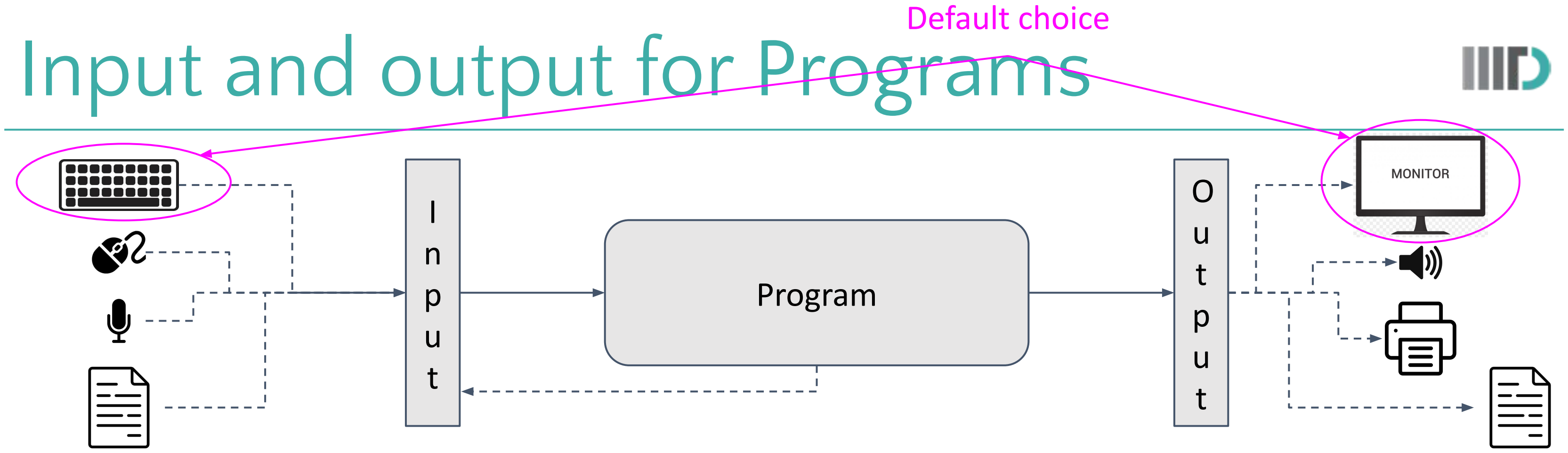
File Handling – Input / Output

Md Shad Akhtar

Assistant Professor

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Input and output for Programs



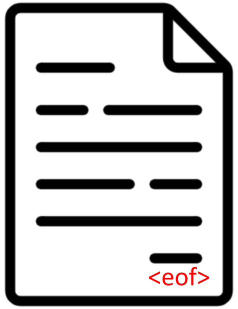
- In general, programs need input to work on.
  - Inputting through keyboard using input() statement/ function
    - Very limiting - user has to type input data every time the program is to be used; data you can give is limited - one line
  - Among others, one way to give input is to have the inputs in a file, and then the program can read these inputs from it

- Each program must have at least one output
  - Displaying output on monitor / console using print()
  - Among others, we can write the output to a file

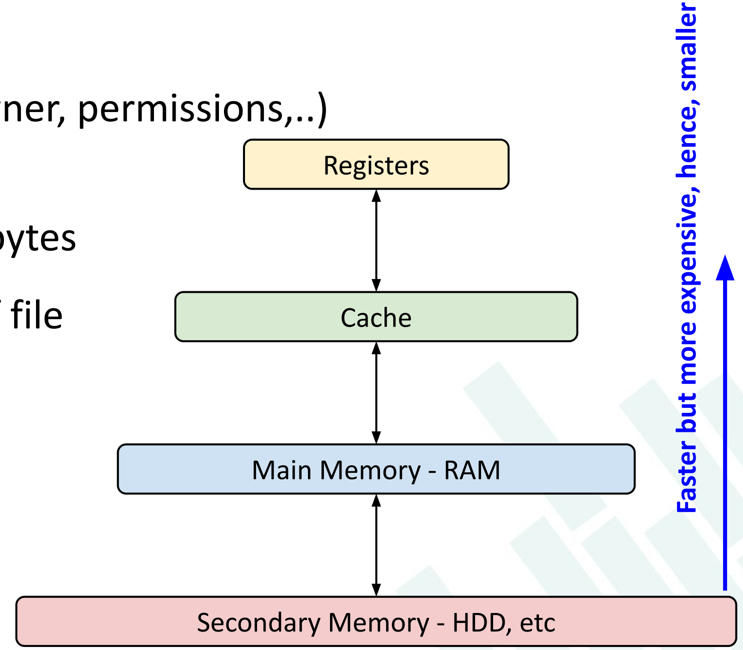
What is a File?

