CIT Students Only

Section-1: Errors and Exceptions : Syntax Errors, Exceptions, Handling Exceptions, Raising Exceptions, User-defined Exceptions, Defining Clean-up Actions, Redefined Clean-up Actions.
Section-2: File Operations: Reading config files in Python, Writing log files in Python, Understanding read functions, read(), readline() and readlines(), Understanding write functions, write() and writelines(), Manipulating file pointer using seek, Programming using file operations Manipulating file pointer using seek().

Section-1: Errors and Exceptions

What are Exception in Python?

In Python, exceptions are events that occur during the execution of a program and disrupt the normal flow of the program. Whenever an exception occurs, the program stops the execution. Therefore,

An exception is a Python object that represents the run-time error that could not be handled by the Python script/program.

Python provides a way to handle these exceptions, so that the code can be executed without any interruption. Exceptions are useful for handling errors, exceptional conditions, or unforeseen events that may occur during the execution of a program. They help prevent the program from crashing and allow developers to handle errors in a controlled manner.

Python has many built-in exceptions and we can also create our own custom exceptions that enable our program to run without interruption and give the output. Here are some examples of commonly used built-in exceptions in Python

Define few Types of Errors and Built-in Exceptions in Python.

Python provides a number of built-in exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

No.	Type of errors	Built-in Exceptions
1	Error occurs when a number is divided by zero.	**ZeroDivisionError**
2	Error occurs when given value is not correct	**ValueError**
3	Error occurs when a name is not found. It may be local or global	**NameError**
4	Error occurs when an operation is applied on objects of different data types	**TypeError**
5	Error occurs when a file or directory is requested but cannot be found.	**FileNotFoundError**

CIT Students Only

6	Error occurs when you attempt to access a list or sequence with an invalid index.	**IndexError**
7	Error occurs when you try to access a dictionary with a key that doesn't exist.	**KeyError**
8	Error occurs if incorrect indentation is given	**IndentationError**
9	Error occurs when Input Output operation fails	**IOError**
10	Error occurs when the end of the file is reached, and yet operations are being performed	**EOFError**

Problem if we don't handle exceptions:
Suppose we have two variables a and b, which take the input from the user and perform the division of these values. If the user entered the zero as the denominator, then it raises ZeroDivision exception because division by 0 is not allowed. **This will interrupt the program execution.**

Application without Exception Handling:

```
a = int(input("Enter numerator: "))
b = int(input("Enter denominator: "))
c = a/b
print("a/b=",c)

# remaining code
print(" Rest of the statements ")
```

Output:
Enter numerator: 7
Enter denominator: 0
Traceback (most recent call last):
  File "c:\Courses\Python 22-23\code\excep1.py", line 3, in <module>
    c = a/b
        ~^~
ZeroDivisionError: division by zero

Explanation:
The above program is syntactically correct, but it stopped with an error **ZeroDivisionError**. That's because a division by zero is not defined and our program doesn't know how to handle it. So it crashes.

CIT Students Only

Illustrate exception handling with a sample program.

Exception handling in Python ensures the programs are not crashed and are executed without interruption.

Full Syntax for Exception Handling:

```
try:
    Run this code
    to capture an exception
except Exception1:
    Run this code
    if this exception occurs
except Exception2:
    Run this code
    if this exception occurs
else:
    Run this code
    if NO exception found
finally:
    ALWAYS
    Run this code
```

- **try block:** Consist of normal statements and throws an exception if any error occurs.
- **except block:** Executes a code as a response to the exception that occurred in the **try** block.
- **else block:** Executes a code if NO exceptions occurred
- **finally Statements:** If we have some code that shall be executed at the end, no matter what happened in **try-except** blocks, we can write it into a **finally** block. **This code will always be executed**, even if an exception remains unhandled.

Method-1: try-except statement:

1. **try block** - Python executes code in the **try** block as a "normal" part of the program. If it contains suspicious code, then it will throw an exception.
2. **except block** - It is the program's response to an exception found in try block. The **except** block block of code will be executed if there is some exception in the **try** block.

Method-1 try - except	Example

try:
Run this code to capture an exception
except:
Run this code if an exception occurs
	`try:`
	`    a = int(input("Enter numerator: "))`
	`    b = int(input("Enter denominator: "))`
	`    c = a/b`
	`    print("a/b=",c)`
	`except:`
	`    print("Can't divide with a zero")`
	Output:
	Enter numerator: 7
	Enter denominator: **0**
	Can't divide with a zero

Method-2 try - except exception_name statement
try: block to run normal code
except exception_name: this block will respond to the given exception_name if it occurs in try block

Method-2 try - except exception	Example

try:
Run this code to capture an exception
except Exception1:
Run this code if this exception occurs
except Exception2:
Run this code if this exception occurs
	`try:`
	`    a = int(input("Enter numerator: "))`
	`    b = int(input("Enter denominator: "))`
	`    c = a/d`
	`    print("a/b=",c)`
	`except ZeroDivisionError:`
	`    print("Can't divide with a zero")`
	`except NameError:`
	`    print("Name is not found")`
	Output:
	Enter numerator: 9
	Enter denominator: 3
	Name is not found

CIT Students Only

Method-3 try - except exception_name - else statement

We can use the **else** statement with the **try-except** statement. The **else** block contains a code that will be executed if NO exceptions occured. The syntax to use the else statement with the try-except statement is given below.

Method-3 try - except exception - else	Example

try:
Run this code to capture an exception
except Exception1:
Run this code if this exception occurs
except Exception2:
Run this code if this exception occurs
else:
Run this code if NO exception satisfied
	try:
	    a = int(input("Enter numerator: "))
	    b = int(input("Enter denominator: "))
	    c = a/b
	    print("a/b=",c)
	except ZeroDivisionError:
	    print("Can't divide with a zero")
	except NameError:
	    print("Name is not found")
	else:
	    print("Executed with NO exceptions")
	Output:
	Enter numerator: 9
	Enter denominator: 3
	a/b= 3.0
	Executed with NO exceptions

Method-4: except statement with exception variable:

We can use the **exception variable** with the **except** statement. It is used with **as** keyword. This variable object will return the cause of the exception.

Method-4: except with exception variable	Example

try:
Run this code to capture an exception
except Exception as variable:
Run this code if an exception occurs variable refers to Exception
	try:
	    a = int(input("Enter numerator: "))
	    b = int(input("Enter denominator: "))
	    c = a/b
	    print("a/b=",c)
	except ZeroDivisionError as z:
	    print("Can't divide with a zero")
	    print(z)

CIT Students Only

	Output: Enter numerator: 7 Enter denominator: **0** **Can't divide with a zero** **division by zero**

CIT Students Only

Applications of Built-in Exceptions

1. **ZeroDivisionError:** This exception is raised when you attempt to divide a number by zero.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero!")
```

Output: `Error: Division by zero!`

2. **ValueError:** This exception is raised when a built-in operation or function receives an argument of the correct type but an inappropriate value.

```
try:
    num = int("abc")
except ValueError:
    print("Error: Invalid value!")
```

Output: `Error: Invalid value!`

3. **TypeError:** This exception is raised when an operation or function is applied to an object of an inappropriate type.

```
try:
    result = 10 + "5"
except TypeError:
    print("Error: Incompatible types!")
```

Output: `Error: Incompatible types!`

4. **FileNotFoundError:** This exception is raised when a file or directory is requested but cannot be found.

```
try:
    file = open("nonexistent_file.txt", "r")
except FileNotFoundError:
    print("Error: File not found!")
```

Output: `Error: File not found!`

5. **IndexError:** This exception is raised when you attempt to access a list or sequence with an invalid index.

```
try:
    my_list = [1, 2, 3]
    print(my_list[5])
except IndexError:
    print("Error: Index out of range!")
```

Output: `Error: Index out of range!`

6. **KeyError:** This exception is raised when you try to access a dictionary with a key that doesn't exist.

