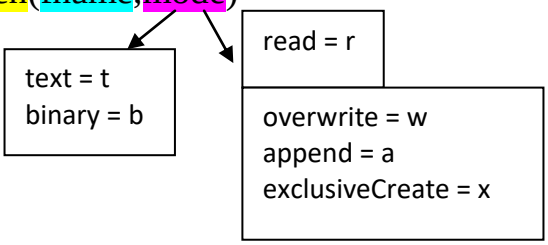
PART A.

Often it is not enough to just display the data on the screen, it becomes necessary to store the data in a manner that can be later retrieved and displayed either in part or in whole. This medium is usually a '**file**' on the disk. We will discuss how file I/O operations can be performed.

OPENING FILES:

FileConnectionObject=open(fname,mode)



- f1 = open('mydoc1', 'r')
- f2 = open('mydoc2', 'w')
- bf1 = open('mydoc3', 'rb')
- bf2 = open('mydoc4', 'wb')

Python File Modes	Mode Description
r	Open a file for reading. (default)
w	Open a file for writing. Creates a new file if it does not exist or overwrite the file if it exists.
a	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
x	Open a file for exclusive creation. If the file already exists, the operation fails.
r+	**Open for read and write**
w+	**Open for read and write (see w above)**
a+	**Open for read and write (see a above)**
t	Open in text mode. (default)
b	Open in binary mode.

PYTHON FILE METHODS

Python FileObject Methods	Methods Description
close()	**Close an open file**. It has no effect if the file is already closed.
flush()	Flush the file buffer
detach()	Separate the underlying binary buffer from the TextIOBase and return it.
fileno()	Return an integer number (file descriptor) of the file.
isatty()	Return True if the file stream is interactive.
read(n)	Read at most **n characters** form 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)	Read and **return one line** from the file. Reads in at most **n** bytes if specified.
readlines(n=-1)	Read and return a **list of lines** from the file. Reads in at most **n** bytes/characters if specified.
seek(offset,from=SEEK_SET)	Change the **file position** to offsetbytes, in reference to from (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(size=None)	Resize the file stream to size bytes. If size is not specified, resize to current location.
writable()	Returns True if the file stream can be written to.
write(s)	Write **string s to the file** and ***return the number of characters*** written.
writelines(lines)	Write **a list of lines** to the file.

PLAINTEXT FILES

Plaintext files contain only basic text characters and do not include font, size, or color information. Text files with the .**txt** extension or Python script files with the **.py** extension are examples of plaintext files. These can be opened with Windows's Notepad or OS X's TextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

BINARY FILES

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense. Every different type of binary file must be handled in its own way.

There are three steps to reading or writing files in Python.

- Call the open() function to return a File object.
- Call the read() or write() method on the File object.
- Close the file by calling the close() method on the File object.

There Are Different File Operations That Can Be Carried Out On A File. These Are:

- Creation of a new file
- Opening / Reading an existing file
- Writing to a file
- Moving to a specific location in a file (seeking)
- Closing a file

Input

The **read()** method is used to read bytes directly into a string, reading at most the number of bytes indicated. If no *size* is given (the default value is set to integer -1) or *size* is negative, the file will be read to the end. It will be phased out and eventually removed in a future version of Python.

The **readline()** method reads one line of the open file (reads all bytes until a line-terminating character like NEWLINE is encountered). The line, including termination character(s), is returned as a string. Like read(), there is also an optional *size* option, which, if not provided, defaults to -1, meaning read until the line-ending characters (or EOF) are found. If present, it is possible that an incomplete line is returned if it exceeds *size* bytes.

The **readlines()** method does not return a string like the other two input methods. Instead, it reads all (remaining) lines and returns them as a list of strings. Its optional argument, *sizhint*, is a hint on the maximum size desired in bytes. If provided and greater than zero, approximately *sizhint* bytes in whole lines are read (perhaps slightly more to round up to the next buffer size) and returned as a list.

Output

The **write()** built-in method has the opposite functionality as read() and readline(). It takes a string that can consist of one or more lines of text data or a block of bytes and writes the data to the file.

