Unit 5 PPS Sem(1)

By Mohd Yasir Sheikh

Q) File handling operations

File handling is a crucial aspect of programming, and it involves performing various operations on files, such as reading from them, writing to them, and manipulating their content.

Opening a File:
To open a file, you use the open() function. It takes two arguments - the file name and the mode in which you want to open the file (read, write, or append).
For example:

```
file = open("example.txt", "r")
```

Reading from a File:
You can use various methods to read from a file, such as read(), readline(), or readlines().

```
content = file.read()  # Reads the entire content of the file
line = file.readline()  # Reads a single line
lines = file.readlines()  # Reads all lines and returns a list
```

Writing to a File:
To write to a file, open it in write mode ("w") or append mode ("a") and use the write() method.

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

Closing a File:
It's important to close a file after performing operations on it using the close() method.

```
file.close()
```

Closing a File:

It's important to close a file after performing operations on it using the close() method.

```
with open("example.txt", "r") as file:
    content = file.read()
```

Checking if a File Exists:

You can use the os.path module to check if a file exists before opening it.

```
import os

if os.path.exists("example.txt"):
    with open("example.txt", "r") as file:
        content = file.read()
else:
    print("File does not exist.")
```

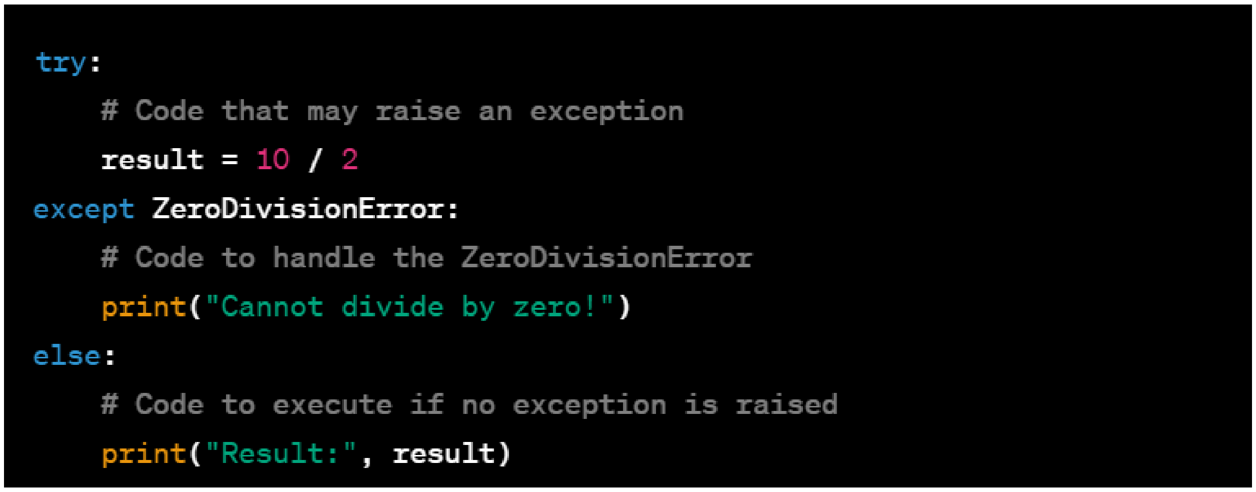
Q) Different types of exceptions

1. **SyntaxError:** Raised when there is a syntax error in the code.
2. **IndentationError:** Raised when there is an incorrect indentation.
3. **NameError:** Raised when a local or global name is not found.
4. **TypeError:** Raised when an operation or function is applied to an object of an inappropriate type.
5. **ValueError:** Raised when a built-in operation or function receives an argument with the right type but an inappropriate value.

6. **FileNotFoundError:** Raised when a file or directory is requested but cannot be found.
7. **IndexError:** Raised when a sequence subscript is out of range.
8. **KeyError:** Raised when a dictionary key is not found.
9. **ZeroDivisionError:** Raised when the second operand of a division or modulo operation is zero.
10. **AttributeError:** Raised when an attribute reference or assignment fails.