```
try:
    my_dict = {"name": "Sneha", "age": 20}
    print(my_dict["gender"])
except KeyError:
    print("Error: Key not found!")
```

Output: `Error: Key not found!`

CIT Students Only

CIT Students Only

How to Raise an Exception in Python?

Raising exceptions

We can force an exception to be raised using the raise keyword in Python. It is useful to perform a specific task by raising an exception.

Syntax:

```
raise exception_name:
```

For example, if the user inputs incorrect data then we can raise a ValueError exception and suggest correct data.

Application: Raise exception

```
try:
    age=int(input("Enter age: "))
    if age<21:
        raise ValueError
    else:
        print("You are eligible to vote.")
except ValueError:
    print("You must be 21 or older to vote.")
```

Output:
Enter age: 20
You must be 21 or older to vote.

Application: Raise exception with Message

```
try:
    score=int(input("Enter score: "))
    if score<0:
        raise ValueError("Score must be a positive number")
    else:
        print("You scored",score)
except ValueError as e:
    print(e)
```

Output:
Enter score: -2
Score must be a positive number

CIT Students Only

How to create, raise and handle user-defined exceptions in Python? Illustrate with a program.

In Python, you can create, raise, and handle **user-defined exceptions** by creating a **custom exception class** that **inherits from the base Exception class**. This allows us to define our own exception types and raise them when needed.

Syntax:

```
# CREATE User-Defined Exception
class user_exception (Exception):
   pass
   or
   methods…

# RAISE User-Defined Exception
try:
   condition
   raise user_exception("message")

# HANDLE User-Defined Exception
except user_exception as variable:
   statements
```

Application-1: User-Defined Exception "VoteNotEligible"

```
# CREATE User-Defined Exception class
class VoteNotEligible(Exception):
    pass

# RAISE User-Defined Exception
try:
    age = int(input("Enter age: "))
    if (age<21):
        raise VoteNotEligible("You must be 21 or older!")
# HANDLE User-Defined Exception
except VoteNotEligible as e:
    print("Error:", str(e))
```

Output:
Enter age: 20
Error: You must be 21 or older!

- In this example, we define (CREATE) a custom exception class called VoteNotEligible. It inherits from the base Exception class.
- To RAISE and HANDLE the exception, we use a try-except block.
- The age variable reads a number and RAISEs a VoteNotEligible exception if age<21. it is caught by the except block, and the error message is printed.

You can also define additional methods and attributes in your custom exception class to provide more information or behavior specific to your needs.
Note that it's generally good practice to create custom exception classes when you need to differentiate between different types of errors or provide more meaningful error messages for specific scenarios.

Application-2: User-Defined Exception "ErrorInCode"

```
# CREATE User-Defined Exception class
class ErrorInCode(Exception):
    def __init__(self,data):
        self.data = data
    def __str__(self):
        return repr(self.data) #Returns printable string of given object

# RAISE User-Defined Exception
try:
    raise ErrorInCode(561)

# HANDLE User-Defined Exception
except ErrorInCode as ecode:
    print("Received Error#", str(ecode))
```

Output:
Received Error# 561

CIT Students Only

CIT Students Only

Write a single except block that handles multiple exceptions and also all kinds of exceptions?

In Python, you can use a single except block to handle multiple exceptions by specifying multiple exception types inside parentheses.

Syntax:

```
try:
   # Code that may raise exceptions
except (ExceptionType1, ExceptionType2, ...):
   # Code to handle the exceptions
```

Application: single except block with multiple exceptions as a tuple

```
try:
    name = input("Enter name: ")
    num = int(input("Enter number: "))
    print(name+num)
except (TypeError, ValueError) as e:
    if type(e) is TypeError:  # both values must be same data type
        print("TypeError:",e)
    elif type(e) is ValueError: # string input can't convert to int
        print("ValueError:",e)
except (Exception):
    print("No exceptions caught")
```

Output1:
Enter name: A
Enter number: 10
TypeError: can only concatenate str (not "int") to str

Output2:
Enter name: A
Enter number: B
ValueError: invalid literal for int() with base 10: 'B'

Explanation:

- The try block contains the code that might raise exceptions.
- The except block consisted of 2 exceptions as a tuple
- If any of the specified exceptions occur, the corresponding except block is executed.
- If none of the specified exceptions match, the control goes to the last except block, which can handle any other exceptions that are not explicitly listed.

CIT Students Only

Implement multiple exception blocks in Python exception handling. Justify with an example.

Certainly! In Python, we can use multiple except blocks to handle different types of exceptions individually. This allows us to specify different error-handling routines for each specific exception.

Here's an example program that demonstrates the usage of multiple exception blocks:

```
try:
    num1 = int(input("Enter the numerator: "))
    num2 = int(input("Enter the denominator: "))
    result = num1 / num2
    print("Result:", result)

