Gayatri Vidya Parishad College of Engineering for Women
(Autonomous)
(Affiliated to Andhra University, Visakhapatnam)
Madhurawada, Visakhapatnam

II B.Tech. - I Semester Regular Examinations, Nov – 2025
PYTHON PROGRAMING - 24CT11RC06
(EEE Branch)

Answers for all Questions

1(a) Explain about applications of Python in different domains. (7M)
Ans:
Python is a versatile, high-level programming language widely used in many technological fields due to its simplicity, readability, and powerful libraries. Some important application areas are:
1. Web Development
Python provides frameworks like Django, Flask, and FastAPI for building dynamic, secure, and scalable web applications.
2. Data Science and Analytics
Libraries such as NumPy, Pandas, and Matplotlib help in data analysis, data visualization, and scientific computation.
3. Machine Learning and AI
Python is the most popular language for AI/ML due to powerful libraries like TensorFlow, PyTorch, and Scikit-learn, used for building intelligent systems.
4. Automation and Scripting
Python is widely used to automate repetitive tasks such as file handling, web scraping, system administration, and software testing.
5. Scientific Computing
Tools like SciPy, SymPy, and Jupyter Notebook make Python useful in research, simulations, mathematical modeling, and engineering applications.
6. Cybersecurity and IoT
Python helps in writing security tools, scanning networks, and developing IoT applications using MicroPython and Raspberry Pi.
Conclusion:
Python's simplicity and powerful library ecosystem make it applicable in almost every domain.

1(b) Define input & output statements. Write a program to add two numbers using input(). (7M)
Ans:
Input Statement:
An **input statement** is used to receive data from the user during program execution. In Python, the function **input()** is used to accept values from the keyboard. It always returns the entered value as a **string**.
Example:
```
name = input("Enter your name: ")
```

Output Statement:
An **output statement** is used to display information or results on the screen.
In Python, the **print()** function is used for output.
Example:

```
print("Hello World")
```

Program:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
sum = a + b
print("Sum =", sum)
```

Output:

Enter first number: 5
Enter second number: 8
Sum = 13

2(a) Explain the historical development and necessity of Python programming? (7M)

Ans:

Historical Development:

- Python was developed by Guido van Rossum in 1989 at CWI, Netherlands.
- First official release: Python 1.0 (1994).
- Significant versions:
  - Python 2.0 (2000)
  - Python 3.0 (2008) – major redesign
- Now maintained by Python Software Foundation.

Necessity of Python

Python became necessary due to several factors that other languages could not satisfy simultaneously:

1. **Simplicity and Readability**
   - Python's syntax closely resembles plain English.
   - Easy to learn for beginners and powerful for experts.
2. **Rapid Application Development**
   - Fewer lines of code compared to languages like C, C++ or Java.
   - Faster development cycle and lower maintenance cost.
3. **Extensive Standard Library and Ecosystem**
   - Built-in modules for file handling, networking, math, OS operations.
   - Additional libraries for almost every domain (NumPy, Pandas, TensorFlow, Django, Flask).
4. **Cross-Platform and Open Source**
   - Runs on Windows, Linux, macOS, mobile, and embedded systems.
   - Completely free and community-driven.
5. **Versatility Across Domains**
   Python is widely used in:
   - Artificial Intelligence and Machine Learning
   - Data Science and Analytics
   - Web Development
   - Automation and Scripting
   - Scientific Computing
   - Cybersecurity
   - Education
6. **Strong Community Support**
   - Millions of developers contribute to packages, forums, documentation, and learning resources.

2 (b). Illustrate with an example the concept of indentation and its significance in Python? (7M)
Ans:
Indentation in Python
Indentation refers to the spaces or tabs placed at the beginning of a line of code. In Python, indentation is not optional—it is a part of the syntax. It is used to define blocks of code such as loops, functions, conditional statements, etc. Unlike languages like C, C++ or Java that use { } braces to group statements, Python uses indentation to represent code blocks.

Significance of Indentation

1. **Defines the structure of a program**
   Python identifies which statements belong to loops, functions or conditions based on indentation.
2. **Improves readability**
   Indented blocks make the program easier to read and understand.
3. **Avoids errors**
   Incorrect indentation leads to errors such as IndentationError or SyntaxError.
4. **Makes Python clean and consistent**
   It enforces uniform code formatting.

Example Demonstrating Indentation
Correct Code

