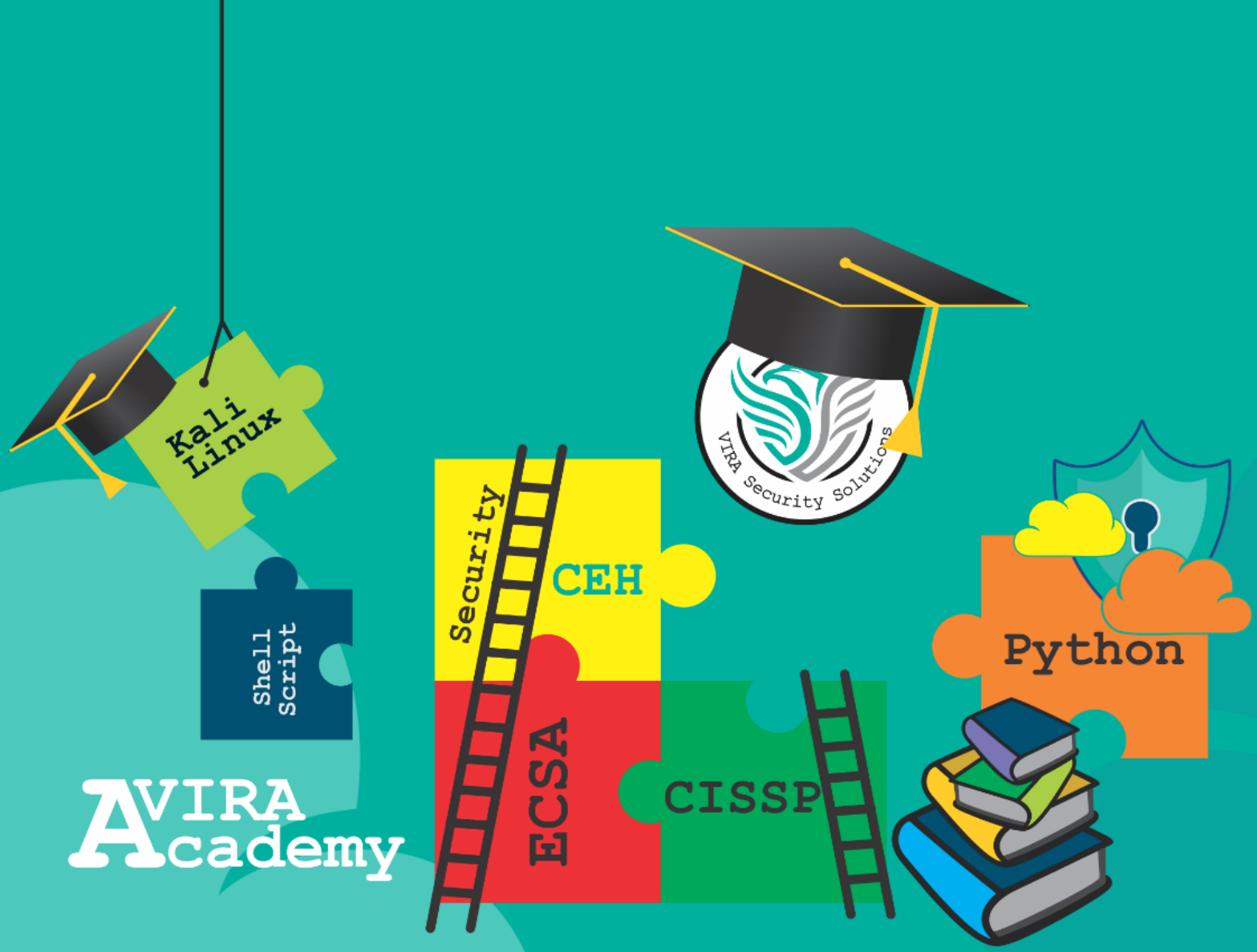

VIRA
Security
Solutions


Kali Linux
Shell Script
VIRA Academy
Security
CEH
ECSA
CISSP
VIRA Security Solutions
Python

Python

Part5

Mohammad Reza Gerami
mrgerami@aut.ac.ir
gerami@virasec.ir

Python Functions

Functions

A **function** is **a block of code** which only **runs** when it is **called**.

You can pass data, known as **parameters**, into a function.

A **function** can **return** data as a **result**.

