PYTHON 3

OBJECT ORIENTED PYTHON

Declaration

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

The documentation string, can be accessed via ***ClassName.__doc__***.

Example

```
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)
```

- The variable *empCount* is a class variable whose value is shared among all the instances of a in this class. (like a static member of Java)
- This can be accessed as *Employee.empCount* from inside the class or outside the class

Example

```
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)
```

- The first method ___*init*__()* is the initialization method
  - Not same as constructors
  - it is called when a new instance of this class is created.

Example

```
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)
```

- first argument to each method is *self*.
- While calling a method from a class, Python adds the *self* argument automatically

Creating Instance 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)
```

```
emp1.salary = 7000
emp1.name = 'xyz'  # Modify 'age' attribute.
del emp1.salary  # Delete 'age' attribute.
```

Setting and getting attributes

Instead of using the normal statements to access attributes, you can use the following functions-

- The **getattr(obj, name[, default])**: to access the attribute of object.
- The **hasattr(obj,name)**: to check if an attribute exists or not.
- The **setattr(obj,name,value)**: to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)**: to delete an attribute.

```
hasattr(emp1, 'salary')    # Returns true if 'salary' attribute exists
getattr(emp1, 'salary')    # Returns value of 'salary' attribute
setattr(emp1, 'salary', 7000) # Set attribute 'age' at 8
delattr(emp1, 'salary')    # Delete attribute 'age'
```

Class variable vs. Instance Variable

```
class MyEmp:
    salary = 3000;

    def __init__(self, salary):
        self.salary = salary

    def prSlr(self):
        print(self.salary," ",MyEmp.salary)


em = MyEmp("40000")
em.prSlr()
```

```
40000    3000
```

Examples – You can Change Type

```
class MyEmp:
    salary = 3000;

    def __init__(self, salary):
        self.salary = salary

    def prSlr(self):
        print(self.salary," ",MyEmp.salary)


em = MyEmp("40000")
em.prSlr()
print(type(em.salary))
em.salary = 50000 #type changed
em.prSlr()
print(type(em.salary))
```

```
40000   3000
<class 'str'>
50000   3000
<class 'int'>
```

You can even create a new member

```
class MyEmp:
    salary = 3000;

    def __init__(self, salary):
        self.salary = salary

    def prSlr(self):
        print(self.salary," ",MyEmp.salary)


em = MyEmp(5000);
em.name = "Kuddus" # you can create
                   # new member variable
print(em.name)
```

You can even create a new member

```
class MyEmp:
    salary = 3000;

    def __init__(self, salary):
        self.salary = salary

    def prSlr(self):
        print(self.salary," ",MyEmp.salary)


em = MyEmp(5000);
print(hasattr(em,"name"))
em.name = "Kuddus" # you can create
                   # new member variable
print(hasattr(em,"name"))
print(em.name)
```

```
False
True
Kuddus
```

- Adding attribute in an object doesn't change the class

```
em = MyEmp(5000);
print(hasattr(em,"name"))
em.name = "Kuddus" # you can create
                   # new member variable
print(hasattr(em,"name"))
print(em.name)

em2 = MyEmp(2000)
print(hasattr(em2,"name")) # False

em2 = em
print(hasattr(em2,"name")) # True
```

These two codes are exactly equivalent

```
class Foo(object):
    def __init__(self, bar):
        self.bar = bar

foo = Foo(5)
```

And this code:

```
class Foo(object):
    pass

foo = Foo()
foo.bar = 5
```

Objects and members

- The members that a class will have is not restricted in the time of class declaration
  - In fact, there is no notion of declaration !!
  - However, there is way to restrict this property
    - Some built in types also do not allow this
- **Most instances store their attributes in a dictionary**
  - a regular Python dictionary like you'd define with {}
- Check this SO answer for details:
  - https://stackoverflow.com/questions/12569018/why-is-adding-attributes-to-an-already-instantiated-object-allowed

Built-In Class Attributes

Every Python class keeps the 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.

obj.__dict__

```
In[4]: class Dummy:
  ...:     pass
  ...:
In[5]: d = Dummy()
In[6]: d.__dict__
Out[6]:
{}
In[7]: d.a = 1
In[8]: d.__dict__
Out[8]:
{'a': 1}
In[9]: d.b = "B"
In[10]: d.__dict__
Out[10]:
{'a': 1, 'b': 'B'}
In[11]: d.c = "You cant see me"
In[12]: d.__dict__
Out[12]:
{'a': 1, 'b': 'B', 'c': 'You cant see me'}
```

obj.__dict__

