Week 2 & 3

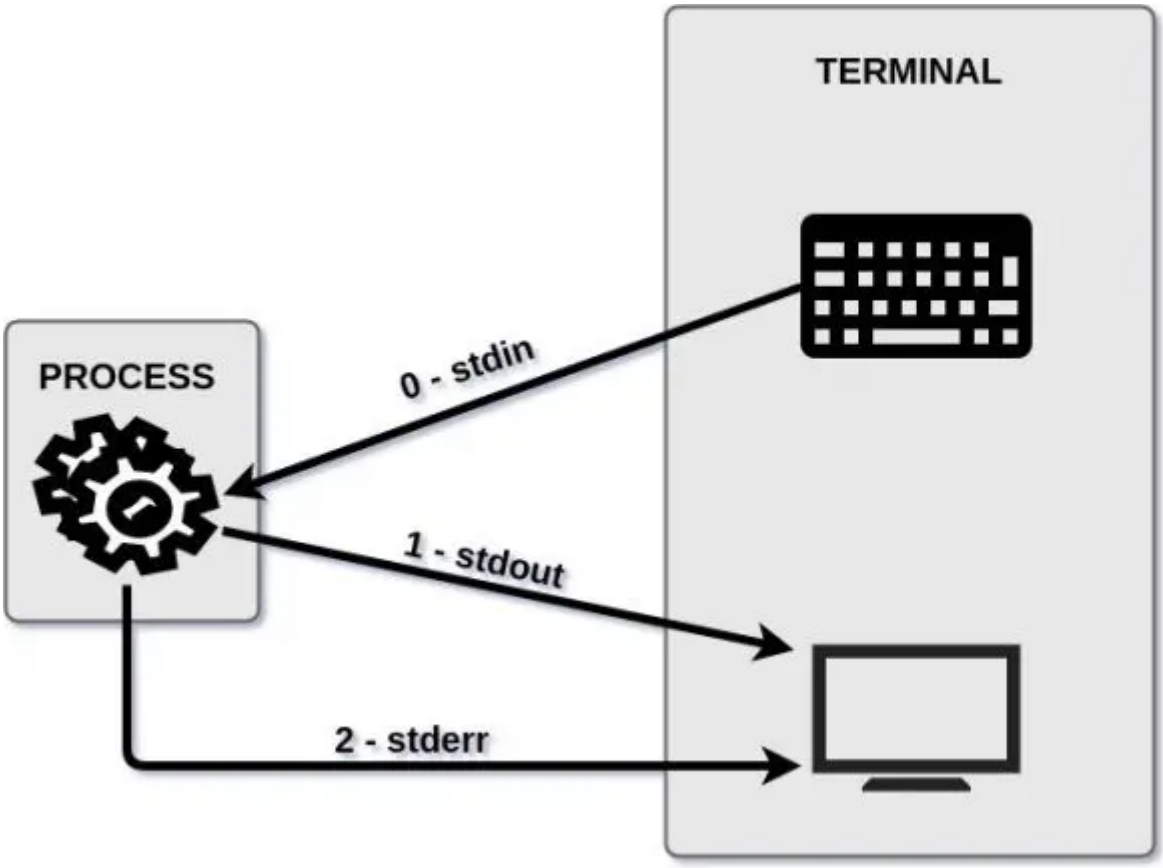
We covered the basic of CommandLine interface both in Microsoft Windows and Linux. The reason to go through this pain was to get everyone familier with a different approach to computers. What you need to understand is that the input stream, output stream and error stream, these all are related to a terminal/command prompt. Historically, before Graphical User Interface was introducted in the personal computer, a computer had just a blinking cursor upon booting.

Terminal

Terminal would act as the Main Center. Monitor, Keyboard and all other Peripheral devices are controlled by the terminal. In addition to that the terminal has two streams. Consider streams as water streams flowing in and flowing out. In our case, data flows in and flows out.

A Process (Computer Program in Execution)

A Computer Program in execution is called a Process. A Process runs under an Operating System. In our case, it's running in a terminal of an operating system. If the program wants to write string/int/float onto the output device (monitor), it has to send that data into the output stream of the terminal. Terminal will then write received string onto the monitor. Similarly, if the program wants to read some data, it has to receive that from standard input stream of the terminal.



Why this Tedious Process?

One may ask, why? Why not instead the program just writes the output itself on the screen.

Computer Abstraction

This kind of flow is called abstraction. A concept you will come across again and again in Operating Systems specially. That means each component should handle a particular task and nothing more. In case of process sending string/integer/float to standard output stream, the process just sends some stream of bytes (0s 1s). The program doesn't know what kind of display (physical screen) is the computer connected with, what is the font and font size, or even what is the position of the cursor right now on the screen. All of the latter details are handled by the terminal program. Just imagine how much work is avoided by following this simple ruling.

