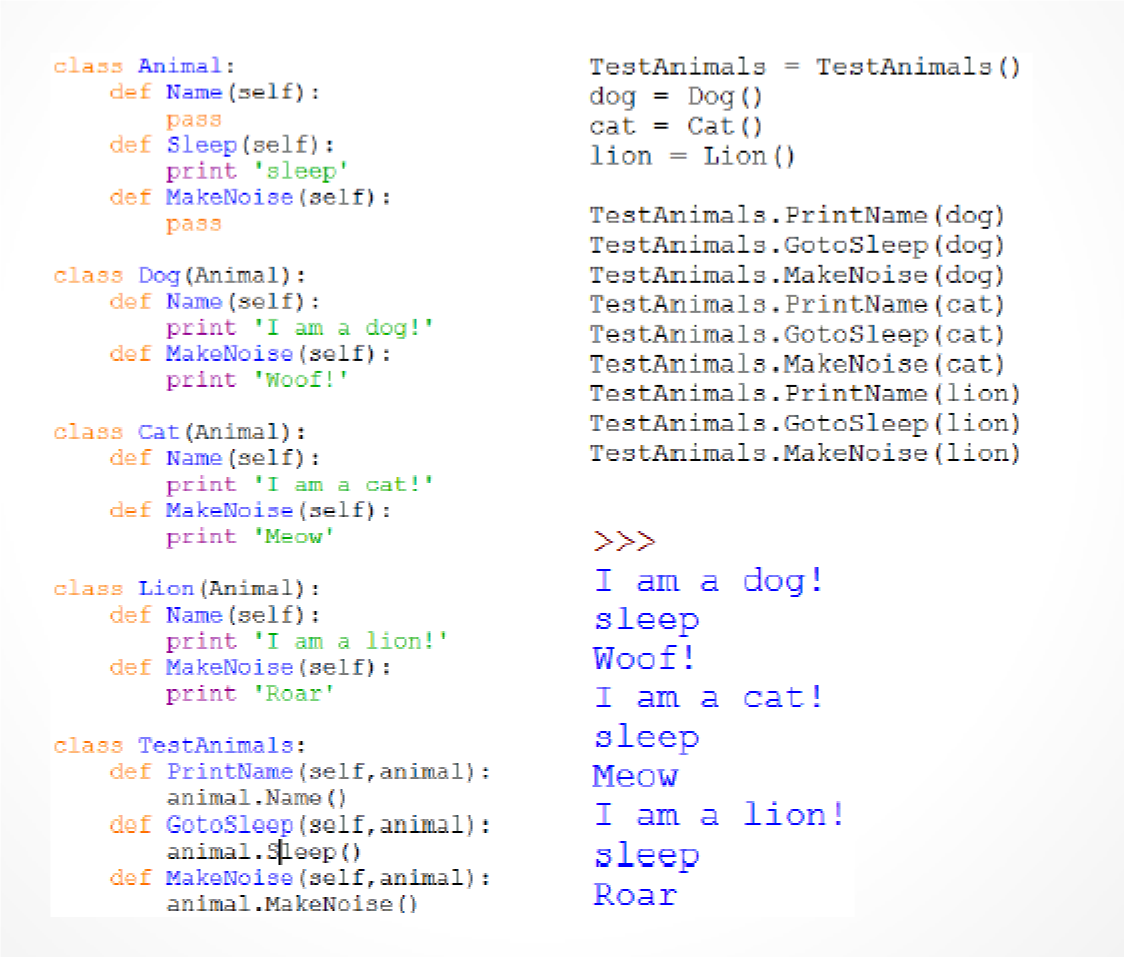
Unary Operators

–	__neg__(self)
+	__pos__(self)



Operator	Class Method
==	__eq__(self, other)
!=	__ne__(self, other)
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)

