File Handling and Exception Handling

Reading Material

PW SKILLS

1. File Handling

- Any programming language must include file handling in order for programs to be able to read from and write to system files. With Python, files handling is easy because of the built-in methods that let you open, read, write, and close files, among other tasks.

Opening and Closing Files

- An integral component of Python programming is file handling. It gives you access to read, write, and alter system files. It's essential to know how to open and close files correctly in order to protect data integrity and avert problems like memory leaks and data corruption.

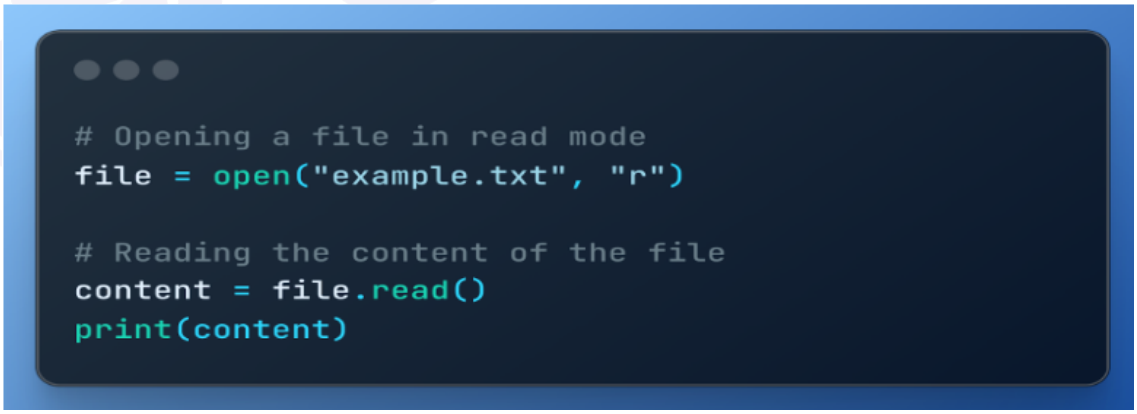
Opening Files

In Python, the built-in open() method is used to open a file. This function requires two main inputs:

Filename: The file you wish to open..
Mode: The desired mode of file opening (e.g., read, write, append).

Common File Modes:

“r”: Read mode. Makes the file readable. The default mode is this one.
“w”: Write mode. Allows you to write to the file. The file will be overwritten if it already exists. A new file will be generated if the existing one doesn't exist.
“a”: Append mode. Allows the file to be appended. There will be more data added to the file's end. A new file will be generated if the existing one doesn't exist.
“r+”: Read and write mode. Opens the file for both reading and writing.



```
# Opening a file in read mode
file = open("example.txt", "r")

# Reading the content of the file
content = file.read()
print(content)
```

Closing Files
It is imperative that you use the close() method to close a file once you have completed working with it. When a file is closed, any modifications made to it are preserved and any related resources are released in a correct manner.

Example:

```
# Closing the file
file.close()
```

Not closing a file can cause a number of issues, like data loss or memory leaks if the program crashes before the file is closed.

Using 'with' Statement for Automatic File Handling.

Python's with statement offers a more practical method of handling files. Even in the event of an exception, this method immediately closes the file after the block of code inside the with statement is run.

Example:

```
# Using with statement to open and read a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# File is automatically closed after the block is executed
```

Reading from Files

- A basic part of handling files in Python is reading from files. It enables you to get data from files for additional processing or examination. Python offers several file reading methods, each appropriate for a distinct set of applications.