In a world without Abstraction

In a world without this concept, insane amount of work would have to be done for each program, one wrote. Forexample getting the physical details of the connected display (it's size in inches, pixels), handle the geometry of the font (that means coloring each pixel), and still after that would have to communicate with the terminal to get to know the current the current position of the cursor. Not to mention, different algorithms for different screens would have to be written. Read More. I hope you understand the concept, I tried to explain during the course of last two lectures.

In according to your experience, when you compiled and run the c++ program using dev-c++, a command window was initiated to handle the need of terminal/commandline with input and output streams, where the program could write data and receive data from.

Is terminal the only way?

Terminal is not the only way to display and receive data from user. Infact, most user level applications have nothing to do with a terminal. When was the last time you used an general purpose application that had a command window? Almost none. It is a common practice to stay with the terminal which learning programming. But advancing into the field, you will write computer programs containing thousands of lines of code without a single `cout` or `print` statment. When designing a Game, instead of `cout`, the output of the program will be setting up `pixel values` on the window provided by the Operating System and program will receive input from keyboard, mouse, joystick from that created window. When designing a general purpose software, the ouput will be setting up text and icons in the provided window. Again we have been using terminal to see the ouput. But it is not the only way to get the output.

Python

`Python` is a high-level, interpreted, scripting language. `Python` itself is just a computer program like any other computer program on your computer like the commands we have seen `ls`, `cd`. As I explained a command is a compiled C, C++ machine code. `Python` is created using C language. So when you downloaded and installed the `python` program, all you got is a machine code of C language. When you type `python` in the terminal and press enter, the operating system does exactly the same as it would do to operate the ls command, nothing more. Python machine code is present at a particular location in the operating system. Please do some research on how operating system locates the location of the python machine code and runs it. There are two ways this machine code can work.

Interactive Mode

If you type `python` in the terminal and press enter, you will see output something like

```
muhammad@debian:~$ python3
Python 3.11.2 (main, Nov 30 2024, 21:22:50) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This represents that now python is incharge, and not the terminal screen. Anything you write will be handled by the python process and not the terminal. You cannot write `ls` or `dir` command at this screen, if you do, you will receive an error. At this prompt, only python statments will be accepted.

Python Source File

Python Interactive mode is useful for small code snipets. As soon as you exit the python interactive mode, the program you wrote will not be saved. Therefore, a more practical way is to do the same as you did with C++ i.e. writing a source file. A Python Source file is in the most simplest wording just a text file, nothing special. Python source file is saved with `py` extension but it is nor compulsory neither does python care about the extension. But extensions are a great way for a person to see and understand the fact that this text file contains a python script.

Text File

`This part is optional and meant only for the curious among you. Everything mentioned is strictly Unix-based systems implementation (i.e., Linux and macOS), Not for Microsoft Windows.`
A number of filetypes exist in a file system. I will demonstrate a text file. A common perception about determining the type of a file is it's extension for example `.txt` represents a text file, `.exe` represents an executable file etc etc. And that may be true from a Person/User's perspective. But how does the computer determine, what kind of file is this. By seeing extension? Absolutely not.
A text file has two pieces of information along with it. One is called as MetaData and other is actuall data the text file contains which in this case is Ascii Text. MetaData contains information about the file itself forexample what is the name of the file, permissions on file, last modification time etc etc. Now let's create a text file using a text editor and examine it. Create two files using `nano` terminal based text editor. One named as `program` and other as `program1.py`. After that inspect both files using `file` command available in GNU/Linux.

```
muhammad@debian:~/IntroToPython$ file program
program: ASCII text

muhammad@debian:~/IntroToPython$ file program.py
program.py: ASCII text
```

Both files are ASCII based text files. There is absolutely no difference between the two files. With the extension and without it, they are identical. When python parses a text file, it does not care if the text file holds an extension. All it's concerned for is the text contained inside of the file should be based on ASCII and should be following strict rules of python (i.e. indentation, next line to end statments etc). `But again, in a nutshell, it is just a text file`. Moving on to see the MetaData of the file. `stat` command can be used for that.

```
muhammad@debian:~/IntroToPython$ stat program
  File: program
  Size: 2           Blocks: 8          IO Block: 4096   regular file
Device: 8,2 Inode: 5010414     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/muhammad)   Gid: ( 1000/muhammad)
Access: 2025-04-01 16:43:49.293258366 +0530
Modify: 2025-04-01 16:43:45.613258393 +0530
Change: 2025-04-01 16:43:45.613258393 +0530
 Birth: 2025-04-01 16:43:45.613258393 +0530
```

The most interesting part in a text file is the actuall data it contains. It is possible to review the data in raw format using `xxd` comamnd.