- A file is an object an operating system allows users to create on secondary memory (i.e., HardDisk drive) which can hold persistent data, i.e. data remains after power off also
  - File is very different from main memory (i.e., RAM) - which is volatile

- A file in any operating system generally consists of:
  - Header: Gives info about the file (size, type, owner, permissions,..)
  - Data: This is the content of the file
    - conceptually contiguous - a sequence of bytes
  - EOF: Special character that indicates the end of file

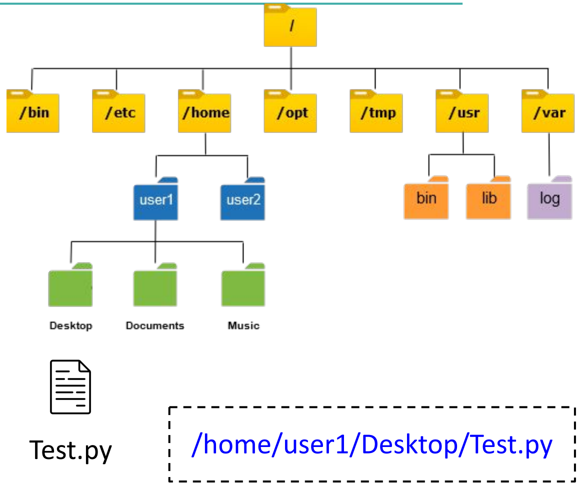


Test.py - 10kb


Registers
Cache
Main Memory - RAM
Secondary Memory - HDD, etc
Faster but more expensive, hence, smaller

Files...

- Files are organized in a directory (folder) structure in the OS
  - Directories are nested
    - the structure starts from root – / signify level 0
    - Following slashed (/) represent nested levels



- Locate a file: Provide full path to the file
  - Directory path
    - Full path: The folder location in the file system usually starting from the root
    - Relative path: Path from the present working directory (pwd) – usually where the program file resides
  - File name: User given file name
  - Extension: Often used to indicate the type of data

File Modality and Encodings

- Files can store anything – text, audio, image, video.
    - Internally, everything is stream of bytes.
- For simplicity, we will focus on text files only, i.e., files whose data is text
- Encoding scheme decides how bytes are mapped to human readable characters
    - ASCII (older - 128 chars) only roman characters
    - Unicode (utf-8) (new - 100K values) – a range of characters

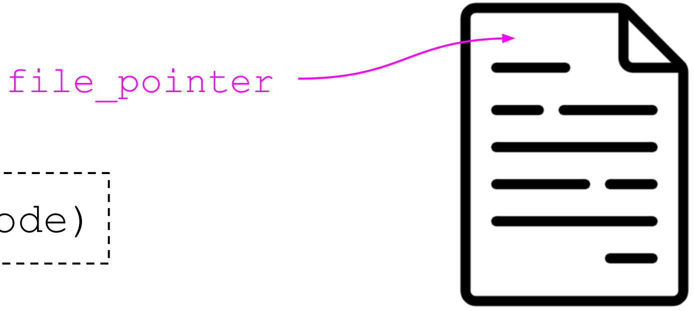
Accessing a file

- **Locate the file**
  - To access a file from a program, have to specify the location of the file to python program, so it can request the OS for the file
- **Open the file**
  - Request access from the OS (taking temporary control of the file)
- **Operate on file**
  - Read, write, or append
  - Depends on the granted access by the OS while opening
    - You can not write or append in a file, if the file was opened only with read permission
- **Close the file**
  - Returning the control of the file back to OS
  - Free up resources

