**Session 2**

Error Handling & Paths

## ERROR HANDLING IN PYTHON

### **ERROR**

* Errors are problems in the program that the program should not recover from.
* If an error occurs at any point in the program, the program exits.
* Error in Python can be of two types i.e. Syntax errors and Exceptions.
* SYNTAX ERROR: -This error is caused by the wrong syntax in the code. -It leads to the termination of the program.
* EXCEPTION ERROR: -Exceptions are raised when the program is syntactically correct but the code results in an error. -This error does not stop the execution of the program, -however, it changes the normal flow of the program.

### EXCEPTIONS

* Exceptions, on the other hand, are errors that occur during the execution of a program.
* Despite the program being syntactically correct, unexpected situations or conditions may lead to Exceptions.
* Unlike Syntax Errors, Exceptions do not bring the program to an abrupt halt.
* ZERO DIVISION ERROR: - An example of exceptions, ZeroDivisionError. Instead of terminating the program, it prints an error message, continues with the else, and finally blocks.

### EXCEPTION HANDLING

Exception handling in Python is a mechanism that allows you to gracefully handle errors that may occur during the execution of your code. It prevents your program from crashing and provides a way to handle unexpected situations. Exception handling is done using try, except, else, and finally blocks.

**Here's a structure of exception handling in Python:**

* **try block**: This is the block of code where you suspect an exception might occur. If an exception occurs, it jumps to the exceptblock.
* **except block**: This block is executed only if an exception of the specified type occurs in the tryblock. You can catch specific exception types or use a more general exception to catch any exception. The caught exception is assigned to the variable e in this example.
* **else block (optional)**: This block is executed if no exception occurs in the tryblock. It is optional**.**
* **finally block (optional):** This block is always executed, regardless of whether an exception occurred in the try block. It is used for cleanup operations, such as closing files or network connections. It is also optional.

**Here's another example with a more specific exception:**

try:

num = int(input("Enter a number: "))

result = 10 / num

except ValueError:

print("Invalid input. Please enter a valid number.")

except ZeroDivisionError:

print("Cannot divide by zero.")

else:

print(f"The result is: {result}")

finally:

print("This block will always be executed.")

In the above example, the program asks the user to enter a number, tries to convert it to an integer, and then divides 10 by that number. If the user enters a non-numeric value, a ValueError is raised, and if the user enters 0, a **ZeroDivisionError** is raised. The appropriate except block is then executed.

**raise**

• instead of using predefined errors we can raise our error

name = input('enter the file name :')

try:

f = open(name+'.txt')

if f.name == 'test.txt':

raise Exception

except FileNotFoundError as e:

print(e)

except Exception:

print("this fine should not be read")

else:

data = f.read()

f.close()

print(data)

finally:

print("sorry")

enter the file name: sample

[Errno 2] No such file or directory: 'sample.txt'

sorry

## Understanding paths in python

Paths refer to locations where files and applications are stored in your computer.

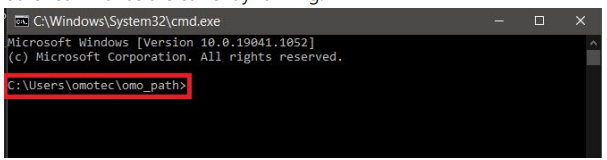
There are a few terms that we need to be aware of before we start.

1. Current working directory current working directory is where the command prompt is currently situated. We are sometimes required to be in the same folder as the file is, to be able to access the files.

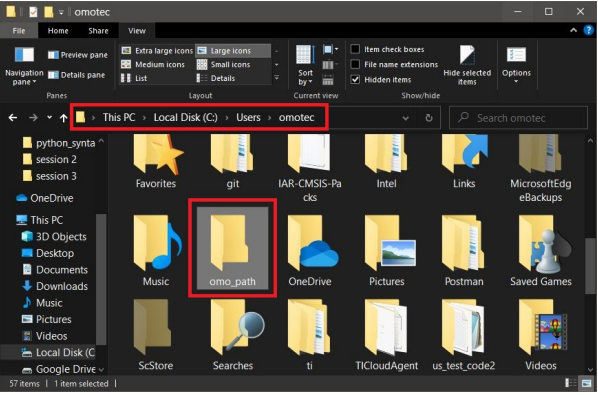
cd command gives you the absolute location of the current working directory.

cd command can also be used to change the directory to the specified one.

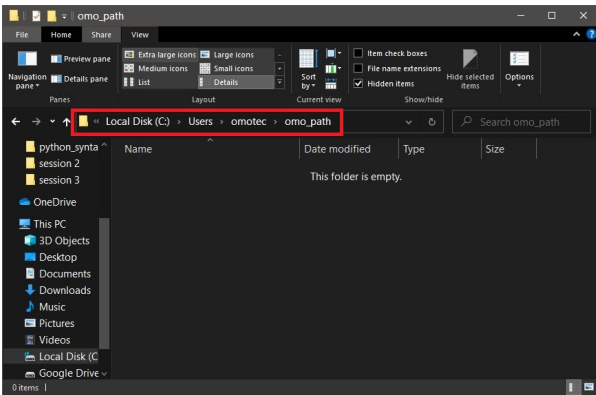
The command prompt also shows the current working directory on every line provided no other commands are currently running.