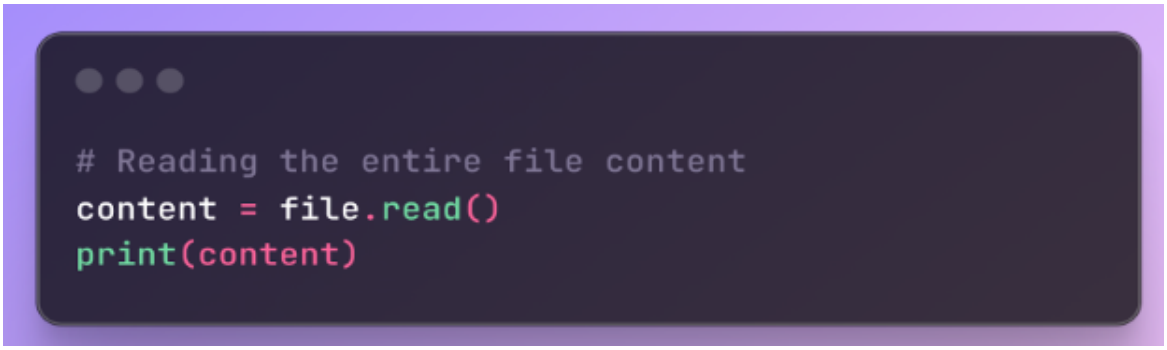
Opening a File for Reading

- Using the 'open()' method, you must open a file in read mode before you may read from it. The "r" parameter specifies the read mode.

```
# Opening a file in read mode
file = open("example.txt", "r")
```

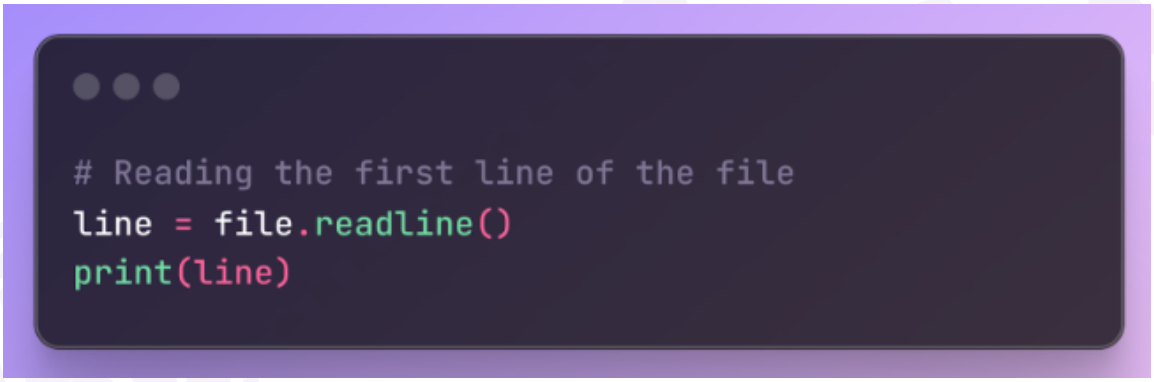
Reading Methods
Once the file is open, there are multiple ways to read the contents in Python:

- **'read()' Method:** reads a single string containing all of the file's content. When you need to process the entire file at once, this is helpful.



```
# Reading the entire file content
content = file.read()
print(content)
```

- **'readline()' method:** Reads one line from the file at a time. This is useful when you need to process the file line by line.



```
# Reading the first line of the file
line = file.readline()
print(line)
```

- **- 'readlines()' method:** Creates a list by reading every line in the file. One line from the file is represented by each element in the list.

```
# Reading all lines into a list
lines = file.readlines()
for line in lines:
    print(line.strip())  # .strip() removes the
newline characters
```

Using a Loop to Read a File
Iterating through the file object in a loop is another effective method of reading a file line by line. This approach uses less memory and is effective with big files.

```python
# Reading file line by line using a loop
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Handling Different Files Types

Python's file reading functions may read a wide range of file types, including text, CSV, JSON, and more. To read CSV files or JSON files, you can need additional libraries like csv or json.

Example: Reading a CSV File

```python
import csv

# Opening and reading a CSV file
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Example: Reading a JSON File

```python
import json

# Opening and reading a JSON file
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)
```

Handling File Not Found Errors

It's critical to handle potential problems, like the file not being found, when reading from a file. You can use a 'try-except' block to handle this.

```
try:
    with open("non_existent_file.txt", "r") as
file:
        content = file.read()
except FileNotFoundError:
    print("File not found. Please check the file
name or path.")
```

Writing to Files

Writing to files is an essential part of managing files. You can use it to generate reports, keep logs, and save data. Python provides a number of methods for writing data to files based on the particular needs of your project.

Opening a File for Writing

Using the 'open()' method, you must open a file in write mode before you can write to it. Depending on whether you wish to add new data or replace the current material, there are various writing styles.