```
In[14]: def meth():
   ...:     print("Add Me")
   ...:
In[15]: def block():
   ...:     print("Add Meh")
   ...:
In[16]: d.block = block
In[17]: d.block()
Add Meh
In[18]: d.__dict__
Out[18]:
{'a': 1, 'b': 'B', 'block': <function __main__.block>, 'c': 'You cant see me'}
```

Garbage Collection

- When a object is no longer referenced, Python automatically destroys it and reclaims the memory

Destructor Function

- __del__()

- The destruction function is called when an object is going to be destroyed

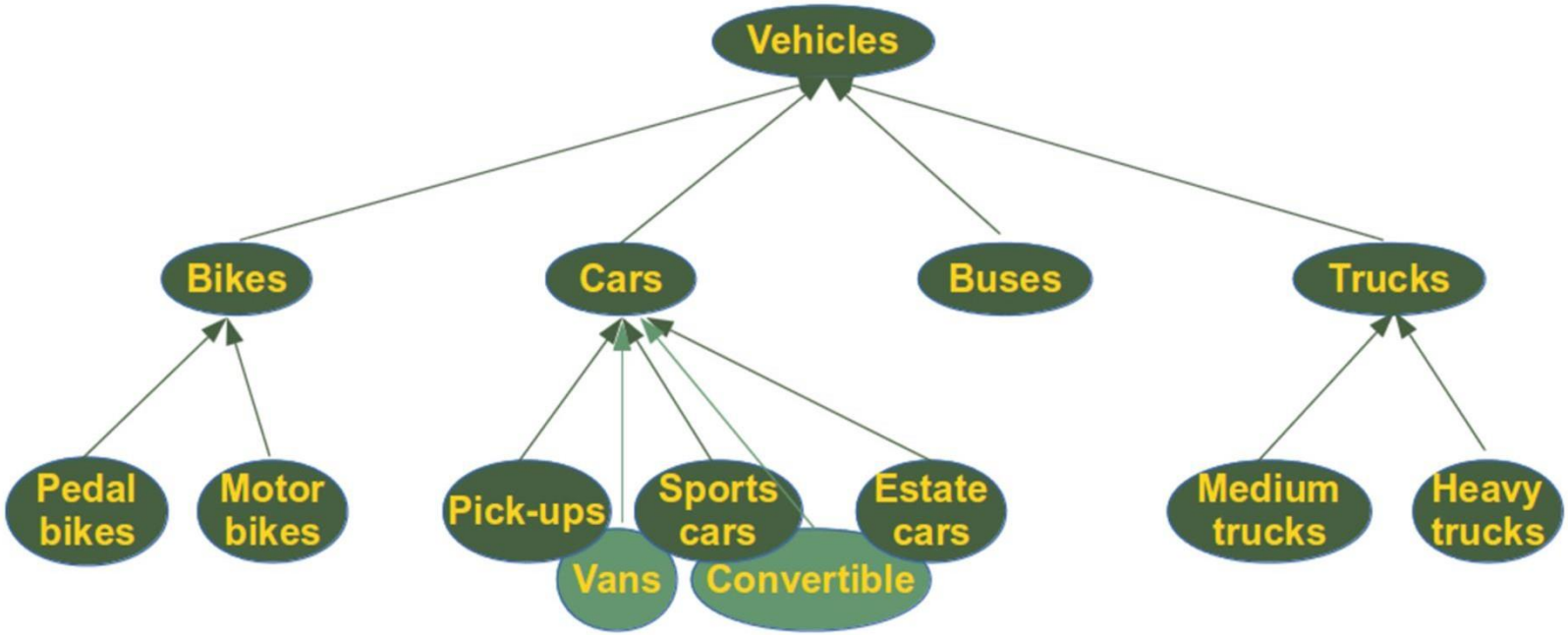
Example

```
class MyEmp:

    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(self.name, " is going to be destroyed!!")