except ValueError:
    print("Invalid input. Please enter a valid integer.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except Exception as e:
    print("An error occurred:", str(e))
finally:
    print("Program execution completed.")
```

In the above program, we have a **try** block that attempts to divide two numbers provided by the user. Let's see what happens with different inputs:

1. If the user enters a non-integer value (e.g., "**abc**") for either num1 or num2, a **ValueError** exception is raised. The program will catch the exception in the first except block and display the "**Invalid input. Please enter a valid integer.**" message.
2. If the user enters a **zero value** for num2, a **ZeroDivisionError** exception is raised. The program will catch the exception in the second except block and display the "**Error: Division by zero is not allowed.**" message.
3. If **any other unexpected exception occurs**, the program will **catch it in the third except block**. It will display the specific error message using str(e), which shows the exception details.

Regardless of the exceptions outcome, the **finally** block will always be executed, printing the "**Program execution completed.**" message.

Hence, The multiple exception blocks allow us to handle different types of errors separately, providing more specific and tailored error messages to the user.

CIT Students Only

Section-2: File Operations

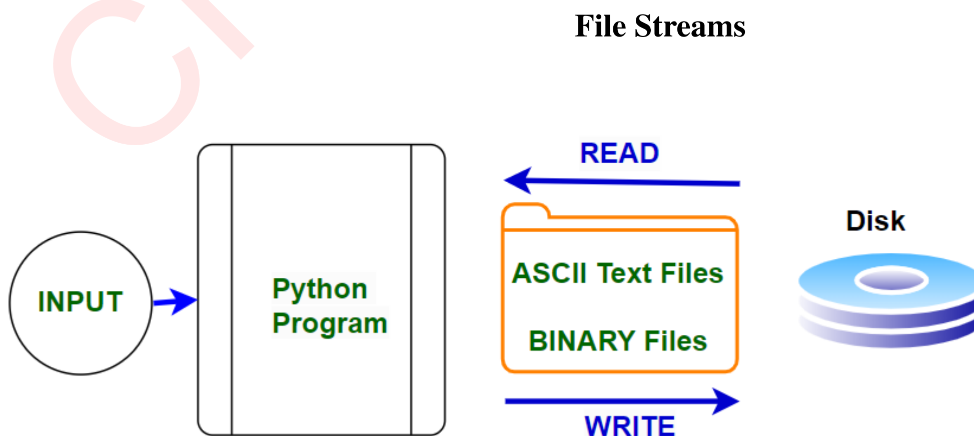
What is a File?

> **A file is a named location on the disk that stores information.**
> **The information in files is stored in a sequence of bytes.**
> **The files store information permanently in non-volatile memory.**
> **We can retrieve information from files as and when we need it.**

Python can handle Two types of file streams.

1. **Text Files** I/O stream**:** The user can create TEXT Files easily using file handling in Python. It stores information in the form of ASCII characters internally and when the file is opened, the content is readable by humans. It can be created by any text editor with a **.txt** or **.rtf** (rich text), or **.json** extension. Since text files are simple, they can be edited by any text editor like Microsoft Word, Notepad, Apple Text edit, etc.
2. **Binary Files** I/O stream**:** The binary files store data as bytes (8 bits) in the form of **0** s or **1** s. It stores information as blocks or bytes in the form of and it is saved with **.bin, .obj, .exe** extensions. Therefore it takes **less space**. The Audio files, Video files, Image files, and PDF files are also a few examples of binary files. Since it is stored in a binary number system format, it is **faster to access** and **more secure** than a text file.

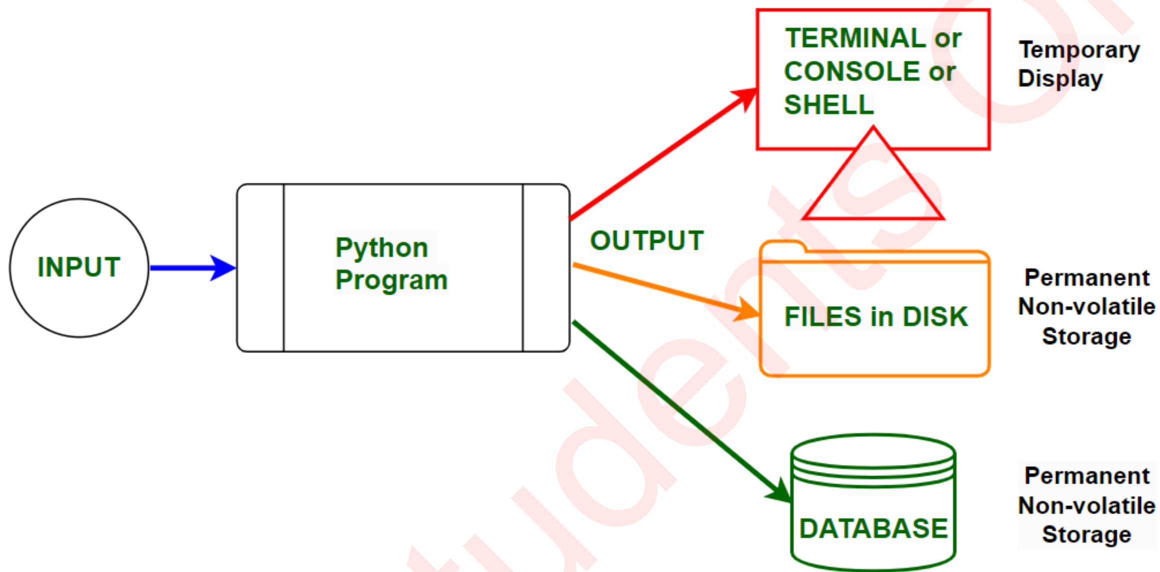
File Streams



What is the Need for Files and file handling?

File handling allows us to ***Preserve the data permanently on the disk for future usage***. This is why we need file handling.

For example, using file storage is a basic necessity to work with large datasets in the fields such as machine learning or data analytics.



CIT Students Only

What is File Handling?

Performing a list of operations on files is called File Handling in Python. Following are the commonly used file-handling operations in Python.

- Opening a file, open()
- Performing some operations in files
  - Reading from a file, read(), realine(), readlines()
  - Writing into a file, write(), writelines()
  - Positioning inside a file, or tell(), seek()
- Closing a file, and close(), with
- Deleting a file, os.remove()

Explain the Python 'open()' built-in function.

open() Method:

Before we can read or write a file in the OS, we need to open that file first. The built-in open () function helps to open a file for reading or writing.

The open() function creates a file object or file handler. This file object is used to call other file methods to manage files.

Syntax:

```
file_object = open(file_name, access mode)
```

Full Syntax:

```
file_object = open(file_name, mode=”r”, buffering=num, encoding=None, errors=None,
newline=None, closed=True
```

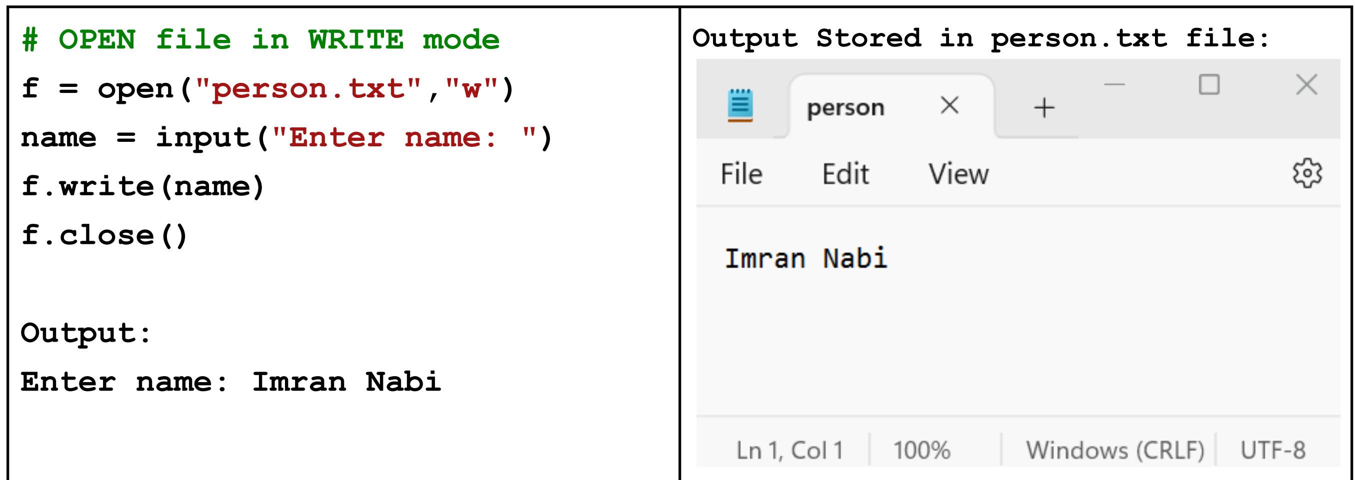
CIT Students Only

Parameter	Description
file_name	A string value that contains the Name of the file to access
mode	**File Access Modes** ● Text file modes: **r, r+, w, w+, a, a+** ● Binary file modes: **rb, rb+, wb, wb+, ab, ab+**
buffering	Positive integer value to set buffer size Text file: **>= 1** buffer size Bin file: **>= 0** buffer size Default size: 4096 - 8192 bytes
encoding	Used to encode or decode a files Used only in Text mode Default value depends on OS [Windows: CP1252, Linux: utf-8]
errors	Refers to how encoding and decoding errors are to be handled Cannot use in Bin mode Values: **strict, ignore, replace,** etc
newline	\n, \r, \t Default None uses \n. It depends on type of file such as .txt or .csv
closed	Boolean value. True if the file is already closed. False if the file is still open.

Access Modes	Description
Text File Modes	
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, it creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

CIT Students Only

a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
x	Creates a new file ONLY if the file Doesn't exist. It returns an error if the file is already there.
Binary File Modes	
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
rb+	Opens a file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, it creates a new file for writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.



CIT Students Only

Code	Output

`# OPEN file to READ and display`
`# Method-1`
`f = open("person.txt","r")`
`print(f.read())`
`f.close()`
	Output read from file & displayed in Terminal:
	
	Imran Nabi

```
# OPEN file to READ and display
# Method-2
f = open("person.txt", mode="r", buffering=10, encoding="utf-8")
if f:
    print(f.read())
    print("File opened")
```

Output read from file & displayed in Terminal:

Imran Nabi
File opened

File attribute variables

- **name**
- **mode**
- **closed** - boolean value (True for closed file; False for Not Closed file)
- **encoding** - the name of encoding (utf-8 for Linux; CP1252 for Windows)

Syntax:

```
file_object.attribute_name
```

```
# Example: File Attributes
f = open("person.txt", mode="r", buffering=10, encoding="utf-8")
print("File attributes are: ")
print("Name: ",f.name)
print("Mode: ",f.mode)
print("Closed status: ",f.closed)
```

CIT Students Only

