## UNIT-4 PART - 1

### Syllabus:

**Part - 1**

File Operations: Understanding read functions, read (), readline () and readlines (), Understanding write functions, write () and writelines (), Manipulating file pointer using seek, Programming using file operations, Reading config files in python, Writing log files in python.

### Part - 2

Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support.

Design with Classes: Objects and Classes, Data modelling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism.

## Files in Python:

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 us with an important feature for reading data from the file and writing data into a file. Mostly, in programming languages, all the values or data are stored in some variables which are volatile in nature. Because data will be stored into those variables during run-time only and will be lost once the program execution is completed. Hence it is better to save these data permanently using 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.

## Types of File in Python: -

There are two types of files in Python and each of them are explained below they are:

* Binary file
* Text file

## Binary files in Python: -

Most of the files that we see in our computer system are called binary files. Example:

* + Document files: .pdf, .doc, .xl
  + Image files: .png, .jpg, .gif, .b
  + Video files: .mp4, .3gp, .mkv,
  + Audio files: .mp3, .wav, .mka,.
  + Database files: .mdb, .accde,.sqlite
  + Archive files: .zip, .rar, .iso, .7z etc.
  + Executable files: .exe, .dll, .class etc.

All binary files follow a specific format. We can open some binary files in the normal text editor, but we can’t read the content present inside the file. That is because all the binary files will be encoded in the binary format, which can be understood only by a computer. For handling such binary files, we need a specific type of software to open it.

# Text files in Python: -

Text files do not have any specific encoding and it can be opened in normal text editor itself. Example:

Web standards: html, XML, CSS, JSON etc. Source code: c, app, js, py, java etc.

Documents: txt, tex, RTF etc.Tabular data: csv, tsv etc.

Configuration: ini, cfg, reg etc.

## Opening and Closing Files

**The open () Method**

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 (filename, access mode)

### Here are parameter details –

**file\_name** − The file\_name argument is a string value that contains the name of the file that youwant 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).

### Here is a list of the different modes of opening a file –

|  |  |  |
| --- | --- | --- |
| **Sno** | **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. |  |

|  |  |
| --- | --- |
| 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:**

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 −

|  |  |
| --- | --- |
| **Sno** | **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. |

**Example:**

f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file #File object attributes

print('Name of the file: ', f.name)

print('Closed or not : ', f.closed) print('Opening mode : ', f.mode) f.close()

## 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. It is a good practice to use the close () method to close a file.

**Syntax:** fileObject.close()

### Example:

f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file #File object attributes

print('Name of the file: ', f.name)

print('Closed or not : ', f.closed) print('Opening mode : ', f.mode) f.close()

## Reading and Writing Files

The file object provides a set of access methods. Now, we will see how to use read (), readline (), readlines () and write (), writelines () methods to read and write files.

## Understanding write () and writelines ()

**The write () Method**

* The write () method writes any string (binary data and text data) to an open file.
* The write () method does not add a newline character ('\n') to the end of the string

**Syntax:** fileObject.write(string)

Here, passed parameter is the content to be written into the opened file.

### Example:

f=open('sample.txt','w') #creates a new file sample.txt give write permissions on file #writing content into file sample.txt using write method

f.write( "Python is a great language.") f.close()

## The writelines () method:

Python file method **writelines ()** writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value

.**Syntax:** fileObject.writelines(sequence)

### Parameters

**Sequence** − This is the Sequence of the strings.

**Return Value**-This method does not return any value.

### Example

f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file #writing content into file using write method

f.writelines (['python is easy\n','python is portable\n','python is comfortable']

)

f.close()

## Understanding read (), readline () and readlines ():

**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.

### Example

f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file #writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable']) f.close()

f=open('sample.txt','r')

#reading first 20 bytes from the file using read() method print(f.read(20))

## The readline () Method

Python file method **readline()**reads one entire line from the file. A trailing newline character is kept in the string. If the *size* argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately.

**Syntax:** fileObject.readline( size )

### Parameters

* **size** − This is the number of bytes to be read from the file.

### Return Value

* This method returns the line read from the file.

### Example

f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file #writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable']) f.close()

f=open('sample.txt','r')

#reading first line of the file using readline() method print(f.readline())

## The readlines () Method

Python file method **readlines()** reads until EOF using readline() and returns a list containing the lines. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totalling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read.

An empty string is returned only when EOF is encountered immediately.

**Syntax:** fileObject.readlines( sizehint )

### Parameters

* **sizehint** − This is the number of bytes to be read from the file.

### Return Value

This method returns a list containing the lines.