Common File Modes for Writing

- **'w':** Write mode. Allows you to write to the file. The file will be overwritten if it already exists. In the event that it doesn't, a new file will be made.
- **'a':** Append mode. Open the file for appending. New data will be placed at the end of the file without altering the existing content. If the file doesn't exist, a new file will be generated.
- **'w+':** Write and read mode. Allows you to write and read in the file. In addition, this option overwrites any existing files.

Example: Opening a File in Write Mode

```
# Opening a file in write mode
file = open("output.txt", "w")
```

Writing Methods

Once the file is open, you can use the following methods to write data to it:

- **'write()' Method:** Adds a string to the document. You must include '\n' to start a new line because this approach does not automatically add a newline character.

```
# Writing a string to the file
file.write("Hello, World!\n")
```

- **'writelines()' Method:** Adds a string list to the file. You must insert \n at the end of each string because this approach does not add newline characters between the strings.

```
# Writing multiple lines to the file
lines = ["First line\n", "Second line\n", "Third line\n"]
file.writelines(lines)
```

Using 'with' Statement for Writing

Python lets you use the with statement for writing files, just like it does for reading. This makes sure the file closes on its own after writing, which is important to avoid data loss and to free up system resources.

```
# Writing to a file using with statement
with open("output.txt", "w") as file:
    file.write("This is a new file.\n")
    file.write("Writing multiple lines.\n")
```

Appending Data to a File

You can open an existing file in append mode ('a') to add new data without overwriting the existing content.

```
# Appending data to the file
with open("output.txt", "a") as file:
    file.write("Appending a new line.\n")
```

Handling Different Data Types

Writing to a file only allows you to write strings; other data types (such numbers and lists) must first be converted to strings.

```
# Writing integers and lists to a file
data = [1, 2, 3, 4, 5]
with open("output.txt", "w") as file:
    file.write("Writing integers: " + str(42) +
"\n")
    file.write("Writing a list: " + str(data) +
"\n")
```

Handling File Writing Errors

It is imperative to address any mistakes, such as disc space limitations or permission difficulties, that may arise during file writing.

```
try:
    with open("output.txt", "w") as file:
        file.write("Attempting to write to the
file.\n")
except IOError:
    print("An error occurred while writing to the
file.")
```

File Modes

Knowing the various writing modes available is crucial when using Python to work with files. These settings control the handling of already-existing data as well as how new data is written to the file. Making the right mode choice guarantees accurate and effective data management.

Common Writing Modes

- **Write Mode ('w'):** This mode allows you to write to a file. The file's content will be overwritten if it already exists. A new file will be generated if the existing one doesn't exist.

```
# Opening a file in write mode
with open("example.txt", "w") as file:
    file.write("This will overwrite any existing
content.\n")
```

- **Append Mode('a'):** In this mode, a file is opened for adding. It lets you append additional information to the end of the file without changing the content that's already there. A new file will be generated if the existing one doesn't exist.

```
# Opening a file in append mode
with open("example.txt", "a") as file:
    file.write("This will be added to the end of
the file.\n")
```

- **- Write and Read Mode('w+'):** This mode allows you to write and read in the same file. It overwrites the current content, just like 'w' mode, except it also lets you read from the file after writing.

```
# Opening a file in write and read mode
with open("example.txt", "w+") as file:
    file.write("Writing and reading in the same
file.\n")
    file.seek(0)  # Move the cursor back to the
beginning of the file
    content = file.read()
    print(content)
```

- **- Append and Read Mode('a+'):** A file can be opened in this mode for both reading and adding. Without changing the current content, you can append data to the end of the file and read from it.

```
# Opening a file in append and read mode
with open("example.txt", "a+") as file:
    file.write("Appending more data to the
file.\n")
    file.seek(0)  # Move the cursor back to the
beginning of the file
    content = file.read()
    print(content)
```

Handling Binary Files
You might have to work with binary files in addition to text files (e.g., pictures, executables). Python offers binary writing modes:

Write Binary Mode ('wb'): Enables the writing of a binary file. The file is overwritten if it already exists.

```
# Writing binary data
with open("example.bin", "wb") as file:
    file.write(b'\x00\xFF')
```

