Defining a Function

- Begin with the keyword **def** followed by the function name and parentheses (**( )**).
  - Any input parameters or arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the ***function*** or ***docstring***.
- The code block within every function starts with a colon (**:**) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller.
  - A return statement with no arguments is the same as return None.

Defining a Function

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str;
   return;

# Now you can call printme function
printme("I'm first call to user defined function!");
printme("Again second call to the same function");
```

```
I'm first call to user defined function!
Again second call to the same function
```

Functions

- def print_hello():# returns nothing
  print “hello”

- def gcd(m, n):
  if n == 0:
  return m # returns m
  else:
  return gcd(n, m % n) recursive call

- def has_args(arg1,arg2=['e', 0]): # returns [9.16,[1,‘b’,‘a’,7]]
  num = arg1 + 4
  mylist = arg2 + ['a',7]
  return [num, mylist]
  has_args(5.16,[1,'b'])

Function Parameter

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

Function Parameter

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Default Arguments

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name;
   print "Age ", age;
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length Arguments

- You may need to process a function for more arguments than you specified while defining the function.

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

```
Output is:
10
Output is:
70
60
50
```

Variable-length Arguments

- *args = list of arguments -as positional arguments
- **kwargs = dictionary - whose keys become separate keyword arguments and the values become values of these arguments.

```
def print_everything(*args):
    for count, thing in enumerate(args):
        print('{0}.{1}'. format(count,thing))


print_everything('apple','banana', 'cabbage')
```

```
0.apple
1.banana
2.cabbage
```

```
0,apple
1,banana
2,cabbage
```

```
def table_thing(**kwargs):
    for name, value in kwargs.items():
        print('{0}={1}'. format(name,value))


table_thing(apple='fruit', cabbage='vegetable')
```

```
apple=fruit
cabbage=vegetable
```

Variable-length Arguments

- You can also use both in the same function definition but *args must occur before **kwargs.

```
def test_kwargs(first, *args, **kwargs):
   print 'Required argument: ', first
   for v in args:
      print 'Optional argument (*args): ', v
   for k, v in kwargs.items():
      print 'Optional argument %s (*kwargs): %s' % (k, v)

test_kwargs(1, 2, 3, 4, k1=5, k2=6)
```

```
Requires argument: 1
Optional argument (*args): 2
Optional argument (*args): 3
Optional argument (*args): 4
Optional argument k1 (*kwargs): 5
Optional argument k2 (*kwargs): 6
```

The *Anonymous* Functions

- You can use the ***lambda*** keyword to create small anonymous functions.
  - These functions are called **anonymous** because they are not declared in the standard manner by using the ***def*** keyword.
- The syntax of *lambda* functions contains only a single statement,

```
lambda [arg1 [,arg2,.....argn]]:expression
```

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;



# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

```
value of total :  30
value of total :  40
```

Python Files I/O-Keyboard Input

- Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard.
  - raw_input
  - input
- The *raw_input([prompt])* function reads one line from standard input and returns it as a string (removing the trailing newline).

```
#!/usr/bin/python

str = raw_input("Enter your input: ");
print "Received input is : ", str
```

```
Enter your input: Hello Python
Received input is :  Hello Python
```

The *input* Function

- The *input([prompt])* function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
#!/usr/bin/python

str = input("Enter your input: ");
print "Received input is : ", str
```

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

Opening and Closing Files

- The file manipulation using a ***file*** object.
- *Open :* Before you can read or write a file, you have to open it using Python's built-in *open()* function.
- This function creates a ***file*** object, which would be utilized to call other support methods associated with it.

Open function

- Syntax `file object = open(file_name [, access_mode][, buffering])`
- **file_name:** is a string value that contains the name of the file.
- **access_mode:** determines the mode in which the file has to be opened, i.e., read, write, append, etc.
  - This is optional parameter and the default file access mode is read (r).
- **buffering:**
  - If the buffering value is set to 0, no buffering takes place.
  - If the buffering value is 1, line buffering is performed while accessing a file.
  - If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size.
  - If negative, the buffer size is the system default (default behavior).

access_mode

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is a file exists. That is, the file is in the append mo exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. Th of the file if the file exists. That is, the file is in file does not exist, it creates a new file for writi
a+	Opens a file for both appending and reading. T end of the file if the file exists. The file opens i file does not exist, it creates a new file for read
ab+	Opens a file for both appending and reading in pointer is at the end of the file if the file exists append mode. If the file does not exist, it creat and writing.

The *file* Object Attributes

- Once a file is opened and you have one *file* object, you can get various information related to that file.

