

File/Stream Handling

Ends at Page 70

Recompiled -
Naiswita Parmar
Assistant Professor,
CSE, SET,
Navrachana University

File Stream Handling

About Files

- File is a **permanent storage** area containing any kind of data stored on secondary storage devices (HDD, SDD, CD, DVD, Card memories etc.)
- Files are **managed by Operating System through File System Applications** like File Allocation Table (**FAT32**), New Technology File System (**NTFS**), **ExtX** in Linux, etc.
- Big data files are managed by Distributed Files Systems such as Hadoop File System (**HDFS**), Google Distributed File System (**GDFS**), Network File System (**NFS**).
- File data is transferred to RAM when in need in form of Stream.

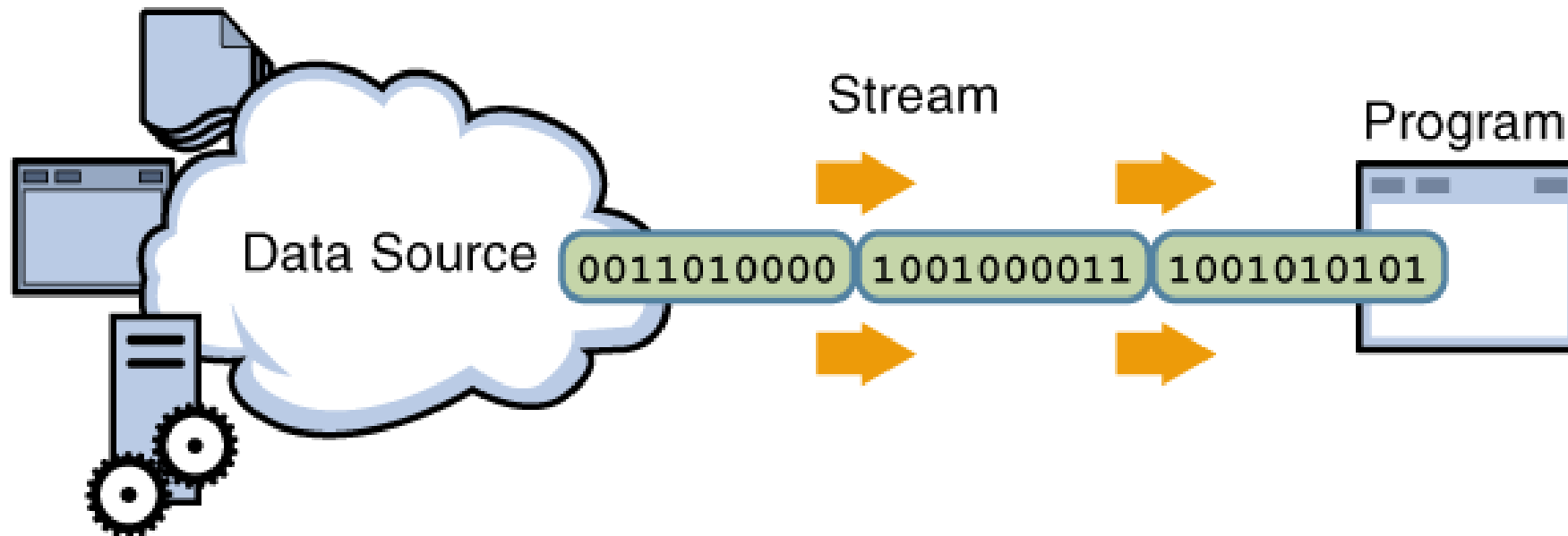
File Stream Handling

About Streams

- Stream is a memory area in RAM with **contiguous array of bytes**.
- Streams are unidirectional**.
- A Stream **has two ends**. One end is attached to Python program and other to **any Input or Output device**.
- Streams based on direction of flow of data are categorized into two
- 1. **Input Stream** : One end of stream is attached to Python program and other end to Input device (keyboard, File storage, Network device). Data flows from device to Python program.
- 2. **Output Stream** : One end of stream is attached to Python program and other end to Output device (VDU, printer, File storage, Network device). Data flows from Python program to device attached.

File Stream Handling

- Python performs I/O through streams.
- Stream is a **sequence of data** (bytes/chars) that either **produces OR consumes information**.
- Streams can be connected to **different devices** – **hard-disk file, console, or Network**.



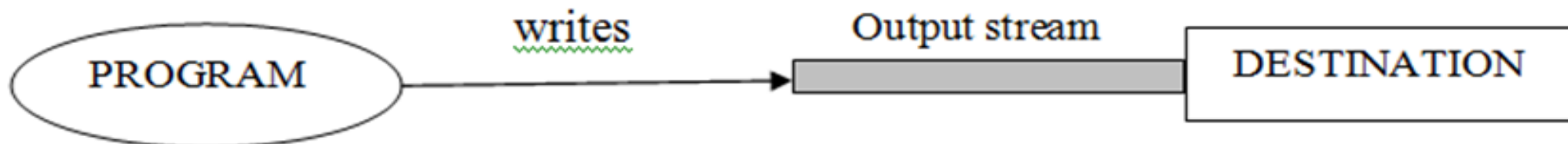
File Stream Handling

- 2 types of streams – **Input and Output**
- **Input stream** – used to read data from file/device
- **Output stream** – used to write data to file/device
- **Note:** Streams are **Uni-directional** (they can transfer data in only one direction)

Reading data into Python Program using Input



WRITING DATA to a DESTINATION using OUTPUT STREAM



Opening a file in Python

Opening a file in Python

- There are two types of files that can be handled in Python:
 - normal text files
 - binary files (written in binary language, 0s, and 1s).
- Opening a file refers to getting the file ready either for reading or for writing.
- This can be done using the [open\(\) function](#).
- This function returns a file object and takes two arguments, one that accepts the **file name** and another that accepts the **mode(Access Mode)**.

Opening a file in Python

- **Note:** The file should exist in the same directory as the Python script, otherwise, the full address of the file should be written.
- **Syntax:** `File_object = open("File_Name", "Access_Mode")`
- **Parameters:**
- **File_Name:** It is the name of the file that needs to be opened.
- **Access_Mode:** Access modes govern the type of operations possible in the opened file. The below table gives the list of all access mode available in python

Python File Functions

- `Open()` – It opens file in different modes
- Syntax
- `Open(file_name, file_opening_mode)`
- `file_opening_mode` :
- “r”- file is opened in read mode, file must exist otherwise it raises `FileNotFoundError`, if file is found the read pointer is placed at first character or byte within a file.
- “w” – file is opened in write mode, if file does not exist it is created and write pointer is placed at beginning of file. If file exist then overwrites the older content to a newer one.
- “a” - file is opened in append mode, if file does not exist it is created and write pointer is placed at beginning of file. If file exist then write pointer is placed at `End_of_File` hence preserving the older content. The data is added to end of file.
- “t” – Text mode, It is accompanied with r,w and a . It is the default mode
- “b” – Binary mode, It is accompanied with r,w and a.

- Read and Write : r+ : Open the file for reading and writing.
- Write and Read : w+ : Open the file for reading and writing. Unlike “r+” is doesn't raise an I/O error if file doesn't exist.
- Append and Read : a+ : Open the file for reading and writing and creates new file if it doesn't exist. All additions are made at the end of the file and no existing data can be modified.

Difference between r+, w+, & a+

- these three modes all allow reading and writing but:
 - for r+ an error occurs if file does not exist.
 - for r+ the existing data is not discarded, but can be overwritten with random access.
 - for w+ the existing data (if any) is always discarded.
 - for a+ the existing data (if any) is not discarded and can not be overwritten, even with random access - data is always written at end of file.

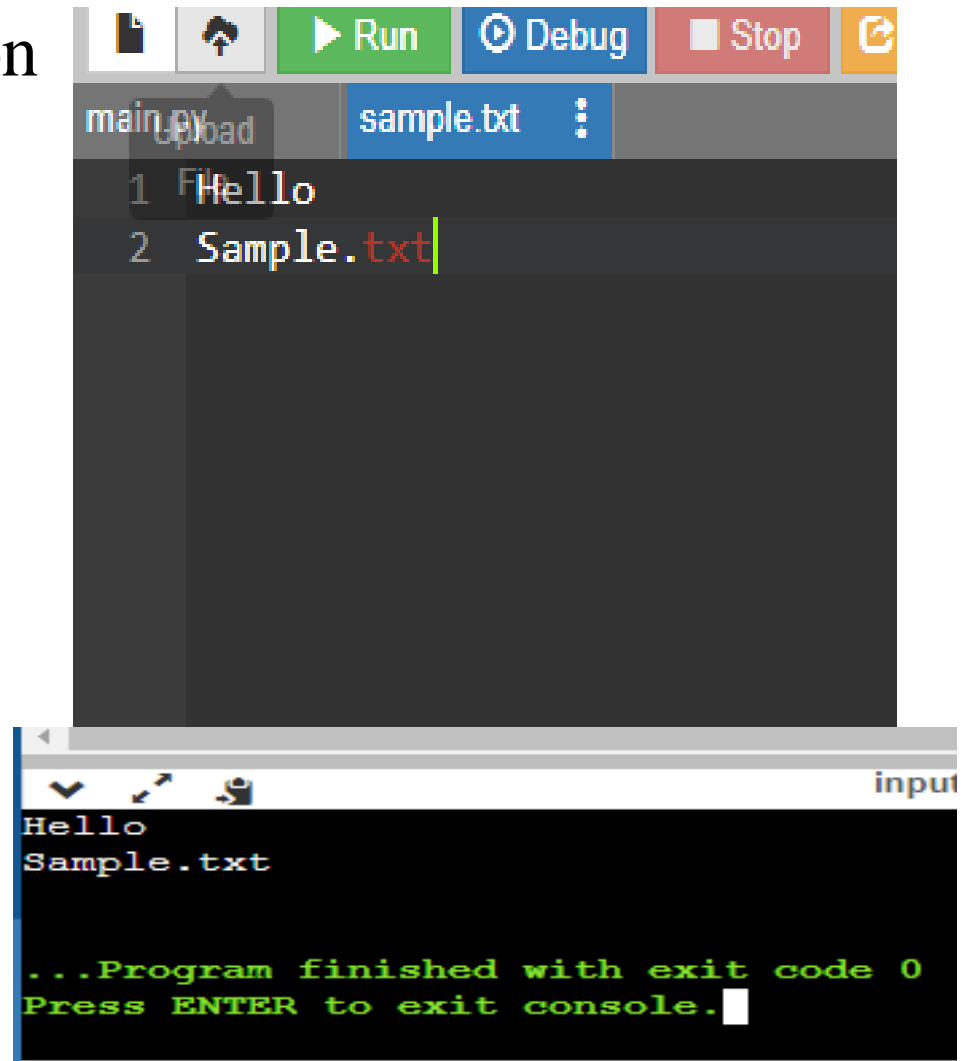
- **Example 1:** Open and read a file using Python
- In this example, we will be opening a file to read-only.