Opening a File

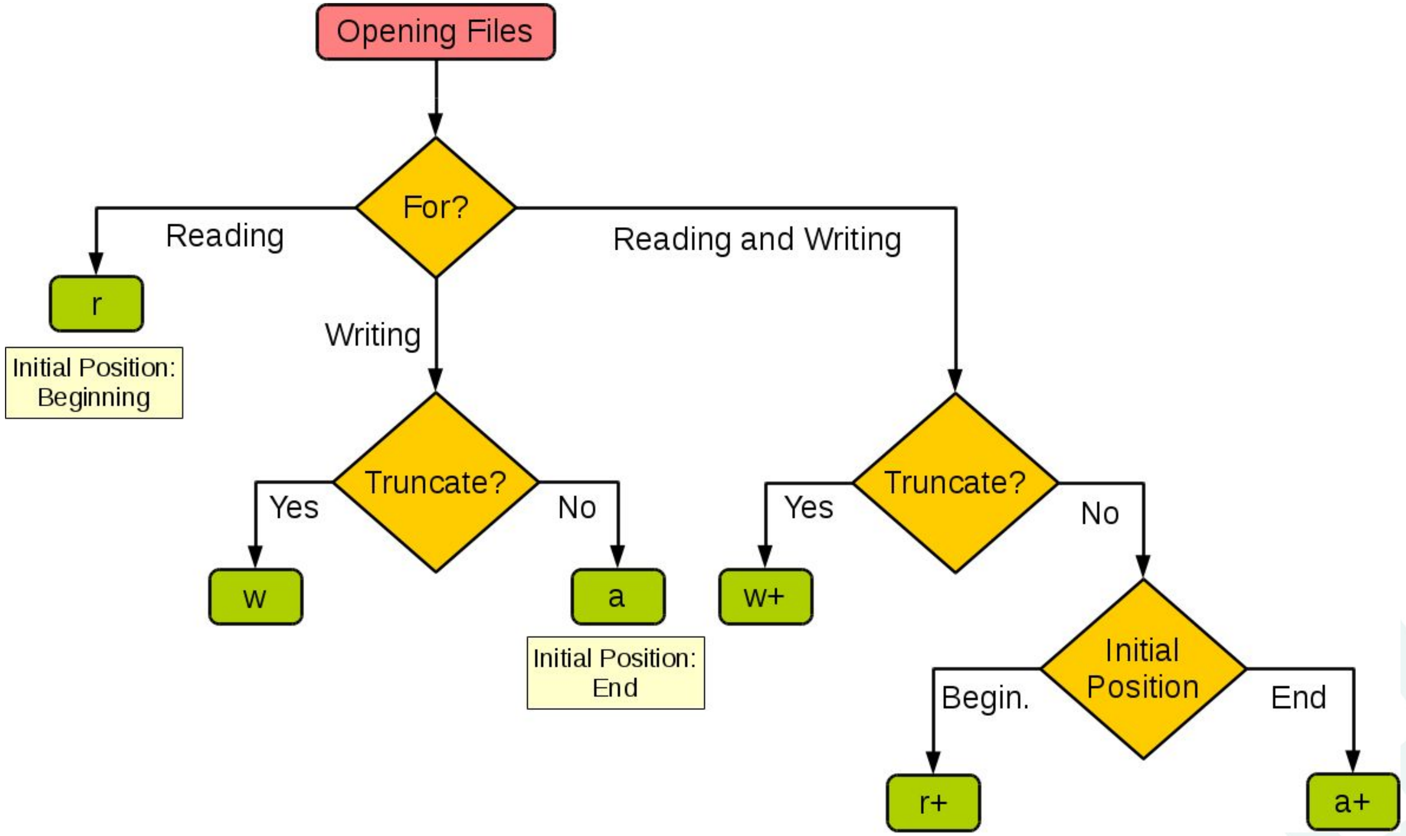
- `open()`
  - Open a file (fname) in requested mode

```
file_pointer = open(fname, mode)
```



- `fname`: full path of the file
- `mode`
  - **Read ('r'):** Can only be used for reading. Writing not permitted. Default value and can be omitted
    - If file is not existing, it will give FileNotFoundError
  - **Write ('w'):** Open a file in write mode, if existing. It will overwrite the content
    - If file is not existing, it will create a new file
  - **Append ('a'):** Open a file in write mode, if existing. It will not overwrite, instead writes at the end
    - If file is not existing, it will create a new file

