PANIMALAR ENGINEERING COLLEGE, CHENNAI

23ES1206 PYTHON PROGRAMMING – V UNIT- QUESTION BANK ANSWERS
ACADEMIC YEAR – 2024-2025 / SEMESTER-II

Prepared By
DR.G.UMARANI SRIKANTH
Professor
Dept of CSE, PEC

==========

PART-A

1.Define a file and give its advantages.

A **file** is a collection of data stored on a computer's storage device, such as a hard disk or SSD. It can store text, images, videos, or any other form of digital information.

Advantages of Using Files

1. **Data Storage** – Files store data permanently, unlike volatile memory (RAM).
2. **Easy Access** – Data can be read, written, and modified efficiently.
3. **Data Sharing** – Files allow easy data transfer between systems.
4. **Organization** – Files help manage large amounts of structured data.
5. **Backup & Recovery** – Important information can be saved and restored.
6. **Security** – Files can be protected with encryption and access controls.

2. Differentiate text file and binary file.

Feature	Text File	Binary File
Storage Format	Stores data as plain text (ASCII/Unicode).	Stores data in binary (0s and 1s).
Extension	`.txt`, `.csv`, `.log`, etc.	`.bin`, `.exe`, `.jpg`, etc.
Readability	Human-readable.	Not human-readable.
Size	Takes more space due to character encoding.	More compact and efficient.
Processing	Can be processed in text editors.	Requires special software to interpret.

3.write the difference between ftell () and fseek().

Function	Purpose	Usage
`ftell()`	Returns the **current file position** (in bytes) from the beginning of the file.	`position = file_object.tell()`
`fseek()`	Moves the file pointer to a **specific position** in the file.	`file_object.seek(offset, whence)`

4. What is command line argument? Give it purposes.

A **command-line argument** is an input value passed to a program when it is executed from the command prompt or terminal. These arguments are accessed in Python using the sys.argv list.

Purpose of Command-Line Arguments

1. **Dynamic Input** – Allows users to provide input without modifying code.
2. **Automation** – Enables scripting and batch processing.
3. **Flexibility** – Users can customize program behavior at runtime.
4. **Efficiency** – Reduces the need for user interaction in automated tasks.

5.Name the different modes used in file handling.

Mode	Description
`r`	Read mode (default), opens file for reading, error if file does not exist.
`w`	Write mode, creates a new file or overwrites existing content.
`a`	Append mode, adds data at the end of the file.
`r+`	Read and write mode, maintains existing content.
`w+`	Write and read mode, overwrites existing content.
`a+`	Append and read mode, retains existing content and allows appending.
`rb`	Read mode for binary files.
`wb`	Write mode for binary files.
`ab`	Append mode for binary files.

6.List down some inbuilt exceptions

Some Inbuilt Exceptions in Python

1. **TypeError** – Invalid operation on different data types.
2. **ValueError** – Incorrect value given to a function.
3. **IndexError** – Accessing an invalid index in a list or tuple.
4. **KeyError** – Accessing a non-existent key in a dictionary.
5. **ZeroDivisionError** – Division by zero is not allowed.
6. **FileNotFoundError** – File does not exist when trying to open it.

7. What are the distinctions between append and write modes?

Mode	Description	Effect on Existing File
`'w'` (write)	Opens a file for writing. **Creates** a new file or **overwrites** if it exists.	Deletes old content and writes new data.
`'a'` (append)	Opens a file for writing. **Creates** a new file or **appends** if it exists.	Keeps old content and adds new data.

8. What is exception chaining in python? provide an example.

Exception chaining occurs when one exception is raised while handling another. It helps in preserving the original exception context using **raise ... from ....**

Syntax:

```
raise NewException("Message") from OriginalException
```

Example:

```
try:
    x = int("abc")          # Causes ValueError
except ValueError as e:
    raise TypeError("Invalid type conversion") from e
```

✅ *Exception chaining helps in debugging by preserving the original error!*

9. How do you define a user-defined exception in Python? write a basic example.

A **user-defined exception** is created by inheriting from the built-in Exception class.