Q) Try with else clause

In Python, the **else** clause in a **try...except...else** block is executed if the code in the **try** block doesn't raise any exceptions. Here's an example of using the **else** clause:



```python
try:
    # Code that may raise an exception
    result = 10 / 2
except ZeroDivisionError:
    # Code to handle the ZeroDivisionError
    print("Cannot divide by zero!")
else:
    # Code to execute if no exception is raised
    print("Result:", result)
```

In this example, if the division in the **try** block doesn't result in a **ZeroDivisionError**, the code in the **else** block will be executed.

You can use the **else** clause to include code that should run only if no exceptions occur, making it useful for scenarios where you want to handle exceptions in one block and execute additional code when no exceptions are raised.

Q) Different file access modes

Read Mode ('r'):

- Open the file for reading.
- File pointer is positioned at the beginning of the file.
- Raises a FileNotFoundError if the file doesn't exist.

Write Mode ('w'):

- Open the file for writing.
- If the file exists, it truncates it to zero length. If not, it creates a new file.
- File pointer is positioned at the beginning of the file.

Append Mode ('a'):

- Open the file for writing.
- If the file exists, it appends data to the end. If not, it creates a new file.
- File pointer is positioned at the end of the file.

Binary Read Mode ('rb'):

- Opens the file for reading in binary mode.
- Similar to 'r' mode but specifically for binary files.

Binary Write Mode ('wb'):

- Opens the file for writing in binary mode.
- Similar to 'w' mode but specifically for binary files.

Binary Append Mode ('ab'):

- Opens the file for appending in binary mode.
- Similar to 'a' mode but specifically for binary files.

Read and Write Mode ('r+'):

- Opens the file for both reading and writing.
- File pointer is positioned at the beginning of the file.

Write and Read Mode ('w+'):

- Opens the file for both writing and reading.
- If the file exists, it truncates it to zero length. If not, it creates a new file.
- File pointer is positioned at the beginning of the file.

Append and Read Mode ('a+'):

- Opens the file for both appending and reading.
- If the file exists, it appends data to the end. If not, it creates a new file.
- File pointer is positioned at the end of the file.

Q) Steps involved in reading a file

Open the File:

- Use the open() function to open the file in read mode ('r').
- Specify the file path or name as an argument to open().

Read Content from the File:

- Choose an appropriate method to read the content from the file.
- Common methods include read(), readline(), or readlines().

Process the Content (Optional):

- Once the content is read, you can process it as needed.
- This may involve parsing lines, extracting information, or performing other operations.

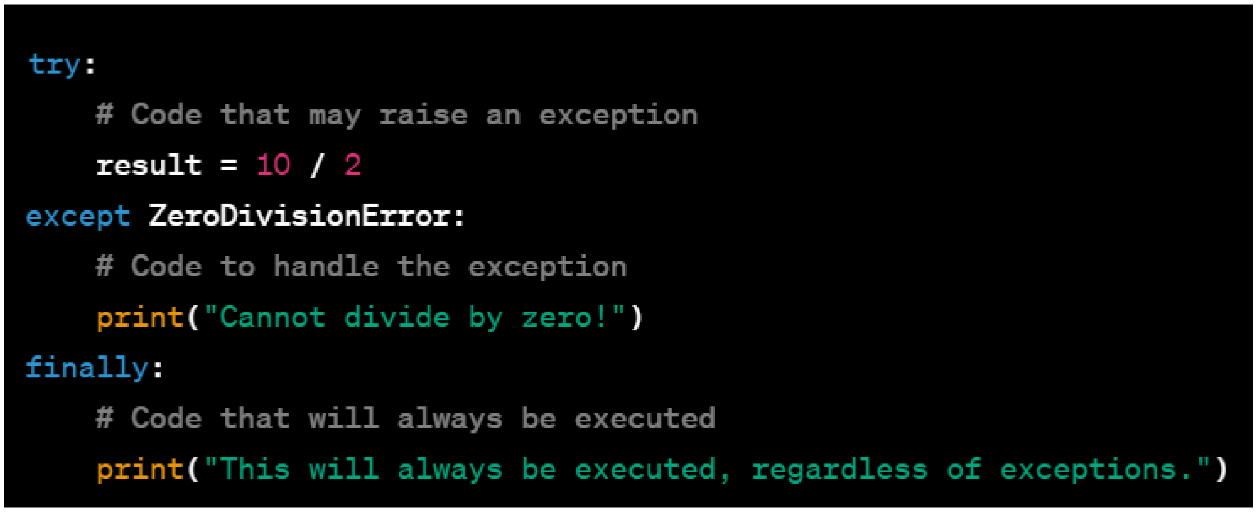
Close the File:

- Use the close() method to close the file when you are done reading.
- Closing the file is important to release system resources and ensure proper file handling.

Q) Working of finally keyword

The finally keyword in Python is used in conjunction with a **try** block and is followed by a block of code. The code within the **finally** block is guaranteed to be executed whether an exception is raised or not. This is useful for performing cleanup actions, such as closing files or network connections, regardless of whether an error occurred.

Here's a general structure of how **try**, **except**, and **finally** work together:



```
try:
    # Code that may raise an exception
    result = 10 / 2
except ZeroDivisionError:
    # Code to handle the exception
    print("Cannot divide by zero!")
finally:
    # Code that will always be executed
    print("This will always be executed, regardless of exceptions.")
```

- The try block contains code that may raise an exception (e.g., **dividing by zero**).
- The except block specifies what to do if a specific exception (**e.g., ZeroDivisionErro**r) occurs.
- The finally block contains code that will be executed no matter what, whether an exception occurred or not.

The **finally** block is commonly used for cleanup operations, ensuring that resources are released or certain actions are taken regardless of the outcome of the **try** block. This makes it useful for tasks like closing files, database connections, or releasing other system resources.

Q) Built-in exceptions in python(any 4)

SyntaxError:

- Raised when the Python interpreter encounters a syntax error in the code.
- Common causes include misspelt keywords, missing colons, or incorrect indentation.

TypeError:

- Raised when an operation or function is applied to an object of an inappropriate type.
- For example, trying to concatenate a string with an integer without conversion.

ValueError:

- Raised when a built-in operation or function receives an argument with the right type but an inappropriate value.
- For instance, attempting to convert a non-numeric string to an integer.

FileNotFoundError:

- Raised when an attempt to open a file or directory fails because the specified file or directory cannot be found.
- It often occurs when using the open() function to access a file that does not exist.

Q) Purpose of using try-except block.

The **try-except** block in Python is used to handle exceptions, which are unexpected events that can occur during the execution of a program. The purpose of using a **try-except** block is to gracefully handle these exceptions and prevent the program from crashing.

Here are the main reasons for using a **try-except** block:

Preventing Program Crashes:

- If an exception occurs within the try block, it would normally terminate the program and display an error message. Using try-except allows you to catch and handle exceptions, preventing the program from crashing.

Graceful Error Handling:

- Instead of letting the default error message be displayed to the user, you can provide custom error messages or alternative actions to take when a specific exception occurs. This makes error messages more user-friendly.

Isolating Problematic Code:

- You can wrap specific sections of code in a try block to isolate and handle exceptions that might occur within that block. This helps in pinpointing the source of errors and makes the code more maintainable.

Resource Cleanup:

- In scenarios where resources need to be released, such as closing files or network connections, the try-except block, along with the finally block, can be used to ensure that cleanup operations are performed even if an exception occurs.

Selective Exception Handling:

- Different except blocks can be used to handle different types of exceptions. This allows you to handle specific errors in a way that makes sense for your application while letting others propagate up the call stack.

Unit 5 PPS Sem(1)

By Mohd Yasir Sheikh