

# Subject Name Solutions

4321602 – Winter 2024

Semester 1 Study Material

*Detailed Solutions and Explanations*

## Question 1(a) [3 marks]

Give the difference between Set and Dictionary in python.

### Solution

Feature	Set	Dictionary
Data Storage	Stores unique elements only	Stores key-value pairs
Order	Unordered collection	Ordered (Python 3.7+)
Duplicates	No duplicates allowed	Keys must be unique
Access	Cannot access by index	Access values by keys
Syntax	{1, 2, 3}	{'key': 'value'}

- **Set:** Collection of unique, unordered elements
- **Dictionary:** Collection of key-value pairs with unique keys

### Mnemonic

“Sets are Unique, Dicts have Keys”

## Question 1(b) [4 marks]

Explain List in Python with example.

### Solution

**List** is an ordered, mutable collection that can store different data types.

**Table of List Operations:**

Operation	Syntax	Example
Create	<code>list_name = []</code>	<code>fruits = ['apple', 'banana']</code>
Access	<code>list[index]</code>	<code>fruits[0]</code> returns 'apple'
Add	<code>append()</code>	<code>fruits.append('orange')</code>
Remove	<code>remove()</code>	<code>fruits.remove('apple')</code>

```
\# Example
numbers = [1, 2, 3, 4, 5]
numbers.append(6) \# [1, 2, 3, 4, 5, 6]
print(numbers[0]) \# Output: 1
```

- **Ordered:** Elements maintain their position
- **Mutable:** Can be modified after creation
- **Flexible:** Stores any data type

### Mnemonic

“Lists are Ordered and Modifiable”

### Question 1(c) [7 marks]

What is Tuple in Python? Write a Python program to swap two tuple values.

#### Solution

**Tuple** is an ordered, immutable collection that stores multiple items.

**Table of Tuple Features:**

Property	Description	Example
<b>Immutable</b>	Cannot change after creation	t = (1, 2, 3)
<b>Ordered</b>	Elements have defined order	Access by index
<b>Duplicates</b>	Allows duplicate values	(1, 1, 2)
<b>Indexing</b>	Access elements by position	t[0]

```
\# Program to swap two tuple values
def swap\_tuple\_values(tup, pos1, pos2):
    \# Convert tuple to list for swapping
    temp\_list = list(tup)

    \# Swap values
    temp\_list[pos1], temp\_list[pos2] = temp\_list[pos2], temp\_list[pos1]

    \# Convert back to tuple
    return tuple(temp\_list)

\# Example usage
original\_tuple = (10, 20, 30, 40, 50)
print("Original tuple:", original\_tuple)

\# Swap values at positions 1 and 3
swapped\_tuple = swap\_tuple\_values(original\_tuple, 1, 3)
print("After swapping:", swapped\_tuple)
```

- **Immutable:** Cannot modify once created
- **Ordered:** Maintains element sequence
- **Heterogeneous:** Can store different data types

#### Mnemonic

“Tuples are Immutable and Ordered”

### Question 1(c OR) [7 marks]

What is Dictionary in Python? Write a Python program to traverse a dictionary using loop.

#### Solution

**Dictionary** is an unordered collection of key-value pairs with unique keys.

**Table of Dictionary Methods:**

Method	Purpose	Example
<b>keys()</b>	Get all keys	dict.keys()
<b>values()</b>	Get all values	dict.values()
<b>items()</b>	Get key-value pairs	dict.items()
<b>get()</b>	Safe key access	dict.get('key')

```

\# Program to traverse dictionary using loops
student\_marks = \{
    {Alice}: 85,
    {Bob}: 92,
    {Charlie}: 78,
    {Diana}: 96,
    {Eve}: 89
\}

print("Dictionary Traversal Methods:")
print("{-"} * 30)

\# Method 1: Traverse keys only
print("{n}1. Keys only:")
for key in student\_marks:
    print(f"    \{key}\}")

\# Method 2: Traverse values only
print("{n}2. Values only:")
for value in student\_marks.values():
    print(f"    \{value}\}")

\# Method 3: Traverse key{-value pairs}
print("{n}3. Key{-Value pairs}")
for key, value in student\_marks.items():
    print(f"    \{key}\}: \{value}\}")

\# Method 4: Using keys() method
print("{n}4. Using keys() method:")
for key in student\_marks.keys():
    print(f"    \{key}\ scored \{student\_marks[key]}\}")



- Key-Value storage: Each key maps to a value
- Unique keys: No duplicate keys allowed
- Fast lookup: O(1) average time complexity