Syntax:

```
class MyException(Exception):
    pass
```

Example:

```
class AgeTooSmallError(Exception):
    pass
```

```
age = int(input("Enter our age: "))
if age < 18:
    raise AgeTooSmallError("Age must be at least 18")
else:
    print("Access granted!")
```

10. What is clean up actions in exception handling?

Clean-up actions ensure that resources (like files or memory) are properly released, even if an exception occurs. This is done using the finally block in Python.

Example:

```
try:
    file = open("data.txt", "r")
except FileNotFoundError:
    print("File not found!")
finally:
    print("Execution completed, cleaning up resources.")
```

✅ *finally ensures proper resource management and prevents leaks!* 🚀

Part -B / iv unit

1. How can you read and write data in a file using Python? provide source code examples for both operations.

Yes. Python provides built-in functions like open(), read(), readline(), readlines(), write(), writelines() and close() to handle file operations. The with statement ensures safe file handling by automatically closing the file, avoiding data loss or memory leaks.

Reading and Writing Data in a File Using Python

Python provides built-in functions to handle file operations, such as reading and writing data using open() with different modes (r, w, a, etc.).

1. Writing Data to a File ("w" mode)

The **w** mode opens the file for writing. If the file exists, it overwrites the content; otherwise, it creates a new file.

Example: Writing to a File

```
# Open the file example.txt in the same folder in write mode
file = open("example.txt", "w")
# Writing data to the file
file.write("Hello, this is a test file.\n")
file.write("Python file handling is easy!\n")
#write lines of text
lines =file.writelines(["Hello, this is a test file.\n" , "hello how are you\n"])
```

```
# Closing the file
file.close()
print("Data written successfully.")
```

output

```
Data written successfully.
```

Now open example.txt, the contents will be

```
Hello, this is a test file.
Python file handling is easy!
Hello,this is a test file.
hello how are you
```

2. Reading Data from a File ("r" mode)

The r mode opens a file for reading. If the file does not exist, it raises a FileNotFoundError.

Example: Reading from a File

```
# Open the file in read mode
file = open("example.txt", "r")  #opens example.txt in read mode
# Reading entire content
content = file.read()
print("File Content:\n", content)
# Closing the file
file.close()
```

output

```
File Content:
 Hello, this is a test file.
Python file handling is easy!
Hello,this is a test file.
hello how are you
```

3. Writing and Reading Together ("w+" mode)

This mode allows both writing and reading but overwrites existing content.

Example: Writing and Reading in the Same File

```
# Open file in write+read mode
file = open("example.txt", "w+")
# Writing to the file
file.write("This is a new content.\n")
file.write("Previous content is overwritten!\n")
file.write("welcome")
# Moving cursor to the beginning
file.seek(0)
# Reading the content
print("Updated File Content:\n",)
with open("example.txt", "r") as f:
```

```
    lines = f.readlines()
print(lines)
# Closing the file
file.close()
```

output

```
Updated File Content:
['This is a new content.\n', 'Previous content is overwritten!\n', 'welcome']
```

4. Appending Data (a mode)

The a mode adds data at the end of an existing file without deleting previous content.

Example: Appending Data to a File

```
# Open file in append mode
file = open("example.txt", "a")
# Adding more data
file.write("Appending new content.\n")
# Closing the file
file.close()
print("Data appended successfully.")
```

output

```
Data appended successfully.
```

5. Reading a File Line by Line

To read large files efficiently, use .readline() or .readlines().

Example: Reading Line by Line

```
# Open file in read mode
file = open("example.txt", "r")
# Reading file line by line
for i in file:
    print(i.strip())
# Closing the file
file.close()
```

output

```
Hello, this is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you
```

Key Points:

- Use w mode to write (overwrites existing content).
- Use r mode to read data.
- Use a mode to append data.
- Always close the file using file.close() or use with open() for automatic closure.
- Use seek(0) to reset the file pointer while reading after writing.