The **writelines()** method operates on a list just like readlines(), but takes a list of strings and writes them out to a file. Line termination characters are not inserted between each line, so if desired, they must be added to the end of each line before writelines() is called.

When reading lines in from a file using file input methods like read() *or* readlines(), *Python does not remove the line termination characters. It is up to the programmer. For example, the following code is fairly common to see in Python code:*

```
>>>f = open('myFile', 'r')
>>>data = [line.strip() for line in f.readlines()]
>>>f.close()
```

Similarly, output methods like write() *or* writelines() *do not add line terminators for the programmer... you have to do it yourself before writing the data to the file.*

Intra-file Motion

The **seek()** method (analogous to the fseek() function in C) moves the file pointer to different positions within the file. The offset in bytes is given along with a *relative offset* location, *whence*. A value of 0, the default, indicates distance from the beginning of a file (note that a position measured from the beginning of a file is also known as the *absolute offset*), a value of 1 indicates movement from the current location in the file, and a value of 2 indicates that the offset is from the end of the file.

If you have used fseek() as a C programmer, the values 0, 1, and 2 correspond directly to the constants SEEK_SET, SEEK_CUR, and SEEK_END, respectively. Use of the seek() method comes into play when opening a file for read and write access.

tell() is a complementary method to seek(); it tells you the current location of the filein bytes from the beginning of the file.

File Iteration

Going through a file line by line is simple:

```
>>>for eachLine in f:
        -----
```

Inside this loop, you are welcome to do whatever you need to with **eachLine**, representing a single line of the text file (which includes the trailing line separators).

Others

The **close()** method completes access to a file by closing it. The Python garbage collection routine will also close a file when the file object reference has decreased to zero. One way this can happen is when only one reference exists to a file, say, fp = open(...), and fp is reassigned to another file object before the original file is explicitly closed. Good programming style suggests closing the file before reassignment to another file object. It is possible to lose output data that is buffered if you do not explicitly close a file.

The **fileno()** method passes back the file descriptor to the open file. This is an integer argument that can be used in lower-level operations such as those featured in the os module, i.e., os.read().

flush() : Rather than waiting for the (contents of the) output buffer to be written to disk, calling the **flush()** method will cause the contents of the internal buffer to be written (or flushed) to the file immediately.

The **TRuncate()** method truncates the file to the size at the current file position or the given *size* in bytes.

PROGRAMMING WITH PYTHON 3.6. – FILES & DIRECTORIES

SET THE WORKSPACE PATH USING THE TERMINAL:

```
>>> import os
>>> os.chdir("your workspace path…….")
```

CREATION OF A NEW FILE

- In order ***to write into a file we need to open it in write 'w', append 'a' or exclusive creation 'x' mode.***
- If the given filename already exist then with the **'w'** mode as it will overwrite into the file.
- A file object represents a connection to a file, not the file itself, but if you open a file for writing that doesn't exist, Python creates the file automatically.
- Parameter "**b**" is used for binary file.
- Writing a string (**for text files**) or sequence of bytes (**for binary files**) is done using **write()** method. This method returns the number of characters written to the file.

```
filename = input('Enter file name: ')
f = open(filename, 'w')
while True:
        aLine = input("Enter a line ('.' to quit): ")
        if aLine != ".":
                f.write('%s%s' % (aLine, os.linesep)
        else:
                break
f.close()
```

OPEN AN EXISTING FILE

```
filename = input('Enter file name: ')
f = open(filename, 'r')
allLines = f.readlines()
f.close()

for eachLine in allLines:
        print eachLine

filename = input('Enter file name: ')
f = open(filename, 'r')

for eachLine in f:
        print eachLine,
f.close()
```

EX1:

CREATING TEXT FILES

The file is created in the current directory because we have not specified the directory, so it has created the file **hello.txt** in the current directory. If we open that file we can see the below output:

```
>>>#file handling operations; writing to a new file hello.txt via script
>>>f = open('hello.txt', 'w', encoding = 'utf-8')
>>>f.write("Hello Python Developers! ")
>>>f.write("Welcome to Python World")
>>>f.close()
>>> f = open('hello.txt')
>>> data=f.read()
>>> data
'Hello Python Developers! Welcome to Python World'


>>> f = open('hello.txt', 'w', encoding = 'utf-8')
>>> f.write("Hello Python Developers! ")
25
>>> f.write("Welcome to Python World")
23
>>> f.flush()
>>> f.close()


>>> f = open('hello.txt')
>>> f
<_io.TextIOWrapper name='hello.txt' mode='r' encoding='cp1252'>
>>> data=f.read()
>>> data
'Hello Python Developers! Welcome to Python World'

>>>f = open('hello.txt', 'w', encoding = 'utf-8')
>>>f.write("Hello Python Developers! ")
>>>f.write("Welcome to Python World ")
>>>mylist =["Apple", "Orange", "Banana"]
>>>f.writelines(mylist)
>>>f.flush()
>>>f.close()
```

The file is written into the disk with the following:

```
>>> f = open('hello.txt')
>>> data=f.read()
>>> data
'Hello Python Developers! Welcome to Python World'AppleOrangeBanana
```

Unlike other languages like C# the **writelines()** function in Python ***writes a sequence of text***. The text is not written line by line in the file. If we want the text to be written line by line in the text file then we have to modify our code and insert a **'\n'**.

```
>>>#file handling operations; writing to a new file hello.txt
>>>f = open('hello.txt', 'w', encoding = 'utf-8')
>>>f.write("Hello Python Developers!\n")
>>>f.write("Welcome to Python World\n")

>>>mylist =["Apple\n", "Orange\n", "Banana\n"]
>>>#writelines() is used to write multiple lines in to the file
>>>f.writelines(mylist)

>>>f.flush()
>>>f.close()
```

Now if we open the file, we could see the following output:

```
>>> f = open('hello.txt')
>>> data=f.read()
>>> data
Hello Python Developers!
Welcome to Python World
Apple
Orange
Banana
```

EX2:

CREATING TEXT FILES USING FUNCTIONS:

Enter the following:

```
>>> def make_text_file():
        a=open("test.txt","w")
        a.write("This is how you create a new text file")
        a.close()
        print("Created test.txt")

>>> make_text_file()
Created test.txt
```

- You start off by creating a new function called make_text_file() .
- You then tell Python to open a file named test.txt .
- Because Python does not find this file, it creates it for you. (Note: if the file did exist, Python would have deleted it and created a new one. The " w " argument tells Python that you intend to write to the file; without it, Python would assume you intend to read from the file and would raise an exception when it found that the file didn't exist).
- Next, you add a line of text to the file, namely: " This is how you create a new text file " .

You will notice that a new file named test.txt has been created. If you double – click it, you will see the text you added. Now, create a program that first checks to see if the file name exists; if so, it will give you an error message; if not, it will create the file. Type in the following code:

```
>>>import os
>>> def make_another_file():
        if os.path.isfile("test.txt"):
                print("You are trying to create a file that already exists!")
        else:
                f=open("test.txt","w")
                f.write("This is how you create a new text file")
                print("Created test.txt because it doesn't exist")

>>> make_another_file()
You are trying to create a file that already exists!
>>>
```

APPENDING TEXT TO A FILE

Till now we have modified *the same file* and every time the file is **overwritten**. Let's now modify the same program and this time ***we will use append mode 'a'.***

```
>>>#file handling operations; writing the file hello.txt in append mode
>>>f = open('hello.txt', 'a', encoding = 'utf-8')
>>>f.write("Now I am appending this line!\n")
>>>f.write("Let's write another list\n")

>>>mylist =["Cabbage\n", "Potato\n", "Tomato\n"]
>>>#writelines() is used to write multiple lines in to the file
>>>f.writelines(mylist)

>>>f.flush()
>>>f.close()
```

*This time the file **hello.txt** is appended. The output is as follows:*

```
>>> f = open('hello.txt')
>>> data=f.read()
>>> data
Hello Python Developers!
Welcome to Python World
Apple
Orange
Banana
Now I am appending this line!
Let's write another list
Cabbage
Potato
Tomato

>>> def add_some_text():
        a=open("test.txt","a")
        a.write("Here is some additional text appended to the file test.txt!")
        print("Appended to test.txt ...")

>>> add_some_text()
Appended to test.txt ...
```

To see the results, go to the directory where Python is installed and open up the test.txt file. You should see the new text appended to the end of the file.