```

### Mnemonic

“Dicts map Keys to Values”

## Question 2(a) [3 marks]

What is Package? List out advantages of using Package.

### Solution

**Package** is a directory containing multiple modules organized together.

**Table of Package Advantages:**

Advantage	Description
<b>Organization</b>	Groups related modules together
<b>Namespace</b>	Avoids naming conflicts
<b>Reusability</b>	Code can be reused across projects
<b>Maintainability</b>	Easier to manage large codebases
<b>Distribution</b>	Easy to share and install

- **Modular structure:** Better code organization
- **Hierarchical namespace:** Prevents name conflicts
- **Code reuse:** Promotes software reusability

### Mnemonic

“Packages Organize Related Modules”

### Question 2(b) [4 marks]

Explain any two package import method with example.

#### Solution

##### Table of Import Methods:

Method	Syntax	Usage
Normal Import	<code>import package.module</code>	Access with full path
From Import	<code>from package import module</code>	Direct module access
Specific Import	<code>from package.module import function</code>	Import specific items
Wildcard Import	<code>from package import *</code>	Import all modules

```
\# Method 1: Normal Import
import mypackage.calculator
result = mypackage.calculator.add(5, 3)
print(f"Normal import result: \{result}\")
```

```
\# Method 2: From Import
from mypackage import calculator
result = calculator.multiply(4, 6)
print(f"From import result: \{result}\")
```

- **Normal import:** Requires full package path
- **From import:** Allows direct module access
- **Specific function import:** Import only needed functions

### Mnemonic

“Import Normally or From Package”

### Question 2(c) [7 marks]

Explain about intra-package reference with example.

#### Solution

**Intra-package reference** allows modules within a package to import from each other.

##### Diagram of Package Structure:

```
mypackage/
  \_ \_ init \_ \_ .py
  math \_ ops /
    \_ \_ init \_ \_ .py
    basic.py
    advanced.py
  utils /
    \_ \_ init \_ \_ .py
    helpers.py
```

##### Table of Reference Types:

Type	Syntax	Usage
Absolute	<code>from mypackage.math_ops import basic</code>	Full path from package root

<b>Relative</b>	<code>from . import basic</code>	Current package
<b>Parent</b>	<code>from .. import utils</code>	Parent package
<b>Sibling</b>	<code>from ..utils import helpers</code>	Sibling package

```
\# Package structure example
\# mypackage/math\_ops/advanced.py
from . import basic \# Relative import from same package
from ..utils import helpers \# Import from sibling package

def power\_operation(base, exp):
    \# Using function from basic module
    if basic.is\_valid\_number(base) and basic.is\_valid\_number(exp):
        result = base ** exp
        \# Using helper function
        return helpers.format\_result(result)
    return None

\# mypackage/math\_ops/basic.py
def is\_valid\_number(num):
    return isinstance(num, (int, float))

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

\# mypackage/utils/helpers.py
def format\_result(value):
    return f"Result: \{value:.2f\}"
```

- **Relative imports:** Use dots (.) for current package
- **Absolute imports:** Full package path
- **Package hierarchy:** Navigate using dot notation

### Mnemonic

“Dots Navigate Package Levels”

## Question 2(a OR) [3 marks]

What is Module? List out advantages of using Module.