open the file using open() function

```
file = open("sample.txt")
```

Reading from file

```
print(file.read())
```



The screenshot shows a code editor window with a dark background. At the top, there are buttons for 'Run' (green), 'Debug' (blue), and 'Stop' (red). Below these, the file 'sample.txt' is open, showing two lines of text: '1 Hello' and '2 Sample.txt'. The cursor is at the end of the second line. Below the code editor, there is a terminal window with a black background. It shows the output of the program: 'Hello' and 'Sample.txt'. At the bottom of the terminal, it says '...Program finished with exit code 0' and 'Press ENTER to exit console.' with a white cursor.

```
main.py 1 Hello
2 Sample.txt
Hello
Sample.txt
...Program finished with exit code 0
Press ENTER to exit console.
```

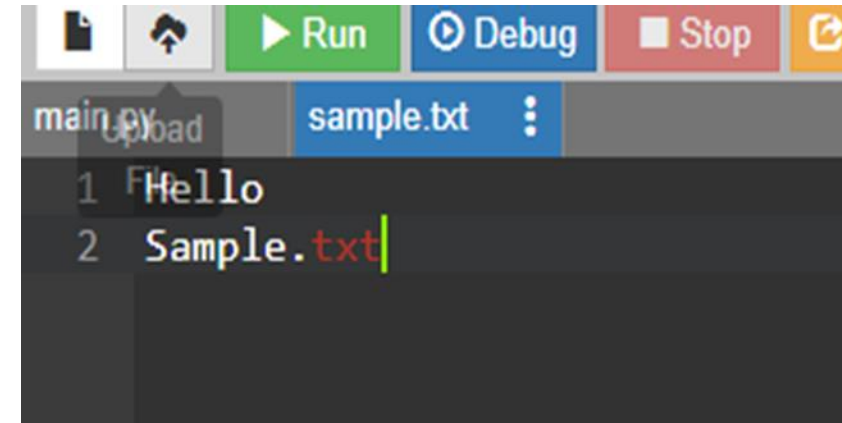
- **Example 2:** Open and write in a file using Python
- In this example, we will be appending new content to the existing file. So the initial file looks like the below:

open the file using open() function

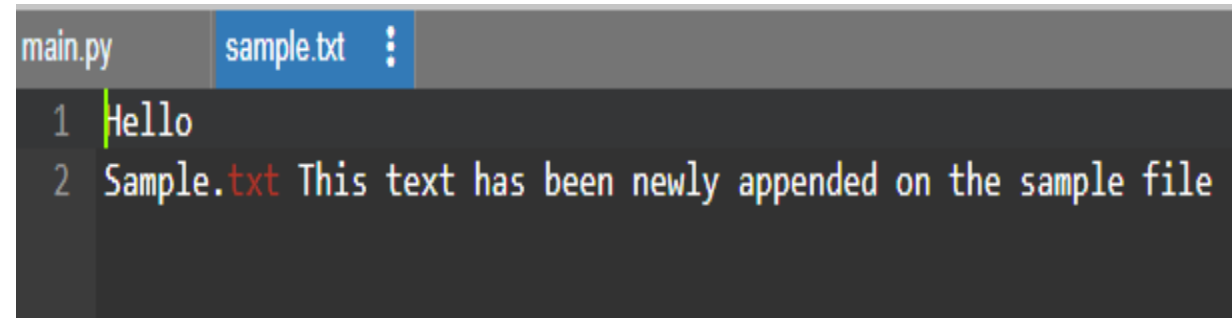
```
file = open("sample.txt", 'a')
```

Add content in the file

```
file.write(" This text has been newly  
appended on the sample file")
```

A screenshot of a code editor interface. At the top, there is a toolbar with icons for file operations and buttons for 'Run' (green), 'Debug' (blue), 'Stop' (red), and a share icon (orange). Below the toolbar, the editor shows two tabs: 'main.py' and 'sample.txt'. The 'sample.txt' tab is active, displaying two lines of text: '1 Hello' and '2 Sample.txt'. The cursor is positioned at the end of the second line.

```
main.py sample.txt  
1 Hello  
2 Sample.txt
```

A screenshot of the same code editor interface. The 'sample.txt' tab is still active. The first line remains '1 Hello'. The second line now reads '2 Sample.txt This text has been newly appended on the sample file'. The cursor is at the end of the second line.

```
main.py sample.txt  
1 Hello  
2 Sample.txt This text has been newly appended on the sample file
```

- **Example 3:** Open and overwrite a file using Python

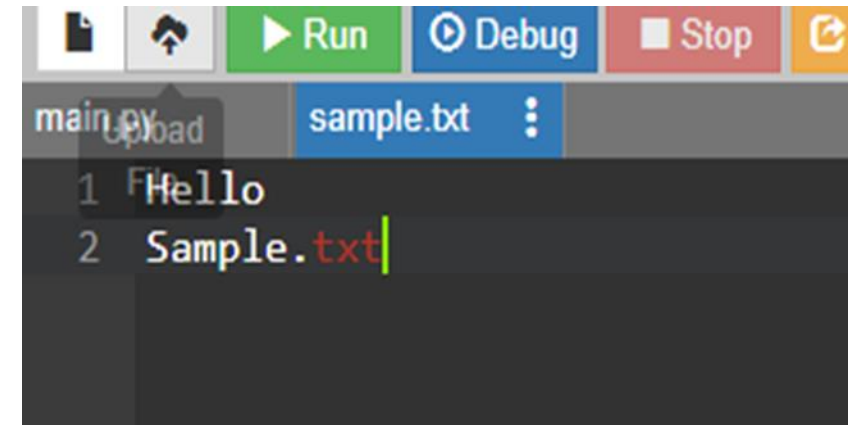
open the file using open() function

```
file = open("sample.txt", 'w')
```

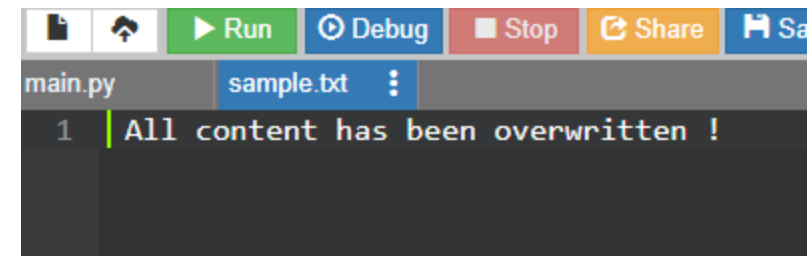
Overwrite the file

```
file.write(" All content has been overwritten !")
```

The above code leads to the following result,

A screenshot of a code editor interface. At the top, there is a toolbar with icons for file operations and buttons for 'Run' (green), 'Debug' (blue), 'Stop' (red), and 'Share' (orange). Below the toolbar, a tab labeled 'sample.txt' is active. The editor shows two lines of text: '1 Hello' and '2 Sample.txt' with a cursor at the end of the second line.

```
1 Hello
2 Sample.txt
```

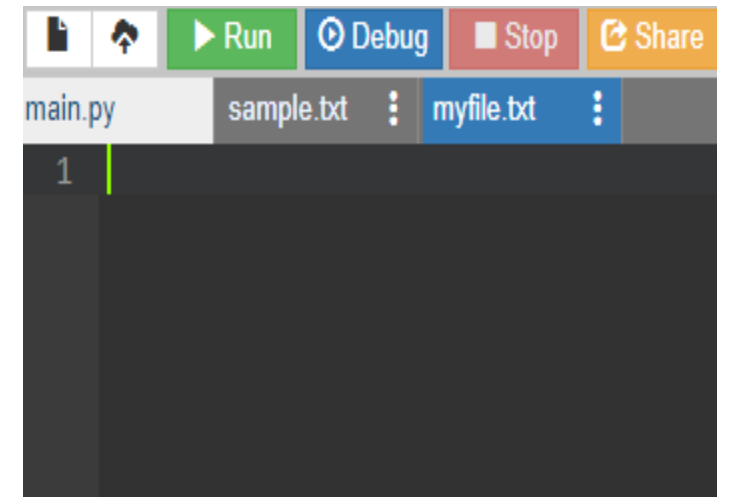
A screenshot of the same code editor interface. The 'sample.txt' tab is still active. The editor now shows a single line of text: '1 All content has been overwritten !' with a cursor at the end of the line.

```
1 All content has been overwritten !
```

- **Example 4:** Create a file if not exists in Python
- The `path.touch()` method of the [pathlib](#) module creates the file at the path specified in the path of the [path.touch\(\)](#).

from pathlib import Path

```
my_file = Path('test1/myfile.txt')  
my_file.touch(exist_ok=True)  
f = open(my__file)
```



Closing a file in Python

Closing a file in Python

- Python automatically closes a file if the reference object of the file is allocated to another file, it is a standard practice to close an opened file as a closed file reduces the risk of being unwarrantedly modified or read.
- Python has a **close()** method to close a file.
- The close() method can be called more than once and if any operation is performed on a closed file it raises a **ValueError**.
- The below code shows a simple use of close() method to close an opened file.

- **Example:** Read and close the file using Python

open the file using open() function

```
file = open("sample.txt")
```

Reading from file

```
print(file.read())
```

closing the file

```
file.close()
```

Now if we try to perform any operation on a closed file like shown below it raises a `ValueError`

```
# open the file using open() function
```

```
file = open("sample.txt")
```

```
# Reading from file
```

```
print(file.read())
```

```
# closing the file
```

```
file.close()
```

```
# Attempt to write in the file
```

```
file.write(" Attempt to write on a closed file !")
```

Output:

```
ValueError: I/O operation on closed file.
```

File Handling Options

Working of open() function

- `f = open(filename, mode)`
- Where the following mode is supported:
- **r**: open an existing file for a read operation.
- **w**: open an existing file for a write operation. If the file already contains some data then it will be overridden.
- **a**: open an existing file for append operation. It won't override existing data.
- **r+**: To read and write data into the file. The previous data in the file will be overridden.
- **w+**: To write and read data. It will override existing data.
- **a+**: To append and read data from the file. It won't override existing data.

a file named “sample”, will be opened with the reading mode.

```
file = open('sample.txt', 'r')
```

This will print every line one by one in the file

for each **in** file:

```
    print (each)
```

Working of read() mode

Python code to illustrate read() mode

```
file = open("file.txt",  
"r")  
print (file.read())
```

Python code to illustrate read() mode character wise

```
file = open("file.txt", "r")  
print (file.read(5))
```

Creating a file using write() mode

Python code to create a file

```
file = open('sample.txt','w')
```

```
file.write("This is the write command")
```

```
file.write("It allows us to write in a particular file")
```

```
file.close()
```

- The close() command terminates all the resources in use and frees the system of this particular program

Working of append() mode

```
file = open('geek.txt','a')  
file.write("This will add this line")  
file.close()
```

With()

Python code to illustrate with()

with open("file.txt") as file:

data = file.read()

do something with data

- **Using write along with the with() function**

with open("file.txt", "w") as f:

f.write("Hello World!!!")