Note that ***write doesn't add line breaks automatically***; you must add one yourself with the escape sequence **\n** wherever you want a line break in the file. The same goes for spaces. If you do not add a space, tab, or line break, the next time you add some text to the file; it will be crammed up against the previous text.

If you use write again, the text is appended to what you wrote before. If the string you pass is more than one line long, more than one line is added to the file:

```
>>> def even_more_text():
        a=open("test.txt","a")
        a.write("""
        here is
        more
        text""")

        print("Appended more text to test.txt ...")

>>> even_more_text()
Appended more text to test.txt ...
>>>
```

You've used a multi - line, triple - quoted string here. In a multi - line string, Python adds line breaks between lines.

CREATING BINARY FILES

We need to write data as binary files such as image files or sound files. We can also write text data in binary, we must supply data in the form of objects that expose data as bytes (e.g., byte strings, bytearray objects, etc.).

EX1:

```
>>> # Write binary data to a file; writing the file hello.dat write binary mode
>>> import os
>>> os.chdir("D:\SOFTWARE TRAINING\TRAINING\COURSES\2-NON SAP\4A-PYTHON\WORKSPACE\6-FILES\EX")
>>>
>>> f = open('hello.dat', 'wb')            #writing as byte strings
>>> f.write(b"I am writing data in binary file!\n")
34
>>> f.write(b"Let's write another list\n")
25
>>> f.close()
>>>
>>> f = open('hello.dat', 'rb')
>>> data = f.read()
>>> data
b"I am writing data in binary file!\nLet's write another list\n"
```

If we need to read or write text from a binary-mode file, we would also use **decode** or **encode** function.

```
>>> # Write binary data to a file; writing the file new.dat write binary mode
>>> f = open('new.dat', 'wb')
>>> text = 'Hello World'
>>> f.write(text.encode('utf-8'))
11
>>> f.close()
>>>
>>> f = open('new.dat', 'rb')
>>> data = f.read()
>>> data
b'Hello World'
>>> text = data.decode('utf-8')
>>> text
'Hello World'
>>> print(text)
Hello World
```

CREATING BINARY FILE IN APPEND MODE

```
>>> # Write binary data to a file; writing the file new.dat write append binary mode
>>> f = open('new.dat', 'ab')
>>> line1 = 'Hello Python Developers!'
>>> line2 = 'Binary files are great'
>>> f.write(line1.encode('utf-8'))
24
>>> f.write(line2.encode('utf-8'))
22
>>> f.close()
>>>
>>> f = open('new.dat', 'rb')
>>> data = f.read()
>>>
>>> data
b'Hello WorldHello Python Developers!Binary files are great'
>>>
>>> text = data.decode('utf-8')
>>> text
'Hello WorldHello Python Developers!Binary files are great'
>>> print(text)
Hello WorldHello Python Developers!Binary files are great
>>>
```

EX2:
CREATING FILES

```
>>> # create a file. If file is existing, it erases and creates a new one
>>> f1 = open('mydoc1', 'w')
>>> # create a file. If file is existing, it appends. Otherwise, it creates
>>> f2 = open('mydoc2', 'a')
>>>
>>>
>>> # binary files
>>> bf1 = open('mydoc3', 'wb')
>>> bf2 = open('mydoc4', 'ab')
```

WRITING DATA

```
>>> #Write data into a file, we can use write() method.
>>> data = "" ;index=1
>>> name = 'user ' + str(index-1)
>>> email = 'user' + str(index-1) + '@email.com'
>>> if index == 1:
                data = '{0:3s} {1:10s} {2:15s}\n'.format(1, 'Name', 'Email')
        else:
                data = '{0:3s} {1:10s} {2:15s}\n'.format(str(index-1), name, email)
>>> f1.write(data)
>>> f2.write(data)

>>>
>>>
>>> # to write data to a binary file; use the encode method of the string object to get a byte
sequence
>>> bf1.write(data.encode('utf-8'))
>>> bf2.write(data.encode('utf-8'))
```