### Solution

**Module** is a Python file containing definitions, statements, and functions.

**Table of Module Advantages:**

Advantage	Description
<b>Code Reusability</b>	Write once, use multiple times
<b>Namespace</b>	Separate namespace for functions
<b>Organization</b>	Better code structure
<b>Maintainability</b>	Easier to debug and update
<b>Collaboration</b>	Multiple developers can work

- **Reusable code:** Functions can be imported anywhere
- **Modular design:** Break large programs into smaller parts
- **Easy maintenance:** Changes in one place affect all imports

### Mnemonic

“Modules Make Code Reusable”

## Question 2(b OR) [4 marks]

Explain any two module import method with example.

### Solution

Table of Module Import Methods:

Method	Syntax	Access Pattern
Direct Import	<code>import module_name</code>	<code>module_name.function()</code>
From Import	<code>from module_name import function</code>	<code>function()</code>
Alias Import	<code>import module_name as alias</code>	<code>alias.function()</code>
Wildcard Import	<code>from module_name import *</code>	<code>function()</code>

```
\# Method 1: Direct Import
import math
result1 = math.sqrt(16)
print(f"Direct import: \{result1}\")
```

```
\# Method 2: From Import
from math import pi, sin
result2 = sin(pi/2)
print(f"From import: \{result2}\")
```

- **Direct import:** Access with module name prefix
- **From import:** Direct function access without prefix
- **Namespace control:** Choose appropriate import method

### Mnemonic

“Import Directly or From Module”

## Question 2(c OR) [7 marks]

Write a program to define a module to find the area and circumference of a circle.

### Solution

```
\# circle\_operations.py (Module file)
import math

def area(radius):
    """Calculate area of circle"""
    if radius {=} 0:
        return 0
    return math.pi * radius * radius

def circumference(radius):
    """Calculate circumference of circle"""
    if radius {=} 0:
        return 0
    return 2 * math.pi * radius

def display\_info(radius):
    """Display circle information"""
    print(f"Circle with radius: \{radius}\")
    print(f"Area: \{area(radius):.2f}\")
    print(f"Circumference: \{circumference(radius):.2f}\")
```

```

\# Constants
PI = math.pi

\# a) Import the module to another program
\# main\_program.py
import circle\_operations

radius = 5
print("Method 1: Import entire module")
area\_result = circle\_operations.area(radius)
circumference\_result = circle\_operations.circumference(radius)

print(f"Area: \{area\_result:.2f}\}")
print(f"Circumference: \{circumference\_result:.2f}\}")

\# b) Import specific function from module
\# specific\_import.py
from circle\_operations import area, circumference

radius = 7
print("\nMethod 2: Import specific functions")
area\_result = area(radius)
circumference\_result = circumference(radius)

print(f"Area: \{area\_result:.2f}\}")
print(f"Circumference: \{circumference\_result:.2f}\}")

```

#### Table of Module Features:

Feature	Implementation
<b>Functions</b>	area(), circumference()
<b>Error Handling</b>	Check for negative radius
<b>Constants</b>	PI value
<b>Documentation</b>	Function docstrings

- **Module creation:** Save functions in .py file
- **Import flexibility:** Whole module or specific functions
- **Code reuse:** Use same functions in multiple programs

#### Mnemonic

“Modules Contain Reusable Functions”

### Question 3(a) [3 marks]

Explain the types of error in Python.

#### Solution

##### Table of Python Error Types:

Error Type	Description	Example
<b>Syntax Error</b>	Wrong Python syntax	Missing colon :
<b>Runtime Error</b>	Occurs during execution	Division by zero
<b>Logical Error</b>	Wrong program logic	Incorrect algorithm
<b>Name Error</b>	Undefined variable	Using undeclared variable
<b>Type Error</b>	Wrong data type operation	String + Integer

- **Syntax errors:** Detected before program runs
- **Runtime errors:** Occur during program execution
- **Logical errors:** Program runs but gives wrong results

#### Mnemonic

“Syntax, Runtime, Logic Errors”

### Question 3(b) [4 marks]

Explain user-defined exception using raise statement with example.

#### Solution

User-defined exceptions are custom error classes created by programmers.