Append Binary Mode('ab'): Allows you to add files to a binary file. The file's end contains new data.Allows you to add files to a binary file. The file's end contains new data.

```
# Appending binary data
with open("example.bin", "ab") as file:
    file.write(b'\x01\x02')
```

Error Handling in Writing Modes
Errors might arise while working with different file modes, such as trying to write to a read-only file or having permission problems. 'Try-except' blocks are useful for handling these errors.

```
try:
    with open("example.txt", "w") as file:
        file.write("Attempting to write data.")
except IOError:
    print("An error occurred while writing to the
file.")
```

File Objects and Methods
File objects are important to Python file handling. You can read from and write to files, among other tasks, using the methods these classes expose. Effective file management requires an understanding of working with file objects and the procedures that go along with them.

File Objects
The 'open()' method in Python returns a file object when you use it to open a file. This object serves as a bridge between the file on disc and your Python code. You can work with the content of the file using a number of methods available to the file object.

```
# Creating a file object by opening a file
file = open("example.txt", "w")
```

Writing Methods for File Objects

- **'write()' method:** One string is written to the file using this method. '(n)' is not added automatically, therefore you have to include it directly if you want each string to end on a new line.

```
# Writing a single string to the file
file.write("Hello, World!\n")
```

- **'writelines()' Method:** Using this method, a list of strings is written to the file. The input for this method is an iterable (such as a list), unlike 'write()'. It adds newline characters to each string manually, just like 'write()' does not.

```
# Writing multiple lines to the file
lines = ["First line\n", "Second line\n", "Third
line\n"]
file.writelines(lines)
```

Closing the File Object

After writing to a file, you must close it. This guarantees the release of system resources and the correct saving of all data. The file object is closed using the 'close()' method.

```
# Closing the file
file.close()
```

After writing to a file, you must close it. This guarantees the release of system resources and the correct saving of all data. The file object is closed using the 'close()' method.

```
# Using with statement to handle file object
with open("example.txt", "w") as file:
    file.write("This will automatically close the
file.")
```

Working with CSV Files

Tabular data can be stored in a popular format called CSV (Comma-Separated Values) files. The csv module in Python offers built-in support for handling CSV files. This guide explains how to use Python to read from and write to CSV files, using real-world examples to help you grasp the ideas.

Introduction to CSV Files

Data is stored in a plain text file called a CSV file, which is separated by commas. A row of data is represented by each line in the file, and individual fields (columns) are separated by commas. Due to their simplicity, human-readable nature, and compatibility with a wide range of programs, including databases and spreadsheets, CSV files are frequently utilised.

Writing to CSV Files

In Python, the csv module must be used in order to write data to a CSV file. You may create and work with CSV files with the aid of the classes and methods in this module.

- **Writing a Single Row:** Using the 'writerow()' function of the 'csv.writer' object to write one row at a time is the most straightforward technique of writing to a CSV file.

test.py

```
import csv

# Writing a single row to a CSV file
with open("output.csv", "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "City"])
#Writing the header
    writer.writerow(["Alice", 30, "New York"])
#Writing a data row
```

A list of values is written as a single row in the CSV file using the 'writerow()' method. A common problem on various platforms is the addition of extra blank lines between rows. This is ensured by the newline='' argument.

- **Writing Multiple Rows:** The 'writerows()' method can be used to write numerous rows at once. A list of lists that each inner list represents a row can be entered into this procedure.

```python
import csv

# Writing a single row to a CSV file
with open("output.csv", "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "City"])
#Writing the header
    writer.writerow(["Alice", 30, "New York"])
#Writing a data row
```

When you need to write more than one row of data, the 'writerows()' method comes in handy because it lets you pass all the data at once.

- **Writing to CSV Files with Dictionaries**

Using dictionaries is an additional method of writing data to CSV files. You can write dictionaries with the keys representing the column headers using the 'DictWriter' class in the 'csv' module.

```python
import csv

# Writing data to a CSV file using DictWriter
with open("output_dict.csv", "w", newline='') as
file:
    fieldnames = ["Name", "Age", "City"]
    writer = csv.DictWriter(file,
fieldnames=fieldnames)

    writer.writeheader()  # Writing the header
    writer.writerow({"Name": "Alice", "Age": 30,
"City": "New York"})  # Writing a data row
    writer.writerow({"Name": "Bob", "Age": 25,
"City": "Chicago"})
```

When you need to write more than one row of data, the 'writerows()' method comes in handy because it lets you pass all the data at once.

