UNIT-4

Modules: A module is a file containing Python definitions and statements. A module can define functions, classes and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use.

Example:
```
# A simple module, calc.py
def add(x, y):
    return (x+y)

def subtract(x, y):
    return (x-y)
```

The import Statement: We can use any Python source file as a module by executing an import statement in some other Python source file. When 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 for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script.

```
import calc
print add(10,2)
print substract(10,2)
```

Output : 12
8

Example 2:
```
import math
num = 4
print(math.sqrt(num))
```

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter, if it's just one module you want to test interactively, use reload(), for ex.

```
reload(module-name)
```

There are more ways to import modules:

1) from -- import statement
2) from -- import * statement
3) renaming the imported module.

1) from -- import statement: The from -- import statement allows you to import specific functions / variables from a module instead of importing everything.

In previous example, when you imported calc into module calc.py, both add and Subtract functions were imported. But what if you only needed the add() function in your code?

```
from calc import add
print (add (1,2))
```

In above example, only the add() function is imported and used. Notice the use of add()? You can now access it directly without using the module name. You can import multiple attributes as well, separating them with a comma in the import statement.

```
from calc import add, Subtract
```

- 2) from -- import * Statement: You can import all attributes of a module using this statement. This will make all attributes of imported module visible in your code.

```
from calc import *
print (add (1,2))
print (sub (3,2))
```

3) Renaming the imported module: You can rename the module you are importing, which can be useful in cases when you want to give a more meaningful name to the module or the module name is too large to use repeatedly. You can use the as keyword to rename it.

vice-versa ↔

```
import calc as calculation
print (calculation.add (1,2))
```

You saved yourself some typing time by renaming :)

NOTE: You can't use calc.add(1,2) anymore, as calc is no longer recongized in your program.

MODULE SEARCH PATH: You may need your modules to be used in different programs/projects and their physical location in the directory can be different. If you want to use a module residing in some other directory, you have some options provided by python.

When you import a module named calculation, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named Calculation.py in a list of directories given by the variable sys.path.

sys.path contains these locations:

- the directory containing the input script (or the current directory)
- PYTHOPATH (a list of directory names, with the same syntax as the shell variable PATH)
- the installation-dependent default.

Assume module-test.py is in the /home/datacamp/ directory, and you moved calculation.py to /home/test/. You can modify sys.path to include /home/test in the list of paths, in which the Python interpreter will search for the module. For this, you need to modify module-test.py in the following way:

```
import sys
sys.path.append('/home/test/')

import calculation
print(calculation.add(1,2))
```

Byte Compiled Files: Importing a module increases the execution time of programs. So python has some tricks to speed it up. One way is to create byte-compiled files with the extension .pyc

Internally, Python converts the source code into an intermediate form called byte code, it then translates this into the native language of your computer and then runs it. This .pyc file is useful when you import the module the next time from a different program - it will be much faster since a portion of the processing required for importing a module is already done.
Also, these byte-compiled files are platform-independent.

NOTE: These .pyc files are usually created in the same directory as the corresponding .py files. If python does not have permission to write to files in that directory, then the .pyc files will not be created.

The dir() function: The dir() function is used to find out all the names defined in a module. It returns a sorted list of strings containing the names defined in a mocdule.

```
import calculation
print (test.add(1,2))
print (dir (calculation))
```

Output:

['builtins', 'cached', 'doc', 'file', 'loader', 'name', 'package', 'spec', 'add', 'sub']

In the output, you can see the names of the functions you defined in the module, add & sub. Attribute __name__ contains the name of the module. All attributes beginning with an underscore are default python attributes associated with a module.

CONCLUSION: Creating a module is required for better management of code and reusability. Python provides you with some built in modules, which can be imported by using the import keyword. Python also allows you to create your own modules and use them in your programs. It gives you byte-compiled files to overcome the cost of using modules, which makes execution faster. You can use dir() to known the attributes defined in the module being used in a program, this can be used both on pre-defined modules as well as user-defined modules.

PYTHON EXCEPTIONS : An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.

Python is an interpreted language. When you are coding in the Python interpreter, you often have to deal with runtime errors. These are runtime exceptions when there is an exception, all execution stops and it displays a red error message on the screen. But if we can handle it, the program will not crash. So an exception is when something goes wrong and your program cannot run anymore.

```
print(1/0)
```

i) ⇒ ZeroDivisionError : division by zero.