```
muhammad@debian:~/IntroToPython$ cat program.py
print("Hello")
print("World")
muhammad@debian:~/IntroToPython$ xxd -b program.py
00000000: 01110000 01110010 01101001 01101110 01110100 00101000  print(
00000006: 00100010 01001000 01100101 01101100 01101100 01101111  "Hello
0000000c: 00100010 00101001 00001010 01110000 01110010 01101001  ").pri
00000012: 01101110 01110100 00101000 00100010 01010111 01101111  nt("Wo
00000018: 01110010 01101100 01100100 00100010 00101001 00001010  rld").
```

I used `-b` flag to receive output in binary values. Otherwise they would have been in hexadecimal format. Examining the output, there are 3 columns. The leftmost represents the binary offset in hexadecimal format, for the time being, just ignore it. The middle column is all binary values. The rightmost is the actuall ASCII text. Please open an online binary to decimal calculator and ASCII Table. We can match the binary values with the ASCII text. The 1st value `01110000` when converted to decimal becomes `112`. Looking at the ASCII table, `112` matches with character `p`. The next value `01110010` transaltes to `114` whose ASCII equivalent is `r` and so on.

> Exercise: Can you spot the next line / new line binary form, it's equivalent decimal form and the ASCII character? Additionally, how many next line characters appear in the above output?

The commands like `file` and `xxd` are super useful. I am showing the most simplified form of these commands to you. Anyone interested may read it more via their documentation pages. Two things should be clear by now

- Python Interpreter is the machine code of a program written in `C` language.
- Python Source file is just a text file.

WorkFlow of Execution of a Program

There are two types of computer programs.

- System Software
- Application Software

System Softwares are the softwares which directly run on the hardware without an intermediatory between. Best example of system sofware is the Operating System. Windows, Linux, MacOS all are system sofwares. The software we write are generally application layered programs. These programs depend on the OS for their execution. Let's say a program that wants to request some input from the user. All it needs to do is to read input from the terminal input stream. The program has no idea of what company the keyboard came from, what layout is being used. All of these details are handled by the OS. The program can never run on a raw hardware i.e. a hardware with no Operating System on it. So the programs we write will be running under the hood of an OS. A program in execution is called a `process`. Process lives in memory (RAM), every python statement written has some appearence in the memory. Forexample the statement `5` is saved in RAM. But here one may ask the questions like

- Can the program just random pick some memory addresses and store it's value in it?
- If yes, what if different parallely running programs try to access the same value?

Turns out that is not how it works. Your Operating System has a very strict control over all hardware resources. No Application level software can access the hardware without the OS's permission. OS acts as a policeman. When a process starts, it has to request the OS for certain amount of memory addresses in the RAM. OS then checks the memory addresses and sees which part of RAM is free i.e. not occupied by any other process. Then the OS will assign the process that chunk of RAM. Access to other hardware resources work in similar way.

Python Syntax

We have seen that a python source file is a text file and python interpreter a machine code has to read it. Now the reading entity is a computer and computers are incredibly dumb entities. Therefore, the set of instructions passed to the python interpreter must be very structured. In order to do that programming languages define a set of rules. Forexample, the `g++` compiler expects every statement to be ended with a semi-colon `;`. Similary, python has a set of rules to be followed.

- Each statement ends with nextline character.
- Single line cannot contain two or more statements.

And that's all. I don't want you to focus on something we are not dealing with at the moment.

Data Types in Python

There are 4 main data types in python. `Integer`, `Float`, `String` and `Bool`. All of these datatypes are self-explanatory.

- Integers : Integers called `int` in `python` store integers as 0, 6, 100, 12354, and also negative number as -7, 5, -2 etc.
- Floats : Floats called `float` in `python` store decimal values 0.2, -1.6, 12.64, etc.
- Strings : Strings called `str` in `python` store strings. Strings are meaningless ASCII characters. These characters are enclosed in quotes `"`. If not enclosed in quotes `"`, python treats them specially considering them variable name, keyword, or function etc. `"This is a string"`, `"sdfertG"`, `"one"` are all examples of strings. One thing to note here is that if an integer is enclosed in quotes, `"2"` it loses it's own value and becomes a string. The integer rules cannot be applied on it, strings rules will be.
- Bool : Bool called `bool` in `python` can hold only two values. `True` and `False`.

In terms of detecting datatypes, python will do that automatically. You don't need to explicity mention any datatype.