Important

split() using file handling

- We can also split lines using file handling in Python.
- This splits the variable when space is encountered.
- You can also split using any characters as we wish.

with open("file.text", "r") as file:

 data = file.readlines()

for line **in** data:

 word = line.split()

 print (word)

Python seek() function

- seek() function is used to **change the position of the File Handle** to a given specific position.
- File handle is like a cursor, which defines from where the data has to be read or written in the file.
- ***Syntax:** `f.seek(offset, from_what)`, where *f* is file pointer*

Parameters:

Offset: Number of positions to move forward

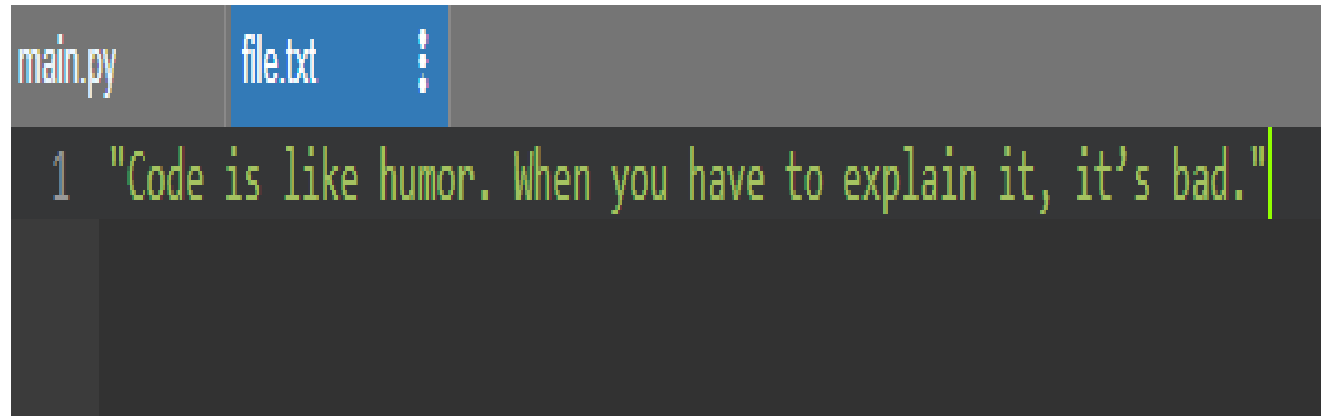
from_what: It defines point of reference.

Returns: Return the new absolute position.

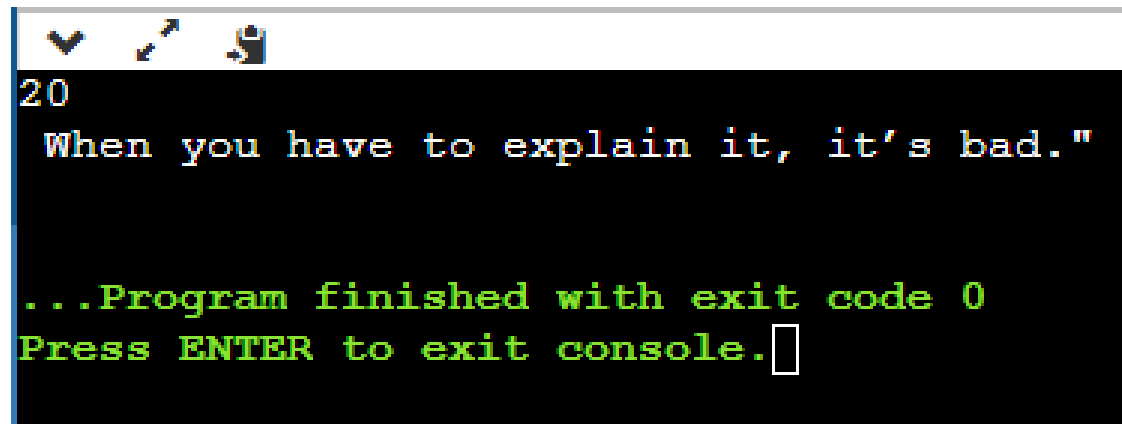
- The reference point is selected by the **from_what** argument. It accepts three values:
 - **0**: sets the reference point at the beginning of the file
 - **1**: sets the reference point at the current file position
 - **2**: sets the reference point at the end of the file
 - By default from_what argument 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 1

```
f = open("file.txt", "r")
f.seek(20)
# prints current position
print(f.tell())
print(f.readline())
f.close()
```



The screenshot shows a code editor with two tabs: 'main.py' and 'file.txt'. The 'file.txt' tab is active and shows a single line of text: `1 "Code is like humor. When you have to explain it, it's bad."`. The text is highlighted in green.



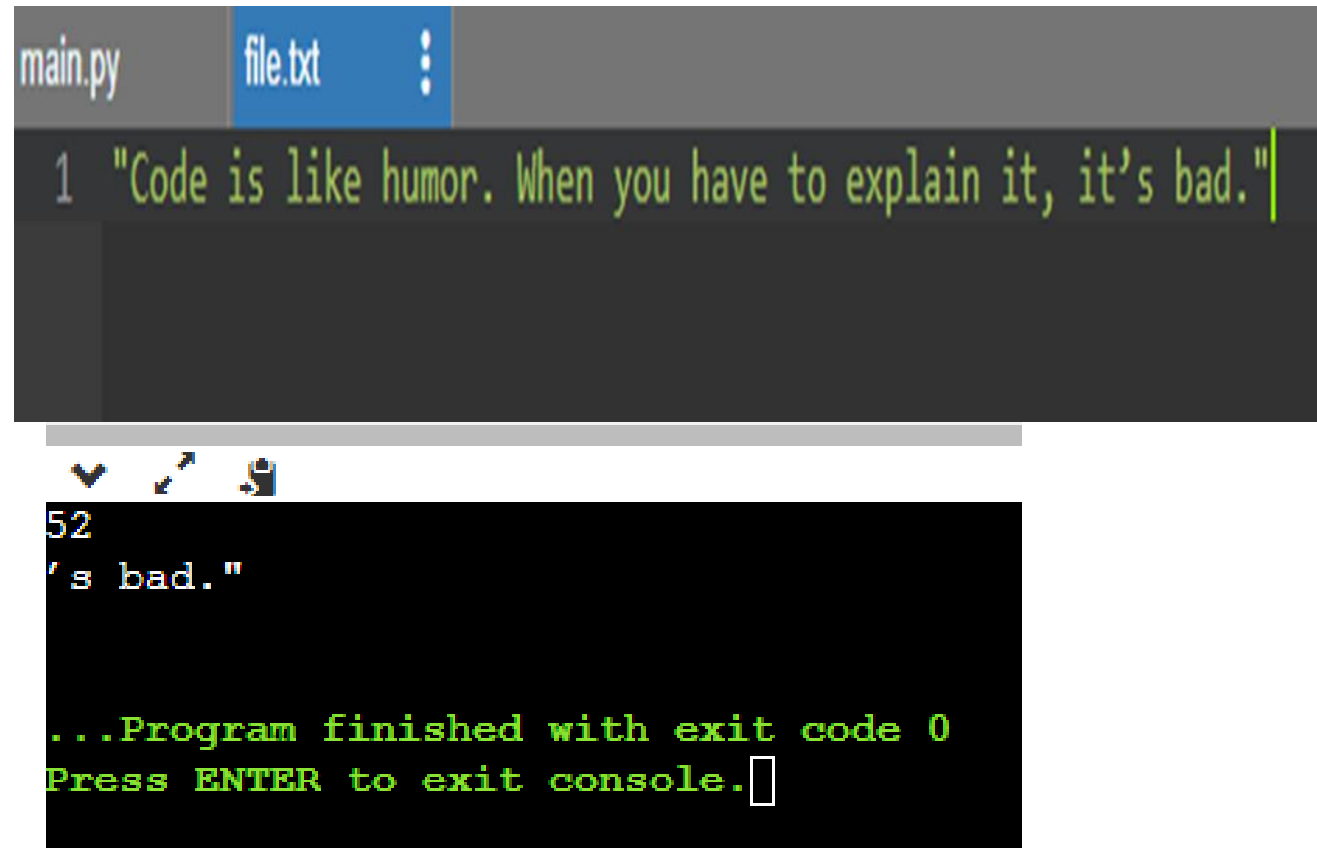
The screenshot shows a terminal window with a black background and green text. The output of the program is as follows:

```
20
  When you have to explain it, it's bad."

...Program finished with exit code 0
Press ENTER to exit console. □
```

- **Example 2:** Seek() function with negative offset only works when file is opened in binary mode. Let's suppose the binary file contains the following text.

```
f = open("file.txt", "rb")
# sets Reference point to tenth
# position to the left from end
f.seek(-10, 2)
# prints current position
print(f.tell())
# Converting binary to string and
print(f.readline().decode('utf-8'))
f.close()
```



The screenshot shows a code editor with two tabs: 'main.py' and 'file.txt'. The 'file.txt' tab is active and contains a single line of text: `1 "Code is like humor. When you have to explain it, it's bad."`. Below the editor is a terminal window. The terminal output shows the number `52` on the first line, followed by `'s bad.'` on the second line. The third line of the terminal output is `...Program finished with exit code 0`, and the fourth line is `Press ENTER to exit console.` with a cursor at the end.

Advantage of Stream based I/O

- ❑ Provides abstraction/simplicity to the programmer –without worrying about the hardware device protocols/details.
- ❑ Device independence - Streams can be connected to any device – memory, file, console, network – with same behavior/usage/methods
- ❑ Less lines of code (LOC) to be written by programmer
- ❑ Provides buffering functionality for faster/efficient I/O operations

Python sys Module

- The **sys module** in Python provides various functions and variables that are used to manipulate different parts of the Python runtime environment.
- It allows operating on the interpreter as it provides access to the variables and functions that interact strongly with the interpreter.
- **sys.version** is used which returns a string containing the version of Python Interpreter with some additional information. This shows how the sys module interacts with the interpreter.

Output:

```
import sys  
print(sys.version)
```

```
3.6.9 (default, Oct 8 2020, 12:12:24)  
[GCC 8.4.0]
```

Input and Output using sys

- The sys modules provide variables for better control over input or output.
- We can even redirect the input and output to other devices.
- This can be done using three variables –
 - stdin
 - stdout
 - stderr

Input and Output using sys

- **stdin**: It can be used to get input from the command line directly. It is used for standard input. It internally calls the input() method. It, also, automatically adds '\n' after each sentence.