```
print("Encoding typ: ",f.encoding)
f.close()
print("Closed status: ",f.closed)
```

Output:
File attributes are:
Name: person.txt
Mode: r
Closed status: False
Encoding typ: utf-8
Closed status: True

Explain file write methods write() and writelines() with examples.

In Python, the **write()** and **writelines()** methods are used to write data to a file. They are both file write methods, but they have different behaviors and use cases. Let me explain each of them with examples:

1. **write() method:**

The write() method is used to write a single string or a sequence of characters to a file.

- Writes the given content from the beginning of the file or
- Overwrites the existing file content if the file already exists.

Syntax:

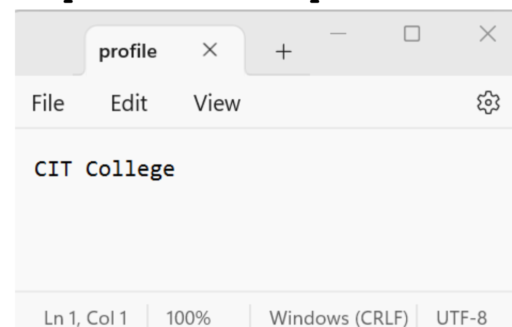
```
file_object = write ( content )
```

file_object: a file handler that represents the file to write;
content: is the string or characters that we want to write to file.

Example:
Assume the file name is "profile.txt" and
Assume the content “CIT College” to be written to the file

```
# OPEN file in WRITE mode
f = open("profile.txt","w")
f.write("CIT College")
f.close()
""" This code will create a file named "profile.txt"
(if it doesn't exist) or overwrite the existing
content with the string "CIT College". """
```

Output Stored in profile.txt file:



2. writelines() method:

The writelines() method is used to write many strings to a file. The writelines() function

- takes an iterable of strings as input (such as a list, tuple, or set) and
- writes each string as a separate line in the file.

Syntax:

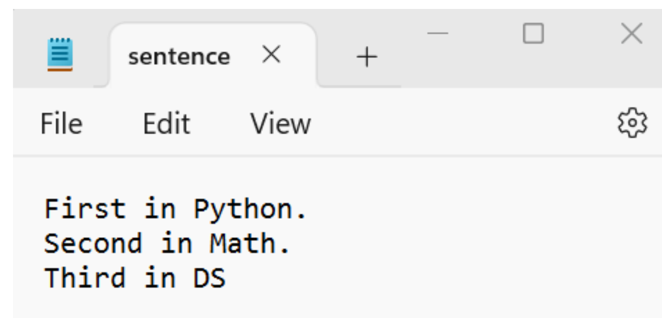
file_object = writelines (iterable)

file_object: a file handler that represents the file to write;
iterable: contains several strings to be written to the file.

Example:
Assume the file name is "sentencesprofile.txt" and
Assume the content “CIT College” to be written to the file

```
# Example: Write many lines into a
file using writelines()
fname = "sentences.txt"
ranks = ["First in Python.\n", "Second
in Math.\n", "Third in DS\n"]

# Open the file in write mode
fobject = open(fname, "w")
# Write the ranks to the file
fobject.writelines(ranks)
# Close the file
fobject.close()
```



Explanation:
This program creates a “**sentences.txt**" file (if it doesn't exist) or overwrites the existing content with each sentence from the **ranks** list written on separate lines.

The **writelines()** method does not automatically add line breaks ("**\n**") between the strings. We must explicitly add them in our strings while writing to file..

The **close()** method after writing is used to save and release file resources.

This is the basic usage of the write() and writelines() methods in Python for writing data to files.

CIT Students Only

CIT Students Only

Explain file read methods read(), readline(), and readlines() with examples.

In Python, the file methods **read(), readline(), and readlines()** are used to read data from a file.

1. **read():** This method reads the entire contents of a file and returns it as a single string.

Syntax:

```
file_object.read(size)
```

Example:

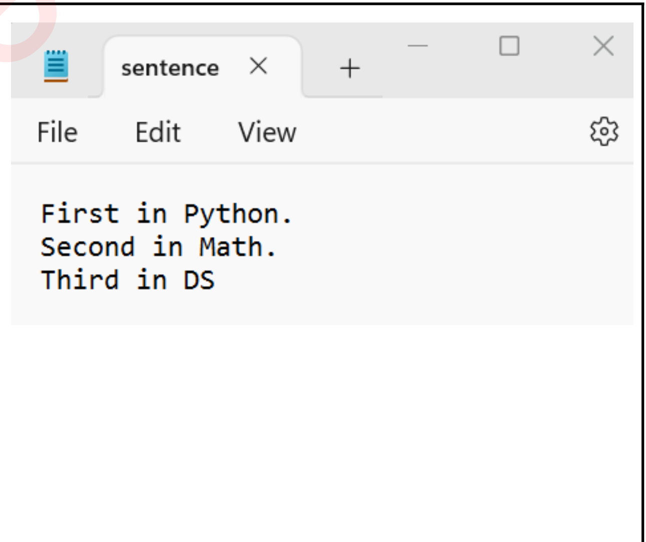
```
# OPEN file to READ and display
f = open("sentences.txt","r")
print(f.read())
f.close()
```

Output:

First in Python.

Second in Math.

Third in DS



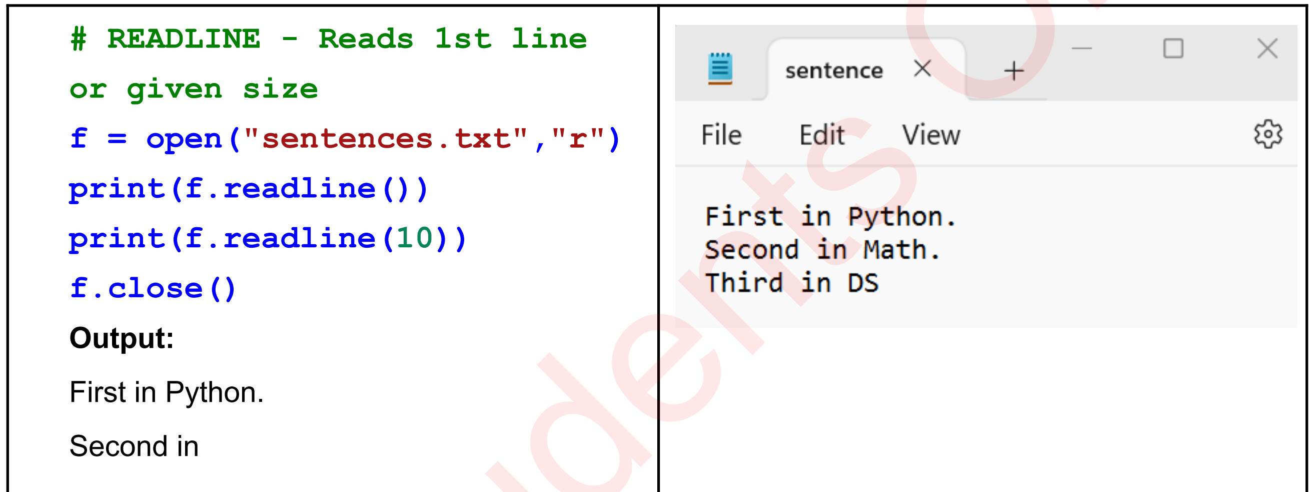
2. **readline():** This method reads a single line from a file and returns it as a string. It moves the file pointer to the next line after reading.

Syntax:

```
file_object.readline(size)
```

size: (optional) An integer value of string size to read. By default, size = -1 which returns an entire string.

Example:



```
# READLINE - Reads 1st line
or given size
f = open("sentences.txt","r")
print(f.readline())
print(f.readline(10))
f.close()
```

Output:

First in Python.

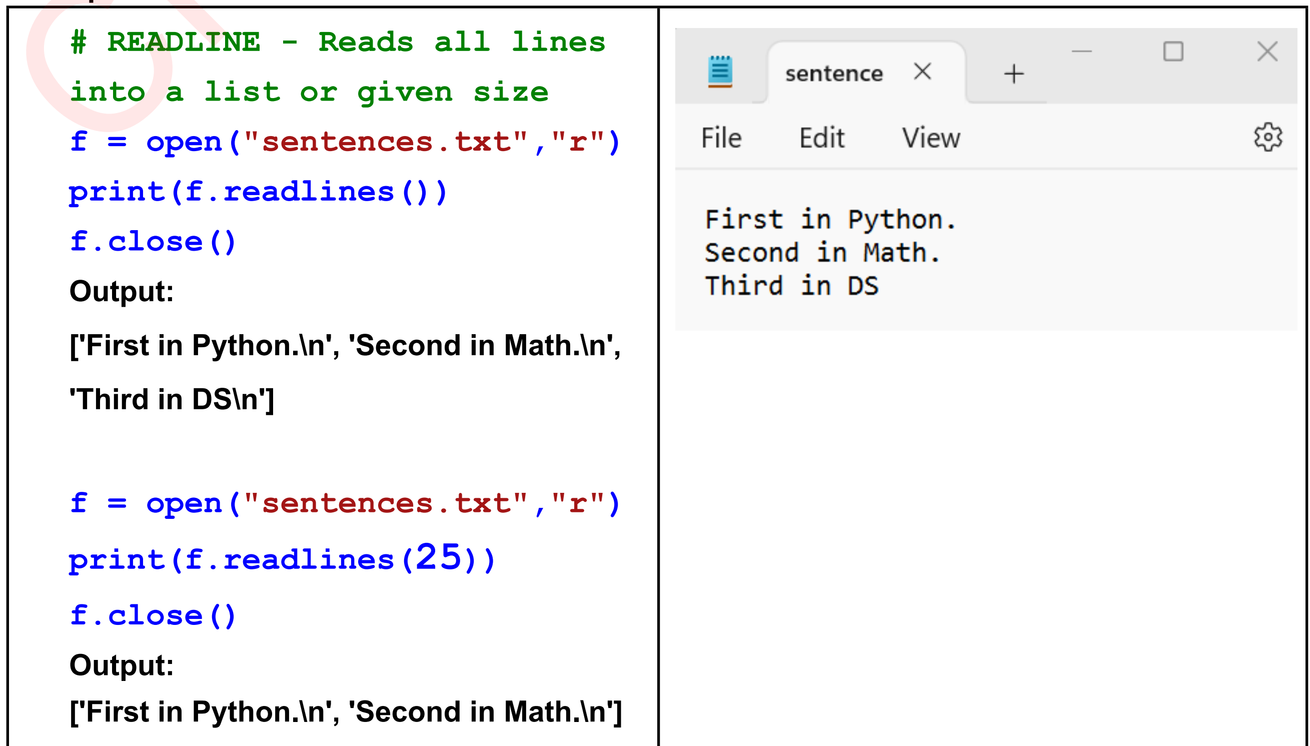
Second in

3. **readlines():** This method reads all the lines of a file and returns them as a list of strings. Each line is stored as a separate element in the list.

Syntax:

```
file_object.readlines(size)
```

Example:



```
# READLINE - Reads all lines
into a list or given size
f = open("sentences.txt","r")
print(f.readlines())
f.close()
```

Output:

['First in Python.\n', 'Second in Math.\n', 'Third in DS\n']

```
f = open("sentences.txt","r")
print(f.readlines(25))
f.close()
```

Output:

['First in Python.\n', 'Second in Math.\n']

CIT Students Only

CIT Students Only

What is close() method? Explain different ways to close a file in Python.

Method-1: close a file using the close() method

When we **open()** a file, Python allocates memory. After completion of the program, the file must be closed to release the memory. We use the **close()** method to close the opened file and release the system resources.

Syntax:

```
file_object.close()
```

Example:

```
# using close() statement
f = open("test.txt")
data = f.read()
f.close()
```

Method-2: close a file using "with" keyword (No need to use close() method)

Alternatively, we can use the "**with**" statement to **automatically handle the closing of the file**.
No need to use a close() statement.

Syntax:

```
with open(file_name, mode) as file_object
```

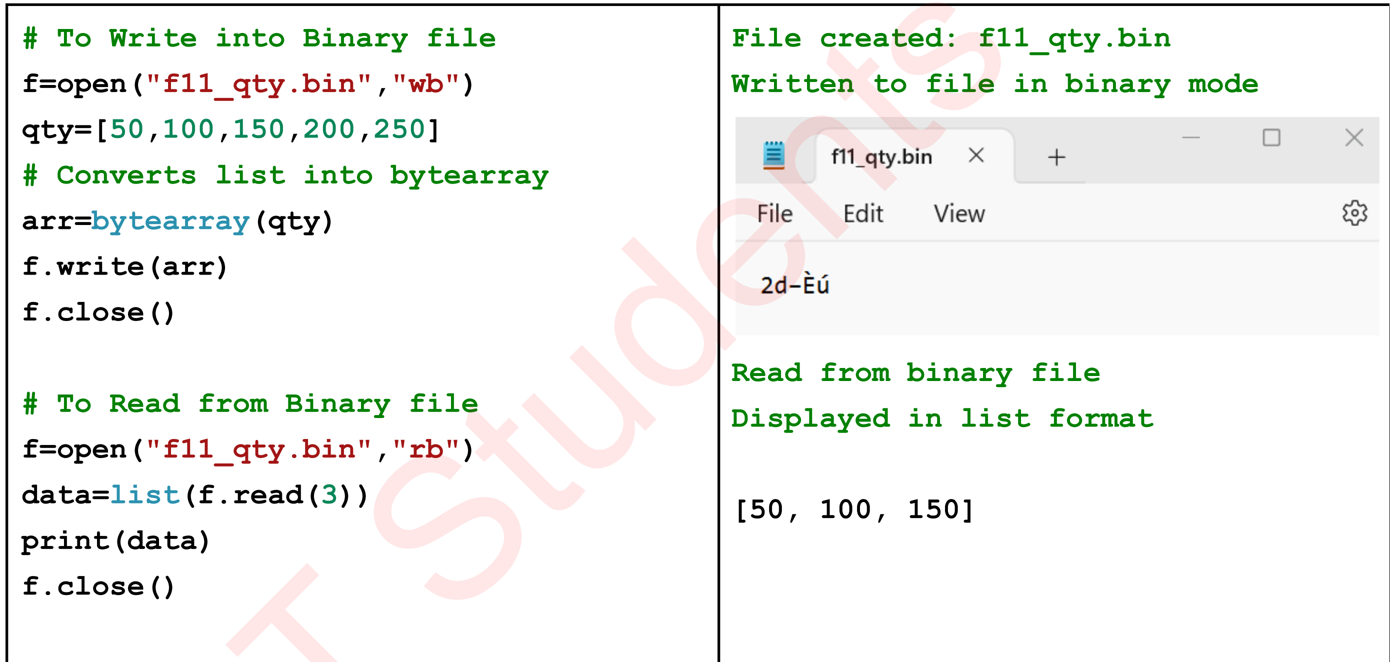
Example:

```
# using with statement
with open('test.txt', 'w+') as f:
    f.write('CIT Python!')
```

CIT Students Only

Write and Read Binary Files in Python

Application: Write a list of numbers into Binary file and Read those numbers from Binary file



```
# To Write into Binary file
f=open("f11_qty.bin","wb")
qty=[50,100,150,200,250]
# Converts list into bytearray
arr=bytearray(qty)
f.write(arr)
f.close()