```
5
```

The above statement is a valid python statement. The way this would work is python interpreter will see it as an integer, will request the OS for memory location to store a single integer, will store the integer in memory, The way Python interpreter would run this program

- Sees a single integer.
- Request the OS for memory location to store a single integer.
- After getting memory address, stores the integer in the memory location.
- As there are no next statement, the python interpreter/ongoing process terminates immediately.
- The memory location occupied will now be marked free for other processes to use.

```
2.0
```

Use a point makes python declear it as floating point value.

```
"This is a string"
```

Using quotes makes it a string.

Standard Output

The most fundamental part in a computer program is to actually get the output from the program. To begin with, we are going to use terminal to get output from the python interpreter. In future other application programs will be used. In order to throw output to the standard output of the terminal, `print()` function is used which looks very strange when compared to c++. We will study more about functions in the upcoming lectures. For now just know this, function is a set of code that either has an affect or returns a value. Function may also take some input as argument for it's successful execution. `print()` function accepts int/float/string as arguments and puts them in the standard output stream of the connected terminal. It is important to note that `print()` does not return anything at all.

```
print(2)
```

In the above python statement, python puts 2 in standard output stream of the terminal which then displays the 2 on the screen.

```
print(3, 5.0)
```

`print()` can accept as many arguments as required, using comma to separate them. This is a different approach from C++, where functions can only accept limited numbers or accepts pointer.

> For those who are Curious. (Optional) → `Study how `printf()` function works in C language. How is it able to accept a number of arguments.`

There are two types of arguments in python functions.

- Positional Arguments
- Keyword Arguments Positional Arguments are expected to appear based on their positions in the function definition.

```
def func(user1, user2)
```

When calling the function above, the values for user1 and user2 can be passed based on their positions.

```
func("bob", "tam")
```

The string `"bob"` is assigned to `user1` and "tam" is assigned to `user2`. This is passing value to a function based on their positions. But what if I want to pass `"bob"` to `user2` and `"tam"` to `user1`. One way is to change their positions.

```
func("tam", "bob")
```

But there is another way. And that is to use the keywords `user1` and `user2`.

```
func(user1="tam", user2="bob")
```

This can also be written as

```
func(user2="bob", user1="tam")
```

Therefore that is the flexibility of using keyword arguments. When using positions, you have to be extra careful. More on keyword and positional arguments in next lectures. For now just know this.

```
print(...., sep, end)
```

`print()` accepts random number of values with any datatype. They are specified by `....` I mentioned. Despite of that `print()` also accepts two other arguments named as `sep` and `end`. Using positions to

specify values, we can never reach `sep` and `end`. They can only be assinged values using `sep` and `end` keyword.

> Sep → `sep (short form of separator) specifies the string to separate output values. It's default value is a single space character.`

> End → `end specifies the string to end the print statement. It's default value is a single next line character.`

```
>>> print("3", 35, 23, True, sep= "_")
3_35_23_True
>>>
```

```
>>> print("3", 35, 23, True, end= ".")
3 35 23 True.>>>
```

```
>>> print("3", 35, 23, True, sep= "_", end= ".")
3_35_23_True.>>>
```

Practice Questions

Conceptional Questions

> What are the three main types of streams in a terminal, and how do they function?

> Why is a terminal considered the "Main Center" for a computer's input and output?

> Define a process in the context of a computer program. How does it interact with the terminal?

> Explain the concept of abstraction in computing. Why is it important in operating systems?

> What challenges would arise in a world without abstraction in computing?

> Why does running a C++ program in Dev-C++ open a command window?

> How do general-purpose applications handle input and output differently from terminal-based programs?

Terminal/CommandLine

> Explain Environment Variables.

> Explain `PATH` environment variable.

> Describe how a Python command is executed in a terminal. How does the operating system locate and run the Python executable?

Python Basics

> What is the difference between running Python in interactive mode and executing a Python script from a file?

> How does the Python interpreter determine the type of a file it is reading? Does it rely on file extensions?

> Why is the .py extension not necessary for a Python script, even though it is commonly used?

Hands-On Tasks

> Write python script that sends the string "Hello, world" to standard output of the terminal.

> Write python script that sends an integer, float, string, boolean separated by backward slash "" value to standard ouput. Leave end as it's default value.

> Using single print statement, write your name and age on two lines.

> Print the following text using print statements.

```
Roses are red,
Violets are blue,
Python is awesome,
And so are you!
```

> Print the result of the Math Expression using a single python statement. Note: You are not allowed to use variables.

```
5+2*(3-4)
```