✅ *Proper file handling ensures data integrity and prevents memory leaks!*

@@

2. What are the ftell() and fseek() methods in file handling? Explain with suitable examples.

ftell() and fseek() Methods in File Handling

In Python, ftell() and fseek() are used for handling file positions while working with files.

1. ftell() Method

- The ftell() function returns the current file pointer position (offset) from the beginning of the file in bytes.
- It helps track where the read/write operations are occurring.

Example: Using ftell()

```
# Open a file in read mode
file = open("example.txt", "r")
# Read some characters
file.read(10)
# Get the current position
position = file.tell()
print("Current file position:", position)
# Close the file
file.close()
```

Output:

Current file position: 10

✅ *This means the cursor is at the 10th byte from the beginning of the file.*

2. fseek() Method

- The fseek(offset, whence) function moves the file pointer to a specified position.
- **Parameters:**
    - offset: Number of bytes to move.
    - whence (optional):
        - 0 → Move from the beginning of the file.
        - 1 → Move relative to the current position.
        - 2 → Move relative to the end of the file.

Example: Using fseek()

```
# Open a file in read mode
file = open("example.txt", "r")
# Move the cursor to the 10th byte from beginning
file.seek(10, 0)  #0- indicates from beginning
# Read from the new position
data = file.read()
print("Data read after seek:", data)
# Close the file
file.close()   # ✅ This moves the file pointer to the 5th byte and reads 10 bytes from there.
```

Already example.txt exist, the contents

```
Hello, this is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you
```

now after executing this program the output becomes as it start read from 10^{th} byte

```
Data read after seek: s is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you
```

---

3. Combined Example: Using ftell() and fseek()

```
# Open file in read mode
file = open("example.txt", "r")
# Read 5 bytes
print("Reading first 5 bytes:", file.read(5))
# Check current position
print("Current position:", file.tell())
# Move to the 2nd byte
file.seek(2, 0)
# Read 5 more bytes
print("Reading after seeking:", file.read(5))
# Close file
file.close()
```

output

```
Reading first 5 bytes: Hello
Current position: 5
Reading after seeking: llo,
```

Key Differences:

Method	Purpose
ftell()	Returns the current file pointer position.
fseek()	Moves the file pointer to a specific position.

✅ *These methods help efficiently navigate large files while reading and writing!* 🚀

@@

3. Write a python program to count the number of words in a given text file.

Input file example.txt

Hello, this is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you

Algorithm to Count Number of Words in a File

1. Start
2. Open the file "example.txt" in read mode.
3. Read the entire content of the file and store it in a variable content.
4. Split content into a list of words using the split() method.
5. Count the number of words using len() and store in word_count.
6. Display word_count.
7. Close the file.
8. Stop

program

```
# Open the file in read mode
file = open("example.txt", "r")
# Read the file content
content = file.read()
# Split the content into words
words = content.split()
# Count the number of words
word_count = len(words)
print("Number of words in the file:", word_count)
# Close the file
file.close()
```

output

Number of words in the file: 21

4. Write a python program to copy the contents of one file to another file and display the contents.

Algorithm to Copy Content from One File to Another

1. **Start**
2. Open the source file "source.txt" in **read** mode using with.
3. Read the content of the source file and store it in variable content.
4. Open the destination file "destination.txt" in **write** mode using with.
5. Write the content into the destination file.
6. Display the message **"Contents of destination file:"**.
7. Open the destination file again in **read** mode.
8. Read and display the contents of the destination file.
9. **Stop**

source.txt

Hello, this is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you

PROGRAM

```
# Open the source file in read mode
with open("source.txt", "r") as source_file:
   # Read the content of the source file
   content = source_file.read()

# Open the destination file in write mode and copy the content
with open("destination.txt", "w") as destination_file:
   destination_file.write(content)

# Display the copied content
print("Contents of destination file:")
with open("destination.txt", "r") as file:
   print(file.read())
```

destination.txt

Hello, this is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you

@@

5.Explain the concept of command-line arguments in python and demonstrate their usage with an example program.