Polymorphism

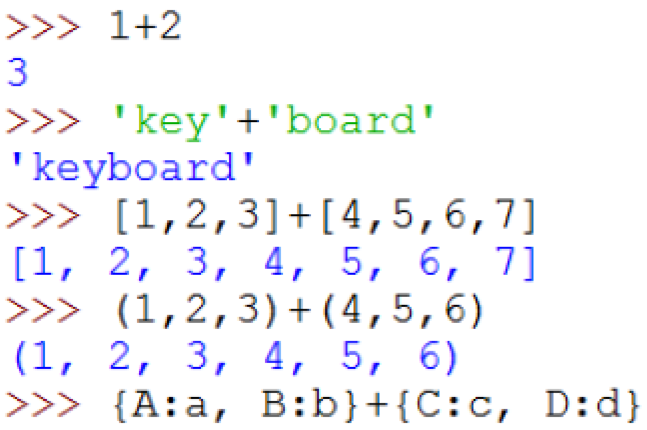


```
class Animal:
    def Name(self):
        pass
    def Sleep(self):
        print 'sleep'
    def MakeNoise(self):
        pass

class Dog(Animal):
    def Name(self):
        print 'I am a dog!'
    def MakeNoise(self):
        print 'Woof!'

class Cat(Animal):
    def Name(self):
        print 'I am a cat!'
    def MakeNoise(self):
        print 'Meow'

class Lion(Animal):
    def Name(self):
        print 'I am a lion!'
    def MakeNoise(self):
        print 'Roar'

class TestAnimals:
    def PrintName(self,animal):
        animal.Name()
    def GotoSleep(self,animal):
        animal.Sleep()
    def MakeNoise(self,animal):
        animal.MakeNoise()
```

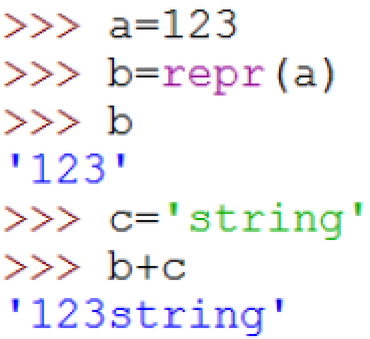
```
TestAnimals = TestAnimals()
dog = Dog()
cat = Cat()
lion = Lion()

TestAnimals.PrintName(dog)
TestAnimals.GotoSleep(dog)
TestAnimals.MakeNoise(dog)
TestAnimals.PrintName(cat)
TestAnimals.GotoSleep(cat)
TestAnimals.MakeNoise(cat)
TestAnimals.PrintName(lion)
TestAnimals.GotoSleep(lion)
TestAnimals.MakeNoise(lion)
```

```
>>>
I am a dog!
sleep
Woof!
I am a cat!
sleep
Meow
I am a lion!
sleep
Roar
```

Polymorphism



```
>>> 1+2
3
>>> 'key'+'board'
'keyboard'
>>> [1,2,3]+[4,5,6,7]
[1, 2, 3, 4, 5, 6, 7]
>>> (1,2,3)+(4,5,6)
(1, 2, 3, 4, 5, 6)
>>> {A:a, B:b}+{C:c, D:d}
```



```
>>> a=123
>>> b=repr(a)
>>> b
'123'
>>> c='string'
>>> b+c
'123string'
```

Importing and Modules

- Use classes & functions defined in another file
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, C++ *include*
- Three formats of the command:

```
import somefile
from somefile import *
from somefile import className
```

- The difference? What gets imported from the file and what name refers to it after importing

import …

```
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text “somefile.” to the front of its name:

```
somefile.className.method(“abc”)
somefile.myFunction(34)
```

*from … import **

```
from somefile import *
```

- *Everything* in somefile.py gets imported
- To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.
- *Take care!* Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method(“abc”)
myFunction(34)
```

from … import …

```
from somefile import className
```

- Only the item *className* in somefile.py gets imported.
- After importing *className*, you can just use it without a module prefix. It’s brought into the current namespace.
- *Take care*! Overwrites the definition of this name if already defined in the current namespace!

`className.method(“abc”)` 🡐 imported

`myFunction(34)` 🡐 Not imported

Directories for module files

- *Where does Python look for module files?*
- The list of directories where Python will look for the files to be imported is sys.path
- This is just a variable named ‘path’ stored inside the ‘sys’ module

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages/setuptools-0.6c5-py2.5.egg’, …]
```

- To add a directory of your own to this list, append it to this list

```
sys.path.append(‘/my/new/path’)
```