emp1 = MyEmp("EMP1")
print(emp1.name)
emp1 = MyEmp("EMP2")
print(emp1.name)
```

```
Running F:/PyCharmWS/Prac/dummy.py
EMP1
EMP1  is going to be destroyed!!
EMP2
```

INHERITANCE

Inheritance



Inheritance

- You create a class by deriving it from a pre-existing class
- The child class inherits the attributes of its parent class
  - you can use those attributes as if they were defined in the child class.
- A child class can also override data members and methods from the parent

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

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

Example

```
class Polygon:

    def __init__(self):
        print("In Polygon-Constructor")

    def intro(self):
        print("I am a polygon")


class Triangle(Polygon):
    def __init__(self):
        print("In Triangle-Constructor")

    def myIntro(self):
        print("I am a triangle")


t = Triangle()
t.intro()
t.myIntro()
```

```
Running F:/PyCharmWS/Prac/dummy.py
In Triangle-Constructor
I am a polygon
I am a triangle
```

```python
class Polygon:

    def __init__(self):
        print("In Polygon-Constructor")
        self.m = 0

    def intro(self):
        print("I am a polygon")


class Triangle(Polygon):
    def __init__(self):
        #super().__init__()
        print("In Triangle-Constructor")

    def myIntro(self):
        print("I am a triangle")


t = Triangle()
t.intro()
t.myIntro()
print(t.m) #Attribute error
```

```python
class Polygon:

    def __init__(self):
        print("In Polygon-Constructor")
        self.m = 0

    def intro(self):
        print("I am a polygon")


class Triangle(Polygon):
    def __init__(self):
        super().__init__()
        print("In Triangle-Constructor")

    def myIntro(self):
        print("I am a triangle")


t = Triangle()
t.intro()
t.myIntro()
print(t.m) #prints 0 now
```

Overriding

```
class Polygon:

    def __init__(self):
        print("In Polygon-Constructor")
        self.m = 0

    def intro(self):
        print("I am a polygon")


class Triangle(Polygon):
    def __init__(self):
        super().__init__()
        print("In Triangle-Constructor")

    def intro(self):
        print("I am a triangle")


t = Triangle()
t.intro() # prints triangle
```

Inheriting Multiple Class

```
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 relationship 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* or is an instance of a subclass of Class.

MORE ON NUMBERS

Numbers

- They are immutable data types.
- This means, changing the value of a number data type results in a newly allocated object

Number objects are created when you assign a value to them. For example-

```
var1 = 1
var2 = 10
```

Deleting reference

You can also delete the reference to a number object by using the **del** statement. The syntax of the **del** statement is –

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the **del** statement. For example-

```
del var
del var_a, var_b
```

Deleting reference

```
i = 5
print(i)
del i
print(i)
```

```
E:\py prac>python prac.py
5
Traceback (most recent call last):
  File "prac.py", line 4, in <module>
    print(i)
NameError: name 'i' is not defined
```

int (signed integers)

- positive or negative whole numbers
- Integers in Python 3 are of unlimited size.
  - Python 2 has two integer types - int and long. There is no **'long integer'** in Python 3 anymore.
- It is possible to represent an integer in hexa-decimal or octal form.

```
>>> number = 0xA0F #Hexa-decimal
>>> number
2575

>>> number=0o37 #Octal
>>> number
```

float (floating point real values)

- represent real numbers
- Floats may also be in scientific notation, with E or e indicating the power of 10
  - 2.5e2 = 2.5 x 102 = 250

complex (complex numbers)

- are of the form a + bj
  - where a and b are floats
  - j (or J) represents the square root of -1

Number Type Conversion

- Python converts numbers internally in an expression containing mixed types to a common type for evaluation.

Explicit Conversion

Type **int(x)** to convert x to a plain integer.

Type **long(x)** to convert x to a long integer.

Type **float(x)** to convert x to a floating-point number.

Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.

Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions.

Some Useful Functions

- abs(x), fabs(x)
- ceil(x),floor(x)
- cmp(x,y)
- exp(x)
- log(x), log10(x)
- sqrt(x)

- **max(x1,x2,…)**
- **min(x1,x2,…)**

abs(x), fabs(x)

The **fabs()** method returns the absolute value of x. Although similar to the abs() function, there are differences between the two functions. They are-

- abs() is a built in function whereas fabs() is defined in math module.
- fabs() function works only on float and integer whereas abs() works with complex number also.