Find Duplicates in Lists


```
list1=['a','b','c','b','d','m','n','n']
```

```
duplicates = []
for value in list1:
    if list1.count(value) > 1:
        duplicates.append(value)
print(duplicates)
```

['b', 'b', 'n', 'n']

```
duplicates = []
for value in list1:
    if list1.count(value) > 1:
        if value not in duplicates:
            duplicates.append(value)
print(duplicates)
```

['b', 'n']

Functions

Creating a Function

```
def my_function():
  print("Hello from a function")
```

Calling a Function

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments

Information can be **passed** into functions **as arguments**.

Arguments are specified after the function name, inside the parentheses.

You can add as many arguments as you want, just separate them with a comma.

```
def my_function(fname):
  print(fname + " Gerami")

my_function("Reza")
my_function("Darya")
my_function("Arya")
```

Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less

Functions

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function(“Darya", “Gerami")
```

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function(“Darya")
```

Functions


```
def jadval_zarb(rows, columns):
        for i in range(1,rows+1):
                for j in range(1,columns+1):
                        #print(i*j, end=' ')
                        print('{:>4}'.format(i* j), end=' ')
                print()

def do_continue():
        choice = str(input('continue(y/n)?')).lower()
        if(choice !='y'):
                return False
        return True

while True:
        rows = int(input("Enter rows:"))
        columns = int(input("Enter columns:"))
        jadval_zarb(rows, columns)
        if(not do_continue()):
                break
```

Functions arguments & parameters

Positional arguments

```
def say_hello(name,emoji):
    print(f'Hellooo {name} {emoji}')

say_hello('Vira','☺')
```

Functions arguments & parameters

Keyword arguments

```
def say_hello(name,emoji):
    print(f'Hellooo {name} {emoji}')

say_hello(name='Vira',emoni='☺')
say_hello(emoni='☺', name='Vira')
```

Functions arguments & parameters

Default arguments

```
def say_hello(name='Vira',emoji='☺'):
    print(f'Hellooo {name} {emoji}')

say_hello('Vira','☺')
say_hello('Reza','☺')
say_hello('Ali','☹')
say_hello('Tommy')
say_hello('☹')
```

Functions return

```
def sum(num1,num2):
    num1 + num2

print(sum(4,5))
```

Functions return


```
def sum(num1,num2):
    return num1 + num2

print(sum(4,5))

total = sum(10,15)
print(sum(10,total))

print(sum(10,sum(14,3)))
```

Functions return

```
def sum(num1,num2):
    def another_func(num1,num2):
      return num1 + num2
    return another_func(num1,num2)


print(sum(4,5))

total = sum(10,15)
print(sum(10,total))
```

Functions return


```
def sum(num1,num2):
    def another_func(n1,n2):
      return n1 + n2
    return another_func(num1,num2)


print(sum(4,5))

total = sum(10,15)
print(sum(10,total))
```

Functions return

```
def sum(num1,num2):
    def another_func(n1,n2):
        return n1 + n2
    return another_func(num1,num2)
    print('Hello')
    print(555)

    print(sum(4,5))

    total = sum(10,15)
    print(sum(10,total))
```

Methods vs Functions


```
list()
print()
max()
min()
input()
```

```
def some_random_func():
    pass

some_random_func
```

Method has to owned by something and who owns a method well whether is to the left of the dot in our case the string owns the method capitalized.

```
str1='hellllooooooo'
str1.capitalize()
str1.upper()
```

Clean Code

```
def is_even(num):
        if num % 2 == 0:
                return True
        elif num % 2 != 0:
                return False

print(is_even(51))
```

```
def is_even(num):
        if num % 2 == 0:
                return True
        else:
                return False

print(is_even(67))
```

Clean Code


```
def is_even(num):
    if num % 2 == 0:
        return True
    else:
        return False

print(is_even(68))
```

```
def is_even(num):
    return num % 2 == 0
print(is_even(68))
```

Args and Kwargs

*args **kwargs

```
def super_func(args):
    return sum(args)

print(super_func(1,2,3,4,5))
```

```
def super_func(*args):
    return sum(args)

print(super_func(1,2,3,4,5))
```

```
def super_func(*args):
    print(*args)
    return sum(args)

print(super_func(1,2,3,4,5))
```