Other modes: r+, w+, a+, rb, wb, ab



https://stackoverflow.com/questions/1466000/difference-between-modes-a-a-w-w-and-r-in-built-in-open-function

Closing a file

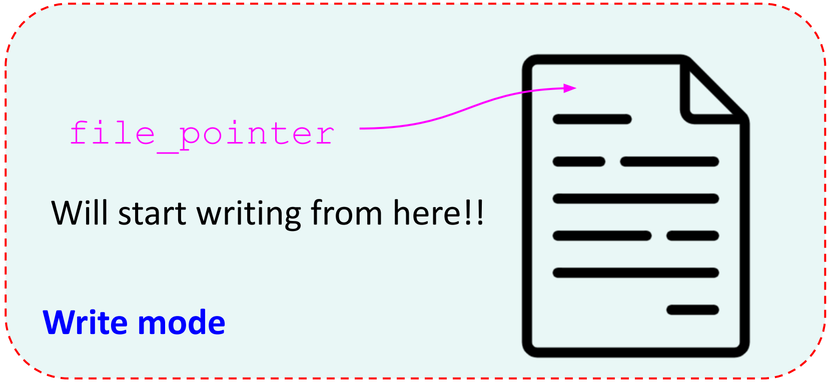
- Once we are done with the reading and writing operations, we need to properly close the file.
- Free up the resources associated with the file.
- Content may be actually written on the disk at the closing time.
    - If not done, the OS will not know when to close it - it will do it sometime in future - strange behaviour possible

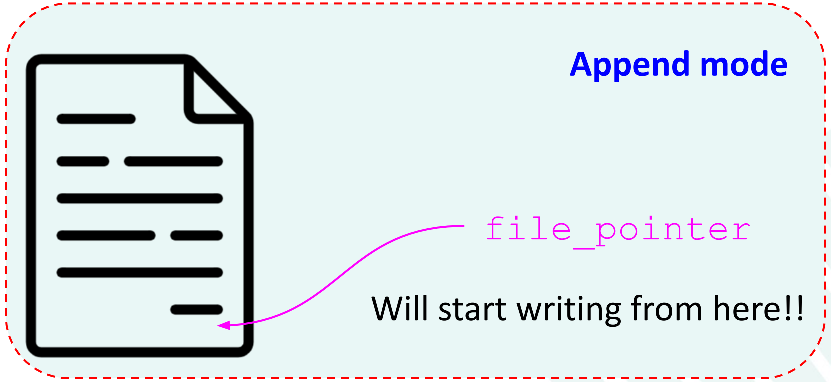
```
f.close()
```

Writing to a file

- File must be opened in write (‘w’) or append (‘a’) mode
  - File is created if does not exist; cleared if it exists
- write()
  - Write on an open file
  - Only strings can be written - other data types must be converted
  - Whitespace characters (e.g., newlines ‘\n’) must be explicitly mentioned.

```
f.write(string)
```





Writing to a file: Examples

```
count = 0
f = open('test.txt', 'w')
count = f.write('Hi, how are you?\nThanks, I\'m fine.')
print(count)
f.close()
```

test.txt

```
Hi, how are you?
Thanks, I'm fine.
```

```
f = open('test.txt', 'w')
f.write('Hi, how are you?\nThanks, I\'m fine.')
f.close()

f = open('test.txt', 'w')
f.write('What about you?')
f.close()
```

test.txt

```
Hi, how are you?
Thanks, I'm fine.
```

content will be deleted ❌

test.txt

```
What about you?
```

```
f = open('test.txt', 'w')
f.write('Hi, how are you?\nThanks, I\'m fine.')
f.close()

f = open('test.txt', 'a')
f.write('What about you?')
f.close()
```

test.txt

```
Hi, how are you?
Thanks, I'm fine.What about you?
```

Reading a file

- If file opened successfully, it will return a valid file_pointer
    - We can use to read (or write) the file
- Python provides a few different functions to read the file

```
f.read()  # reads the entire file, returns a string
f.read(size) # reads size bytes
f.readline() # reads one line, returns a string
f.readlines() # reads a list of strings, each item being a line
```

- To loop over lines in a file, it provides an efficient way

```
for line in f:
    #do something with line
```