It is not possible to divide 1 by 0, so the program has to stop, it cannot execute anymore. So it raises a ZeroDivisionError. This is an built-in exception in Python.

ii) TypeError:

```
a = 2
b = 'DataCamp'
print(a+b)
```

unsupported operand type(s) for +

Built-In Exceptions:

There are four main components of exception handling:

i) Try: It will run the code block in which you except an error to occur.

ii) Except: Here, you will define the type of exception you except in the try block (built in or custom)

iii) Else: If there isn't any exception, then this block of code will be executed (Consider this as a remedy or a fallback option if you except a part of your script of produce an exception).

iv) Finally: Irrespective of whether there is an exception or not, this block of code will always be executed.

Common Exceptions:

1. Zero division error: occurs when a number is divided with zero.
2. Name Error: when name is not found. local or global
3. Indentation Error: If incorrect indentation is given
4. IOError: when Input ouput operation fails
5. EOFError: When the end of file is reached, and yet operations are being performed.

Exceptions handling in Python:

try

{ Run this code }

except

{ Run this code if an exception occurs }

Syntax:

```
try:
    # block of code
except Exception 1:
    # block of code
except Exception 2:
    # block of code
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

```
try:
    # block of code
except exception 1:
    # block of code
else:
    # this code executes if no except block is executed
```

Example:

```
try:
    a = int (input ("Enter a"))
    b = int (input ("Enter b"))
    c = a/b;
    print ("a/b = %d %c)

except Exception:
    print ("can't divide by zero")

else:
    print ("Hi i am else block")
```

Output:

```
Enter a  10
Enter b  2
a/b = 5
Hi I am else block.
```

The except statement with no exceptions:

Python provides the flexibility not to specify the name of exception with the except statement.

```
try:
    a = int (input ("a"))
    b = int (input ("b"))
    c = a/b
    print ("a/b = %d" %c)
except:
    print ("can't divide by zero")
else:
    print ("Hi I am else block")
```

Output :

a : 10
b : 0
Can't divide by zero.

Points to remember:

1. Python facilitates us to not specify the exception with the except Statement.
2. We can declare multiple exceptions in the except Statement since the try block may contain the Statements which throws the different type of exceptions.
3. We can also specify an else block along with the try-except statement which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

Declaring multiple exceptions:

The ~~one~~ python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions

```
try:
    a = 10/0;
except ArithmeticError, StandardError:
    print("Arithmetic Exception")

else:
    print("Successfully Done")
```

Output: Arithmetic Exception

- The finally block: We can use the finally block with the try block in which, we can pace the important code which must be executed before the try statement throws an exception.

```
try:
    fileptr = open("file.txt", "r")
    try:
        fileptr.write("Hi I am good")
finally:
    fileptr.close()
    print("file closed")
except:
    print("Error")
```

Output: file closed
Error

ASSERTION IN PYTHON

Python assert keyword is defined as a debugging tool that tests a condition. The assertions are mainly the assumptions that asserts or state a fact confidently in the program. For example, while writing a division function, the divisor should not be zero, and you assert that division is not equal to zero. It is simply a boolean expression that has a condition or expression checks if the condition returns true or false. If it is true, the program does not do anything, and it moves to the next line of code. But if it's false, it raises an AssertionError exception with an optional error message.

The main task of assertions is to inform the developers about unrecoverable errors in the program like "file not found", and it is right to say that assertions are internal self-checks for the program. They work by declaring some conditions as impossible in your code. If one of the conditions does not hold, that means there is a bug in the program.

Why Assertion is used:

It is a debugging tool, and its primary task is to check the condition. If it finds that condition is true, it moves to the next line of code, and if not then stops all its operations and throws an error. It points out the error in the code.

Where Assertion in Python used:

1) Checking the outputs of the functions
2) Used for testing the code.
3) In checking the values of arguments.
4) Checking the valid input.

```
def avg (scores):
    assert len(scores)!=0 "The list is empty"
    return sum(scores)/len(scores)

scores2 = [67,59,86,75,92]
print ("The average of scores2:", avg (scores2))