- **Writing to CSV Files with Dictionaries**

Sometimes, certain characters in your data, such commas, could cause issues for the CSV structure. These situations are handled automatically by the 'csv' module, which encloses fields containing special characters in quotes.

test.py

```
import csv

# Writing data with special characters to a CSV
file
with open("output_special.csv", "w", newline='')
as file:
    writer = csv.writer(file)
    writer.writerow(["Product", "Price"])
    writer.writerow(["Gadget, Model X", "19.99"])
# The comma in the product name is handled
automatically
```

To maintain the integrity of the CSV structure, the 'csv' module makes sure that fields containing commas or other special characters are appropriately quoted.

2. Exception Handling

Python's exception handling mechanism handles exceptional circumstances or runtime faults that interfere with a program's normal operation.

Programs may end suddenly when an error occurs if exception handling isn't used, which could result in data loss or other problems. Exception handling enables the software to gracefully handle errors.

Exception vs. Errors

- **Syntax:** The interpreter notices these at compile time. They arise from improper syntax and need to be corrected before running the code.
- **Exceptions**: These happen when Python encounters an unforeseen circumstance during runtime, like attempting to divide by zero or attempting to access an erroneous index within a list.

Try-Except Blocks

In Python, exceptions that may arise during program execution are handled with "try-except" blocks. These blocks give you the ability to "try" a block of code and "accept" possible errors, giving you a gracious manner to handle and respond to problems.

'Try-except' blocks are mostly used to keep programs from crashing in the event of a mistake, enabling them to either continue operating or handle errors in a way that is intuitive for the user.

Basic Structure of Try-Except Blocks

The basic structure of a 'try-except' block consists of:

- **try:** The block of code you want to test for errors.
- **except:** The block of code that will execute if an error occurs in the 'try' block.
- **else:** A block that executives if no exceptions were raised in the 'try'block.
- **finally:** A block that executes whether an exception occurred or not, often used for cleanup actions.

```
test.py

try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
```

Example: Handling a Division by Zero Error
An example of using a 'try-except' block to handle a 'ZeroDivisionError' is provided here.

```
test.py

try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

Handling Multiple Exceptions
Python enables the handling of multiple exceptions by including numerous 'except' clauses. Multiple exceptions can be managed in a single block by utilising a tuple.

```
test.py

try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not
allowed.")
except ValueError:
    print("Error: Invalid input, please enter a
number.")
```

The code within the `try` block may raise a `ZeroDivisionError` if there is an attempt to divide by zero, or a `ValueError` if the input is not a valid numerical value. The accompanying `except` blocks are responsible for capturing these specific exceptions and presenting a suitable error message to the user.

Using a General Exception

If you have uncertainty regarding the specific type of exception that might occur, you can employ a generic 'Exception' class to capture any occurrence.

test.py

```
try:
    result = 10 / int(input("Enter a number: "))
except Exception as e:
    print(f"An error occurred: {e}")
```

The `except Exception as e` block captures any exception that occurs and assigns it to the variable `e`. The error message is then shown using the `print` function.

Try-Except-Finally Block

The 'finally' block is executed unconditionally, irrespective of the occurrence of an exception, and is commonly employed for resource cleanup, such as shutting files.

test.py

```
try:
    file = open("example.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("Error: File not found.")
finally:
    file.close()
    print("File closed.")
```

A `FileNotFoundError` is raised if the file cannot be located, and the `finally` block makes sure that the file is closed whether or not an exception is raised.

Multiple Except Clauses

Multiple 'except' clauses in Python let you handle distinct kinds of exceptions on their own. This lets you respond or take precise actions according to the kind of mistake that happens.

Multiple 'except' clauses allow you to customize your error handling for various scenarios, improving the readability and robustness of your code.

Basic Structure of Multiple Except Clauses

- Several 'except' blocks can be added to a try block in order to handle various exceptions.
- The 'except' blocks are examined sequentially, with the execution of the first one that corresponds with the raised exception.

test.py