Reading a file: Examples

test.txt
```
Hi, how are you?
Thanks, I'm fine.
```

```
f = open('test.txt', 'r')
print(f.read())
f.close()
```

```
Hi, how are you?
Thanks, I'm fine.
```

```
f = open('test.txt', 'r')
print(f.read(10))
f.close()
```

```
Hi, how ar
```

```
f = open('test.txt', 'r')
print(f.readline())
f.close()
```

```
Hi, how are you?
```

```
f = open('test.txt', 'r')
print(f.readlines())
f.close()
```

```
['Hi, how are you?\n', "Thanks, I'm fine."]
```

```
f = open('test.txt', 'r')
for line in f:
    print(line)
f.close()
```

```
Hi, how are you?

Thanks, I'm fine.
```

```
f = open('test.txt', 'r')
print(f.read())
print(f.read(10))
f.close()
```

```
Hi, how are you?
Thanks, I'm fine.

```

Observe a blank (new) line here

Writing and Reading in binary mode

```
binary_data = b"\x48\x65\x6C\x6C\x6F"
f = open('file.bin', 'wb')
f.write(binary_data)
f.close()
```

```
f = open('file.bin', 'wb')
binary_data = bytearray([0xFF, 0x00, 0x7F, 0x80])
f.write(binary_data)
```

```
f = open('test.txt', 'rb')
print(f.readline())   # b'Hi, how are you?\n'
f.close()
```

Using `with` to work with files

test.txt

```
Hi, how are you?
Thanks, I'm fine.
```

- Often programmers forget to close a file - then the file remains open and file resources remain occupied.
    - The output written to the file might stay in buffer until file is closed and modifications might not be visible on disk.
- A better way to handle the file is using '`with`'

```python
with open('test.txt', 'r') as f:
     lst = f.readlines()

for line in lst:
     print(line)
```

```
Hi, how are you?

Thanks, I'm fine.
```

- Code-block is where you use the opened file and save data in data structures; the file will be closed after the code-block.

Handline whitespace characters

- Reading
  - Often when we read data from files, the lines have trailing whitespaces or newline characters.
  - You should explicitly handle them
    - Use `strip()`, `lstrip()`, or `rstrip()` functions on individual line to get rid of them
- Writing
  - Whitespace characters (e.g., newlines ‘\n’) must be explicitly mentioned.

Other important functions

- `file_pointer.tell()` # returns the current position of the file_pointer within the file
- `file_pointer.seek(offset, from)` # moves the file_pointer to a new position (`from+offset`).
    - offset = number of bytes to move
    - from defines the starting position for offset:
        - `os.SEEK_SET` or `0` **# beginning of the file. By default**
        - `os.SEEK_CUR` or `1` **# current position in the file**
        - `os.SEEK_END` or `2` **# end of the file**
    - With python 3+, from can only be `os.SEEK_SET` and offset must only be positive [in text mode].
        - `file_pointer.seek(0, 1)` # Allowed but will not have an effect on the file_pointer
        - `file_pointer.seek(0, 2)` # Allowed but will return blank

Negative offset and other positions are allowed in binary mode!!

Other important functions

```
f = open('test.txt', 'r')
print(f.readline())

position = f.tell()
print ("Current file position : ", position)

# Reposition pointer
f.seek(10, 0)
print ("Again read String is : ", f.read(10))
f.close()
```

```
Hi, how are you?

Current file position : 17
Again read String is : e you?
I'm
```

Opened in read-binary mode

```
f = open('test.txt', 'rb')
print(f.readline())

position = f.tell()
print ("Current file position : ", position)

# Reposition pointer
f.seek(-10, 1)
print ("Again read String is : ", f.read(10))
f.close()
```

```
b'Hi, how are you?\n'
Current file position : 17
Again read String is : b' are you?\n'
```

Other important functions

- Some other helper functions with the os module [`import os`]
    - `os.rename(current_file_name, new_file_name)` # rename the file.
    - `os.remove(file_name)` # remove the file. Use with caution.
    - `os.mkdir("newdir")` # make new directory
    - `os.chdir("newdir")` # change the current working directory to a different directory
    - `os.getcwd()` # get the current working directory
    - `os.rmdir('dirname')` # remove the directory. Use with caution.

In-class exercise