### Example

f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file #writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable']) f.close()

f=open('sample.txt','r')

#reading all the line of the file using readlines() method print(f.readlines())

## Manipulating file pointer using seek():

**tell ():** The tell () method tells you the current position within the file

**Syntax:** file\_object.tell()

### Example:

# Open a file

fo = open("sample.txt", "r+") str = fo.read(10)

print("Read String is : ", str) # Check current position = fo.tell()

print("Current file position : ", position)

**seek ():** The seek (offset, from\_what) method changes the current file position.

**Syntax:** f.seek(offset, from\_what) #where f is file pointer

### Parameters:

**Offset:** Number of postions to move forward

**from\_what:** It defines point of reference.

**Returns:** Does not return any value

The reference point is selected by the **from\_what** argument. It accepts three values:

**0:** sets the reference point at the beginning of the file **1:** sets the reference point at the current file position **2:** sets the reference point at the end of the file

By default from\_what argument is set to 0.

**Note:** Reference point at current position / end of file cannot be set in text mode except when offset is equal to 0.

### Example:

# Open a file

fo = open("sample.txt", "r+")

str = fo.read(10) print("Read String is : ", str)

# Check current 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

## File processing operations:

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.

1. **os.rename():** The rename() method takes two arguments, the current filename and the new filename.(to rename file)

**Syntax:** os.rename(current\_file\_name, new\_file\_name)

### Example:

**import os** **os.rename(‘sample.txt’,’same.txt’)**

1. **os.mkdir():** The mkdir() method takes one argument as directory name, that you want to create.(This method is used to create directory)

**Syntax:** os.mkdir(directory name)

### Example:

**import os**

### os.mkdir(‘python’) # Creates python named directory

1. **os.rmdir():** The rmdir() method takes one argument as directory name, that you want to remove.( This method is used to remove directory)

**Syntax:** os.rmdir(directory name)

### Example:

**import os**

### os.rmdir(‘python’) # removes python named directory

1. **os.chdir():** The chdir() method takes one argument as directory name which we want to change.( This method is used to change directory)

**Syntax:** os.chdir(newdir)

### Example:

**import os**

### os.chdir(‘D:\>’) # change directory to D drive

**os.remove():** The remove() method takes one argument, the filename that you want to remove.( This method is used to remove file)

**Syntax:** os.remove(filename)

### Example:

**import os**

### os.remove(‘python,txt’) # removes python.txt named file

**os.getcwd():** The getcwd() method takes zero arguments,it gives current working director.

**Syntax:** os.getcwd() **Example:**

### import os

**os.getcwd( ) # it gives current working directory**

**Writing and Reading Data from a Binary File :-**

Binary files store data in the binary format (0’s and 1’s) which is understandable by the machine. So when we open the binary file in ourmachine, it decodes the data and displays in a human- readable format. **Example:**

#Let’s create some binary file. my\_file = open("bfile.bin", "wb+") message = "Hello Python"

file\_encode = message.encode("ASCII") my\_file.write(file\_encode) my\_file.seek(0) bdata = my\_file.read() print("Binary Data:", bdata) ntext = bdata.decode("ASCII")

print("Normal data:", ntext)

In the above example, first we are creating a binary file ‘bfile.bin’ withthe read and write access and whatever data you want to enter into the file must be encoded before you call the write method.

Also, we are printing the data without decoding it, so that we can observehow the data exactly looks inside the file when it’s encoded and we are also printing the same data by decoding it so that it can be readable by humans.

### Output:

Binary Data: b’Hello Python’Normal data: Hello Python

### WRITING AND READING CONFIG FILES IN PYTHON

Config files help creating the initial settings for any project, they help avoiding the hardcoded data. For example, imagine if you migrate your server to a new host and suddenly your application stops working, now you have to go through your code and search/replace IP address of host at all the places. Config file comes to the rescue in such situation. You define the IP address key in config file and use it throughout your code. Later when you want to change any attribute, just change it in the config file. So this is the use of config file.

### Creating and writing config file in Python

In Python we have configparser module which can help us with creation of config files (.ini format).

### Program:

from configparser import ConfigParser #Get the configparser object config\_object = ConfigParser()

#Assume we need 2 sections in the config file, let's call them USERINFO and SERVERCONFIG

config\_object["USERINFO"] = { "admin": "Chankey Pathak", "loginid": "chankeypathak", "password": "tutswiki"

}

config\_object["SERVERCONFIG"] = {

"host": "tutswiki.com",

"port": "8080",

"ipaddr": "8.8.8.8"

}

#Write the above sections to config.ini file with open('config.ini', 'w') as conf:

config\_object.write(conf)

Now if you check the working directory, you will notice config.ini file has been created, below is its content.