Some Useful Functions

- **modf(x)**
  - The fractional and integer parts of x in a two-item tuple.
  - Both parts have the same sign as x.
  - The integer part is returned as a float.

modf(x)

```
math.modf(100.12) :  (0.12000000000000455, 100.0)
math.modf(100.72) :  (0.7199999999999989, 100.0)
math.modf(119) :  (0.0, 119.0)
math.modf(math.pi) :  (0.14159265358979312, 3.0)
```

Some Useful Functions

- **round(x [,n])**
  - x rounded to n digits from the decimal point.
  - Python rounds away from zero as a tie-breaker:
  - round(0.5) is 1.0 and round(-0.5) is -1.0

Some Useful Functions

```
round(70.23456) :  70
round(56.659,1) :  56.7
round(80.264, 2) :  80.26
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

Random Number Functions

import random

random.random()

Function	Description
choice(seq)	A random item from a list, tuple, or string.
randrange ([start,] stop [,step])	A randomly selected element from range(start stop, step).
random()	A random float r, such that 0 is less than or equa to r and r is less than 1.

Random Number Functions

seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle(lst)	Randomizes the items of a list in place. Returns None.
uniform(x, y)	A random float r, such that x is less than or equal to r and r is less than y.

Trigonometric Functions

acos(x)	
asin(x)	
atan(x)	
hypot(x, y)	Return the Euclidean norm, sqrt(x*x + y*y).

sin(x)	
tan(x)	
cos(x)	

degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

import math

Constants
math.pi
math.e

MORE ON STRINGS

Strings are immutable

```
In[1]: str = "Hello"
In[2]: str[1] = 'B'
Traceback (most recent call last):
  File "C:\Program Files\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py", line 2881, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-2-f8d39658be9a>", line 1, in <module>
    str[1] = 'B'
TypeError: 'str' object does not support item assignment
```

In Python3 all strings are Unicode

```
In[12]: print("আমার সোনার বাংলা")
আমার সোনার বাংলা
```

What happens here ?

```
a = "Dog"
b = "eats"
c = "treats"

print (a)
a = a + " " + b + " " + c
print (a)
```

String Formatting

- Like printf
- print(string % tuple)

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

When the above code is executed, it produces the following result –

```
My name is Zara and weight is 21 kg!
```

There are many String functions

- Find it from the book and play around

MORE ON LISTS

Lists are Sequence

- The most basic data structure in Python is the **sequence**
  - Each element has an index
- Lists, Tuples, Strings are sequences
- There are certain things you can do with all the sequence types.
  - These operations include indexing, slicing, adding, multiplying, and checking for membership

Negative index is possible

- Try it yourself

More List Functions

len(list)

Gives the total length of the list.

---

max(list)

Returns item from the list with max value.

---

min(list)

Returns item from the list with min value.

---

list(seq)

Converts a tuple into list.

More List Functions

list.append(obj)

Appends object obj to list

---

list.count(obj)

Returns count of how many times obj occurs in list

---

list.extend(seq)

Appends the contents of seq to list

list.index(obj)

Returns the lowest index in list that obj appears

---

list.insert(index, obj)

Inserts object obj into list at offset index

GuessTheOutput

```
In[2]: l = [1,2,3,4]
In[3]: l2 = [5,6]
In[4]: l.append(l2)
```

GuessTheOutput

```
In[2]: l = [1,2,3,4]
In[3]: l2 = [5,6]
In[4]: l.append(l2)
In[5]: l
Out[5]:
[1, 2, 3, 4, [5, 6]]
```

More List Functions

list.pop(obj=list[-1])

Removes and returns last object or obj from list

---

list.remove(obj)

Removes object obj from list

---

list.reverse()

Reverses objects of list in place

---

list.sort([func])

Sorts objects of list, use compare func if given

Deleting a value from list

```
#!/usr/bin/python3

list = ['physics', 'chemistry', 1997, 2000]

print (list)

del list[2]

print ("After deleting value at index 2 : ", list)
```

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :  ['physics', 'chemistry', 2000]
```

List Comprehension

```
string = 'Hello'
L = [1,14,5,9,12]
M = ['one', 'two', 'three', 'four', 'five', 'six']
```

List comprehension	Resulting list
`[0 for i in range(10)]`	`[0,0,0,0,0,0,0,0,0,0]`
`[i**2 for i in range(1,8)]`	`[1,4,9,16,25,36,49]`
`[i*10 for i in L]`	`[10,140,50,90,120]`
`[c*2 for c in string]`	`['HH','ee','ll','ll','oo']`
`[m[0] for m in M]`	`['o','t','t','f','f','s']`
`[i for i in L if i<10]`	`[1,5,9]`
`[m[0] for m in M if len(m)==3]`	`['o','t','s']`

Practice

- **Example 1** Write a program that generates a list L of 50 random numbers between 1 and 100.
- **Example 2** Replace each element in a list L with its square
- **Example 3** Count how many items in a list L are greater than 50
- **Example 4** Given a list L that contains numbers between 1 and 100, create a new list whose first element is how many ones are in L, whose second element is how many twos are in L, etc.