```
x = 10
if x > 5:
    print("x is greater than 5")   # Indented block
    print("This line is also part of the if block")

print("This line is outside the if block")
```

Output

```
x is greater than 5
This line is also part of the if block
This line is outside the if block
```

Incorrect Indentation Example

```
if x > 5:
print("x is greater")  # Error: not indented
```

Error

```
IndentationError: expected an indented block
```

3 (a) Explain the different types of Python operators with examples? (7M)

Ans:
Python operators are special symbols used to perform operations on variables and values.
They are classified into **seven main types**:
1. Arithmetic Operators
Used to perform mathematical operations.

Operator	Description	Example
+	Addition	10 + 5 → 15
-	Subtraction	10 - 3 → 7
*	Multiplication	4 * 5 → 20
/	Division	10 / 2 → 5.0
%	Modulus (remainder)	10 % 3 → 1
**	Exponent	2 ** 3 → 8
//	Floor division	10 // 3 → 3

Example:

```
a = 10
```

```
b = 3
print(a + b)    # 13
print(a % b)    # 1
```

2. Relational (Comparison) Operators

Used to compare two values; returns True or False.

Operator	Meaning	Example
==	Equal to	5 == 5 → True
!=	Not equal to	5 != 3 → True
>	Greater than	7 > 4 → True
<	Less than	3 < 5 → True
>=	Greater or equal	6 >= 6 → True
<=	Less or equal	4 <= 7 → True

Example:

```
x = 10
print(x > 5)    # True
```

3. Logical Operators

Used to combine conditions; return True or False.

Operator	Meaning
and	True only if both conditions are true
or	True if any one condition is true
not	Reverses the condition

Example:

```
x = 10
print(x > 5 and x < 20)  # True
print(not(x > 5))        # False
```

4. Assignment Operators

Used to assign values to variables.

Operator	Meaning	Example
=	Assign	x = 5
+=	Add and assign	x += 3 → x = x + 3
-=	Subtract and assign	x -= 2
*=	Multiply and assign	x *= 4
/=	Divide and assign	x /= 2
%=	Modulus assign	x %= 3
**=	Power assign	x **= 2
//=	Floor assign	x //= 2

Example:

```
x = 5
x += 2
print(x)  # 7
```

5. Bitwise Operators

Operate on bits (0s and 1s) of numbers.

Operator	Meaning	Example
&	Bitwise AND	5 & 3 → 1
|	Bitwise OR	`5
^	XOR	5 ^ 3 → 6
~	Bitwise NOT	~5 → -6
<<	Left shift	5 << 1 → 10
>>	Right shift	5 >> 1 → 2

Example:

```
a = 5   # 0101
b = 3   # 0011
print(a & b)  # 1
```

6. Membership Operators

Check membership in sequences like lists, strings, tuples.

Operator	Meaning	Example
in	True if value found	"a" in "apple" → True
not in	True if value not found	3 not in [1,2,3] → False

Example:

```
print(4 in [1, 2, 3])       # False
print("py" in "python")     # True
```

7. Identity Operators

Used to check if two variables refer to the **same memory object**.

Operator	Meaning
is	True if both reference same object
is not	True if objects are different

Example:

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a is b)      # True (same object)
print(a is c)      # False (same values, different object)
```

Python provides a rich set of operators to perform arithmetic, logical, relational, bitwise, membership, identity, and assignment operations. Operators make Python programming powerful and expressive.

3 (b) Define Loops? Write a python program to find factorial of number using while loop? (7M)

Ans:

Define Loops

A **loop** is a control structure that allows a set of statements to be executed repeatedly as long as a given condition is true. Loops help to perform repetitive tasks efficiently without writing the same code multiple times.

Python mainly supports two loops:

1. **for loop** – repeats over a sequence (list, string, range, etc.)
2. **while loop** – repeats as long as a condition is true

1. Syntax of for Loop

```
for variable in sequence:
   statement(s)
```

Example

```
for i in range(1, 6):
   print(i)
```

2. Syntax of while Loop

```
while condition:
   statement(s)
```

Python Program to Find Factorial of a Number Using while Loop

```
# Program to find factorial of a number using while loop

num = int(input("Enter a number: "))
fact = 1
i = 1

while i <= num:
   fact = fact * i
   i = i + 1

print("Factorial of", num, "is", fact)
```

Output:

Enter a number: 4
Factorial of 4 is 24

4 (a) Define Conditional Statements? Write a Python program using if-elif-else to categorize a number as positive, negative, or zero? (7M)

Ans:

Define Conditional Statements

Conditional statements are decision-making statements that allow a program to execute certain blocks of code based on whether a condition is **true** or **false**. They help control the flow of a program.