[USERINFO]

admin = Chankey Pathak password = tutswiki loginid = chankeypathak

[SERVERCONFIG]

host = tutswiki.com ipaddr = 8.8.8.8

port = 8080

### Reading a key from config file:

So we have created a config file, now in your code you have to read the configuration data so that you can use it by “keyname” to avoid hardcoded data, let’s see how to do that

### Program:

from configparser import ConfigParser

#Read config.ini file config\_object = ConfigParser() config\_object.read("config.ini")

#Get the password

userinfo = config\_object["USERINFO"] print("Password is{}".format(userinfo["password"])) output:

Password is tutswiki

# Updating a key in config file

Suppose you have updated the password for chankeypathak user. You can update the same in config file using below:

from configparser import ConfigParser #Read config.ini file

config\_object = ConfigParser() config\_object.read("config.ini")

#Get the USERINFO section

userinfo = config\_object["USERINFO"]

#Update the password userinfo["password"] = "newpassword"

#Write changes back to file

with open('config.ini', 'w') as conf: config\_object.write(conf)

# WRITING LOG FILES IN PYTHON:

Logging is a very useful tool in a programmer’s toolbox. It can help you develop a better understanding of the flow of a program and discover scenarios that you might not even have thought of while developing.

Logs provide developers with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information, like which user or IP accessed the application. If an error occurs, then they can provide more insights than a stack trace by telling you what the state of the program was before it arrived at the line of code where the error occurred.

By logging useful data from the right places, you can not only debug errors easily but also use the data to analyze the performance of the application to plan for scaling or look at usage patterns to plan for marketing.

Python provides a logging system as a part of its standard library, so you can quickly add logging to your application. In this article, you will learn why using this module is the best way to add logging to your application as well as how to get started quickly, and you will get an introduction to some of the advanced features available.

## The Logging Module

The [logging module](https://realpython.com/python-logging-source-code/) in Python is a ready-to-use and powerful module that is designed to meet the needs of beginners as well as enterprise teams. It is used by most of the third-party Python libraries, so you can integrate your log messages with the ones from those libraries to produce a homogeneous log for your application.

Adding logging to your Python program is as easy as this:

import logging

With the logging module imported, you can use something called a “logger” to log messages that you want to see. By default, there are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:

* DEBUG
* INFO
* WARNING
* ERROR
* CRITICAL

The logging module provides you with a default logger that allows you to get started without needing to do much configuration. The corresponding methods for each level can be called as shown in the following example:

import logging

logging.debug('This is a debug message')

logging.info('This is an info message')

logging.warning('This is a warning message')

logging.error('This is an error message')

logging.critical('This is a critical message')

The output of the above program would look like this:

WARNING:root:This is a warning message

ERROR:root:This is an error message

CRITICAL:root:This is a critical message

The output shows the severity level before each message along with root, which is the name the logging module gives to its default logger. (Loggers are discussed in detail in later sections.) This format, which shows the level, name, and message separated by a colon (:), is the default output format that can be configured to include things like timestamp, line number, and other details.

Notice that the debug() and info() messages didn’t get logged. This is because, by default, the logging module logs the messages with a severity level of WARNING or above.

### Logging Variable Data

In most cases, you would want to include dynamic information from your application in the logs. You have seen that the logging methods take a string as an argument, and it might seem natural to format a string with variable data in a separate line and pass it to the log method. But this can actually be done directly by using a format string for the message and appending the variable data as arguments. Here’s an example:

import logging

name = 'John'

logging.error('%s raised an error', name)

ERROR:root:John raised an error

The arguments passed to the method would be included as variable data in the message.

While you can use any formatting style, the [f-strings](https://realpython.com/python-f-strings/) introduced in Python 3.6 are an awesome way to format strings as they can help keep the formatting short and easy to read:

import logging

name = 'John'

logging.error(f'{name} raised an error')

ERROR:root:John raised an error

### Capturing Stack Traces

The logging module also allows you to capture the full stack traces in an application. [Exception information](https://realpython.com/python-exceptions/) can be captured if the exc\_info parameter is passed as True, and the logging functions are called like this:

import logging

a = 5

b = 0

try:

c = a / b

except Exception as e:

logging.error("Exception occurred", exc\_info=True)

ERROR:root:Exception occurred

Traceback (most recent call last):

File "exceptions.py", line 6, in <module>

c = a / b

ZeroDivisionError: division by zero

[Finished in 0.2s]

If exc\_info is not set to True, the output of the above program would not tell us anything about the exception, which, in a real-world scenario, might not be as simple as a ZeroDivisionError. Imagine trying to debug an error in a complicated codebase with a log that shows only this:

ERROR:root:Exception occurred

Here’s a quick tip: if you’re logging from an exception handler, use the logging.exception() method, which logs a message with level ERROR and adds exception information to the message. To put it more simply, calling logging.exception() is like calling logging.error(exc\_info=True). But since this method always dumps exception information, it should only be called from an exception handler. Take a look at this example:

import logging

a = 5

b = 0

try:

c = a / b

except Exception as e:

logging.exception("Exception occurred")

ERROR:root:Exception occurred

Traceback (most recent call last):

File "exceptions.py", line 6, in <module>

c = a / b

ZeroDivisionError: division by zero

[Finished in 0.2s]

Using logging.exception() would show a log at the level of ERROR. If you don’t want that, you can call any of the other logging methods from debug() to critical() and pass the exc\_info parameter as True