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Default = 0

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

The *close()* Function

- The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file.
  - It is a good practice to use the close() method to close a file.
- Syntax `fileObject.close();`

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

```
Name of the file:  foo.txt
```

Reading and Writing Files

- The *file* object provides a set of access methods.
  - *read()* and *write()* methods to read and write files.

Syntax `fileObject.write(string);`

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");

# Close opend file
fo.close()
```

```
Python is a great language.

Yeah its great!!
```

The *read()* Method

- Syntax `fileObject.read([count]);`

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

- Passed parameter is the number of bytes to be read from the opened file.

```
Read String is :  Python is
```

File Positions

- The *tell()* method tells you the current position within the file.
  - The next read or write will occur at that many bytes from the beginning of the file.
- The *seek(offset[, from])* method changes the current file position.
  - The *offset* indicates the number of bytes to be moved.
  - The *from* specifies the reference position from where the bytes are to be moved.
- *From* is set to 0,
  - it means use the beginning of the file as the reference position
- 1: uses the current position as the reference position.
- 2: the end of the file would be taken as the reference position.

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str

# Check current position
position = fo.tell();
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# Close opend file
fo.close()
```

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming and Deleting Files

- Python **os** module provides methods that help you perform file-processing operations, such as *renaming* and *deleting* files.
- The *rename()* Method

```
os.rename(current_file_name, new_file_name)
```

```
#!/usr/bin/python
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

The *remove()* Method

```
os.remove(file_name)
```

```
#!/usr/bin/python
import os

# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python

- The **os** module has several methods that help you create, remove, and change directories.
- The *mkdir()* Method

```
#!/usr/bin/python
import os

# Create a directory "test"
os.mkdir("test")
```

The *chdir()* Method

```
#!/usr/bin/python
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

- The *getcwd()* Method

```
#!/usr/bin/python
import os

# This would give location of the current directory
os.getcwd()
```

The *rmdir()* Method

```
#!/usr/bin/python
import os

# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
  - The attributes are data members (class variables and instance variables) and methods, accessed via dot notation (.).
- **Class variable:** A variable that is shared by all instances of a class.
  - Class variables are defined within a class but also outside any of the class's methods.
  - Class variables aren't used as frequently as instance variables are.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

Creating Class

- The ***class*** statement creates a new class definition.

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

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.

- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

EXAMPLE

- The variable *empCount* is a class variable whose value would be shared among all instances of a this class.
  - This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method.
  - Python automatically calls when you create a new instance of this class.
- You declare other class methods 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.

```
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
```

Creating instance objects

- To create instances of a class, you call the class using class name and pass in whatever arguments its __*init*__ method accepts.

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

- Accessing attributes

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

Example

```
#!/usr/bin/python

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

"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
```

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

Built-In Class Attributes

- Every Python class keeps following **built-in attributes** and they can be accessed using **dot (.)** operator like any other attribute:
- **__dict__ :** Dictionary containing the class's namespace.
- **__doc__ :** Class documentation string or None if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined.
  - This attribute is "__main__" in interactive mode.
- **__bases__ :** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Example

```
#!/usr/bin/python

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

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Built-in Function *dir*

- The built-in function ***dir*** will give a list of names comprising the methods and attributes of an object.

```
>>>print dir(Exception)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__getitem__', '__getslice__', '__hash__', '__init__', '__new__', '__reduc
e__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__',
'__str__', '__subclasshook__', '__unicode__', 'args', 'message']
```

- You can also get help using the help method: help (Exception).

Destroying Objects (Garbage Collection)

- Python deletes unneeded objects (built-in types or class instances) automatically to free memory space.
- Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection.
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.
  - An object's reference count changes as the number of aliases that point to it changes.

Destroying Objects

- An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary).
- The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope.
- When an object's reference count reaches zero, Python collects it automatically.

```
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>
```

EXAMPLE

- This __del__() destructor that prints the class name of an instance that is about to be destroyed.

```
#!/usr/bin/python

class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

```
3083401324 3083401324 3083401324
Point destroyed
```

Class Inheritance

- You can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
```

- The child class inherits the attributes of the parent class
  - you can use those attributes as if they were defined in the parent class.
- A child class can also override data members and methods from the parent.

EXAMPLE