Python provides the following conditional statements:

- **if**
- **if-else**
- **if-elif-else**

These statements evaluate conditions and execute different blocks accordingly.

1. Syntax of if Statement

```
if condition:
   statement(s)
```

2. Syntax of if-else Statement

```
if condition:
   statement(s)
else:
```

```
    statement(s)
```

3. Syntax of if-elif-else Statement

```
if condition1:
    statement(s)
elif condition2:
    statement(s)
elif condition3:
    statement(s)
else:
    statement(s)
```

Python Program Using if-elif-else to Categorize a Number

```
# Program to categorize a number as positive, negative, or zero

num = float(input("Enter a number: "))

if num > 0:
    print("The number is Positive")
elif num < 0:
    print("The number is Negative")
else:
    print("The number is Zero")
```

Output:

```
Enter a number: 4
The number is Positive
```

4(b) Explain about Membership & Identity Operator with examples? (7M)

Ans:

Membership Operators

Membership operators are used to **test whether a value exists in a sequence** such as a list, string, tuple, or set.

Python provides two membership operators:

Operator	Description
in	Returns True if the value is present in the sequence
not in	Returns True if the value is NOT present in the sequence

Examples

```
# Membership operator examples

text = "python"
numbers = [1, 2, 3, 4]
print("p" in text)        # True
print("z" not in text)    # True
print(3 in numbers)       # True
print(10 not in numbers)  # True
```

Identity Operators

Identity operators are used to **check whether two variables refer to the same memory location**

(same object).

Operator	Description
is	True if both variables refer to the same object
is not	True if they refer to different objects

Examples

Identity operator examples

```
a = [1, 2, 3]
b = a          # b refers to same object as a
c = [1, 2, 3]  # c is a different object with same values

print(a is b)      # True (same memory location)
print(a is c)      # False (different objects)
print(a is not c)  # True
```

5(a) Demonstrate slicing and indexing operations on lists and strings with examples (7M)

Ans:

- **Indexing in Python**

Indexing means accessing individual elements of a sequence using their position (index).
Python supports indexing for sequences like:

- Strings
- Lists
- Tuples

Indexes are integers that specify the position of an element in the sequence.

Types of Indexing

1. Positive Indexing
   - Starts from 0
   - The first element has index 0
   - The second element has index 1, and so on.

2. Negative Indexing
   - Starts from -1 (last element)
   - The last element has index -1
   - The second last has index -2, and so on.

Importance of Indexing:

- To access individual characters/elements
- To perform slicing
- Helps in data manipulation
- Makes retrieval fast and efficient

(A) Indexing in Strings

```
s = "PYTHON"

print(s[0])    # P  (first character)
print(s[3])    # H
print(s[-1])   # N  (last character)
print(s[-3])   # H
```

(B) Indexing in Lists

```
lst = [10, 20, 30, 40, 50]
```

```
print(lst[0])    # 10
print(lst[2])    # 30
print(lst[-1])   # 50
print(lst[-4])   # 20
```

- **Slicing in Python**

Slicing is a technique used to extract a portion (subset) of a sequence such as:

- Strings
- Lists
- Tuples

It allows you to access multiple elements at once by specifying a start, end, and step.

Slicing Syntax

sequence[start : end : step]

Meaning:

- start → index where the slice begins (included)
- end → index where the slice ends (excluded)
- step → difference between consecutive indices (optional)

(A) Slicing in Strings

```
s = "PYTHON"

print(s[0:3])    # PYT
print(s[2:])     # THON
print(s[:4])     # PYTH
print(s[::2])    # PTO
print(s[::-1])   # NOHTYP  (reverse string)
```

(B) Slicing in Lists

```
lst = [10, 20, 30, 40, 50, 60]

print(lst[1:4])      # [20, 30, 40]
print(lst[:3])       # [10, 20, 30]
print(lst[3:])       # [40, 50, 60]
print(lst[::2])      # [10, 30, 50]
print(lst[::-1])     # [60, 50, 40, 30, 20, 10]  (reverse list)
```

Use of slicing:

- Extracting substrings or sublists
- Reversing sequences
- Getting every Nth element
- Efficient data manipulation

5(b) Explain dictionary operations such as adding, updating, and deleting key-value pairs with examples. (7M)

Ans:

Dictionary in Python

A dictionary is a built-in data structure in Python used to store data in the form of key–value pairs. It is unordered, mutable, and indexed, and each key in a dictionary must be unique and immutable (like strings, numbers, tuples).