```
try:
    # Code that may raise exceptions
except ExceptionType1:
    # Handle ExceptionType1
except ExceptionType2:
    # Handle ExceptionType2
```

Example: Handling Multiple Exceptions
This is a simple example showing how to utilize numerous "except" clauses to handle distinct exceptions.

test.py

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not
allowed.")
except ValueError:
    print("Error: Invalid input, please enter a
valid number.")
```

The code tries to divide 10 by the number entered by the user. A 'ZeroDivisionError' is raised if the user inputs 0; this is handled by the relevant 'except'block. A 'ValueError' is produced if the user enters a non-numeric value. The second 'except' block handles this circumstance by displaying the relevant error message.

Catching Multiple Exceptions in One Block
Python additionally enables you to catch multiple exceptions with a tuple of exception types in a single 'except' block.

```python
# test.py
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ZeroDivisionError, ValueError) as e:
    print(f"Error: {e}")
```

The ‘except’ block in this example handles both ‘ZeroDivisionError’ and ‘ValueError’, simplifying the code while maintaining the provision of distinct error messages for each kind of exception.

Else Clause in Exception Handling

When handling exceptions in Python, the 'else' clause is an optional block that runs in the event that the 'try' block raises no exceptions. It offers a tidy method of dividing the code that must execute only if the 'try' block completes successfully.

The 'else' clause comes in handy when you wish to execute some code separately from the exception-handling code, solely in the event that the code inside the 'try' block is successful.

Basic Structure of Try-Except-Else

After the 'except' block(s), the 'else' clause executes solely in the event that the 'try' block fails to generate an exception.

```python
# test.py
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
else:
    # Code to execute if no exceptions were raised
```

Example: Using Else with Exception Handling

Here is a basic illustration of how to utilize the 'else' clause in exception handling:

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not
allowed.")
except ValueError:
    print("Error: Invalid input, please enter a
valid number.")
else:
    print(f"Result: {result}")
```

The ‘else’ block runs and outputs the result after the ‘try’ block tries to divide 10 by a user-supplied integer and no problems (like ‘ZeroDivisionError’ or ‘ValueError’) arise.
In the event that an exception does arise, the ‘else’ block is bypassed and the corresponding ‘except’ block takes care of it.

Finally Clause in Exception Handling
Whether or not an exception is raised, the 'finally' clause in Python is an optional block that comes after the 'try' and 'except' blocks. It is used for code that needs to run at all times, like resource cleanup (releasing locks, closing files, etc.).

The 'finally' clause's main goal is to guarantee that a specific code is always run, regardless of errors, which is essential for preserving the program's stability and integrity.

Basic Structure of Try-Except-Finally
Whether an exception is caught or not, the 'finally' block always executes after the 'try' and 'except' blocks.

```python
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
finally:
    # Code that always executes
```

Example: Using Finally for Resource Cleanup
'Finally' blocks are frequently used to make sure that resources, such files or network connections, are terminated correctly even in the event of a processing fault.

```python
try:
    file = open("example.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("Error: The file was not found.")
finally:
    file.close()
    print("File closed.")
```

The ‘except’ block handles a ‘FileNotFoundError’ when a file cannot be opened or read from after an effort to do so by the ‘try’ block. The ‘finally’ block makes sure that the file
is closed regardless of whether an exception occurs, which is essential for preventing resource leaks.

When to Use the Finally Clause

- **Resource Management:** When you need to ensure that resources, like files, database connections, or sockets, are released regardless of what occurs in the 'try' block, use 'finally'.

- **Cleanup Activities:** 'finally' clauses work well for cleanup tasks like recording final messages, committing or rolling back transactions, or deleting temporary files.

Example: Handling Multiple Exceptions with Finally
Imagine the following situation: You manage a number of errors, but you still need to make sure that cleanup code runs:

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not
allowed.")
except ValueError:
    print("Error: Invalid input, please enter a
valid number.")
finally:
    print("Execution completed, whether successful
or not.")
```

The ‘try’ block may raise either a ‘ZeroDivisionError’ or a ‘ValueError’, which are handled by their respective ‘except’ blocks. Regardless of the outcome, the ‘finally’ block executes, signaling that the program has completed its execution and ensuring that any necessary final steps are taken.