scores1 = [ ]
print("The Average of scores1:", avg (scores1))
```

Output:

The average of scores2 : 75.8

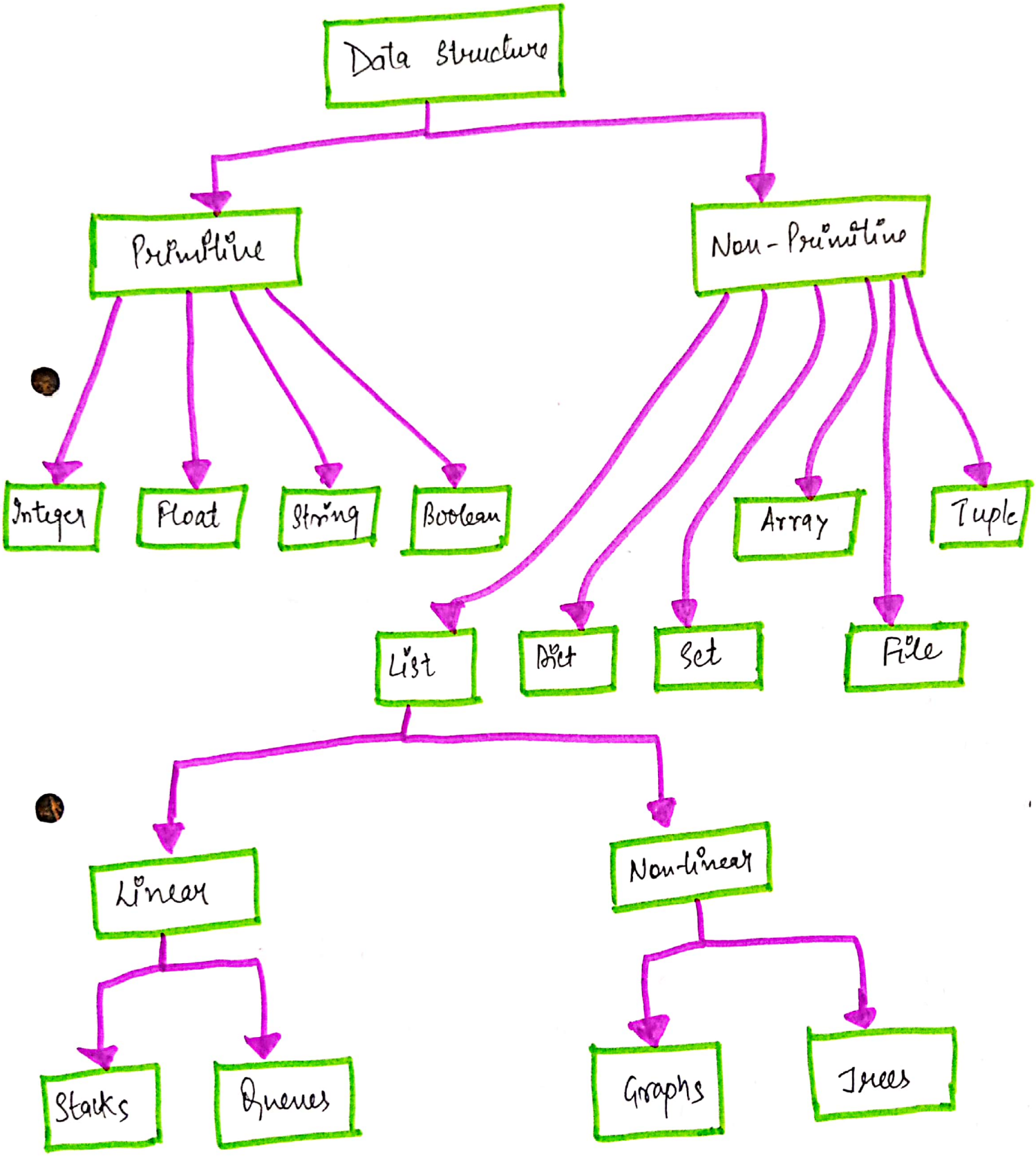
AssertionError: The list is empty.

Explanation: In the above example, we have passed a non-empty list scores2 and an empty list scores1 to the avg() function. We received an output for scores2 list successfully, but after that, we got an error AssertionError: List is empty. The assert condition is satisfied by the scores2 list and lets the program to continue to run. However, scores1 doesn't satisfy the condition and

- gives an AssertionError.

Practical Application: This has a much greater utility in testing and Quality assurance role in any development domain. Different types of assertions are used depending upon the application.

ABSTRACT DATA TYPES (ADT)



An Abstract data-type, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

1) ADT is reusable, robust, and is based on principles of Object Oriented Programming (OOP) and Software Engineering (SE).

2) An ADT can be re-used at several places and it reduces coding efforts.

3) Encapsulation ensures that data cannot be corrupted.

4) ADT ensures a robust data structure.

1 Robust:

ADT approach is based on SE concepts of coupling and cohesion Coupling property determines

- How strongly two separate parts of ~~coupling and~~ programs are linked together.
- Extend to which changes made in one part impacts the other parts of a software module.

In other words,

→ Earlier if a programmer wanted to read a file, the whole code was written to read the physical file device. So that is how Abstract Data type (ADT) came into existence.

→ The code to read a file was written and placed in a library and made available for everyone's use. This concept of ADT is being used in the

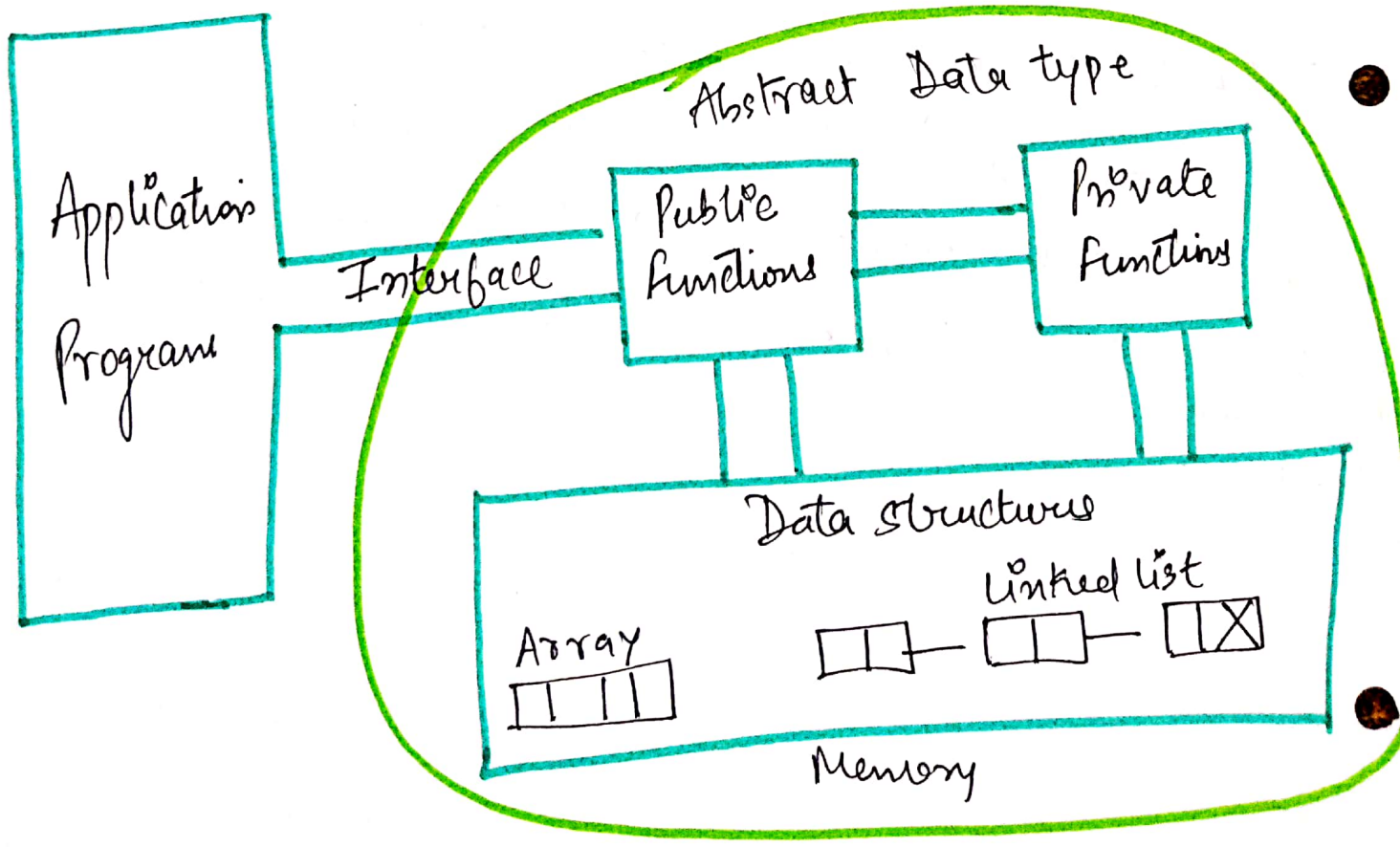
- modern languages nowadays.

Example: The code to read the Keyboard is an ADT. It has a data structure, a character, and a set of operations that can be used to read that data structure.