Definition
A dictionary is a collection of data values stored in the form:
key : value
It is written using **curly brackets { }**.

Example
```
student = {
   "name": "John",
   "age": 20,
   "course": "Python"
}
```

Features of Dictionary
- Stores data as key–value pairs
- Mutable (can be changed)
- Keys must be unique
- Values can be any type (string, number, list, etc.)
- Indexed using keys, not positions
- Very fast lookup due to hashing

1. Adding Key–Value Pairs

You can add a new key–value pair by simply assigning a value to a new key.

Example
```
student = {"name": "John", "age": 20}
# Adding a new key-value pair
student["course"] = "Python"
print(student)
```

Output
```
{'name': 'John', 'age': 20, 'course': 'Python'}
```

2. Updating Key–Value Pairs

Updating is done by assigning a new value to an existing key or using the update() method.

Example 1: Using assignment
```
student = {"name": "John", "age": 20}
# Update existing key
student["age"] = 21
print(student)
```

Example 2: Using update()
```
student.update({"age": 22, "course": "AI"})
print(student)
```

3. Deleting Key–Value Pairs
Python provides multiple methods to delete items from a dictionary.

(a) delete a specific key using del
```
student = {"name": "John", "age": 20, "course": "Python"}

del student["age"]
print(student)
```

(b) remove and return a value using pop()
```
student = {"name": "John", "age": 20}

removed_value = student.pop("age")
print(removed_value)    # 20
print(student)
```

(c) remove the last inserted item using popitem()
```
student = {"name": "John", "age": 20}

student.popitem()
print(student)
```

(d) remove all items using clear()
```
student = {"name": "John", "age": 20}

student.clear()
print(student)   # {}
```

6(a) Write a short note on Advanced Data Structures in python? (7M)

Ans

Definition:

A Data Structure is a method of organizing, storing, and managing data in a computer so that it can be accessed and used efficiently. It determines how data is arranged in memory and what operations can be performed on it.

1. Built-in Data Structures in Python

a) List

- Ordered
- Mutable (modifiable)
- Allows duplicate values
- Can store mixed data types

Example:
```
fruits = ["apple", "banana", "cherry"]
fruits.append("mango")
```

b) Tuple

- Ordered
- Immutable
- Faster than lists
- Used when data should not be changed

Example:
```
t = (10, 20, 30)
```

c) Set

- Unordered
- No duplicate elements
- Used for membership testing and removing duplicates

Example:

```
s = {1, 2, 3}
s.add(4)
```

d) Dictionary

- Stores data as key–value pairs
- Mutable
- Very fast for lookups

Example:

```
student = {"name": "Ravi", "age": 20}
student["age"] = 21
```

2. Advanced (or user-defined) Data Structures in Python

Python supports advanced data structures using libraries like collections and queue.

a) Stack

- Follows LIFO (Last In, First Out)
- Implemented using lists or collections.deque

Example:

```
stack = []
stack.append(10)
stack.append(20)
stack.pop()
```

b) Queue

- Follows FIFO (First In, First Out)
- Implemented using collections.deque

Example:

```
from collections import deque
queue = deque([1, 2, 3])
queue.append(4)
queue.popleft()
```

c) Deque (Double-ended queue)

- Supports insertion/deletion from both ends
- Faster than list operations

Example:

```
from collections import deque
d = deque([1, 2, 3])
d.appendleft(0)
```

d) Linked List

- Not built-in, but can be created using classes
- Each node contains data and a pointer to the next node

Example:

```
class Node:
    def __init__(self, data):
        self.data = data
```

```
    self.next = None
```

e) Heap

- Implements priority queues
- Uses heapq module

Example:

```
import heapq
h = [20, 10, 30]
heapq.heapify(h)
```

f) Graph

- Can be implemented using dictionaries of lists/sets

Example:

```
graph = {
   "A": ["B", "C"],
   "B": ["A", "D"]
}
```

6(b) What is comprehension? Write examples of list, set comprehensions? (7M)

Ans:

Comprehension is a concise and powerful way to create new sequences (like lists, sets, or dictionaries) from existing iterables in a single line of code. It combines looping, expression, and optional conditions into one compact statement.