**Table of Exception Components:**

Component	Purpose	Example
Class Definition	Create custom exception	<code>class CustomError(Exception):</code>
Raise Statement	Trigger the exception	<code>raise CustomError("message")</code>
Error Message	Describe the problem	Informative text
Exception Handling	Catch custom exception	<code>except CustomError:</code>

```
\# Define custom exception
class AgeValidationError(Exception):
    def \_\_init\_\_(self, age, message="Invalid age provided"):
        self.age = age
        self.message = message
        super().\_\_init\_\_(self.message)

def validate\_age(age):
    if age {} 0:
        raise AgeValidationError(age, "Age cannot be negative")
    elif age {} 150:
        raise AgeValidationError(age, "Age cannot exceed 150")
    else:
        print(f"Valid age: \{age}\")

\# Using the custom exception
try:
    validate\_age(-5)
except AgeValidationError as e:
    print(f"Error: \{e.message}\, Age: \{e.age}\")
```

- **Custom exception class:** Inherits from Exception
- **Raise statement:** Manually trigger exceptions
- **Meaningful messages:** Help debug problems

#### Mnemonic

“Raise Custom Exceptions for Validation”

### Question 3(c) [7 marks]

Explain try-except-finally clause with example.



## Solution

**Try-except-finally** provides complete exception handling mechanism.

**Table of Exception Handling Blocks:**

Block	Purpose	Execution
<b>try</b>	Code that might raise exception	Always executed first
<b>except</b>	Handle specific exceptions	Only if exception occurs
<b>else</b>	Code when no exception	Only if no exception
<b>finally</b>	Cleanup code	Always executed

```
\# Complete exception handling example
def divide\_numbers():
    try:
        print("Starting division operation...")

        \# Get input from user
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))

        \# Perform division
        result = num1 / num2

    except ValueError:
        print("Error: Please enter valid numbers only")
        return None

    except ZeroDivisionError:
        print("Error: Cannot divide by zero")
        return None

    except Exception as e:
        print(f"Unexpected error occurred: \{e\}")
        return None

    else:
        print(f"Division successful: \{num1\} \{num2\} = \{result\}")
        return result

    finally:
        print("Division operation completed")
        print("Cleaning up resources...")

\# Example usage
result = divide\_numbers()
if result:
    print(f"Final result: \{result\}")
```

**Flow Diagram:**

```
flowchart LR
    A["try block"] --> B["Exception occurred?"]
    B -- Yes --> C["except block"]
    B -- No --> D["else block"]
    C --> E["finally block"]
    D --> E
    E --> F["End"]
```

- **try:** Contains risky code
- **except:** Handles specific errors
- **finally:** Always executes for cleanup

### Mnemonic

“Try-Except-Finally Always Cleans”

### Question 3(a OR) [3 marks]

What is built-in exception? List out any two with their meaning.

#### Solution

**Built-in exceptions** are predefined error types in Python.

**Table of Built-in Exceptions:**

Exception	Meaning	Example
<b>ValueError</b>	Invalid value for correct type	<code>int("abc")</code>
<b>TypeError</b>	Wrong data type operation	<code>"5" + 5</code>
<b>IndexError</b>	List index out of range	<code>list[10]</code> for 5-item list
<b>KeyError</b>	Dictionary key not found	<code>dict["missing_key"]</code>
<b>ZeroDivisionError</b>	Division by zero	<code>10 / 0</code>

**Two Main Built-in Exceptions:**

- **ValueError:** Occurs when function receives correct type but invalid value
- **TypeError:** Occurs when operation performed on inappropriate data type

### Mnemonic

“Built-in Exceptions Handle Common Errors”

### Question 3(b OR) [4 marks]

Explain try-except clause with example.