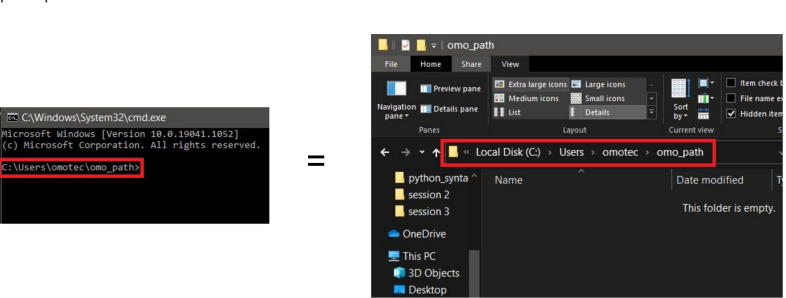


The path shown is a folder in your computer.

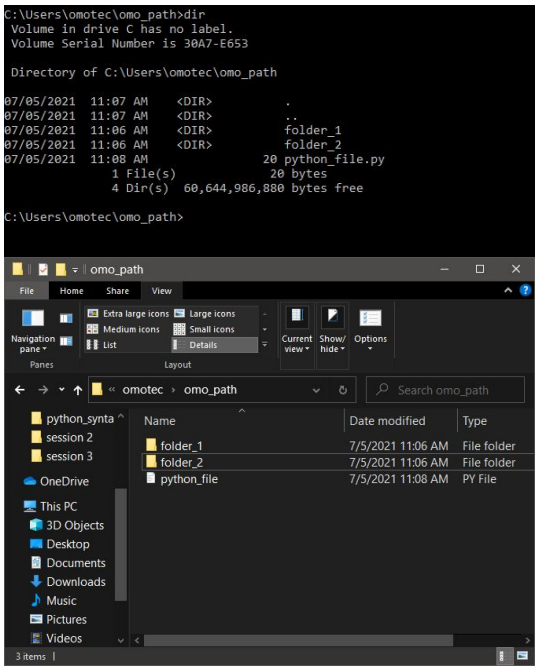


which can also be opened via the file explore If we need to access any files in this folder it can be done by opening a command prompt in this folder

 Opening a folder in File Explorer is similar to being in the same directory in the command prompt.



2. The **dir** command can be used to list all the files and folders in a directory.

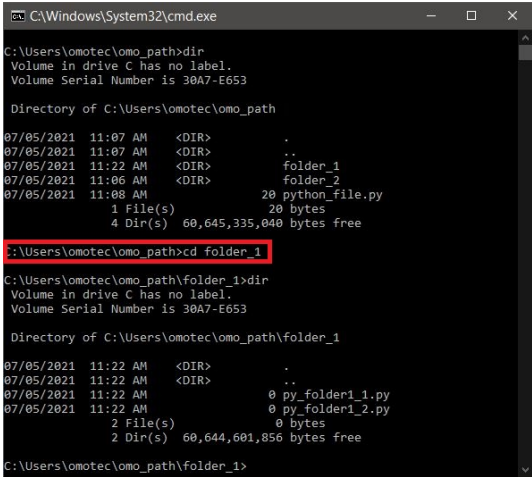


**2 types of paths can be defined in any operating system.**

1) Absolute path 2) Relative path

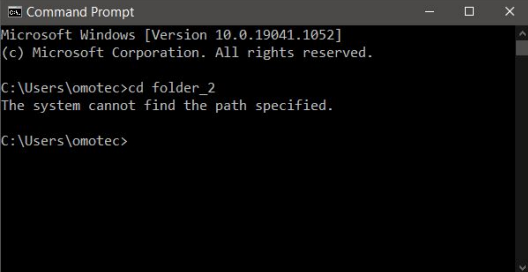
If you are in the folder in which you want to access a file there is no need to be concerned about absolute and relative paths. What we mean by being in a folder is that the current working directory should correspond to the folder in which the file to be accessed has to be. Even moving around the folders using the cd command does not require any use of relative or absolute path if we are accessing a folder that is already in the current working directory.

For eg: If we want to change the directory and go to the folder\_1 directory we can do that directly without any absolute or relative paths.



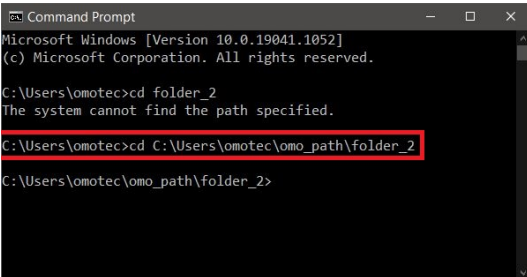
**Absolute Path**

Absolute paths are always referenced concerning base drive. These paths can be used irrespective of the current working directory. Let's try accessing the folder\_2 directory from the omotec folder



The below error pops up as it cannot find the folder\_2 directory in the omotec folder.

This is a good scenario where we can use the absolute path. It's just a choice we make; we could have also used relative paths.



This time no errors are shown as we are using absolute paths. So it goes through every folder we specified in the new path and then tries to find folder\_2

**Relative Path**

Relative paths are always referenced to the current directory/folder. These paths can only be used if we are the reference directory. Let's try accessing the folder\_2 directory from the omotec folder using relative path.

