File Handling

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 Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future 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 (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.

```
>>> 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)

```
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
```

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 the **close()** method available in Python.

```
f = open("test.txt")
# 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")
   # perform file operations
finally:
   f.close()
```

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 files) is done using the write() method. This method returns the number of characters written to the file.

```
with open("test.txt",'w') as f:
    f.write("This is a test file\n")
    f.write("By Python with Naveen \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.

Example

```
file_name = input("Enter a File Name : ")
file = open(file_name,"a")
text = input("Enter Some Text")
file.write(text)
file.close()
print("Text Written to File")
```

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.

```
>>> f = open("test.txt",'r')
>>> f.read(4)    # read the first 4 data   :    'This'
>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'
```

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.

We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

```
>>> 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 is both 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.

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

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

Lastly, the readlines() method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

```
# Example to open a file and read data from it.
# by default the open function will take "r" mode

file = open("test.txt","r")
text = file.read() # reading the entire contents of the file.
print(text)
file.close()

# reading 1st line from test.txt

file = open("test.txt")
text = file.readline()
print(text)
file.close()
```

```
# reading 1st line 8char's from test.txt

file = open("test.txt")
text = file.readline(8)
print(text)
file.close()

# reading all lines from test.txt

file = open("test.txt")
no_lines = file.readlines()
print(no_lines)
print("3rd line : ",no_lines[2])
file.close()

# Exception Handling in File Handling
# If the Given file is not available the open function will
# return FileNotFoundError


try:
    file = open("sample.txt")
    no_lines = file.readlines()
    print(no_lines)
    file.close()
except FileNotFoundError:
    print("File not Found")
```

```
# Create a new file and write employee information into the file.
# like Employee idno, name, salary

idno = int(input("Employee IDNO :"))
name = input("Employee NAME : ")
salary = float(input("Employee SALARY : "))

file = open("employee_info.txt","w")

file.write(idno)
file.write(name)
file.write(salary)

file.close()
print("Data Written to File")
```

Output

```
Employee IDNO :101

Employee NAME : Ravi

Employee SALARY : 185000.00

Traceback (most recent call last):

  File "F:/Naveen Class Room/Python 8pm/File Handling/Demo10.py", line 12, in <module>

    file.write(idno)

TypeError: write() argument must be str, not int
```

The above example will give exception because we cannot write int object or float object into the file using write method.

The Write method is used to write only string objects.

Note: To Overcome the above problem we need to work with python pickle package.