Definition

The arguments that are given after the name of the program in the command line shell of the operating system are known as Command Line Arguments.

introduction

- Command-line arguments allow users to pass values to a Python script when it is executed from the terminal or command prompt. This enables dynamic input to a program without hardcoding values or using input prompts.
- Command-line arguments in Python make scripts versatile, configurable, and suitable for automation. They are especially useful in real-world applications like batch processing and automation scripts.

Concept

- You can pass values to a Python program from command line.
- Python collects the arguments in a list object.
- In Python, command-line arguments are accessed using the **sys** module.
- The **sys.argv** list stores command-line arguments:
  - sys.argv[0]: The name of the script.
  - sys.argv[1], sys.argv[2], etc., are the additional arguments.
- All arguments are treated as **strings**, so type conversion may be required.

Syntax

```
import sys
print(sys.argv)
```

Example program #save file as ex1.py

```
import sys
# Check if correct number of arguments are passed
if len(sys.argv) != 3:
   sys.exit(1)
# Read command-line arguments
num1 = int(sys.argv[1])
num2 = int(sys.argv[2])
# Add and display result
result = num1 + num2
print(f"Sum of {num1} and {num2} is: {result}")
```

execution

1.open command prompt

2.type the path in which ex1.py resides **#C:\Python311>python ex1.py 10 20**

output

Sum of 10 and 20 is: 30

How it works?

- The script expects two numbers as input.
- It reads them using sys.argv[1] and sys.argv[2].
- Converts them to integers and adds them.
- Prints the result.

6.Discuss syntax errors and exceptions in python, providing examples to illustrate their differences.

Syntax Errors vs Exceptions in Python

- In Python, syntax errors and exceptions are two types of errors that occur during the execution of a program, but they occur at different stages and have different causes.
- While syntax errors must be corrected before the code runs, exceptions can be anticipated and handled gracefully using exception handling techniques, thus improving program robustness and reliability.

I. Syntax Errors

Definition:

Syntax errors occur when the Python parser detects incorrect syntax. These errors are raised before the program is run, during the parsing (compilation) stage.

Characteristics:

- Prevent the code from being executed.
- Often due to incorrect use of keywords, missing punctuation, indentation errors, etc.

Example 1: Missing Colon

```
if x > 5
   print("x is greater than 5")
```

Output:

SyntaxError: expected ':'

Example 2: Incorrect Indentation

```
def greet():
print("Hello")
```

Output:

IndentationError: expected an indented block

Example 3 : Unclosed parentheses, brackets, or braces

```
print("Hello"  # Error
```

output

SyntaxError: incomplete input

Example 4 : Misuse of keywords

```
return = 5  # Error: can't use 'return' as variable
```

output

SyntaxError: invalid syntax

Example 5 : Using undeclared variables in expressions

```
print(x + y)  # If x or y is not defined earlier
```

output

NameError: name 'x' is not defined

Example 6: Wrong string syntax

```
print('Hello")  # Error: mismatched quotes
```

output

SyntaxError: unterminated string literal

II. Exceptions

Definition:

Exceptions are runtime errors that occur after the syntax is validated, but while the program is running. These are caused by illegal operations like dividing by zero, accessing out-of-range list indices, or using undefined variables.

Characteristics:

- Do not prevent the program from being parsed.
- Can be handled using try-except blocks.

Example 1: Division by Zero

```
a = 10
b = 0
print(a / b)
```

Output:

ZeroDivisionError: division by zero

Example 2: Handling Exception

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

Output:

Cannot divide by zero

Example 3: File Not Found

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

Output:

File not found

Example 4 : Index out of range

```
fruits = ["apple", "banana", "cherry"]
try:
    print(fruits[3])
except IndexError:
    print("Index out of range!")
```

output
Index out of range!