#### Solution

**Try-except** handles exceptions that might occur during program execution.

**Table of Exception Handling:**

Component	Purpose	Syntax
<b>try</b>	Code that might fail	<code>try:</code>
<b>except</b>	Handle specific exception	<code>except ErrorType:</code>
<b>Multiple except</b>	Handle different errors	Multiple except blocks
<b>General except</b>	Catch any exception	<code>except:</code>

```

\# Example of try{-except clause}
def safe\_division():
    try:
        \# Code that might raise exceptions
        dividend = int(input("Enter dividend: "))
        divisor = int(input("Enter divisor: "))

        result = dividend / divisor
        print(f"Result: \{dividend\} \{divisor\} = \{result\}")

    except ValueError:
        print("Error: Please enter valid integers only")

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

    except Exception as e:
        print(f"An unexpected error occurred: \{e\}")

    print("Program continues after exception handling")

\# Example usage
safe\_division()

```

- **try block:** Contains potentially risky code
- **except block:** Handles specific exception types
- **Multiple handlers:** Different exceptions handled differently

#### Mnemonic

“Try Risky Code, Except Handles Errors”

### Question 3(c OR) [7 marks]

Write a program to catch on Divide by zero Exception with finally clause.

#### Solution

```

\# Program to handle divide by zero with finally clause
def advanced\_calculator():
    """Calculator with comprehensive exception handling"""

    try:
        print("=== Advanced Calculator ===")
        print("Enter two numbers for division")

        \# Input section
        numerator = float(input("Enter numerator: "))
        denominator = float(input("Enter denominator: "))

        print(f"\{n\}Attempting to divide \{numerator\} by \{denominator\}...")

        \# Critical operation that might fail
        if denominator == 0:
            raise ZeroDivisionError("Division by zero is not allowed")

        result = numerator / denominator

        \# Success message
        print(f" Division successful!")
        print(f" Result: \{numerator\} \{denominator\} = \{result:.4f\}")
    
```

```

        return result

except ZeroDivisionError as zde:
    print(f" Zero Division Error: \{zde}\}")
    print(" Please use a non{-zero denominator}")
    return None

except ValueError as ve:
    print(f" Value Error: Invalid input provided")
    print(" Please enter numeric values only")
    return None

except Exception as e:
    print(f" Unexpected error: \{e}\}")
    return None

finally:
    print("{n}" + "="*40)
    print("CLEANUP OPERATIONS:")
    print("{- Closing calculator session}")
    print("{- Saving operation log}")
    print("{- Releasing memory resources}")
    print("{- Calculator shutdown complete}")
    print("="*40)

\# Example usage with multiple test cases
def test\_calculator():
    """Test the calculator with different scenarios"""

    test\_cases = [
        (10, 2),    \# Normal division
        (15, 0),    \# Zero division
        (7, 3),     \# Normal division with decimal
    ]

    for i, (num, den) in enumerate(test\_cases, 1):
        print(f"{n}{-}{-}{-} Test Case }\{i}\ {-}{-}{-}")
        print(f"Testing: \{num}\ \{den}\")

        \# Simulate user input (in real scenario, this would be interactive)
        \# This is just for demonstration
        print(f"Simulating division of \{num}\ by \{den}\")

        try:
            if den == 0:
                raise ZeroDivisionError("Cannot divide by zero")
            result = num / den
            print(f"Result: \{result}\")
        except ZeroDivisionError as e:
            print(f"Error caught: \{e}\")
        finally:
            print("Test case completed")

\# Run the calculator
if \_\_name\_\_ == "\_\_main\_\_":
    result = advanced\_calculator()

    if result is not None:
        print(f"{n}Final calculated result: \{result}\")
    else:
        print("{n}Calculation failed due to errors")

```

### Table of Exception Handling Features:

Feature	Implementation
<b>ZeroDivisionError</b>	Specific handling for division by zero
<b>ValueError</b>	Handle invalid input types
<b>Generic Exception</b>	Catch unexpected errors
<b>Finally Block</b>	Always execute cleanup code

### Exception Handling Flow:

flowchart LR

```
A[Start Division] --> B[try block]
B --> C{Exception?}
C -- ZeroDivisionError --> D[Handle Zero Division]
C -- ValueError --> E[Handle Invalid Input]
C -- Other --> F[Handle Generic Error]
C -- None --> G[Success Result]
D --> H[finally block]
E --> H
F --> H
G --> H
H --> I[End Program]
```