Args and Kwargs

*args **kwargs

```
def varArguments(*args):
    total = 0;
    for number in args:
        total = total + number;
    return total;

print("1 + 2 + 3 = {0}".format(varArguments(1, 2, 3)));

print("1 + 2 + 3 + 4 = {0}".format(varArguments(1, 2, 3, 4)));

print("1 + 2 + 3 + 4 + 5 = {0}".format(varArguments(1, 2, 3, 4, 5)));
```

```
1 + 2 + 3 = 6
1 + 2 + 3 + 4 = 10
1 + 2 + 3 + 4 + 5 = 15
```

Args and Kwargs

*args **kwargs

```
def varArguments(number, *args):
    print("number = ", number);
    print("args = ", args);


varArguments(1, 2, 3);
```

```
number =  1
args =  (2, 3)
```

Args and Kwargs

*args **kwargs

```
def SomeFunction(*args, *args) #ERROR

def SomeFunction(*args, param1, param2) #ERROR

def SomeFunction(param1, param2, *args) #Correct
```

Args and Kwargs

*args **kwargs

```
def varArguments(number, *args):
    print("number = {0}".format(number));
    print("args = {0}".format(args));

varArguments(1, *(2,3,4), *(5,6,7,8));
```

```
number = 1
args = (2, 3, 4, 5, 6, 7, 8)
```

Args and Kwargs

*args **kwargs

*kwargs vs *args :

key value to save like a dictionary

```
def varArguments(**kwargs):
    print(kwargs);

varArguments(person1 = " Darya ", person2 = " Arya ", person3 = " Reza ");
```

{'person1': Darya', 'person2': Arya', 'person3': Reza'}

Exercise Functions


```
def highest_even(li):
        evens = []
        for item in li:
                if item % 2 == 0:
                        evens.append(item)
        return max(evens)


print(highest_even([10,2,3,4,8,11,15,13]))
```

Scope

What variables do I have access to?

```
print(name())

total = 100    -> global var
print(name())

def some_func():
        total = 100

print(total)
```

Scope

What variables do I have access to?

```
if True:
        x = 10


def some_func():
        total = 100

print(x)
```

```
if True:
        x = 10

def some_func():
        total = 100
        print(x)
```

Scope

local

```
a = 1
def confusion():
    a = 5
    return a

print(a)
print(confusion())
```

```
a = 1
def confusion():
    a = 5
    return a

print(confusion())
print(a)
```

```
a = 1
def confusion():
    return a

print(confusion())
print(a)
```

Scope

parent

```
a = 1
def parent():
    a = 10
    def confusion():
        a = 5
        return a
    return confusion()

print(parent())
print(a)
```

Scope

global

```
a = 1
def parent():
    def confusion():
        return a
    return confusion()

print(parent())
print(a)
```

Scope

python built-in functions

```
a = 1
def parent():
    def confusion():
        return sum
    return confusion()

print(parent())
print(a)
```

Global Keywords


```
total = 0
def count():
        global total
        Total += 1
        return total

count()
count()
print(count())
```

```
total = 0
def count(total):
         total
        total += 1
        return total

count(total)
count(total)
print(count(total))
print(count(count(count(total))))
```

Classes and Objects

Classes and Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class
To create a class, use the keyword class:

Example
Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Classes and Objects

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

Classes and Objects

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Classes and Objects

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note:
The __init__() function is called automatically every time the class is being used to create a new object.

Classes and Objects

object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Classes and Objects

object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
     print(f'Hi my name is {self.name} and I\’m {self.age}')

p1 = Person("John", 36)
p1.myfunc()
```

Classes and Objects

Note: The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words myobject and abc instead of self:

```
class Person:
  def __init__(myobject, name, age):
    myobject.name = name
    myobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Classes and Objects

Modify Object Properties

You can modify properties on objects like this:

Example
Set the age of p1 to 40:

```
p1.age = 40
```

Classes and Objects

Delete Object Properties

You can delete properties on objects by using the del keyword:

Example
Delete the age property from the p1 object:

```
del p1.age
```

Classes and Objects

Delete Objects

You can delete objects by using the del keyword:

Example
Delete the p1 object:

```
del p1
```

Classes and Objects

The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Example

```
class Person:
  pass
```

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.

Simply,

- A module is a file consisting of Python code.
- A module can define functions, classes and variables.
- A module can also include runnable code.