# To Read from Binary file
f=open("f11_qty.bin","rb")
data=list(f.read(3))
print(data)
f.close()
```

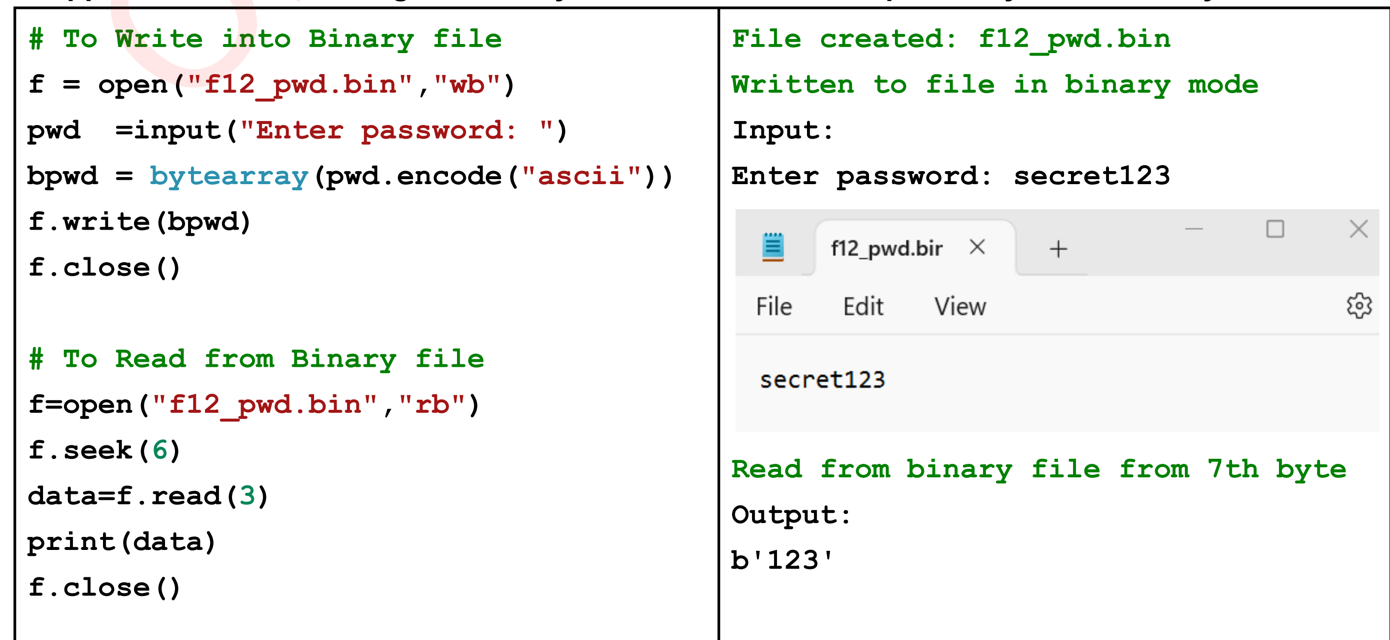
```
File created: f11_qty.bin
Written to file in binary mode
```



```
Read from binary file
Displayed in list format

[50, 100, 150]
```

Application: Write a string into Binary file and Read it from specific byte from Binary file



```
# To Write into Binary file
f = open("f12_pwd.bin","wb")
pwd  =input("Enter password: ")
bpwd = bytearray(pwd.encode("ascii"))
f.write(bpwd)
f.close()

# To Read from Binary file
f=open("f12_pwd.bin","rb")
f.seek(6)
data=f.read(3)
print(data)
f.close()
```

```
File created: f12_pwd.bin
Written to file in binary mode
Input:
Enter password: secret123
```



```
Read from binary file from 7th byte
Output:
b'123'
```

Application: Create QR code in .png binary mode that can be scanned from mobile

First install the qrcode module
C:/> pip install qrcode

<table>
<tr>
<td>

```
import qrcode
#Generate QR Code
img=qrcode.make('Sr Software Engineer,
                leadertain.com, India')
img.save('job.png')
```

</td>
<td> </td>
</tr>
</table>

CIT Students Only

CIT Students Only

Manipulate a file pointer in Python using tell() and seek()? Explain with a sample program

The file operations read/write are done at the cursor position. We can **get** and **move** the **current position of the cursor** in a file.

Python uses two methods **tell()** and **seek()** to position the cursor at a specific location in a file.

1. **tell()**
2. **seek()**

tell() - This function returns the current position of the cursor in the file. **Syntax:** tell( )
seek() - This function is used to set or change the current cursor position within the file. The seek() allows us to access various parts of an opened file according to our requirements. This enables us to perform **read-write operations anywhere in a file**. **Syntax:** seek(offset, from_where) **offset:** (Required) indicates a number of characters (or bytes) to move the cursor and sets the current cursor position in a file. **from_where:** (Optional) starting point from where the cursor position is measured. If this is not given, then Python uses the current position within a file. ● **0** from the beginning of the file ● **1** from the current cursor position in the file (file to be in Binary mode) ● **2** from the end of the file (file to be in Binary mode)

When we open a file in read mode, the cursor position will be at 0. We use seek() to change that position.

Application: Illustrate tell() and seek() methods in Python.

```
# Open the file in read mode
f = open("f5.txt", "r+")

position = f.tell() # Get the current position in the file
print("Current position:", position)

f.seek(4)  # Move 4 bytes from the beginning of the file
print("New position:", f.tell())
```

CIT Students Only

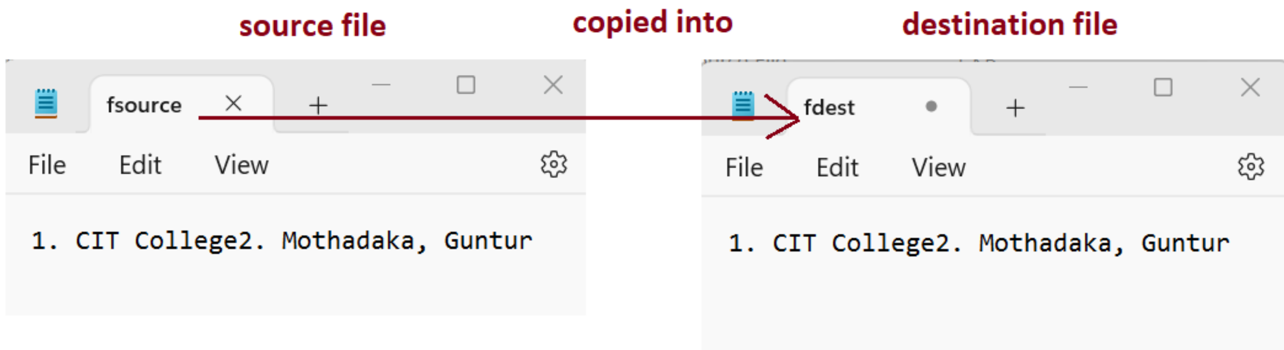
```
data = f.readline()
print("Data from the new position:", data)
f.seek(0, 2)  # Move 0 bytes from the end of the file
print("New position:", f.tell())
data = f.read()
print("Data from the end of the file:", data)
f.close()
```

Output: Current position: 0 New position: 4 Data from the new position: Engineering College New position: 47 Data from the end of the file:	**f5.txt** CIT Engineering College Python Programming

Application:Program to copy data from one file into another file

```
#Program to copy data from one file into another file
f1 = open ("fsource.txt", "w"); #w opens file for both Read & write
f1.write("1. CIT College")
f1.write("2. Mothadaka, Guntur")
f1.close()
f1 = open ("fsource.txt", "r") #r opens file for Read
data = f1.read()
f2 = open ("fdest.txt", "w") #w opens file for write
f2.write(data)
f1.close()
f2.close()
```

Result:



CIT Students Only

How to READ Config files in Python

Config files help create the initial settings for any project, they help avoid hardcoded data within a program. Config files are used to store key-value pairs or some configurable information such as Username, Password, IP address, Port number, Domain name, etc.

Such information is best stored in Config files than in Application Programs. This is because Config files are easier to update than Application programs.

A Python application program can use **configparser** module and **ConfigParser()** method to create and read the config files. The Python config files have the extension **.ini.**
The config files have several [sections]. Each section will have several settings as key=value pairs.

Example config file: f8_lead.ini