- **Specific exception handling:** ZeroDivisionError caught separately
- **Finally clause:** Always executes for cleanup
- **Resource management:** Proper cleanup regardless of errors

### Mnemonic

“Finally Always Cleans Up Resources”

### Question 4(a) [3 marks]

Define: File, Binary File, Text File

### Solution

#### Table of File Definitions:

Term	Definition	Example
<b>File</b>	Named storage location on disk	document.txt, image.jpg
<b>Binary File</b>	Contains non-text data in binary format	.exe, .jpg, .mp3, .pdf
<b>Text File</b>	Contains human-readable text characters	.txt, .py, .html, .csv

#### Detailed Definitions:

- **File:** A collection of data stored on storage device with a unique name
- **Binary File:** Stores data in binary format (0s and 1s), not human-readable
- **Text File:** Contains ASCII or Unicode characters, human-readable format

### Mnemonic

“Files store data, Binary=Machine, Text=Human”

### Question 4(b) [4 marks]

Explain write() and writelines() function with example.

## Solution

### Table of Write Functions:

Function	Purpose	Parameter	Usage
<code>write()</code>	Write single string	String	<code>file.write("Hello")</code>
<code>writelines()</code>	Write list of strings	List/Sequence	<code>file.writelines(["line1", "line2"])</code>

```
\# Example demonstrating write() and writelines()
def demonstrate_write_functions():

    \# Using write() function
    with open("write_demo.txt", "w") as file:
        file.write("Hello World!\n")
        file.write("This is line 2\n")
        file.write("This is line 3\n")

    \# Using writelines() function
    lines = [
        "First line using writelines\n",
        "Second line using writelines\n",
        "Third line using writelines\n"
    ]

    with open("writelines_demo.txt", "w") as file:
        file.writelines(lines)

    print("Files created successfully!")

\# Run the demonstration
demonstrate_write_functions()
```

### Key Differences:

- **write()**: Writes one string at a time
- **writelines()**: Writes multiple strings from a sequence
- **Newlines**: Must be added manually with `\n`
- **Return value**: Both return number of characters written

## Mnemonic

“write() Single, writelines() Multiple”

## Question 4(c) [7 marks]

Explain `tell()` and `seek()` function with example.

## Solution

**File pointer functions** control position within a file for reading/writing.

### Table of Position Functions:

Function	Purpose	Return/Parameter	Usage
<code>tell()</code>	Get current position	Returns current byte position	<code>pos = file.tell()</code>
<code>seek(offset, whence)</code>	Move to specific position	offset: bytes, whence: reference	<code>file.seek(10, 0)</code>

**Seek Whence Values:**

<b>Value</b>	<b>Reference Point</b>	<b>Description</b>
<b>0</b>	Beginning of file	Absolute positioning
<b>1</b>	Current position	Relative to current
<b>2</b>	End of file	Relative to end

```

\# Complete example of tell() and seek() functions
def demonstrate\_file\_positioning():

    \# Create a sample file
    sample\_text = "Hello World! This is a sample file for demonstrating tell() and seek() functions."

    with open("position\_demo.txt", "w") as file:
        file.write(sample\_text)

    \# Demonstrate tell() and seek()
    with open("position\_demo.txt", "r") as file:

        \# Initial position
        print(f"1. Initial position: \{file.tell()\}")

        \# Read first 5 characters
        data1 = file.read(5)
        print(f"2. Read \{\{data1\}\}, current position: \{file.tell()\}")

        \# Move to position 15
        file.seek(15)
        print(f"3. After seek(15), position: \{file.tell()\}")

        \# Read next 10 characters
        data2 = file.read(10)
        print(f"4. Read \{\{data2\}\}, current position: \{file.tell()\}")

        \# Move to beginning using seek(0, 0)
        file.seek(0, 0)
        print(f"5. After seek(0,0), position: \{file.tell()\}")

        \# Move to end using seek(0, 2)
        file.seek(0, 2)
        print(f"6. After seek(0,2), position: \{file.tell()\}")

        \# Move backward from current position
        file.seek(-10, 1)
        print(f"7. After seek(-10,1), position: \{file.tell()\}")

        \# Read remaining content
        remaining = file.read()
        print(f"8. Remaining content: \{\{remaining\}\}")

\# Example with binary file
def binary\_file\_positioning():

    \# Create binary file
    data = b"Binary file positioning example"

    with open("binary\_demo.bin", "wb") as file:
        file.write(data)

    \# Read with positioning
    with open("binary\_demo.bin", "rb") as file:
        print(f"\nBinary file position: \{file.tell()\}")

        \# Read first 6 bytes
        chunk1 = file.read(6)
        print(f"Read: \{chunk1\}, position: \{file.tell()\}")

        \# Jump to position 20