Key Differences

Feature	Syntax Error	Exception
Occurrence Stage	During parsing (before execution)	During execution (runtime)
Type of Error	Structural or formatting	Logical or operational
Detection	Interpreter halts immediately	Interpreter halts when it occurs
Handling	Cannot be handled	Can be handled using try-except
Example	if x > 5 (missing :)	10 / 0 → ZeroDivisionError

@@

7. What is exception handling in Python, and how does the try-except block work? provide an example.

Exception Handling in Python
Definition:
Exception handling in Python is a mechanism that allows you to handle errors (called **exceptions**) that occur during program execution, without stopping the program abruptly. It ensures the normal flow of the program even when unexpected events occur. Exception handling is essential for building **reliable**, **user-friendly**, and **error-resilient** Python applications. The try-except block provides a clean and efficient way to catch and respond to errors.

Why Exception Handling?
- Prevents program crashes
- Provides meaningful error messages
- Enables alternate actions or recovery
- Helps in debugging and fault tolerance

Common Exceptions in Python

Exception Type	Description
ZeroDivisionError	Division by zero
IndexError	List index out of range
ValueError	Invalid value
TypeError	Operation on incompatible types
FileNotFoundError	File doesn't exist

---

try-except Block in Python

Syntax:

```
try:
   risky_code()
except SpecificError:
   handle_error()
else:
   execute_if_no_error()
finally:
   always_execute()
```

Example-1 / Handling ZeroDivisionError

```
try:
   num1 = int(input("Enter numerator: "))
   num2 = int(input("Enter denominator: "))
   result = num1 / num2
   print("Result:", result)
except ZeroDivisionError:
   print("Error: Cannot divide by zero.")
except ValueError:
   print("Error: Please enter valid integers.")
else:
   print("Division successful.")
finally:
   print("Program completed.")
```

Output Sample-1

```
Enter numerator: 34
Enter denominator: 2
Result: 17.0
Division successful.
Program completed.
```

Output Sample-2

```
Enter numerator: 10
Enter denominator: 0
Error: Cannot divide by zero.
Program completed.
```

Example-2

```
#open a file and handles a FileNotFoundError exception if the file does not exist
filename = input("Enter the filename to open: ")
try:
   with open(filename, 'r') as file:
      content = file.read()
      print("File content:\n", content)
except FileNotFoundError:
   print("Error: File not found. Please check the filename.")
else:
  print("program executed successfully")
```

```
finally:
  print("program finished")
```

output-1

Enter the filename to open: file.txt
Error: File not found. Please check the filename.
program finished

output-2

Enter the filename to open: example.txt
File content:
 Hello, this is a test file.
Python file handling is easy!
Hello, this is a test file.
hello how are you

program executed successfully
program finished

@@

8. **Explain about raising exceptions with an example program**.

definition

In Python, **exceptions** are used to indicate that an error has occurred during program execution. Sometimes, we need to **intentionally raise exceptions** to signal that something has gone wrong according to our logic. This is done using the **raise** keyword.

Need- Why Raise Exceptions?

- To enforce business logic or validation rules.
- To prevent incorrect operations or invalid input.
- To make the code more readable and robust.
- To give informative error messages

Syntax

```
raise ExceptionType("Custom error message")
```

Built-in Exceptions Commonly Raised

Exception	**Purpose**
ValueError	Raised when a value is inappropriate
TypeError	Raised when a function receives wrong type
ZeroDivisionError	Raised when dividing by zero
RuntimeError	Raised for general runtime issues

In the following example

 the calculate_bmi() function validates input.
 If invalid input is found, it **raises a ValueError**.
 The try-except block **catches the error** and prints a user-friendly message.
 The finally block **executes cleanup** or follow-up actions regardless of errors.

Example program

```
def calculate_bmi(weight, height):
    if weight <= 0:
        raise ValueError("Weight must be greater than zero.")
    if height <= 0:
        raise ValueError("Height must be greater than zero.")
    bmi = weight / (height ** 2)
    return bmi
try:
    w = float(input("Enter weight (kg): "))
    h = float(input("Enter height (m): "))
    result = calculate_bmi(w, h)
    print(f"Our BMI is: {result:.2f}")
except ValueError as ve:
    print("Input Error:", ve)
finally:
    print("BMI calculation attempt completed.")
```

output

```
Enter weight (kg): 90
Enter height (m): -160
Input Error: Height must be greater than zero.
BMI calculation attempt completed.
```