```
[Login]
weblink = leadertain.com
username = master
password = pwd12345

[Log]
logfilepath = c:\log\
logfilename = f8_lead.log
loglevel = Debug
format = (message)

[Database]
host = localhost
port = 8080

[API]
key = A1234567890
```

To read a config file from a Python program, we need to
- Import module
  - **import configparser**
- Create an object
  - **config = configparser.ConfigParser()**
- Config **obj.read(file_name)** - reads given config file
- Config **obj.get(section, key)** - reads & returns the specified string value from config file
- Config **obj.getint(section, key)** - reads & returns the specified int value from config file

CIT Students Only

Application: Read a config file from Python program

```
import configparser

def read_config(file_path):
    config = configparser.ConfigParser()
    config.read(file_path)

    if 'Login' in config:
      # Accessing configuration values
      link = config.get('Login',"weblink")
      id   = config.get('Login','username')
      pwd  = config.get('Login','password')
    else:
      print('Login section is not found.')

    if 'Database' in config:
    # Accessing values from different sections
      db_host = config.get('Database', 'host')
      db_port = config.getint('Database', 'port')
    else:
      print('Database section is not found.')

    if 'API' in config:
      api_key = config.get('API', 'key')
    else:
      print('API section is not found.')

    # Process/print the configuration values
    print(f'Web Link: {link}')
    print(f'Name: {id}')
    print(f'Password: {pwd}')

    print(f"Database Host: {db_host}")
```

```
    print(f"Database Port: {db_port}")

    print(f"API Key: {api_key}")

# Provide the path to your configuration file
path = 'f8_lead.ini'
read_config(path)

"""
Output:
Web Link: leadertain.com
Name: master
Password: pwd12345
Database Host: localhost
Database Port: 8080
API Key: A1234567890
"""
```

****
[OPTIONAL / AWARENESS ONLY] Start
****

How to CREATE Config files in Python [OPTIONAL / AWARENESS ONLY]

To create a config file from a Python program, we need to

- Import module
  - **import configparser**
- Create an object
  - **config = configparser.ConfigParser()**
- Config **obj.set(section, key, value)** - sets a new section and settings with key=value pair
- Config **obj[section]={key1 : value1, key2:value2, ...}** - also adds a new section and settings with key=value pairs
- Config `obj["Section"]["Key"]="Value"` - changes a setting (value of the given key)
- Config `obj["Section"].update({"key":"value"})` - adds a new setting (key=value pair)

CIT Students Only

CIT Students Only

Application: Program to CREATE a config file with several sections and key=value pair settings

```
import configparser

# CREATE OBJECT
config = configparser.ConfigParser()

# ADD SECTION
config.add_section("Login")
# ADD SETTINGS TO SECTION
config.set("Login", "weblink", "leadertain.com")
config.set("Login", "userName", "master")
config.set("Login", "password", "pwd12345")

# ADD NEW SECTIONS AND SETTINGS
config["Log"]={
    "LogFilePath":"c:\\log\\",
    "LogFileName" : "f8_lead.log",
    "LogLevel" : "Info"
    }
config["Database"]={
    "host":"localhost",
    "port" : 8080
    }
config["API"]={
    "key":"A1234567890"
    }

# SAVE CONFIG FILE
with open("f8_lead.ini", 'w') as configObj:
    config.write(configObj)
    configObj.flush()
    configObj.close()
```

CIT Students Only

```
print("Config file 'f8_lead.ini' created")

# PRINT FILE CONTENT
with open("f8_lead.ini", "r") as f:
    settings = f.read()
    print("Content of the config file are:\n")
    print(settings)

# CHANGE CONFIG FILE
# UPDATE A FIELD VALUE
config["Log"]["LogLevel"]="Debug"
# ADD A NEW FIELD UNDER A SECTION
config["Log"].update({"format":"(message)"})

# SAVE THE SETTINGS TO THE FILE
with open("f8_lead.ini","w") as f:
    config.write(f)

# PRINT FILE CONTENT AFTER CHANGE
with open("f8_lead.ini", "r") as f:
    settings = f.read()
    print("Content of the config file after change are:\n")
    print(settings)
```

```
"""
Output:
Config file 'f8_lead.ini' created
Content of the config file are:
[Login]
weblink = leadertain.com
username = master
password = pwd12345
```

```
[Log]
logfilepath = c:\log\
logfilename = f8_lead.log
loglevel = Info
[Database]
host = localhost
port = 8080
[API]
key = A1234567890
Content of the config file after change are:

[Login]
weblink = leadertain.com
username = master
password = pwd12345
[Log]
logfilepath = c:\log\
logfilename = f8_lead.log
loglevel = Debug
format = (message)
[Database]
host = localhost
port = 8080
[API]
key = A1234567890
"""
```

****

[OPTIONAL / AWARENESS ONLY] End

****

CIT Students Only

CIT Students Only

Writing log files in Python

Logging:
Logging is the process of writing information into log files. Log files capture useful information of several events at Operating System, Software Application, or Networking.

Purpose of Logging:
The log files are helpful for

- Information gathering
- Troubleshooting errors
- System audit
- Application bottlenecks
- Performance statistics
- Monitoring policy violations

Logging Module:
Python provides "**logging**" module as a part of its standard library, so we can quickly add logging to our application program.

```
import logging
```

Now, we can use "logger" to log messages. There are 5 security levels of logging and each level logs in increased severity or detail. The defined levels, in order of increasing severity, are the following:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

By default, the logger logs messages starting from WARNING, ERROR and CRITICAL; it doesn't log for DEBUG and INFO.

Application: Logging to Terminal/Console

```
import logging
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

Output:

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

CIT Students Only

basicConfig Loging:

We can use the basicConfig(**kwargs) method to configure the logging.

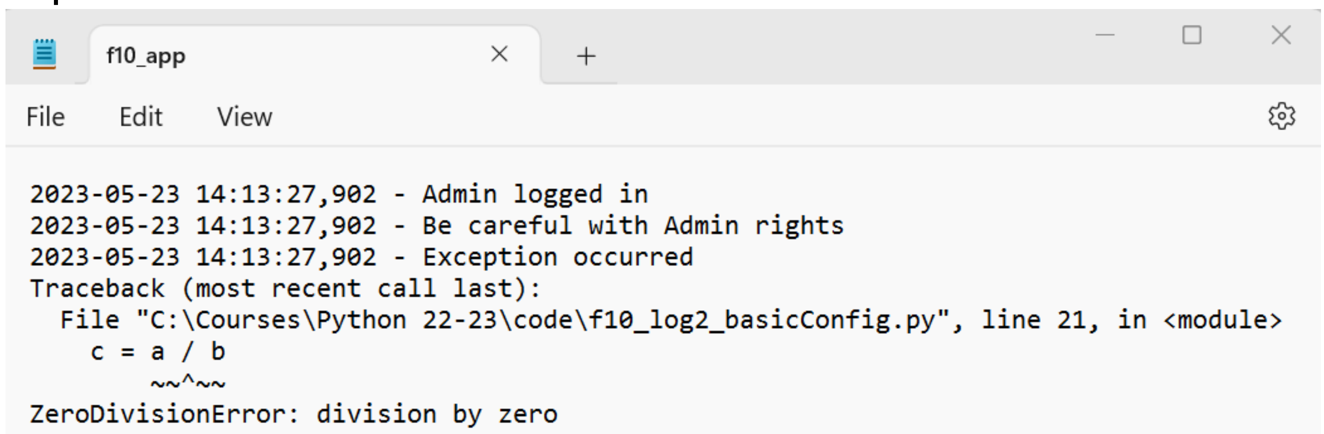
- level: specified severity level
- filename: specifies the file
- filemode: the file is opened in this mode. The default is a (append)
- format: format of the log message.