- Consider we want to maintain a record of quantity
- of items sold in a sale. At least three possibilities of data structure are there. One can use a linear list or an array or may be a vector to save space and dynamic memory allocation or a linked list so that there is no missing item. A user does not need to know the hidden implementations as long as the data is being entered and obtained correctly.

Abstract Data type Model:

There is an interface between Applications Program and the Abstract data type present at the right. ADT consists of the data structures and the functions (private and public) which are interconnected with each other Since, they are entirly present in the ADT so they are out of scope of the Application Program.



Python File Handling: File handling is an important part of any web application.
Python has several functions for creating, reading, updating, and deleting files.
The Key function for working with files in Python is the open() function.
The open() function takes two parameters; filename and mode.
There are four different methods (modes) for opening a file:

- 'r' - Read - Default value. Open a file for reading, error if file does not exist
- 'a' - Append - Opens a file for appending, creates the file if it does not exist
- 'w' - Write - Opens a file writing, creates the file if it does not exist
- 'x' - Create - Creates the specified file, returns an error if the file exists.

In addition you can specify if the file should be handled as binary or text mode.

- 't' - Text - Default value. Text mode
- 'b' - Binary - Binary Mode (e.g. images)

SYNTAX: To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because 'r' and 't' are for read and text respectively for text are the default values, you do not need to specify them.

NOTE: Make sure the file exists, or else you will get an error.

PYTHON FILE OPEN :

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
```