Comprehensions make the code:

- Shorter
- More readable
- Faster

Types of Comprehensions

1. List Comprehension → creates a list
2. Set Comprehension → creates a set
3. Dictionary Comprehension → creates a dictionary
4. Generator Expression → creates a generator object

1. List Comprehension

Syntax:

```
[expression for item in iterable if condition]
```

Example 1: Create a list of squares

```
squares = [x*x for x in range(1, 6)]
print(squares)       # Output: [1, 4, 9, 16, 25]
```

Example 2: Even numbers from 1 to 10

```
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens)         # Output: [2, 4, 6, 8, 10]
```

2. Set Comprehension

Syntax:

```
{expression for item in iterable if condition}
```

Example 1: Create a set of cubes

```
cubes = {x**3 for x in range(1, 6)}
print(cubes)        # Output: {1, 8, 27, 64, 125}

Example 2: Unique even numbers
even_set = {x for x in [1,2,2,3,4,4,5] if x % 2 == 0}
print(even_set)     # Output: {2, 4}
```

7(a) Explain different types of function arguments in python with examples? (7M)

Ans:

Python allows several types of arguments to provide flexibility when calling functions.

The main types are:

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable-Length Arguments (*args, **kwargs)

1. Positional Arguments

Arguments are passed in the same order as parameters are defined.

Example

```
def add(a, b):
   print(a + b)

add(10, 20)   # Output: 30
```

2. Keyword Arguments

Arguments are passed using parameter names, so order does not matter.

Example

```
def student(name, age):
   print("Name:", name)
   print("Age:", age)

student(age=20, name="Ravi")
```

3. Default Arguments

A parameter is given a default value.

If no value is passed, the default is used.

Example

```
def greet(name="Guest"):
   print("Hello", name)

greet()         # Output: Hello Guest
greet("Asha")   # Output: Hello Asha
```

4. Variable-Length Arguments

Python provides two special symbols:

a) *args → Non-keyword variable arguments

Used to pass any number of positional arguments.

Example

```
def total(*numbers):
    print(sum(numbers))

total(10, 20, 30)   # Output: 60
```

b) **kwargs → Keyword variable arguments

Used to pass any number of keyword arguments.

Example

```
def info(**details):
    print(details)

info(name="Ravi", age=22)
# Output: {'name': 'Ravi', 'age': 22}
```

7(b) Describe how to create and import modules in Python with examples? (7M)

Ans

Modules in Python

A module in Python is a file that contains Python code such as variables, functions, classes, or statements.

Modules help in:

- Reusing code
- Organizing programs
- Reducing complexity

A module is simply a **.py file**.

1. Creating a Module

To create a module, write your Python code in a separate file and save it with a .py extension.

Example: mymodule.py

```
# This is a module file

def add(a, b):
    return a + b

def greet(name):
    return "Hello " + name
```

This file (mymodule.py) is a module.

2. Importing a Module

Python provides multiple ways to import modules.

A. Using import statement

Example

```
import mymodule
print(mymodule.add(10, 20))
print(mymodule.greet("Ravi"))
```

Explanation:
We access functions using the module name as a prefix.

B. Using from module import function
Example
```
from mymodule import add

print(add(5, 10))
```
Explanation:
Only the specified function is imported. No need to use module name.

C. Importing all functions
Example
```
from mymodule import *
print(greet("Asha"))
```
Note: Not recommended for large programs.

D. Importing module with an alias
Example
```
import mymodule as m

print(m.add(3, 7))
```

3. Built-in Modules Example
Python also provides built-in modules.
Example: Using the math module
```
import math

print(math.sqrt(25))
```

8(a) Define a Lambda Function in Python and give an example of its usage with the filter() built-in function. (7M)

Ans:

- **Lambda Function in Python**

A lambda function is a small, anonymous (nameless) function in Python.
It is used when we need a simple function for a short period of time.
Lambda functions:

- Are defined using the lambda keyword
- Can have any number of arguments
- Contain only one expression
- Return the value of that expression automatically

Syntax
lambda arguments: expression
Example: Basic Lambda Function
```
square = lambda x: x * x
print(square(5)) #25
```

- **filter() Function in Python**

The filter() function is a built-in function in Python used to filter elements from an iterable (like a list, tuple, or set) based on a condition.
It returns only those elements for which the function returns True.