Application: Logging using basicConfig to a file

```
import logging
#Logging saved in a file from INFO and up
logging.basicConfig(level=logging.INFO, filename='f10_app.log',
filemode='w', format='%(asctime)s - %(message)s')

logging.info("Admin logged in")
logging.warning('Be careful with Admin rights')
#Logging exception data
a = 7
b = 0
try:
  c = a / b
except Exception as e:
  logging.error("Exception occurred", exc_info=True)
  # exc_info=True gives details of an error. Without this, no errors are
shown
```

Output:




```
2023-05-23 14:13:27,902 - Admin logged in
2023-05-23 14:13:27,902 - Be careful with Admin rights
2023-05-23 14:13:27,902 - Exception occurred
Traceback (most recent call last):
  File "C:\Courses\Python 22-23\code\f10_log2_basicConfig.py", line 21, in <module>
    c = a / b
        ~~^~~
ZeroDivisionError: division by zero
```

CIT Students Only

List and explain any three clean–up actions supported by Python.

Python exceptions provide a mechanism for handling and recovering from errors or exceptional situations in a program. They allow you to separate the normal flow of code from error-handling code. Here are three common clean-up actions supported by Python exceptions:

1. Resource Release:

Even in the presence of exceptions, we must release the external resources such as **files, network connections, or database connections** properly.

Exceptions can occur during the usage of these resources, and it's essential to clean up after them. Python's exception handling allows us to use the "**finally**" block to ensure that resources are released regardless of whether an exception occurred.

Example:

```
try:
    f = open("test.txt","r")
    # Perform file operations
except FileNotFoundError:
    print("File not found.")
finally:
    f.close()  # File is closed even if an exception occurs
```

Output:

`File not found.` (If test.txt is not available)

If the file exists, then the file gets automatically closed after the execution of **finally** block.

The code in the "**finally**" block is ALWAYS get executed regardless of whether an exception occurred or not. It provides a convenient way to release resources and perform clean-up actions.

2. Automatic File Cleanup:

- When working with files, it's important to ensure that they are cleaned up after their usage, even if an exception occurs.
- When we open a file using the open() method, we must use the close() method to close the file, If not, the Python Garbage Collector will clean up the memory allocated for the file. But depending on Python Garbage Colletor is risky because it cleans only after the exit of the whole program.
- However, when we open a file using the "with" keyword, then it closes the file automatically after its use. Here, we do not need to use the close() method.

Syntax:

```
with open(filename, mode) as file_object:
  # Code to manipulate file
```

Note: The "**with**" statement guarantees that the **file** will be closed and cleaned up, making it a safe and convenient way to handle temporary files. Here, we do not need to manually close the file.

3. Restoring State:
Exceptions can also be used to restore the state of an object or system to a known state before an error occurred. This is commonly achieved using the **try...except...else** construct.

- The **try** block contains the code that may raise an exception,
- the **except** block handles the exception, and
- the **else** block executes if no exception occurs.

This allows us to restore the state or take appropriate actions before continuing the program.

Syntax:

```
try:
    # Perform operations that may raise an exception
except ValueError:
    # Handle the specific exception
    # Restore the object or system to a known state
else:
    # No exception occurred
    # Continue with the program
```

CIT Students Only

CIT Students Only

File processing operations

Python **os** module provides methods to perform file-processing operations, such as renaming and deleting files. We must first module to call the methods for file processing.

```
import os      # We must first import os module to call the methods for file processing.
```

1. os.rename()

We can use rename() method to rename a file. It takes two arguments, the current filename and the new filename.

Syntax:

```
os.rename(current_file_name, new_file_name)
```

Example: users1.txt will be renamed to users2.txt

```
import os
os.rename('users1.txt','users2.txt')
```

2. os.mkdir()

The mkdir() method creates a new directory. It takes one argument as a directory name

Syntax:

```
os.mkdir(directory name)
```

Example: A new directory "project" will be created in the current directory

```
import os
os.mkdir('project') # Creates python named directory
```

3. os.rmdir()

The rmdir() method removes the specified directory. It takes one argument as a directory name that needs to be removed.

Syntax:

```
os.rmdir(directory name)
```

Example:

```
import os
os.rmdir('project') # removes the directory named "project"
```

CIT Students Only

4. **os.chdir()**

The chdir() method changes the directory. It takes one argument as a directory name to which we want to change.

Syntax:

```
os.chdir(newdir)
```

Example:

```
import os
os.chdir('D:\>') # change directory to D drive
```

5. **os.remove()**

The remove() method removes the given file. It takes one argument as a filename that we want to remove.
Syntax:
Example: import os

Syntax:

```
os.remove(filename)
```

Example:

```
import os
os.remove('users2,txt') # removes the file names users.txt
```

6. **os.getcwd()**

The getcwd() method gives current working directory. It takes zero arguments.

Syntax:

```
os.getcwd()
```

Example:

```
import os
os.getcwd( )      # Gives current working directory
```

CIT Students Only

Reference

Exception Class Hierarchy

A class hierarchy consists of a number of exceptions distributed across different base class types. In any programming application, errors occur when something unexpected happens. The errors can result from improper arithmetic calculations, a full or near-full memory space, formatting errors, or invalid file references that raise an error.

The series of errors raised are considered as exceptions since they're non-fatal and allow the execution of the program to continue. It also enables you to throw an exception to explicitly catch or rescue the exception raised.

The exception hierarchy is determined by the inheritance structure of various exception classes. All raised exceptions are instances of a class derived from the baseException class. A try:except clause, using a particular class, can handle all exceptions derived from that class.

The hierarchy of the major built-in exceptions include:

BaseException

- Exception
  - **ArithmeticError**
  - **FloatingPointError**
  - **OverflowError**
  - **ZeroDivisionError**
  - **AssertionError**

Most classes use keyword exception in the baseException and Exception in its parent classes. Subclasses use the word error. A Python program inherits a series of abcError classes.

CIT Students Only

BaseException Class

The BaseException class is the base class of all built-in exceptions in Python program. A raised Python exception should be inherited from other classes. The BaseException class takes in a tuple of arguments passed during the creation of a new instance of the class. In most cases, a single string value is passed as an exception which indicates the specific error message.

The class also includes the with_traceback(tb) method to explicitly pass the 'tb' argument to the traceback information.

Exception Class

The *Exception* class has a variety of subclasses that handle the majority of errors in Python. Some of these subclasses include:

1. **ArithmeticError:** This acts as a base class for various arithmetic errors, such as when you're trying to divide a number by zero.
2. **AssertionError:** This is an error generated when a call to [assert] statement can't be completed or it fails.
3. **BufferError:** In Python, applications have access to a low-level memory stream in the form of buffers. If the buffer memory fails, an error is generated: BufferError.
4. **EOFError:** This type of error is raised when an input () reaches the end of a file without finding the specified file or any data.
5. **ImportError:** To perform advanced functions or work with complex programs in Python, you have to import modules. If the import statement fails, an importError will be raised.
6. **LookupError:** Just like in arithmeticError, a LookupError acts as the base class from which is inherited by other subclasses. If an improper call is made to the subclass then a LookupError is raised.
7. **MemoryError:** If a Python program runs out of memory or there is no enough memory stream to run the application, a MemoryError is raised.
8. **NameError:** This is an error that occurs when you try to call an unknown name or use an identifier with an invalid name.
9. **OSError:** An error raised due to the failure of an operating system
10. **ValueError:** It occurs if a method receives parameters of the correct type, but the actual value passed is invalid due to some reasons.

CIT Students Only

KeyboardInterrupt This occurs when the user presses Ctrl+C key combination. This creates an interrupt to the executing script.

GeneratorExit The generator is a particular iterator that simplifies the iteration process with constantly changing values. It allows you to use the yield statement inside the generator code block. This type of exception allows Python to generate a new value when a call to the next() function is made. When the generator.close() method is invoked, and a generatorExit instance is raised.

SystemExit When you call *sys.exit()* method, a systemExit exception is raised. The sys.exit() function call closes the current executing script and then closes the Python shell. Since this is an exception, you catch the exception before the script shuts down and exit by responding to the systemExit exception immediately.