To open the file, use the built-in open() function.
The open() function returns a file object, which has a read() method for reading the content of the file:

eg:
```
f = open("demofile.txt", "r")
print(f.read())
```

Output:
```
Hello! Welcome to demofile.txt
This file is for testing purposes.
```

Read only parts of the file:

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example: Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Output: Hello

Read Lines: You can return one line by using the readline() method:

Example1: Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

Output: Hello! Welcome to demofile.txt

Eg 2: Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

Output: Hello! Welcome to demofile.txt
This file is for testing purposes.

By looping through the lines of the file, you can read the whole file, line by line:

Example: Loop through the file by line:

```
f = open("demofile.txt", "r")
for x in f:
    print(x)
```

CLOSE FILES: It is a good practice to always close the file when you are done with it.

Example: Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Output: Hello! Welcome to demofile.txt

NOTE: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

PYTHON FILE WRITE:

- Write to an Existing File: To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file.

"w" - Write - will overwrite any existing content.

Example: Open the file "demofile2.txt" and append the content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()
#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Output: Hello!

content!

Example: Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content")
f.close()
```

- # open and read the file after the appending:

```
f = open("demofile3.txt", "r")
print(f.read())
```

Output: Woops! I have deleted the content!

NOTE: The 'w' method will overwrite the entire file.

Create a New file: To create a new file in Python, use the open() method, with one of the following parameters:

- "x" - Create - will create a file, returns an error if the file exist.
- "a" - Append - will create a file if the specified file does not exist
- "w" - Write - will create a file if the specified file does not exist.

Example: Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created

Example: Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

PYTHON DELETE A FILE:

To delete a file, you must import the OS module, and run its os.remove() function:

Example: Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if file exist: To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example: Check if file exists, then delete it:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

You can only remove empty folders

Delete Folder: To delete an entire folder, use the os.rmdir() method:

```
import os
os.rmdir("myfolder")
```

File Positions: The tell() Methods tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the begining of the file.

The seek(offset, [, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If from is set to 0, it means use the begining of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference positions.

```
# open a file
format1 = open ("format1.txt", "r+")
    str = format1.read (10)
print ("Read String is :", str)
# check current position
position = format1.tell ()
print ("Current file position", position)
# Reposition pointer at the begining once again
position = format1.seek (0,0)
    str = format1.read (10)
```

```
print ("Again read string is ", Str)
# close opend file
fo.close()
```

Output:

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

Renaming Method: 1) The rename() Method

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

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

Example:

```
import os
os.rename("test1.txt", "test2.txt")
```

2) 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.

Example:

```
import os
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.

1) 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.

```
import os
os.mkdir("test")
```

2) 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.

```
import os
os.chdir("/home/newdir")
```

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

```
import os
os.getcwd() # will give you the location of the current directory
```

4) 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.

```
import os
os.rmdir("/tmp/test")
```

It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

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 Os module: The os module of Python allows you to perform operating system dependent operations such as making a folder, listing contents of a folder, know about a process, end a process etc. It has methods to view environment variables of the operating system on which Python is working on and many more.