Bansilal Ramnath Agrawal Charitable Trust's

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY

Department of Engineering &Applied Sciences

F. Y. B. Tech.

Course Material (A Brief Reference Version for Students)

Course: Python for Engineers

UNIT-IV : **Python File Operation**

Unit IV - Python file operation
Reading config files in python, Writing log files in python, Understanding read functions, read(), readline() and readlines(). Understanding write functions, write() and writelines(). Manipulating file pointer using seek. Programming using file operations.

1. How Python handles files:

If you are working in a large software application where they process a large number of data, then we cannot expect those data to be stored in a variable as the variables are volatile in nature. Hence when are you about to handle such situations, the role of files will come into the picture. As files are non-volatile in nature, the data will be stored permanently in a secondary device like Hard Disk and using python we will handle these files in our applications.

Let's take an **Example** of how normal people will handle the files. If we want to read the data from a file or write the data into a file, then, first, we will open the file or will create a new file if the file does not exist and then perform the normal read/write operations, save the file, and close it. Similarly, we do the same operations in python using some in-built methods or functions.

Types of File in Python

There are two types of files in Python and each of them are explained below in detail with examples. They are Binary file and Text file

Binary files in Python

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

- Document files: **.pdf, .doc, .xls etc**
- Image files: **.png, .jpg, .gif, .bmp etc**
- Video files: **.mp4, .3gp, .mkv, .avi etc**
- Audio files: **.mp3, .wav, .mka, .aac etc**
- Database files: **.mdb, .accde, .frm, .sqlite etc**
- 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's because all the binary files will be encoded in the binary format, which can be understood only by a computer or machine.

For handling such binary files, we need a specific type of software to open it.
For Example, you need Microsoft word software to open .doc binary files. Likewise, you need a pdf reader software to open .pdf binary files and you need a photo editor software to read the image files and so on.

Text files in Python

Text files don't have any specific encoding and it can be opened in normal text editor itself.

- **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.

In this tutorial, we will see how to handle both text as well as binary files with some classic examples.

Python File I/O

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk). Since RAM (random access memory) is a volatile memory, (which lose its data when the computer is turned off), we use files for further use of the data by permanently storing them. When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (i.e. perform operation)
3. Close the file

Opening Files in Python

Python has a built-in *open()* function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.
Examples:

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python38/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read *r*, write *w* or append *a* to the file. We can also specify if we want to open the file in text mode or binary mode. The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

Examples:

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
```

```
f = open("img.bmp",'r+b') # read and write in binary mode
```

Unlike other languages, the character *a* does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings). Moreover, the default encoding is platform dependent. In windows, it is cp1252 but utf-8 in linux. So, we must not rely on the default encoding or else our code will behave differently in different platforms. Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

Example:

```
f = open("test.txt", mode='r', encoding='utf-8')
```

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file. Closing a file will free up the resources that were tied with the file. It is done using *close()* method available in Python. Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

Example:

```
f = open("test.txt", encoding = 'utf-8')
# perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try…finally block.

```
try:
   f = open("test.txt", encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we can close the file properly even if an exception is raised that causes program flow to stop. The best way to close a file is by using the *with* statement. This ensures that the file is closed when the block inside the *with* statement is exited. We do not need to explicitly call the close() method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
  # perform file operations
```

Writing to Files in Python

In order to write into a file in Python, we need to open it in write *w*, append *a* or exclusive creation *x* mode. We need to be careful with the *w* mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.
Writing a string or sequence of bytes (for binary file) is done using the ***write()*** method. This method returns the number of characters written to the file.

```
  f.write("my first file\n")
  f.write("This file\n\n")
  f.write("contains three lines\n")
```

This program will create a new file named *test.txt* in the current directory if it does not exist. If it does exist, it is overwritten. We must include the newline characters ourselves to distinguish the different lines.

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

Following is the syntax for **writelines()** method −

fileObject.writelines(sequence)

Parameters

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

Return Value

This method does not return any value.

Example:

```
f = open("mytext.txt", "a")
f.writelines(["Well done!", "Students"])
f.close()
```

Reading Files in Python

To read a file in Python, we must open the file in reading *r* mode. There are various methods available for this purpose. We can use the *read(size)* method to read in the *size* number of data. If the *size* parameter is not specified, it reads and returns up to the end of the file.
We can read the *text.txt* file we wrote in the above section in the following way:

```
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '

>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'

>>> f.read()  # further reading returns empty sting
''
```

We can see that the *read()* method returns a newline as *'\n'*. once the end of the file is reached, we get an empty string on further reading.

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

Following is the syntax for **readline()** method −

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

Read the first line of the file "demofile.txt":

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

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

Following is the syntax for **readlines()** method −

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:

Return all lines in the file, as a list where each line is an item in the list object:

```
f = open("mytext.txt", "r")
print(f.readlines())
```

File Position

*We can change our current file cursor (position) using the ***seek()*** method. Similarly, the ***tell()*** method returns our current position (in number of bytes).
* will be covered in detailed too

```
>>> f.tell()    # get the current file position
56

>>> f.seek(0)   # bring file cursor to initial position
0

>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a for loop. This method is efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

In this program, the lines in the file itself include a newline character *\n*. So, we use the end parameter of the *print()* function to avoid two newlines when printing.

Alternatively, we can use the *readline()* method to read individual lines of a file. This method reads a file till the newline, including the newline character.

Example:

```
>>> f.readline(1)
'This is my first file\n'

>>> f.readline(2)
'This file\n'

>>> f.readline(3)
'contains three lines\n'