```



```

file.seek(20)
chunk2 = file.read(7)
print(f"Read: \{chunk2\}, position: \{file.tell()\}")

```

```

\# Run demonstrations
demonstrate\_file\_positioning()
binary\_file\_positioning()

```

#### Position Control Diagram:

```

flowchart LR
    A[File Start] --> B["tell(): 0"]
    B --> C["read(5)"]
    C --> D["tell(): 5"]
    D --> E["seek(15)"]
    E --> F["tell(): 15"]
    F --> G["seek(0,2)"]
    G --> H["tell(): End"]

```

- **tell():** Returns current byte position in file
- **seek():** Moves file pointer to specified position
- **Positioning:** Essential for random file access
- **Binary mode:** Works with byte positions

#### Mnemonic

“tell() Position, seek() Movement”

### Question 4(a OR) [3 marks]

What is Absolute and Relative file path?

#### Solution

##### Table of Path Types:

Path Type	Description	Example
<b>Absolute Path</b>	Complete path from root directory	/home/user/documents/file.txt
<b>Relative Path</b>	Path relative to current directory	../documents/file.txt

##### Path Symbols:

Symbol	Meaning	Example
/	Root directory (Linux/Mac)	/home/user/
C:\	Drive letter (Windows)	C:\\Users\\Documents\\
.	Current directory	./file.txt
..	Parent directory	../folder/file.txt

- **Absolute:** Complete path from system root
- **Relative:** Path from current working directory

#### Mnemonic

“Absolute from Root, Relative from Current”

### Question 4(b OR) [4 marks]

Explain about various mode to open binary and text file.

## Solution

### Table of File Opening Modes:

Mode	Type	Purpose	File Pointer
'r'	Text	Read only	Beginning
'w'	Text	Write (overwrites)	Beginning
'a'	Text	Append	End
'rb'	Binary	Read binary	Beginning
'wb'	Binary	Write binary	Beginning
'ab'	Binary	Append binary	End
'r+'	Text	Read and write	Beginning
'w+'	Text	Write and read	Beginning

```
\# Examples of different file modes
def demonstrate\_file\_modes():

    \# Text file modes
    with open("text\_file.txt", "w") as f: \# Write mode
        f.write("Hello World")

    with open("text\_file.txt", "r") as f: \# Read mode
        content = f.read()
        print(f"Text content: \{content\}")

    \# Binary file modes
    data = b"Binary data example"
    with open("binary\_file.bin", "wb") as f: \# Write binary
        f.write(data)

    with open("binary\_file.bin", "rb") as f: \# Read binary
        binary\_content = f.read()
        print(f"Binary content: \{binary\_content\}")

demonstrate\_file\_modes()
```

- **Text modes:** Handle string data with encoding
- **Binary modes:** Handle raw bytes without encoding
- **Plus modes:** Allow both reading and writing

## Mnemonic

“Text for Strings, Binary for Bytes”

## Question 4(c OR) [7 marks]

Write a Python program to write student's subject record like branch name, semester, subject code and subject name in the binary file.