Modules

Consider a module to be the same as a code library.
A file containing a set of functions you want to include in your application.

Create a Module
To create a module just save the code you want in a file with the file extension .py:

Example
Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Modules

Use a Module

Now we can use the module we just created, by using the import statement:

Example:
Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

Modules

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example
Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Modules

Example
Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Modules

Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

Example
Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Modules

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example
Import and use the platform module:

```
import platform

x = platform.system()
print(x)
```

Modules

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module.
The dir() function:

Example
List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Note: The dir() function can be used on *all* modules, also the ones you create yourself.

Modules

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

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

Modules

The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax −

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

Modules

Import From Module

You can choose to import only parts from a module, by using the from keyword.
Example
The module named mymodule has one function and one dictionary:

mymodule.py

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example
Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module.
Example: `person1["age"]`, **not** `mymodule.person1["age"]`

Modules

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script −

```
#!/usr/bin/python

# Import module support
import support

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

Modules

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

The from...import Statement

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax −

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

For example, to import the function fibonacci from the module fib, use the following statement

from fib import Fibonacci

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

Modules

The from...import * Statement
It is also possible to import all names from a module into the current namespace by using the following import statement
from modname import *

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Modules

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences:

The current directory.

If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.

If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

Modules

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories.
The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

set PYTHONPATH = c:\python36\lib;
And here is a typical PYTHONPATH from a UNIX system −

set PYTHONPATH = /usr/local/lib/python

Files I/O

Printing to the Screen

The simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows −

```
#!/usr/bin/python

print "Python is really a great language,", "isn't it?"
```

This produces the following result on your standard screen

```
Python is really a great language, isn't it?
```

Files I/O

Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are −

raw_input
input

Files I/O

The raw_input Function

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
```

Files I/O

The raw_input Function

This prompts you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this −

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

Files I/O

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
```

This would produce the following result against the entered input −

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

Files I/O

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

The open Function
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.

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

Files I/O

Opening and Closing Files

Here are parameter details −

•file_name − The file_name argument is a string value that contains the name of the file that you want to access.

•access_mode − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. 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).

Files I/O

Sr.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**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.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**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.
6	**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.
7	**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.

Files I/O

Sr.No.	Modes & Description
8	**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.**
9	**a** **Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.**
10	**ab** **Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.**
11	**a+** **Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.**
12	**ab+** **Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.**

The *file* Object Attributes

The *file* Object Attributes

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

Here is a list of all attributes related to file object −

Sr.No.	Attribute & Description
1	**file.closed** Returns true if file is closed, false otherwise.
2	**file.mode** Returns access mode with which file was opened.
3	**file.name** Returns name of the file.
4	**file.softspace** Returns false if space explicitly required with print, true otherwise.

The *file* Object Attributes


```
#!/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
```

This produces the following result −

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

The close() Method

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()
```

The close() Method


```
#!/usr/bin/python

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

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

Reading and Writing Files

The file object provides a set of access methods to make our lives easier. We would see how to use read() and write() methods to read and write files.

The write() Method
The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string −

Syntax

fileObject.write(string)

Reading and Writing Files


```
#!/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()
```

The read() Method

The read() method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

The read() Method


```
#!/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()
```

Renaming and Deleting Files

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

The rename() Method
The rename() method takes two arguments, the current filename and the new filename.

Syntax

os.rename(current_file_name, new_file_name)

Renaming and Deleting Files

Following is the example to rename an existing file test1.txt −

```
#!/usr/bin/python
import os
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

Renaming and Deleting Files

The remove() Method
You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax
os.remove(file_name)

Following is the example to delete an existing file test2.txt

```
#!/usr/bin/python
import os
# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The os module has several methods that help you create, remove, and change directories.

The mkdir() Method
You can use the mkdir() method of the os module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax
os.mkdir("newdir")

Directories in Python


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

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

Directories in Python

The chdir() Method
You can use the chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

```
os.chdir("newdir")
```

Directories in Python


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

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

Directories in Python

The getcwd() Method
The getcwd() method displays the current working directory.

Syntax
os.getcwd()

Following is the example to give current directory −

```
#!/usr/bin/python
import os
# This would give location of the current directory
os.getcwd()
```

The rmdir() Method

The rmdir() method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax
os.rmdir('dirname')

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

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

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