Date: 24/12/25

—-------------------------

To-Do:

- Exception Handling
- File Handling

—-------------------------

Exception Handling:

Python exception handling allows a program to handle unexpected events without crashing.

➔ Handling simple exception:

```
n=10
try:
        Res = n/0
except ZeroDivisionError:
        print("Cant be divided by zero")
```

➔ The try block contains code that may fail and except block catches the error, printing a a safe msg instead of stopping the program

➔ Difference between exception and error:

- **Error**- serious problem in the program logic that cannot be handled. Examples include syntax and memory errors. This stops the program from running.
- **Exception**- less severe problems that occur at runtime and can be managed using exception handling. Example- invalid input, missing files. This can be handled using runtime.

➔ Syntax and usage:

```
Try:
   #code
Except SomeException:
  #code
Else:
   #code
Finally:
  #code
```

Try: runs the risky code that might cause an error.
Except: catches and handles the error if one occurs.
Else: Executes only if no exception occurs in try.
Finally: runs regardless of what happens useful for cleanup tasks like closing files.

➔ Example:

```
try:
   n=0
   res = 100/n
expect ZeroDivisionError:
   print("You cant divide by zero!")
```

```
expect ValueError:
   print(“Enter a valid number!”)
else:
   print(“result  is”,res)
finally:
   print(“Execution complete.”)
```

➔ Try block attempts division, expect blocks catch specific warriors,else block executes only if no error , while finally block always runs, signaling end of execution.

➔ **<u>PYTHON CATCHING EXPECTIONS:</u>**

1. <u>Catching specific exceptions:</u>

   ```
   Try:
      x=int(“str”)  #this will give valueErroe since str cant be converted into integer
      inv=1/x    #since x has no value, this will cause ZeroDivisionError
   Except ValueError:
      print(“notvalid”)
   Except ZeroDivisionError:
      print(“zero value, cant solve”)
   ```

   In output a valueError occurs because”str” cannot be converted to an integer. If conversion had succeeded but x were 0, a ZeroDivisionError would have been caught instead.

2. <u>Catching multiple exceptions:</u>
   We can catch multiple exceptions in a single block if we need to handle them in the same way or we can separate them in different types of exceptions that require different handling.

   ```
   a=[“10”,”twenty”,30]   #mixed int andn str
   Try:
      Total = int(a[0]) + int(a[1])   #twenty cannot be converted into int
   Except (ValueError, TypeError) as e:
       print(“error”,e)
   Except IndexError:
      print(“Index out of range.”)
   ```

3. <u>Catch all handlers and their risks:</u>
   Sometimes we may use a catchall handler to catch any expectation , but it can hide useful debugging infooooo.

```
Try:
    Res = “100”/20
Expect :
    print(“something went wrnggggg!!!”)
```

➔ Raise an exception:

```
def set(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print(f"Age set to {age}")

try:
    set(-5)
except ValueError as e:
    print(e)
```

➔ Custom Exceptions:

```
class AgeError(Exception):
    pass

def set(age):
    if age < 0:
        raise AgeError("Age cannot be negative.")
    print(f"Age set to {age}")

try:
    set(-5)
except AgeError as e:
    print(e)
```

➔ Advantages:
- Improved reliability
- Separation of concernns
- Cleaner code
- Helpful debugging

➔ Disadvantages:
- Performance overhead
- Added complexity
- Security risks

<u>File Handling:</u>

- File handling refers to the process of performing operations on a file, such as creating, opening, reading, writing and closing it through a programming interface.

- Why do we need file handling:
  - To store data permanently, even after the program ends.
  - To process large files efficiently without using much memory.
  - To automate tasks like reading configs or saving outputs.
- <u>Opening a file:</u>
  To open a file, we can use open() function, which requires file-path and mode as arguments.

  ```
  file= open('filename.txt','mode')
  ```

  filename.txt= name of file to be opened.
  mode=mode in which you want to open the file(read,write,append)

  Example:
  ```
  f = open("geek.txt", "r")
  print(f)
  ```

- <u>Closing a file:</u>
  The file.close() method closes the file and releases the system resources. If the file was opened in write or append mode, closing ensures that all changes are properly saved.

- <u>Checking file properties:</u>
  f.name, f.mode, f.closed(returns in true or false)

- <u>Reading a file:</u>
  Reading a file can be achieved by file.read() which reads the entire content of the file.

  ```
  content=file.read()
  print(content)
  file.close()
  ```

- <u>Writing a file:</u>
  Writing to a file is doe using the mode "w". This creates a new file if it doesn't exist, or overwrites the existing file if it does.

  ```
  f = open("abc.txt","w")
  f.write("hello, im the first here!")
  f.close()
  ```

➔ Using “with” statement

“With” statement automatically handles opening and closing of the file.

```
with open(“abc.txt”,”r” ) as f:
    Content = file.read()
    print(content)
```