- **Example:**

Python3

```
import sys
```

```
for line in sys.stdin:
```

```
    if 'q' == line.rstrip():
```

```
        break
```

```
    print(f'Input : {line}')
```

```
print("Exit")
```

```
hello
Input : hello

world
Input : world

q
Exit

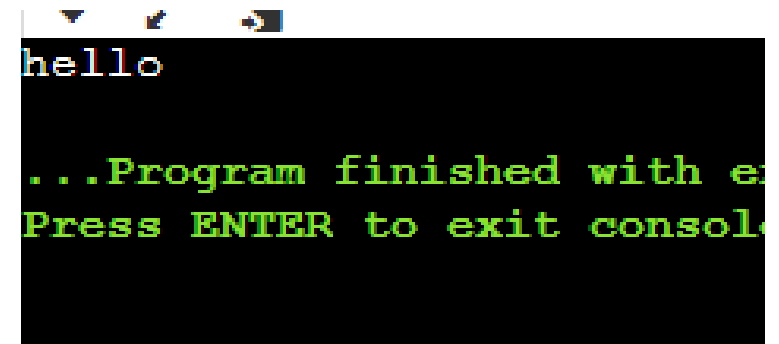
...Program finished
Press ENTER to exit
```

- **stdout**: A built-in file object that is analogous to the interpreter's standard output stream in Python.
- `stdout` is used to display output directly to the screen console.
- Output can be of any form, it can be output from a print statement, an expression statement, and even a prompt direct for input.
- By default, streams are in text mode. In fact, wherever a print function is called within the code, it is first written to **`sys.stdout`** and then finally on to the screen.

- **Example:**

```
import sys
```

```
sys.stdout.write('Hello')
```



- **stderr:** Whenever an exception occurs in Python it is written to sys.stderr.

```
import sys
```

```
def print_to_stderr(*a):
```

```
    # Here a is the array holding the objects
```

```
    # passed as the argument of the function
```

```
    print(*a, file = sys.stderr)
```

```
print_to_stderr("Hello World")
```

```
main.py
1  import sys
2
3
4  def print_to_stderr(*a):
5
6      # Here a is the array holding the objects
7      # passed as the argument of the function
8      print(*a, file = sys.stderr)
9
10 print_to_stderr("Hello World")
11

Hello World

...Program finished with exit code 0
Press ENTER to exit console.
```

Command Line Arguments

- When the python program is executed from a Command Line Shell/Terminal, the words written after name of python program file is/are known as command line arguments.
- By default those values are taken up as Strings
- The arguments are read in python using “sys” library (eg. Import sys)
- Eg
- #python myprogram.py hello how are you
- #sys.argv[0]=" myprogram.py "
- #sys.argv[1]="hello"
- #sys.argv[2]="how"
- #sys.argv[3]="are"
- #sys.argv[4]="you"

```
import sys
# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)
# Arguments passed
print("\nName of Python script:", sys.argv[0])
print("\nArguments passed:", end = " ")
for i in range(1, n):
    print(sys.argv[i], end = " ")
```

```
# Addition of numbers
Sum = 0

for i in range(1, n):
    Sum += int(sys.argv[i])

print("\n\nResult:", Sum)
```

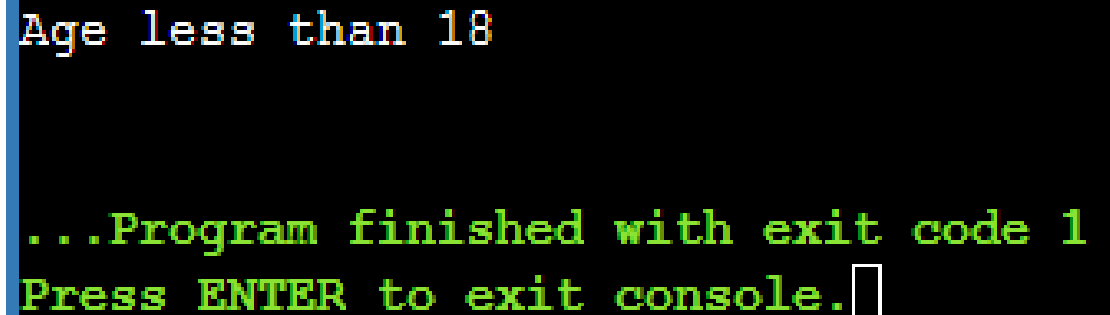
Command Line Arguments

- Instead of hard coding variable values directly within a program such values can be provided as input during program execution from command shell.
- Mostly it is used to built Operating system utilities executing through command shell.
- Using command shell the utility remains light weight rather than using heavy GUI which may not be supported by some hardware.

Exiting the Program

- `sys.exit([arg])` can be used to exit the program. The optional argument `arg` can be an integer giving the exit or another type of object. If it is an integer, **zero is considered “successful termination”**.
- **Note:** A string can also be passed to the `sys.exit()` method.

```
import sys
age = 17
if age < 18:
    # exits the program
    sys.exit("Age less than 18")
else:
    print("Age is not less than 18")
print("end")
```

A terminal window with a black background and green text. The first line shows "Age less than 18" in a monospaced font. The second line shows "...Program finished with exit code 1". The third line shows "Press ENTER to exit console." followed by a white cursor box.

```
Age less than 18
...Program finished with exit code 1
Press ENTER to exit console.
```

Python Modules

- A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

A simple module, calc.py

```
def add(x, y):  
    return (x+y)
```

```
def subtract(x, y):  
    return (x-y)
```

importing module calc.py

```
import calc
```

```
print(calc.add(10, 2))
```

- **The from import Statement**
- Python's *from* statement lets you import specific attributes from a module without importing the module as a whole.

```
# importing sqrt() and factorial from the  
# module math  
from math import sqrt, factorial
```

```
# if we simply do "import math", then  
# math.sqrt(16) and math.factorial()  
# are required.  
print(sqrt(16))  
print(factorial(6))
```

Shell
4.0
720
>

- **Import all Names – From import * Statement**
- The * symbol used with the from import statement is used to import all the names from a module to a current namespace.
- **Syntax:**
- **from module_name import ***
- The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the  
# module math  
from math import *
```

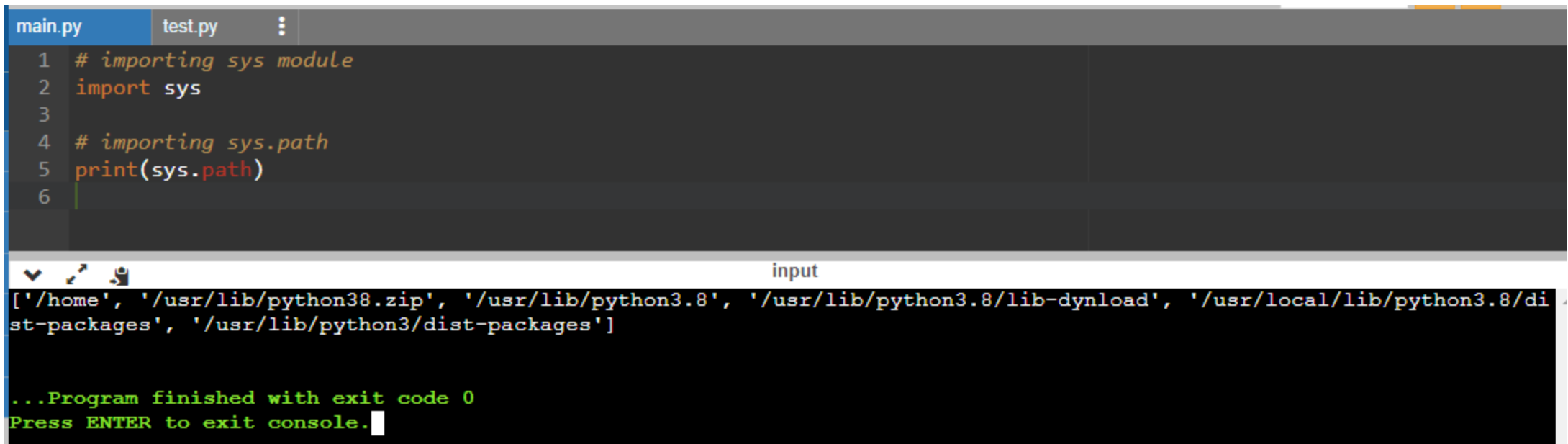
```
# if we simply do "import math", then  
# math.sqrt(16) and math.factorial()  
# are required.  
print(sqrt(16))  
print(factorial(6))
```

Shell
4.0
720
>

Locating Modules

- Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the [sys.path](#). Python interpreter searches for the module in the following manner –
- First, it searches for the module in the current directory.
- If the module isn't found in the current directory, Python then searches each directory in the shell variable [PYTHONPATH](#). The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.
-

- `# importing sys module`
- `import sys`
- `# importing sys.path`



The screenshot shows a code editor with two tabs: 'main.py' and 'test.py'. The 'main.py' tab is active, displaying the following Python code:

```
1 # importing sys module
2 import sys
3
4 # importing sys.path
5 print(sys.path)
6
```

Below the code editor is a terminal window titled 'input'. It displays the output of the Python script:

```
['/home', '/usr/lib/python3.8.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload', '/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages']
```

At the bottom of the terminal, it says: "...Program finished with exit code 0" and "Press ENTER to exit console."

- **Importing and renaming module**
- We can rename the module while importing it using the as keyword.
- **Example: Renaming the module**

```
# importing sqrt() and factorial from the  
# module math  
import math as nuv
```

```
# if we simply do "import math", then  
# math.sqrt(16) and math.factorial()  
# are required.  
print(nuv.sqrt(16))  
print(nuv.factorial(6))
```

```
4.0  
720  
> |
```


Random Module

- Python - Random Module
- The random module is a built-in module to generate the pseudo-random variables. It can be used perform some action randomly such as to get a random number, selecting a random elements from a list, shuffle elements randomly, etc.

Example: random()

```
>>> import random
>>> random.random()
0.645173684807533
```

```
# Import built-in module
random
import random
print(dir(random))
```

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
 'SystemRandom', 'TWOPI', '_Sequence', '_Set', '__all__', '__builtins__',
 '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', '_accumulate', '_acos', '_bisect', '_ceil',
 '_cos', '_e', '_exp', '_inst', '_log', '_os', '_pi', '_random',
 '_repeat', '_sha512', '_sin', '_sqrt', '_test', '_test_generator',
 '_urandom', '_warn', 'betavariate', 'choice', 'choices', 'expovariate',
 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate',
 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange',
 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
 'vonmisesvariate', 'weibullvariate']
```