Example 1 Write a program that generates a list `L` of 50 random numbers between 1 and 100.

```
L = [randint(1,100) for i in range(50)]
```

Example 2 Replace each element in a list `L` with its square.

```
L = [i**2 for i in L]
```

Example 3 Count how many items in a list `L` are greater than 50.

```
len([i for i in L if i>50])
```

Example 4 Given a list `L` that contains numbers between 1 and 100, create a new list whose first element is how many ones are in `L`, whose second element is how many twos are in `L`, etc.

```
frequencies = [L.count(i) for i in range(1,101)]
```

Another example The `join` method can often be used with list comprehensions to quickly build up a string. Here we create a string that contains a random assortment of 1000 letters.

```
from random import choice
alphabet = 'abcdefghijklmnopqrstuvwxyz'
s = ''.join([choice(alphabet) for i in range(1000)])
```

MORE ON DICTIONARIES

Dictionary – Restrictions on KEY

- Only one item per key is allowed
- Value can be any type of object
- However, key must be IMMUTABLE
- For using a user defined object as a key you need to override two methods
  - __hash__(self)
  - __eq__(self, other)
- Details : stackoverflow.com/questions/490181 5 /object-of-custom-type-as-dictionary-key

Dictionary Functions

dict.clear()

Removes all elements of dictionary *dict.*

---

dict.copy()

Returns a shallow copy of dictionary *dict.*

---

dict.fromkeys()

Create a new dictionary with keys from seq and values *set* to *value*.

---

dict.get(key, default=None)

For *key* key, returns value or default if key not in dictionary.

Dictionary Functions

dict.has_key(key)

Removed, use the **in** operation instead.

---

dict.items()

Returns a list of *dict*'s (key, value) tuple pairs.

---

dict.keys()

Returns list of dictionary dict's keys.

---

dict.setdefault(key, default=None)

Similar to get(), but will set dict[key]=default if *key* is not already in dict.

Dictionary Functions

dict.update(dict2)

Adds dictionary *dict2*'s key-values pairs to *dict.*

---

dict.values()

Returns list of dictionary *dict*'s values.

MODULES

Modules

- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A **module is a Python object** with arbitrarily named attributes that you can bind and reference.

Example

- The Python code for a module named aname normally resides in a file named aname.py. Here is an example of a simple module, support.py

```
def print_func( par ):
   print "Hello : ", par
   return
```

Importing Modules

```
import module1[, module2[,... moduleN]
```

```
# Import module support
import support
# Now you can call defined function that module as follows
support.print_func("Zara")
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur.

```
from modname import name1[, name2[, ... nameN]]
```

Executing modules

- Within a module, the module's name (as a string) is available as the value of the global variable __name__.

- The code in the module will be executed, just as if you imported it,
but with the __name__ set to "__main__"

Executing Modules

```
#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
if __name__ == "__main__":
    f=fib(100)
    print(f)
```

Namespaces and Scoping

- Variables are names (identifiers) that map to objects.
- A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

Namespaces and Scoping

- A Python statement can access variables in a *local namespace* and in the *global namespace*.
- If a local and a global variable have the same name, the local variable shadows the global variable.
- In order to assign a value to a global variable within a function, you must first use the global statement

GuessTheOutput

```
def f():
    A = 5
```

```
A = 1
print(A)
f()
print(A)
```

GuessTheOutput

```
def f():
    global A
    A = 5


A = 1
print(A)
f()
print(A)
```

FILE I/O

Opening Files

```
file object = open(file_name [, access_mode][, buffering])
```

```
# 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)
fo.close()
```

This produces the following result-

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

Closing Files

```
fileObject.close();
```

Writing in Files

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

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

Reading Files

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

Resources Used

- Tutorials Point
- Google
- Python Documentation

Any Questions??