CLOSING A FILE

```
>>> #If file operations done, you should call close() method to close file.
>>> f1.close()
>>> f2.close()
>>> bf1.close()
>>> bf2.close()
```

To read data per line from a file, we use readline() method.

```
import sys
#######################################
print('opening files…')
f1 = open('mydoc1', 'r')
f2 = open('mydoc2', 'r')
bf1 = open('mydoc3', 'rb')
bf2 = open('mydoc4', 'rb')
#######################################
# reading data
def reading_data(f,ty):
        while True:
                data = f.readline()
                if (data == "") or (data == None) or (data == b''):
                        break
                if ( ty == "t"):
                        sys.stdout.write(data)
                else:
                        # use decode method to get text data from a byte sequence
                        sys.stdout.write(data.decode('utf-8'))

print('for mydoc1>>>>>')
reading_data(f1,'t')
print('>>>>>>>>>>>>>>>>')
print('for mydoc2>>>>>')
reading_data(f2,'t')
print('>>>>>>>>>>>>>>>>')
print('for mydoc3>>>>>')
reading_data(bf1,'b')
print('>>>>>>>>>>>>>>>>')
print('for mydoc4>>>>>')
reading_data(bf2,'b')
print('>>>>>>>>>>>>>>>>')
#######################################
# close all
print('closing files…')
f1.close()
f2.close()
bf1.close()
bf2.close()
```

WITH CONSTRUCT:

Most of the times many developers forget to close the file. Python supports **with** construct. For example: **with** open("test.txt",'w') **as** f:
After this colon block of python, file handling code follows. In this case no need to explicitly close the file. The Python compiler will take care of it. So it is a better option to use this clause. Let's see an example below using the **with** construct:

```
>>> # Write binary data to a file; writing the file somefile.dat in write binary mode
>>> import os
>>> os.chdir("D:\SOFTWARE TRAINING\TRAINING\COURSES\2-NON SAP\4A-PYTHON\WORKSPACE\6-FILES\EX")
>>> with open("somefile.dat","wb") as f:
        f.write(b"Hello world\n")
        f.write(b"This is a demo using with \n\n")
        f.write(b"This file contains three lines\n")
        #no need to explicitly use f.close

12
28
31
>>>
>>> # Write binary data to a file; writing the file somefile.dat in append binary mode
>>> with open("somefile.dat",'ab') as f:
        f.write(b"Hello world again\n")
        f.write(b"This is a demo using with \n\n")
        f.write(b"This file contains three more lines after appending data\n")
        #no need to explicitly use f.close

18
28
57
>>>
>>> # Read binary data from a file; writing the data to the screen
>>> with open("somefile.dat",'rb') as f:
        data = f.read()
        text = data.decode('utf-8')
        print(text)

Hello world
This is a demo using with
This file contains three lines
Hello world again
This is a demo using with
This file contains three more lines after appending data
```

MOVING TO A SPECIFIC LOCATION IN A FILE (SEEKING)

Change the file position to offsetbytes, in reference to from (start, current, end).

```
>>> # Read binary data to a file;
>>> with open("somefile.dat",'rb') as f:
        #reading first 10 characters
        data = f.read(10)
        text = data.decode('utf-8')
        print(text)
        #after reading 10 characters, move the cursor to initial position
        f.seek(0)
        data = f.read()
        text1 = data.decode('utf-8')
        print("\n")
        print(text1)

Hello worl
0
Hello world
This is a demo using with
This file contains three lines
Hello world again
This is a demo using with
This file contains three more lines after appending data
>>>
```

In the first line of the output, first 10 characters were written i.e., “Hello worl”.
After we seek the file pointer to the initial position, we use a read() with empty parameters to read everything.

The **with** construct automatically closes the file, so if we try to read, there will be an error.

```
>>> data = f.read()
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
data = f.read()
ValueError: read of closed file
>>>
```