Working with Modules

- **sys.path** is a built-in variable within the sys module that returns the list of directories that the interpreter will search for the required module.
- When a module is imported within a Python file, the interpreter first searches for the specified module among its built-in modules. If not found it looks through the list of directories defined by **sys.path**.
- **Note:** sys.path is an ordinary list and can be manipulated.

```
import sys
```

```
print(sys.modules)
```

```

ule '_weakref' (built-in)>, 'time': <module 'time' (built-in)>, 'zipimport': <module 'zipimport' (frozen)>, '_codecs':
<module '_codecs' (built-in)>, 'codecs': <module 'codecs' from '/usr/lib/python3.8/codecs.py'>, 'encodings.aliases': <mo
odule 'encodings.aliases' from '/usr/lib/python3.8/encodings/aliases.py'>, 'encodings': <module 'encodings' from '/usr/
lib/python3.8/encodings/__init__.py'>, 'encodings.utf_8': <module 'encodings.utf_8' from '/usr/lib/python3.8/encodings/
utf_8.py'>, '_signal': <module '_signal' (built-in)>, '__main__': <module '__main__' from 'main.py'>, 'encodings.latin_
1': <module 'encodings.latin_1' from '/usr/lib/python3.8/encodings/latin_1.py'>, '_abc': <module '_abc' (built-in)>, 'a
bc': <module 'abc' from '/usr/lib/python3.8/abc.py'>, 'io': <module 'io' from '/usr/lib/python3.8/io.py'>, '_stat': <mo
odule '_stat' (built-in)>, 'stat': <module 'stat' from '/usr/lib/python3.8/stat.py'>, '_collections_abc': <module '_coll
ections_abc' from '/usr/lib/python3.8/_collections_abc.py'>, 'genericpath': <module 'genericpath' from '/usr/lib/python
3.8/genericpath.py'>, 'posixpath': <module 'posixpath' from '/usr/lib/python3.8/posixpath.py'>, 'os.path': <module 'pos
ixpath' from '/usr/lib/python3.8/posixpath.py'>, 'os': <module 'os' from '/usr/lib/python3.8/os.py'>, '_sitebuiltins':
<module '_sitebuiltins' from '/usr/lib/python3.8/_sitebuiltins.py'>, '_locale': <module '_locale' (built-in)>, '_bootlo
cale': <module '_bootlocale' from '/usr/lib/python3.8/_bootlocale.py'>, 'types': <module 'types' from '/usr/lib/python3
.8/types.py'>, 'importlib._bootstrap': <module 'importlib._bootstrap' (frozen)>, 'importlib._bootstrap_external': <modu
le 'importlib._bootstrap_external' (frozen)>, 'warnings': <module 'warnings' from '/usr/lib/python3.8/warnings.py'>, 'i
mportlib': <module 'importlib' from '/usr/lib/python3.8/importlib/__init__.py'>, 'importlib.machinery': <module 'import
lib.machinery' from '/usr/lib/python3.8/importlib/machinery.py'>, 'importlib.abc': <module 'importlib.abc' from '/usr/l
ib/python3.8/importlib/abc.py'>, '_operator': <module '_operator' (built-in)>, 'operator': <module 'operator' from '/us
r/lib/python3.8/operator.py'>, 'keyword': <module 'keyword' from '/usr/lib/python3.8/keyword.py'>, '_heapq': <module '_
heapq' (built-in)>, 'heapq': <module 'heapq' from '/usr/lib/python3.8/heapq.py'>, 'itertools': <module 'itertools' (bui
lt-in)>, 'reprlib': <module 'reprlib' from '/usr/lib/python3.8/reprlib.py'>, '_collections': <module '_collections' (bu
ilt-in)>, 'collections': <module 'collections' from '/usr/lib/python3.8/collections/__init__.py'>, '_functools': <modul
e '_functools' (built-in)>, 'functools': <module 'functools' from '/usr/lib/python3.8/functools.py'>, 'contextlib': <mo
dule 'contextlib' from '/usr/lib/python3.8/contextlib.py'>, 'importlib.util': <module 'importlib.util' from '/usr/lib/p
ython3.8/importlib/util.py'>, 'mpl_toolkits': <module 'mpl_toolkits' from '/usr/lib/python3/dist-packages/mpl_toolkits/
__init__.py'>, 'apport_python_hook': <module 'apport_python_hook' from '/usr/lib/python3/dist-packages/apport_python_ho
ok.py'>, 'sitecustomize': <module 'sitecustomize' from '/usr/lib/python3.8/sitecustomize.py'>, 'site': <module 'site' f
rom '/usr/lib/python3.8/site.py'>}

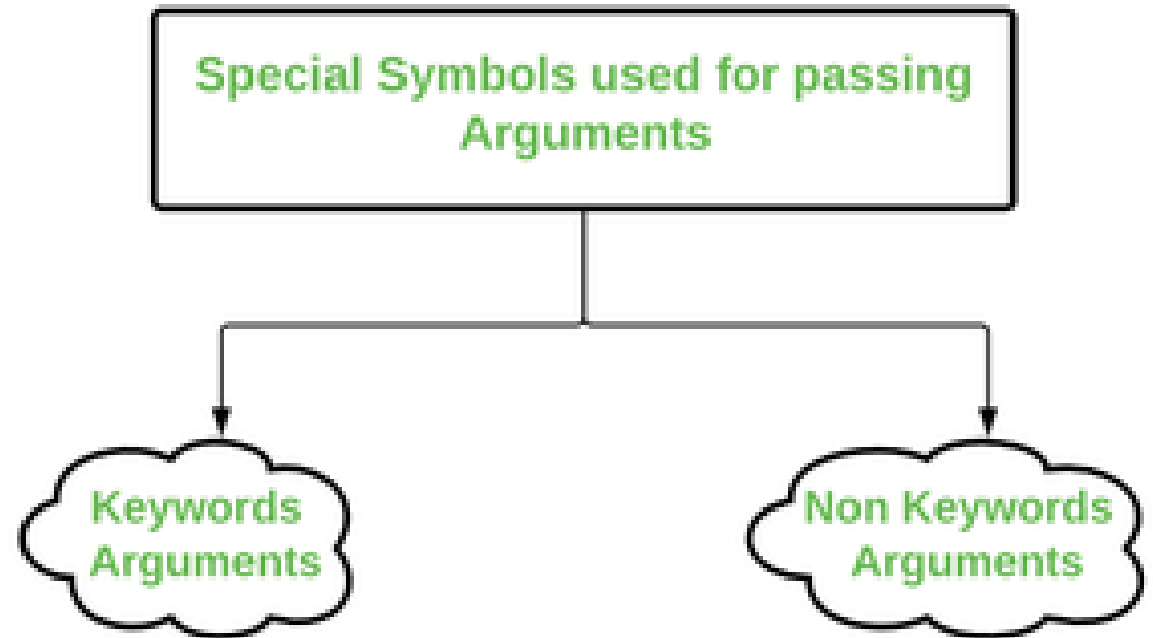
```

***args **kwargs**

Important

***args and **kwargs in Python**

- There are two special symbols:
- **Special Symbols Used for passing arguments:-**
- *args (Non-Keyword Arguments)
- **kwargs (Keyword Arguments)
- **Note:** *“We use the “wildcard” or “*” notation like this – *args OR **kwargs – as our function’s argument when we have doubts about the number of arguments we should pass in a function.”*



What is Python **args* ?

- The special syntax **args* in function definitions in python is used to pass a variable number of arguments to a function.
- It is used to pass a non-key worded, variable-length argument list.
- The syntax is to use the symbol *** to take in a variable number of arguments; by convention, it is often used with the word *args*.
- What **args* allows you to do is take in more arguments than the number of formal arguments that you previously defined.
- With **args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).

- For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together.
- It can be done using `*args`.
- Using the `*`, the variable that we associate with the `*` becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as `map` and `filter`, etc.

```
def myFun(*argv):  
    for arg in argv:  
        print(arg)
```

```
myFun('Hello', 'Welcome', 'to', 'NUV')
```

```
Shell  
Hello  
Welcome  
to  
NUV  
> |
```


What is Python ****kwargs**

- The special syntax ***kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list.
- We use the name *kwargs* with the double star.
- The reason is that the double star allows us to pass through keyword arguments (and any number of them).
- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it.
- That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

```
def myFun(**kwargs):  
    for key, value in kwargs.items():  
        print("%s == %s" % (key, value))
```

Driver code

```
myFun(first='Education', mid='beyond', last='books')
```

Shell

```
first == Education  
mid == beyond  
last == books  
> |
```

Python Lambda Functions

Python Lambda Functions

- **Python Lambda Functions** are anonymous function means that the function is without a name.
- As we already know that the *def* keyword is used to define a normal function in Python.
- Similarly, the *lambda* keyword is used to define an anonymous function in Python.

Python Lambda Function Syntax

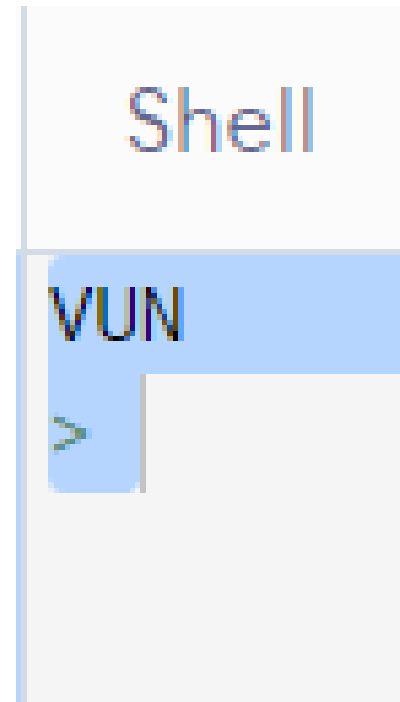
- *Syntax: lambda arguments: expression*
- *This function can have any number of arguments but only one expression, which is evaluated and returned.*
- *One is free to use lambda functions wherever function objects are required.*
- *You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.*
- *It has various uses in particular fields of programming, besides other types of expressions in functions.*

```
str1 = 'NUV'
```

```
# lambda returns a function object
```

```
rev_upper = lambda string: string.upper()[::-1]
```

```
print(rev_upper(str1))
```



```
format_numeric = lambda num: f"{num:e}" if isinstance(num, int) else  
f"{num:,.2f}"
```

```
print("Int formatting:", format_numeric(1000000))
```

```
print("float formatting:", format_numeric(999999.789541235))
```

Shell

```
Int formatting: 1.000000e+06  
float formatting: 999,999.79  
> |
```

Example 1: Python Lambda Function with List Comprehension

```
is_even_list = [lambda arg=x: arg * 10 for x in range(1, 5)]
```

```
# iterate on each lambda function
```

```
# and invoke the function to get the calculated value
```

```
for item in is_even_list:
```

```
    print(item())
```

Shell

10

20

30

40

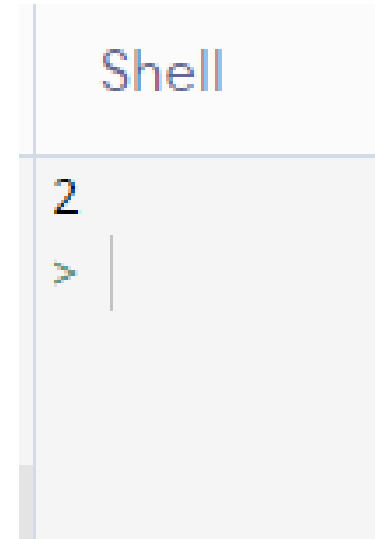
> |

Example 2: Python Lambda Function with if-else

Example of lambda function using if-else

Max = lambda a, b : a if(a > b) else b

print(Max(1, 2))



```
Shell
2
> |
```

Map vs Filter

- **Map** takes all objects in a list and allows you to apply a function to it

```
def square(num):  
    return num * num  
  
nums = [1, 2, 3, 4, 5]  
mapped = map(square, nums)  
  
print(*nums)  
print(*mapped)
```

```
1 2 3 4 5  
1 4 9 16 25
```

- **Filter** takes all objects in a list and runs that through a function to create a new list with all objects that return True in that function.

```
def is_even(num):  
    return num % 2 == 0  
  
nums = [2, 4, 6, 7, 8]  
filtered = filter(is_even, nums)  
  
print(*nums)  
print(*filtered)
```

```
2 4 6 7 8  
2 4 6 8
```

Using lambda() Function with filter()

- **Example 1: Filter out all odd numbers using filter() and lambda function**
- Here, `lambda x: (x % 2 != 0)` returns True or False if x is not even. Since `filter()` only keeps elements where it produces True, thus it removes all odd numbers that generated False.