>>> f.readline()
''
```

Manipulating file pointer using seek()

Python file method seek() sets the file's current position at the offset. The whence argument is optional and default to 0, which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end. There is no

return value. Note that if the file is opened for appending using either ‘a’ or ‘a+’, any seek() operations will be undone at next write. If the file is only opened for writing in append mode using ‘a’, this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode ‘a+’). If the file is opened in text mode using ‘t’, only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seek able.

Syntax:-

fileobject.seek(offset[,whence])

Parameters:

1. offset- This is the position of the read / write pointer within the file.
2. whence- This is optional and default to 0 means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file’s end.

Return Value:

This method does not return any value.

Example:

The following example shows the usage of seek() method.

Python is a great language

Python is a great language

```
#open and read the file after the appending:
f = open("mytextfile.txt", "r")
print(f.read(4))
print(f.read(4))
print(f.tell())
print(f.seek(10))
print(f.read())
```

When we run this above program we get the following result:

```
This is my first python file This file contains three lines All the Best
This
is
8
10
first python file This file contains three lines All the Best
```

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)

Example

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

```
import os

# Rename a file from test1.txt to test2.txt

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

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)

Example

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

```
import os

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

Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
read(n)	Reads at most n characters from the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(n=-1)	Reads and returns one line from the file. Reads in at most n bytes if specified.
readlines(n=-1)	Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified.
seek(offset,from =SEEK_SET)	Changes the file position to offset bytes, in reference to from (start, current, end).
tell()	Returns the current file location.
writable()	Returns True if the file stream can be written to.

write(s)	Writes the string s to the file and returns the number of characters written.
writelines(lines)	Writes a list of lines to the file.

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")
```

Example

Following is the example to create a directory *test* in the current directory −

```
import os

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

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')

Example

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.

```
import os

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

Errors and Exceptions:

There are atleast 2 kinds of errors like syntax errors and exceptions.

Syntax errors:

Also known as Parsing errors.

So lets say for eg:

```
while True print(“Hello world!”)
```

^ invalid syntax

File “<stdin>”, line1

Since a colon is missing before it, an error is caused by the token preceding the arrow. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions:

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks −

try:
   You do your operations here;
   ....
except *ExceptionI*:
   If there is ExceptionI, then execute this block.
except *ExceptionII*:
   If there is ExceptionII, then execute this block.
   ....
else:
   If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example:

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

Config files

ConfigObj is a simple but powerful config file reader and writer: an *ini file round tripper*. Its main feature is that it is very easy to use, with a straightforward programmer's interface and a simple syntax for config files. It has lots of other features though:

- Nested sections (subsections), to any level
- List values
- Multiple line values
- String interpolation (substitution)
- Integrated with a powerful validation system
    - including automatic type checking/conversion
    - repeated sections
    - and allowing default values
- When writing out config files, ConfigObj preserves all comments and the order of members and sections
- Many useful methods and options for working with configuration files (like the 'reload' method)
- Full Unicode support

The config files that ConfigObj will read and write are based on the 'INI' format. This means it will read and write files created for ConfigParser [4].

Keywords and values are separated by an '=', and section markers are between square brackets. Keywords, values, and section names can be surrounded by single or double quotes. Indentation is not significant, but can be preserved. Subsections are indicated by repeating the square brackets in the section marker. You nest levels by using more brackets. You can have list values by separating items with a comma, and values spanning multiple lines by using triple quotes (single or double).

First we need to write a config.ini file:

```
[USERINFO]
admin = Python for Engineers
loginid = FY_Python
password = newpassword

[SERVERCONFIG]
host = Vishwakarma Institute of Information Technology
```

```
port = 8080
ipaddr = 8.8.8.8
```

Writing a Config file with .py extension

```
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:

In the Console we observe the data in (config.ini) file

Lets change the contains of the config file by creating a new python file and accessing the config files.

```
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": "Python for Engineers",
    "loginid": "FY_Python",
    "password": "easytolearn"
}

config_object["SERVERCONFIG"] = {
    "host": "Vishwakarma Institute of Information Technology",
    "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)
```

Output:

In the Console we observe the data in (config.ini) file as well as in the config.ini files

Log files

Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the *level* or *severity*.

Logging provides a set of convenience functions for simple logging usage. These are **debug()**, **info()**, **warning()**, **error()** and **critical()**. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

Task you want to perform	**The best tool for the task**
Display console output for ordinary usage of a command line script or program	**print()**
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	**logging.info()** (or **logging.debug()** for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	**warnings.warn()** in library code if the issue is avoidable and the client application should be modified to eliminate the warning **logging.warning()** if there is nothing the client application can do about the situation, but the event should still be noted

Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	**logging.error()**, **logging.exception()** or **logging.critical()** as appropriate for the specific error and application domain

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

The default level is WARNING, which means that only events of this level and above will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console.

Logging to a file

A very common situation is that of recording logging events in a file, so let's look at that next. Be sure to try the following in a newly-started Python interpreter, and don't just continue from the session described above:

```
import logging
logging.basicConfig(filename='example.log',level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

And now if we open the file and look at what we have, we should find the log messages:

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
```

This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to DEBUG, all of the messages were printed.

The call to **basicConfig()** should come *before* any calls to **debug()**, **info()** etc. As it's intended as a one-off simple configuration facility, only the first call will actually do anything: subsequent calls are effectively no-ops.

If you run the above script several times, the messages from successive runs are appended to the file *example.log*. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the *filemode* argument, by changing the call in the above example to:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.