@@

9. Discuss about various operations performed in a file by giving examples.

1. Opening a File

syntax

open(filename, mode)

Modes:

- 'r' – Read (default)
- 'w' – Write (overwrites file)
- 'a' – Append
- 'x' – Create new file
- 'b' – Binary mode (e.g., 'rb', 'wb')
- 't' – Text mode (default)

2.read() – Read Entire File or Specific Number of Characters

Syntax:

file.read([size])

Example:

```
with open("example.txt", "r") as file:
    content = file.read(5)  #Reads first 5 bytes alone
    print(content)
```

3. readline() –

The readline() method reads one line from the file at a time, including the newline character (\n) at the end.

Syntax:

```
file.readline()
```

Example:

```
with open("sample.txt", "r") as file:
    line = file.readline()   # Useful for processing files line-by-line
    print("First Line:", line)
```

4. readlines()

The readlines() method reads all the lines of a file and returns them as a list of strings, where each line is an item in the list.

Syntax:

```
file.readlines()
```

Example:

```
with open("sample.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())   # Returns a list of lines, including \n.
```

5. write()

The write() method writes a string to an open file. It does not add a newline (\n) automatically.

Syntax:

```
file.write(string)
```

Example:

```
with open("output.txt", "w") as file:
    file.write("Hello, Python!")  #Overwrites the file if it exists.
```

6.writelines()

The writelines() method writes a list of strings to a file. It does not automatically insert newline characters (\n), so you must include them manually.

Syntax:

```
file.writelines(list_of_strings)
```

Example:

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("output.txt", "w") as file:
    file.writelines(lines)
```

7. tell()

The seek() method is used to **move the cursor (file pointer)** to a specified position within the file. It takes an **offset** (the number of bytes to move) and an optional **whence** parameter, which determines from where the offset is calculated

Syntax:

```
file.tell()
```

Example:

```
with open("sample.txt", "r") as file:
    print("Pointer at:", file.tell())
    file.read(5)
    print("Pointer after reading 5 chars:", file.tell())  Returns position in bytes.
```

8.seek()

The seek() method is used to move the cursor (file pointer) to a specified position within the file. It takes an offset (the number of bytes to move) and an optional whence parameter, which determines from where the offset is calculated

Syntax:

```
file.seek(offset, whence)
```

- offset: number of bytes
- whence: 0 (start), 1 (current), 2 (end)

Example:

```
with open("sample.txt", "r") as file:
    file.seek(6)                                        # Move the file cursor to byte position 6
    print("After seeking to position 6:", file.read(5))  # Read 5 characters from there
```

9. Append Operation (a mode)

Syntax:

```
open("filename", "a")
```

Example:

```
with open("log.txt", "a") as file:
    file.write("New log entry\n")
```

Adds data to the end without overwriting existing content.

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

10. Explain about exception chaining in python with an example program.

Definition:

- Exception chaining is a feature that allows you to preserve the trace of exceptions, enhancing the debugging process by providing additional context about the root cause of an error.
- When handling a caught exception and raising a new one, Python can automatically chain them using the `from` keyword.
- Exception chaining is a way of linking multiple exceptions together, so that we can see the cause and effect of errors in code.
- Exception chaining can help us debug code more easily, and also provide more informative error messages to users.

What is Exception Chaining in Python Language?

Exception chaining in Python is a mechanism that allows you to associate one exception with another, creating a chain of exceptions. This is useful when you want to capture and convey additional context about an exception without losing information about the original exception that occurred. Exception chaining was introduced in Python 3.3 to enhance the clarity of error reporting and debugging.

Here's how exception chaining works:

1. An initial exception (the "inner" or "original" exception) occurs in our code. This exception might be raised for various reasons, such as a specific error condition or a failure during an operation.
2. You can capture this initial exception using a try-except block and then raise a new exception while chaining the original exception.
3. The chained exception provides a way to carry the context of the original exception, including its type, message, and traceback, while also allowing you to add additional context or information to the error message.

Here's an example of exception chaining:

```
try:
    # Code that may raise an initial exception
    x = 10 / 0
except ZeroDivisionError as e:
    # Capture the initial exception and raise a new exception while chaining it
    raise ValueError("An error occurred during division") from e
```

output

```
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero
ValueError: A value error occurred due to division by zero.
```

How it works

- An initial ZeroDivisionError exception occurs when attempting to divide by zero.
- We capture this exception using except and then raise a new ValueError exception, chaining the ZeroDivisionError using the from keyword.
- The ValueError exception now carries information about the original ZeroDivisionError, allowing us to provide additional context in the error message.

PART -C

1.**Discuss the Python codes to print try, except, and finally block statements.**

The try, except, and finally blocks in Python are powerful tools for handling exceptions. They enhance the robustness of programs by managing errors efficiently and ensuring critical cleanup operations are always performed.

Python provides robust support for **exception handling** using three main blocks:

Block	Purpose
`try`	Contains code that might raise an exception.
`except`	Handles the exception if it occurs in the try block.
`finally`	Contains code that executes **regardless** of whether an exception occurs.

Basic Structure of Try-Except-Finally

```
try:
    # Code that may raise an exception
except SomeException:
    # Code that runs if an exception occurs
finally:
    # Code that always runs
```

Benefits of Using try, except, finally

- Improves reliability of the program.
- Avoids crashes due to runtime errors.
- Ensures cleanup actions (like closing files or releasing memory) are always performed.
- Makes debugging and maintenance easier and clearer.

I. Exception Occurs

- The try block runs first.
- Division by zero causes a ZeroDivisionError.
- The except block catches and handles the error.
- The finally block runs **always**, whether there is an error or not.

program

```
try:
    print("In try block")
    x = 10 / 0  # Will raise ZeroDivisionError
except ZeroDivisionError:
    print("In except block: Cannot divide by zero")
finally:
    print("In finally block")
```

Output:
In try block
In except block: Cannot divide by zero
In finally block

II. No Exception Occurs

- No exception occurs, so except block is skipped.
- finally still executes.

program

```
try:
    print("In try block")
    x = 10 / 2
    print("Result:", x)
except ZeroDivisionError:
    print("In except block")
finally:
    print("In finally block")
```

Output:
In try block
Result: 5.0
In finally block

III. With Multiple Except Blocks

- Used for **resource cleanup** like closing files or releasing memory.
- Runs **whether or not** an exception is raised.
- Ensures critical code runs even if an error occurs or a return is hit.

program

```
try:
    num = int("abc")  # ValueError
except ValueError:
    print("Caught a ValueError")
except ZeroDivisionError:
    print("Caught a ZeroDivisionError")
finally:
    print("Finally block executed")
```

Output:
Caught a ValueError
Finally block executed

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

2. Write a python program that prompts the user to enter an integer and raises a ValueError if the input is not a valid integer.

This program demonstrates how to safely handle user input by catching a ValueError when the user enters a non-integer value. It uses try, except, and finally blocks to ensure robustness.

- **try block**: Attempts to convert the user input to an integer.
- **except ValueError**: Catches error if the input is not a valid integer.
- **finally block**: Executes regardless of whether an error occurred, typically used for final messages.

Benefits

- Prevents the program from crashing due to invalid input.
- Informs the user of their mistake clearly.
- Demonstrates defensive programming through exception handling.
- Ensures clean exit or continuation regardless of input.

Algorithm: Handle Integer Input with Exception Handling

1. **Start**
2. **Display** the message: "Enter an integer: "
3. **Read** the user input as a string and store it in user_input.
4. **Try** the following:
    - Convert user_input to an integer and store it in number.
    - **Display**: "You entered: " followed by the integer number.
5. **If ValueError occurs** (i.e., input is not a valid integer):
    - **Display**: "Error: That is not a valid integer."
6. **Finally**, regardless of success or error:
    - **Display**: "Input operation completed."
7. **Stop**

PROGRAM