Syntax
filter(function, iterable)
Parameters

- function → A function that returns True or False
- iterable → A sequence (list, tuple, set, etc.)

Note:
filter() returns an iterator, so it is usually converted to a list using list().

Example 1: Filter even numbers using filter()

```
numbers = [1, 2, 3, 4, 5, 6]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

Output:
[2, 4, 6]

Example 2: Filter names with length > 4

```
names = ["Ram", "Sita", "Krishna", "Arjun"]

long_names = list(filter(lambda n: len(n) > 4, names))
print(long_names)
```

Output:
['Krishna', 'Arjun']

8(b) What is PIP in Python? Explain how packages are installed and managed using PIP? (7M)
Ans:
PIP in Python:
PIP stands for **P**ackage **I**nstaller for **P**ython.
It is a command-line tool used to install, upgrade, and manage external Python packages from the Python Package Index (PyPI).
PIP makes it easy to add additional libraries that are not included in the Python standard library.

Common Uses of PIP

- Installing new packages
- Uninstalling packages
- Upgrading packages
- Viewing installed packages
- Checking package versions

1. Installing Packages Using PIP
Basic Syntax

```
pip install package_name
```

Example
Install the NumPy library:

pip install numpy

2. Upgrading a Package
pip install --upgrade package_name
Example
pip install --upgrade pandas

3. Uninstalling a Package
pip uninstall package_name
Example
pip uninstall requests

4. Viewing All Installed Packages
pip list

5. Viewing Details of a Particular Package
pip show package_name
Example
pip show flask

6. Installing from a Requirements File
Used for large projects where multiple packages are required.
Command
pip install -r requirements.txt

9(a) Define classes and objects? Illustrate with a program defining a class and creating objects? (7M)
Ans
Definition of Class
A class is a blueprint or template for creating objects.
It defines:

- Data (variables/attributes)
- Behavior (functions/methods)

A class does not occupy memory until an object is created.

Definition of Object
An object is an instance of a class.
It represents a real-world entity and has:

- State (data values)
- Behavior (methods)

Objects allow us to use the properties and methods defined inside a class.

Illustration with Program
Program: Defining a Class and Creating Objects

```
class Student:
    # Constructor
    def __init__(self, name, marks):
        self.name = name        # attribute
        self.marks = marks      # attribute
```

```
    # Method to display details
    def display(self):
        print("Name:", self.name)
        print("Marks:", self.marks)

# Creating objects of the class
s1 = Student("Ravi", 85)
s2 = Student("Asha", 92)

# Calling methods using objects
s1.display()
s2.display()
```

Output:
Name: Ravi
Marks: 85
Name: Asha
Marks: 92

Explanation of Program

- Line 1–5: A class Student is defined with attributes name and marks.
- __init__() is the constructor that initializes the object.
- display() is a method used to print the details.
- Objects s1 and s2 are created by calling the class with appropriate values.
- Each object has its own copy of attributes.

9(b) How does Python handle exceptions using try-except-finally blocks? (7M)

Ans

Exception Handling in Python

Exception handling is used to manage errors that occur during program execution without stopping the entire program.
Python provides the try–except–finally mechanism to handle such errors safely.

Components of Exception Handling

1. try block
    - Contains the code that may raise an exception.
2. except block
    - Catches and handles the exception.
    - Multiple except blocks can be used for different error types.
3. finally block
    - Contains code that always executes, whether an exception occurs or not.
    - Often used for closing files, releasing resources, etc.

General Syntax

```
try:
    # risky code
except ExceptionType:
    # handling code
finally:
    # code that always runs
```

Example Program

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)

except ZeroDivisionError:
    print("Error: Cannot divide by zero.")

except ValueError:
    print("Error: Invalid input. Please enter a number.")

finally:
    print("Execution completed.")
```

Explanation

- try:
  The program tries to convert input to an integer and divide 10 by that number.
- except ZeroDivisionError:
  Executes if the user enters 0, preventing division by zero.
- except ValueError:
  Executes if the user enters a non-numeric value.
- finally:
  Always runs, whether exception occurs or not.