```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Super function (python 3.x)

```
class Demo:
    __x = 0

    def __init__(self, i):
        self.__i = i
        Demo.__x += 1

    def __str__(self):
        return str(self.__i)

    def hello(self):
        print("hello " + self.__str__())

    @classmethod
    def getX(cls):
        return cls.__x

class SubDemo(Demo):
    def __init__(self, i, j):
        super().__init__(i)
        self.__j = j

    def __str__(self):
        return super().__str__() + "+" + str(self.__j)
```

```
a = SubDemo(12, 34)
a.hello()
print("a.__x =", a.getX())
b = SubDemo(56, 78)
b.hello()
print("b.__x =", b.getX())
print()
print("a.__x =", a.getX())
print("b.__x =", b.getX())
```

static function uses cls parameter

```
hello 12+34
a.__x = 1
hello 56+78
b.__x = 2

a.__x = 2
b.__x = 2
```

Multiple Inheritance

```
class A:          # define your class A
.....

class B:           # define your calss B
.....

class C(A, B):    # subclass of A and B
.....
```

- You can use **issubclass()** or **isinstance()** functions to check a relationships of two classes and instances.
- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* is an instance of a subclass of Class

Polymorphism

- The term *polymorphism*, in the OOP, refers to the ability of an object to adapt the code to the type of the data.
- Polymorphism has *two* major applications in an OOP language.
  - An object may provide different implementations of one of its methods depending on the type of the input parameters.
  - code written for a given type of data may be used on data with a derived type, i.e. methods understand the class hierarchy of a type.

Example

- All animals "talk", but they have different “talk” behavior.
- The "talk" behavior is thus *polymorphic* in the sense that it is *realized differently depending on the animal*.
- The abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name
    def talk(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat('Missy'),
           Cat('Mr. Mistoffelees'),
           Dog('Lassie')]

for animal in animals:
    print animal.name + ': ' + animal.talk()
```

```
prints the following:

Missy: Meow!
Mr. Mistoffelees: Meow!
Lassie: Woof! Woof!
```

Overriding Methods

- You can always override your parent class methods.

```
#!/usr/bin/python

class Parent:          # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

```
Calling child method
```

Base Overloading Methods

- Following table lists some generic functionality that you can override in your own classes.

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)*

Overloading Operators

- You could define the __*add*__ method in your class to perform vector addition and then the plus operator would behave as per expectation

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

```
Vector(7,8)
```

Overloading Operators

```
1  import math
2
3  class Circle:
4
5      def __init__(self, radius):
6          self.__radius = radius
7
8      def setRadius(self, radius):
9          self.__radius = radius
10
11     def getRadius(self):
12         return self.__radius
13
14     def area(self):
15         return math.pi * self.__radius ** 2
16
17     def __add__(self, another_circle):
18         return Circle( self.__radius + another_circle.__radius )
19
20 c1 = Circle(4)
21 print(c1.getRadius())
22
23 c2 = Circle(5)
24 print(c2.getRadius())
25
26 c3 = c1 + c2 # This became possible because we have overloaded + operator by addi
27 print(c3.getRadius())
```

```
1  4
2  5
3  9
```

OPERATOR	FUNCTION	METHOD DESCRIPTION
`+`	`__add__(self, other)`	Addition
`*`	`__mul__(self, other)`	Multiplication
`-`	`__sub__(self, other)`	Subtraction
`%`	`__mod__(self, other)`	Remainder
`/`	`__truediv__(self, other)`	Division
`<`	`__lt__(self, other)`	Less than
`<=`	`__le__(self, other)`	Less than or equal to
`==`	`__eq__(self, other)`	Equal to
`!=`	`__ne__(self, other)`	Not equal to
`>`	`__gt__(self, other)`	Greater than
`>=`	`__ge__(self, other)`	Greater than or equal to
`[index]`	`__getitem__(self, index)`	Index operator
`in`	`__contains__(self, value)`	Check membership
`len`	`__len__(self)`	The number of elements
`str`	`__str__(self)`	The string representation

```python
import math

class Circle:

    def __init__(self, radius):
        self.__radius = radius

    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def area(self):
        return math.pi * self.__radius ** 2

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius

    def __lt__(self, another_circle):
        return self.__radius < another_circle.__radius

    def __str__(self):
        return "Circle with radius " + str(self.__radius)

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2
print(c3.getRadius())

print( c3 > c2) # Became possible because we have added __gt__ method

print( c1 < c2) # Became possible because we have added __lt__ method

print(c3) # Became possible because we have added __str__ method
```

```
4
5
9
True
True
Circle with radius 9
```

Data Hiding

- An object's attributes may or may not be *visible* outside the class definition.
- You can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders.

```
#!/usr/bin/python

class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Data Hiding

- Python protects those members by internally changing the name to include the class name.
- You can access such attributes as *object._className__attrName*.
- If you would replace your last line as following, then it would work for you:

```
.....................
print counter._JustCounter__secretCount
```

```
1
2
2
```