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
```

```
final_list = list(filter(lambda x: (x % 2 != 0), li))  
print(final_list)
```

Shell

```
[5, 7, 97, 77, 23, 73, 61]  
> |
```

- **Example 2: Filter all people having age more than 18, using lambda and filter() function**

Python 3 code to people above 18 yrs

ages = [13, 90, 17, 59, 21, 60, 5]

adults = list(filter(lambda age: age > 18, ages))

print(adults)

Shell

[90, 59, 21, 60]

> |

Using lambda() Function with map()

- The map() function in Python takes in a function and a list as an argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.
- **Example 1: Multiply all elements of a list by 2 using lambda and map() function**

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
```

```
final_list = list(map(lambda x: x*2, li))  
print(final_list)
```

```
Shell  
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]  
> |
```

- **Example 2: Transform all elements of a list to upper case using lambda and map() function**

```
animals = ['dog', 'cat', 'parrot', 'rabbit']
```

```
uppered_animals = list(map(lambda animal: animal.upper(), animals))
```

```
print(uppered_animals)
```

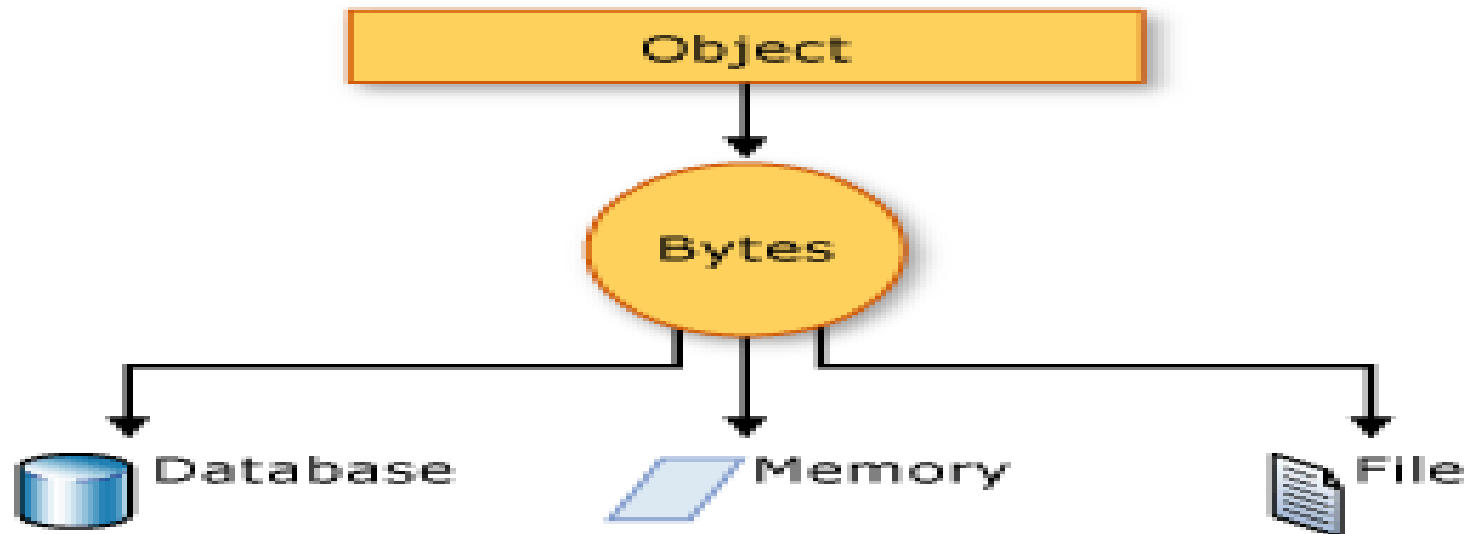
Shell

```
['DOG', 'CAT', 'PARROT', 'RABBIT']
```

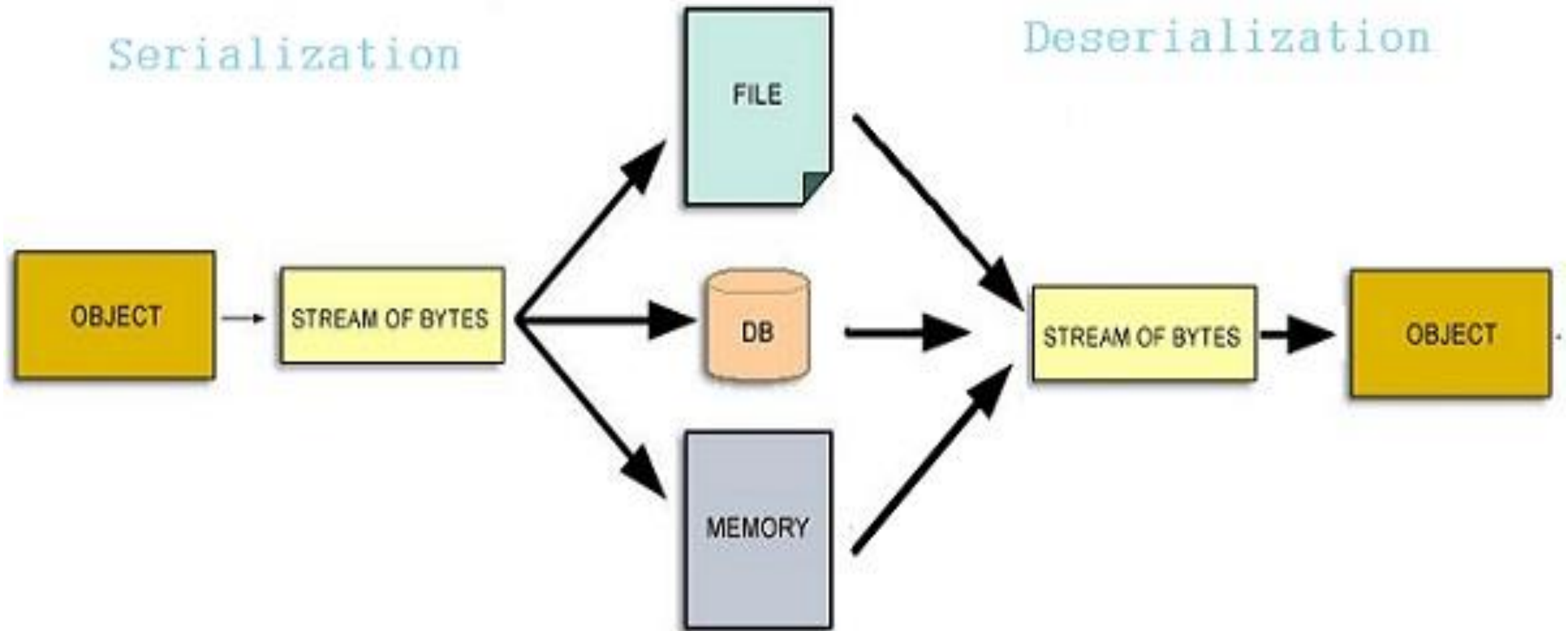
```
> |
```

Object Serialization – Saving Objects to File

- Serialization is a process of converting complex data types such as objects to array of bytes or any simple text type so that it can be written/persisted to a file, database or exported to any different platform.
- Deserialization is a process of reconverting back array of bytes to objects of underlying programming language object.
- It is the [process of writing the state of an object to a byte-stream.](#)



Object De-Serialization – Reading Objects from File



JSON and Pickle Serialization

- JSON is a text serialization format (it outputs unicode text, although most of the time it is then encoded to utf-8), while pickle is a binary serialization format.
- JSON is human-readable, while pickle is not.
- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific.
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes;
- Pickle can represent an extremely large number of Python types.

```
import pickle # A library which is used to write python objects/data in binary
form to store those details permanently in a file
class Student:
    pass
st1=Student()
st2=Student()
mydictw={
    st1.stu_rno
:{"rno":st1.stu_rno,"name":st1.stu_name,"weight":st1.stu_wt},
    st2.stu_rno
:{"rno":st2.stu_rno,"name":st2.stu_name,"weight":st2.stu_wt},
}
fw=open("studentobj.txt","wb") #wt - write mode (text) b-binary
pickle.dump(mydictw,fw)#will save object data to a file //Serialization
/Marshalling /pickling - converting python objects to binary format
fw.close()
```