```
def get_integer_input():
    try:
        user_input = input("Enter an integer: ")
        number = int(user_input)
        print("You entered:", number)
    except ValueError:
        print("Error: That is not a valid integer.")
    finally:
        print("Input operation completed.")
# Run the function
get_integer_input()
```

Output

Case 1: Valid input

```
Enter an integer: 42
You entered: 42
Input operation completed.
```

Case 2: Invalid input
Enter an integer: hello
Error: That is not a valid integer.
Input operation completed.

3.reads a file and handles a FileNotFoundError if the file does not exist.

Handling file operations is crucial in Python. When a file is not found, Python raises a FileNotFoundError. To manage such errors gracefully, we use the try, except, and finally blocks.

- try block: Tries to open and read a file.
- except block: Catches the FileNotFoundError if the file doesn't exist.
- finally block: Executes regardless of error – used for cleanup or final messages.

Benefits of Handling FileNotFoundError:

1. **Prevents Crashes**: Avoids program termination when a file doesn't exist.
2. **User-Friendly Messages**: Provides clear error messages, guiding users to fix the issue.
3. **Improves Robustness**: Ensures predictable behavior, even when errors occur.
4. **Resource Management**: Ensures files are properly closed, even in case of an error.
5. **Better Debugging**: Logs or displays errors for easier troubleshooting

Algorithm for File Reading with Exception Handling

1. Start
2. Define a function read_file(filename).
3. Try to:
    - Open the file in read mode.
    - Read and display the content of the file.
    - Close the file.
4. If FileNotFoundError occurs, display an error message: "The file does not exist".
5. Finally, display: "File read operation completed".
6. Call read_file() with an existing file and a non-existing file.
7. Stop.

PROGRAM

```
def read_file(filename):
    try:
        file = open(filename, 'r')
        content = file.read()
        print("File contents:\n", content)
        file.close()
    except FileNotFoundError:
        print(f"Error: The file '{filename}' does not exist.")
    finally:
        print("File read operation completed.\n")
# Test with an existing file
read_file("existing_file.txt")
# Test with a non-existing file
read_file("missing_file.txt")
```

example Runs:

If "existing_file.txt" exists:
File contents:
Hello, this is a test file.
File read operation completed.

If "missing_file.txt" does not exist:
Error: The file 'missing_file.txt' does not exist.
File read operation completed.

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@2

4.Program that attempts to access an index in a list and handles an IndexError exception if the index is out of range

In Python, an IndexError occurs when a program attempts to access an index that is out of range for a list. Proper exception handling using try, except, and finally ensures the program does not crash and provides meaningful feedback to the user.

- **try block**: Used to access a list element by index.
- **except block**: Catches the IndexError if the index is out of bounds.
- **finally block**: Executes regardless of whether an error occurred.

Benefits of Exception Handling in This Program

- Prevents program from crashing on out-of-range access.
- Enhances user experience by displaying informative messages.
- Demonstrates structured programming with clear separation of normal and error flows.
- Ensures final actions (like logging or cleanup) are always executed.

Short Algorithm for Accessing List Element with Index Error Handling

1. Start
2. Define a function access_list_element(my_list, index).
3. Try to:
    - Access the element at the given index in my_list.
    - Print the element if found.
4. If IndexError occurs, display an error message indicating the index is out of range.
5. Finally, print: "End of index access operation".
6. Test the function with a valid and an invalid index.
7. Stop.

program

```
def access_list_element(my_list, index):
    try:
        print("Attempting to access index", index)
        element = my_list[index]
        print("Element at index", index, "is:", element)
```

```
    except IndexError:
        print("Error: The index", index, "is out of range for the list.")
    finally:
        print("End of index access operation.\n")
# Sample list
numbers = [10, 20, 30, 40, 50]
# Test with a valid index
access_list_element(numbers, 2)
# Test with an invalid index
access_list_element(numbers, 10)
```

OUTPUT

```
Attempting to access index 2
Element at index 2 is: 30
End of index access operation.

Attempting to access index 10
Error: The index 10 is out of range for the list.
End of index access operation.
```

@@