Example output:

```
Enter a number: 2
Result: 5.0
Execution completed.
```

10(a) What is Inheritance? Explain different types of Inheritances in python? (7M)

Ans

Inheritance in Python

Inheritance is an important feature of Object-Oriented Programming (OOP) that allows a class (child class) to acquire the properties and methods of another class (parent class).

It promotes code reusability, reduces duplication, and supports hierarchical relationships.

- Parent/Super Class: The class whose properties are inherited
- Child/Sub Class: The class that inherits properties

Types of Inheritance in Python

Python supports five types of inheritance:

1. Single Inheritance

A child class inherits from one parent class.

Example

```
class A:
    def show(self):
        print("Class A")
```

```
class B(A):
    pass

obj = B()
obj.show()
```

2. Multiple Inheritance
A child class inherits from more than one parent class.
Example
```
class A:
    def methodA(self):
        print("From A")

class B:
    def methodB(self):
        print("From B")

class C(A, B):
    pass

obj = C()
obj.methodA()
obj.methodB()
```

3. Multilevel Inheritance
A class is derived from another derived class (grandparent → parent → child).
Example
```
class A:
    def showA(self):
        print("A")

class B(A):
    def showB(self):
        print("B")

class C(B):
    def showC(self):
        print("C")

obj = C()
obj.showA()
obj.showB()
obj.showC()
```

4. Hierarchical Inheritance
Multiple child classes inherit from the same parent class.
Example
```
class A:
    def show(self):
```

```
        print("Parent A")

class B(A):
    pass

class C(A):
    pass

obj1 = B()
obj2 = C()
obj1.show()
obj2.show()
```

5. Hybrid Inheritance

A combination of two or more types of inheritance (e.g., Hierarchical + Multiple). Python uses Method Resolution Order (MRO) to avoid conflicts.

Example

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

10(b) Explain about file handling operations in Python? (7M)

Ans

File Handling Operations in Python

File handling in Python allows programs to create, read, write, and modify files stored on the disk. Python provides a built-in function open() along with various file methods to perform file operations efficiently.

1. Opening a File

Python uses the open() function to open a file.

Syntax

```
file_object = open("filename", "mode")
```

Common File Modes

Mode	Description
'r'	Read mode (default). File must exist.
'w'	Write mode. Creates new file or overwrites existing one.
'a'	Append mode. Adds data to the end of the file.
'r+'	Read and write.
'w+'	Write and read. Overwrites existing file.
'a+'	Append and read.

Mode	Description
'b'	Binary mode (add to any mode like rb, wb).

2. Reading from a File
Python provides several methods to read data:
(a) read() – reads entire file
```
f = open("sample.txt", "r")
data = f.read()
print(data)
f.close()
```
(b) readline() – reads one line
```
f = open("sample.txt", "r")
line = f.readline()
print(line)
f.close()
```
(c) readlines() – reads all lines into a list
```
f = open("sample.txt", "r")
lines = f.readlines()
print(lines)
f.close()
```

3. Writing to a File
Using write()
```
f = open("sample.txt", "w")
f.write("Hello Python!\n")
f.close()
```
Using writelines() (for list of strings)
```
f = open("data.txt", "w")
f.writelines(["Line1\n", "Line2\n", "Line3\n"])
f.close()
```

4. Appending Data
```
f = open("sample.txt", "a")
f.write("\nThis line is added.")
f.close()
```

5. Closing a File
Always close the file after operations to free system resources.
```
f.close()
```

6. Using with Statement (Best Practice)
Automatically closes the file after work is done.
```
with open("sample.txt", "r") as f:
    content = f.read()
    print(content)
```

7. File Object Properties
Attribute Description
f.name name of file

Attribute	Description
f.mode	file opening mode
f.closed	returns True if file is closed

Example:

```
with open("sample.txt") as f:
    print(f.name)
    print(f.mode)
```

8. File Handling Example (Write → Read)

```
# Writing
with open("demo.txt", "w") as f:
    f.write("Python File Handling\n")
    f.write("Second Line")

# Reading
with open("demo.txt", "r") as f:
    print(f.read())
```

9. Reading and Counting Lines, Words, Characters

```
with open("test.txt", "r") as f:
    text = f.read()

chars = len(text)
words = len(text.split())
lines = len(text.split("\n"))

print("Characters:", chars)
print("Words:", words)
print("Lines:", lines)
```

Prepared by
D Srinivas Reddy
Assistant Professor
EEE Dept.