JSON and Pickle Serialization

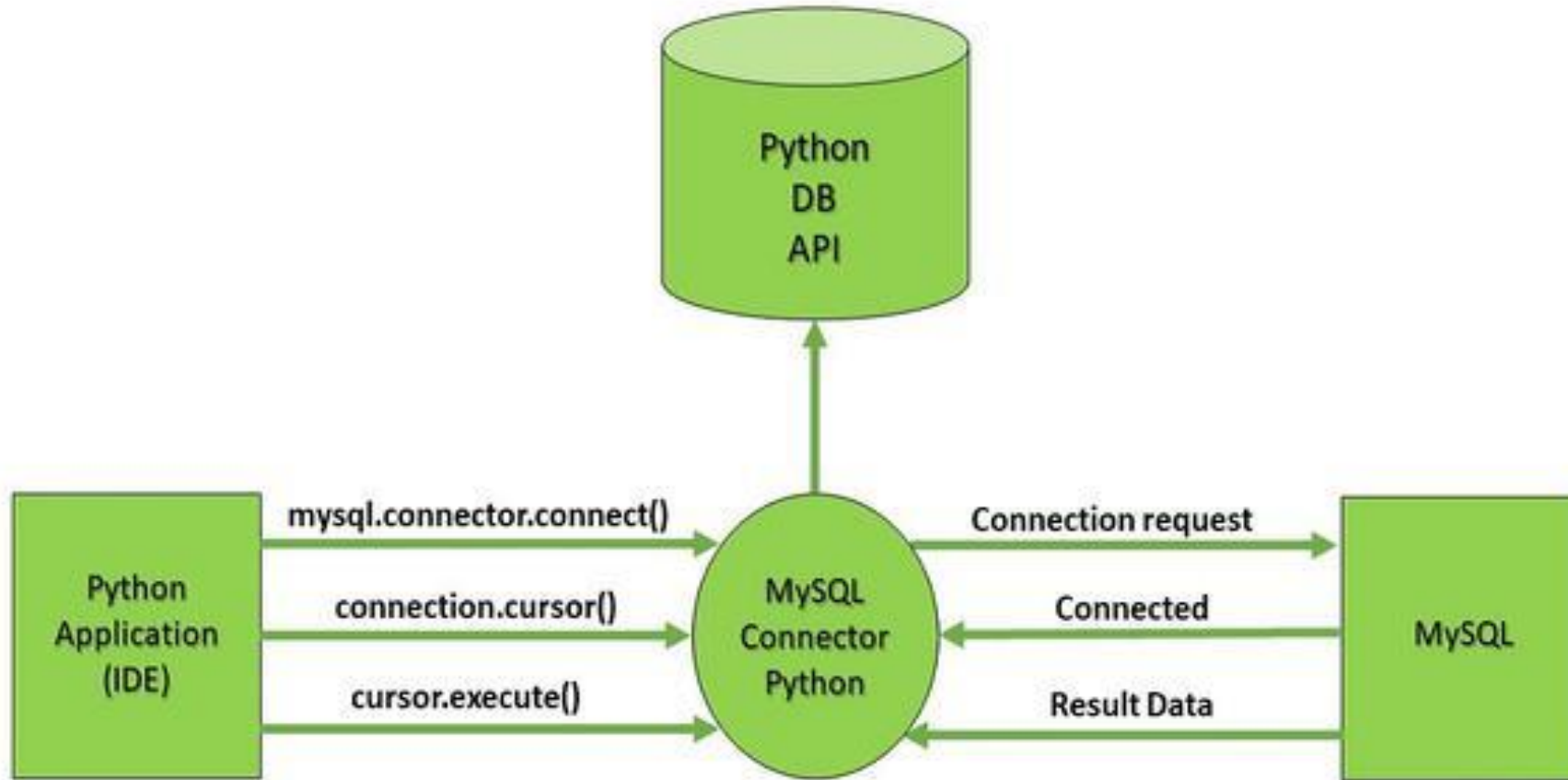
```
fr=open("studentobj.txt","rb") #rb - read binary mode
mydictr=pickle.load(fr) #deserialization, unmarshalling, unpickling means
converting binary object to python objects back again in memory
for k,v in mydictr.items():
    print(k,v)
fr.close()
```

Writing and Reading data with Excel

- Using pandas library in python, data can be read and write through Excel.

```
df=pd.read_excel("product_details.xlsx")
tmpdata={      "ItemId" : 23,    #Dictionary data
              "ItemName" : "Abcd Efg",
              "ItemPrice" : 34.89    }
df=df.append(tmpdata,ignore_index=True)
#df.insert(2,)
writer=pd.ExcelWriter("product_details.xlsx",engine="xlsxwriter") #Opening a excel file
df.to_excel(writer,sheet_name='Sheet1',index=False) # Writes data to Excel in Memory
representation in RAM (temporary)
writer.save() # Saves data to excel file permanently.
#####
df=pd.read_excel("product_details.xlsx") #read_excel functions reads data from excel file and
stores data to df.
print(df.loc[df.ItemId==int(vid),["ItemId","ItemPrice"]]) # loc is function filters/retrieval data
from a DataFrame
print(df.loc[df.ItemPrice>5000,:])
df=df.drop(df.loc[df.ItemId==int(srno),:].index[0]) #drop based on index deletes the row
```

Python Database Connectivity



Python Database Connectivity

- To access any database Python requires the database specific driver
- If python wants to access MySQL database, then the connector of MySQL database is required for Python.
- The “MySQL connector” is a driver for Python by which it can talk to MySQL database.
- Use PIP to install "MySQL Connector".
- `import mysql.connector` # mysql connector/driver is loaded into memory
- `mydb = mysql.connector.connect(host="localhost", user="username", password="password")` # Establishes connection to the database
- `mycursor = mydb.cursor()` #creates a statement object. It is able to execute any SQL statement which can create table, insert records, delete records, display records etc.
- `mycursor.execute(sql, val)` #Executes SQL query stored in cursor object
- `mydb.commit()` # Makes permanent changes to the database file
- `rs=myc.fetchall()` #Stores result (one or more rows) in form of Tuple from execute function after executing the SQL query.

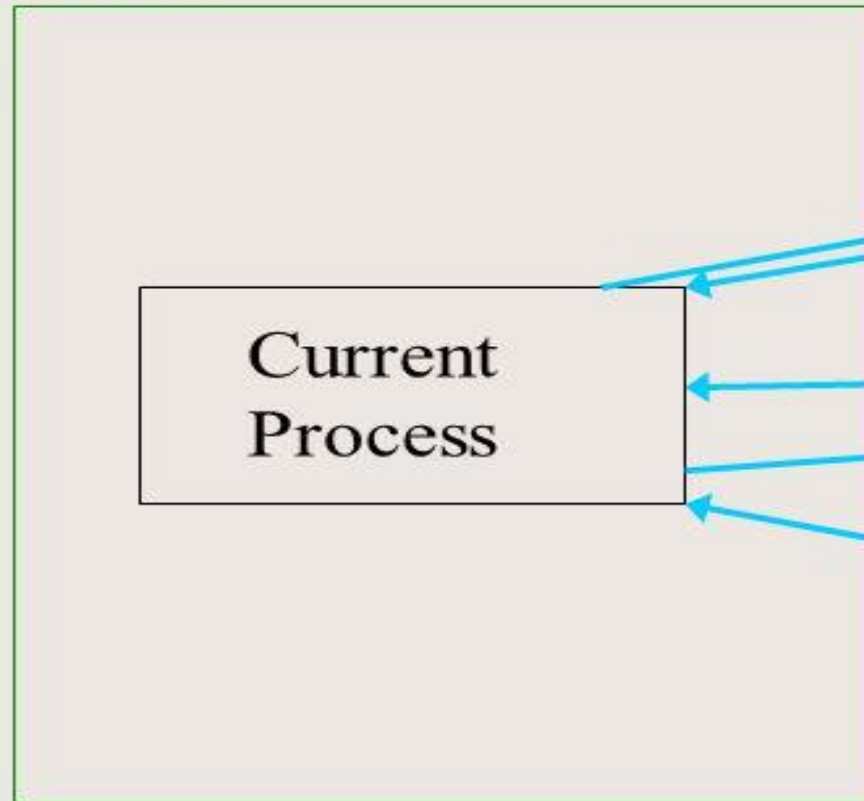
Multithreading in Python

Multi Tasking

- Modern OS - Windows 10, Linux, MacOS
- Important feature provided by OS – so that user can perform different tasks/programs at same time (browsing, editing word, playing video, etc.)
- Execute multiple programs/tasks at same time/concurrently
- Single-Processor is executing only one program at a time - switches between them very fast – appears multiple tasks are running at same time (context-switch).
- Multiple programs run as separate processes concurrently in single CPU.

CPU Time Slicing

CPU



Main Memory

**Allocate CPU
to Process 1**

**Allocate CPU
to Process 2**

**Allocate CPU
to Process 3**

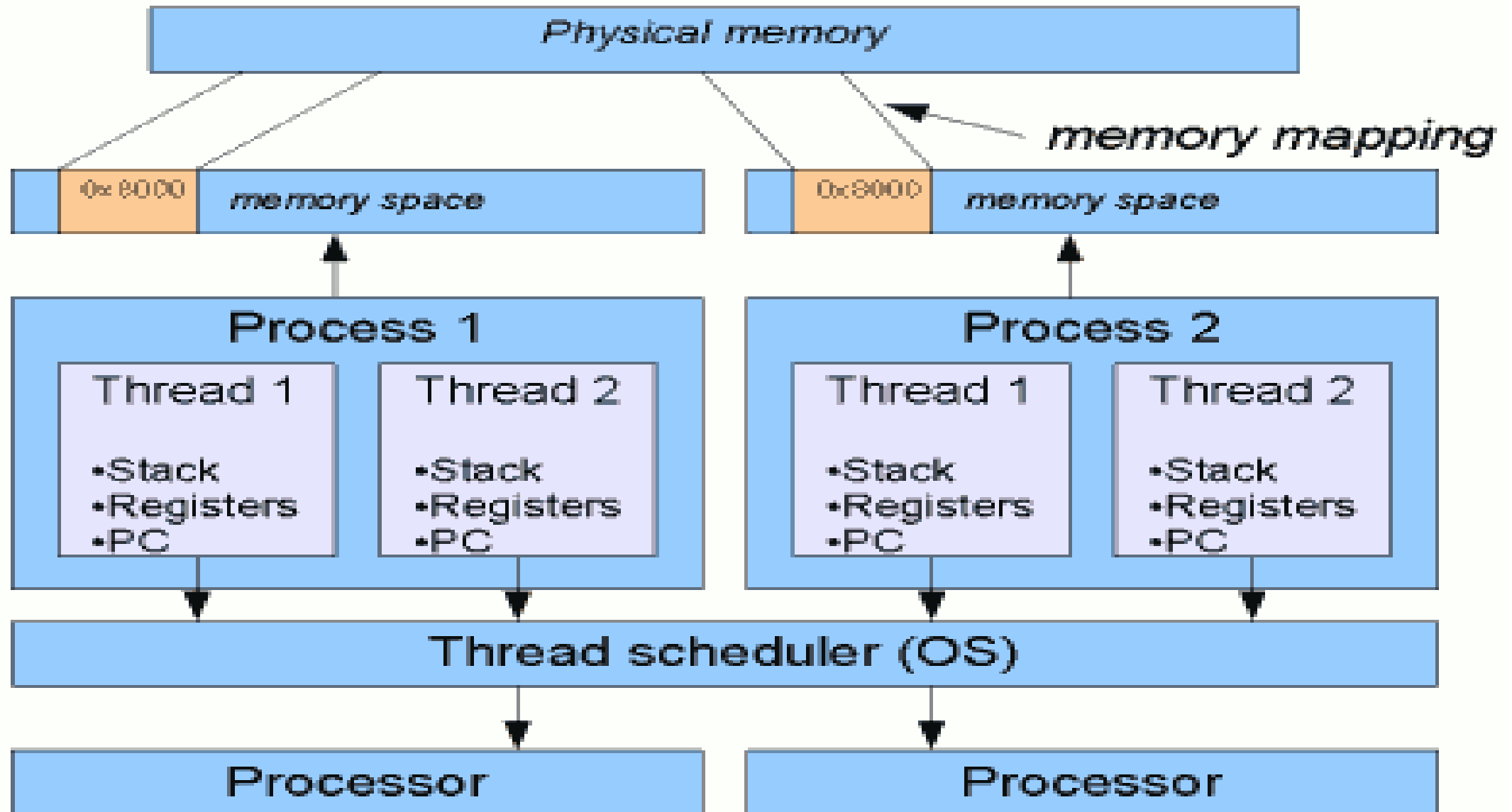
Process 1

Process 2

Process 3

Repeat until all processes have completed.

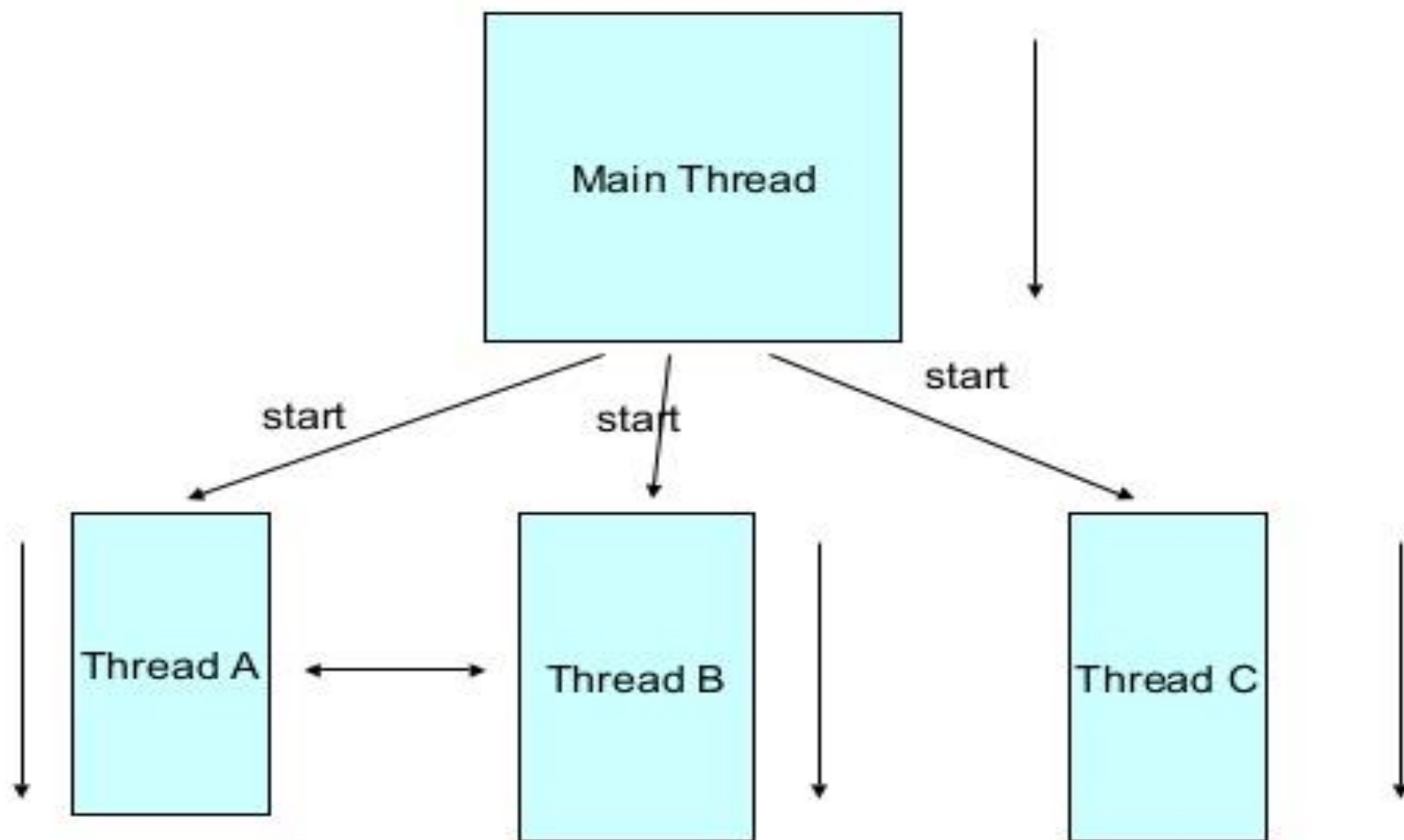
OS Scheduler - Process v/s Thread



Types of Multi Tasking

- **Process based multi tasking**
 - process is an instance of a computer program that is being executed. E.g. MS Excel and MS Word
 - Occupies address space (memory) and CPU cycles
 - a process may be made up of multiple threads of execution
- **Thread based multi tasking**
 - **Thread is a light weight process (less memory and CPU cycles)**
 - smallest unit of a dispatchable code by Scheduler
 - Single Python program - perform two or more tasks simultaneously

A Multithreaded Program



Threads may switch or exchange data/results

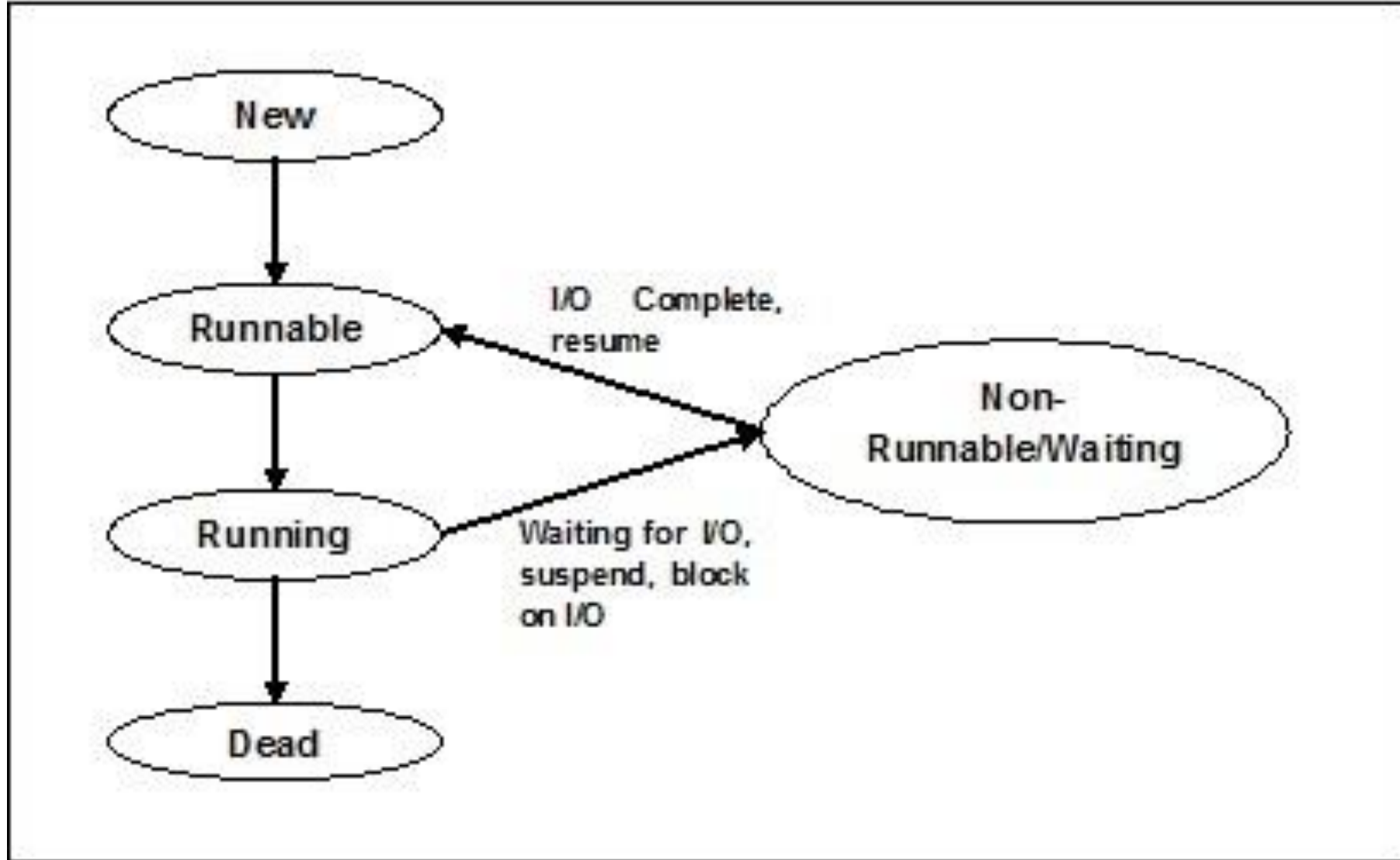
Advantage of using Threads

- A multi threaded program - contains two or more parts/threads that **run concurrently** and each thread performs different task **at the same time**
- Efficient and faster applications
- Maximum use of system resources/ multiple CPUs
- Concurrency and **multi-user support** can be provided
- **No blocking, faster response** to user requests
- **Network based/web based** server applications like FTP Server, Web Server, Mail Server, Messaging servers handles multiple client request in separate thread

How to create a Thread

- Thread in Python can be created by using threading package.
- Eg. **import threading** #imports the required package to create thread
- Now create object of Threading class
- **t1 = threading.Thread(target=print_num, args=(1,5))** #Thread object is created which creates thread also known as born state of thread. The first argument is name of the code which is written within a function. The second argument is no. of arguments required to pass within a function.
- **t1.start()** #Method must be invoked after creating a Thread. It is the method which activates the thread and sends to scheduler for execution. This state of the thread is also known as Runnable state.
- **t1.join()** #Method ensures that Main thread always ends at last. Before main thread ends child thread completes there process. This method must be invoked on all child threads to ensure that the main thread ends at last.

Lifecycle of Thread



References

- Michael H. Goldwasser, David Letscher , “Object-oriented Programming in Python “ , Pearson Prentice Hall, 2008
- Dusty Phillips , “Python 3 Object Oriented Programming” ,PACLT publications.
- Retrieved and referred from docs.python.org, July 2021.
- Retrieved and referred from <https://www.pcmag.com/encyclopedia>, July 2021.
- Retrieved and referred from <https://stackoverflow.com>, July 2021.
- Retrieved and referred from <https://wikipedia.org>, July 2021.
- Retrieved and referred from <https://w3schools.com>, July 2021.