## Solution

```
import pickle
import os

class StudentSubjectRecord:
    """Class to handle student subject records"""

    def __init__(self, branch\_name, semester, subject\_code, subject\_name):
        self.branch\_name = branch\_name
        self.semester = semester
        self.subject\_code = subject\_code
```

```

        self.subject\_name = subject\_name

def \_\_str\_\_(self):
    return f"Branch: \{self.branch\_name\}, Semester: \{self.semester\}, Code: \{self.subject\_code\}"

def write\_student\_records():
    """Write student records to binary file"""

    \# Sample student records
    records = [
        StudentSubjectRecord("Information Technology", 2, "4321602", "Advanced Python Programming"),
        StudentSubjectRecord("Information Technology", 2, "4321601", "Database Management System"),
        StudentSubjectRecord("Computer Engineering", 3, "4330701", "Data Structure"),
        StudentSubjectRecord("Information Technology", 2, "4321603", "Web Development"),
        StudentSubjectRecord("Computer Engineering", 3, "4330702", "Computer Networks")
    ]

    \# Write records to binary file using pickle
    try:
        with open("student\_records.bin", "wb") as binary\_file:
            pickle.dump(records, binary\_file)

        print(" Student records written to binary file successfully!")
        print(f" Total records written: \{len(records)\}")

    except Exception as e:
        print(f" Error writing to binary file: \{e\}")

def read\_student\_records():
    """Read student records from binary file"""

    try:
        if not os.path.exists("student\_records.bin"):
            print(" Binary file not found!")
            return

        with open("student\_records.bin", "rb") as binary\_file:
            records = pickle.load(binary\_file)

        print("{n}" + "="*60)
        print("STUDENT SUBJECT RECORDS FROM BINARY FILE")
        print("="*60)

        for i, record in enumerate(records, 1):
            print(f"\{i\}. \{record\}")

        print("="*60)
        print(f"Total records read: \{len(records)\}")

    except Exception as e:
        print(f" Error reading from binary file: \{e\}")

def add\_new\_record():
    """Add new student record to existing binary file"""

    try:
        \# Read existing records
        records = []
        if os.path.exists("student\_records.bin"):
            with open("student\_records.bin", "rb") as binary\_file:
                records = pickle.load(binary\_file)

```

```

    \# Get new record details
    print("{n}Enter new student record details:")
    branch = input("Branch Name: ")
    semester = int(input("Semester: "))
    code = input("Subject Code: ")
    subject = input("Subject Name: ")

    \# Create new record
    new\_record = StudentSubjectRecord(branch, semester, code, subject)
    records.append(new\_record)

    \# Write back to file
    with open("student\_records.bin", "wb") as binary\_file:
        pickle.dump(records, binary\_file)

    print(" New record added successfully!")

except Exception as e:
    print(f" Error adding new record: \{e\}")

def search\_records\_by\_branch(branch\_name):
    """Search records by branch name"""

    try:
        if not os.path.exists("student\_records.bin"):
            print(" Binary file not found!")
            return

        with open("student\_records.bin", "rb") as binary\_file:
            records = pickle.load(binary\_file)

        \# Filter records by branch
        filtered\_records = [record for record in records
                            if record.branch\_name.lower() == branch\_name.lower()]

        if filtered\_records:
            print(f"{n}Records for \{branch\_name\}:")
            print("{-"} * 40)
            for record in filtered\_records:
                print(f" \{record\}")
        else:
            print(f"No records found for branch: \{branch\_name\}")

    except Exception as e:
        print(f" Error searching records: \{e\}")

\# Main program execution
def main():
    """Main function to demonstrate binary file operations"""

    print("=== STUDENT SUBJECT RECORD MANAGEMENT ==={n}")

    \# Write initial records
    print("1. Writing student records to binary file...")
    write\_student\_records()

    \# Read and display records
    print("{n}2. Reading records from binary file...")
    read\_student\_records()

    \# Search by branch

```

```

print("\n3. Searching records by branch...")
search\_records\_by\_branch("Information Technology")

\# Display file information
if os.path.exists("student\_records.bin"):
    file\_size = os.path.getsize("student\_records.bin")
    print(f"\nBinary file size: \{file\_size\} bytes")

\# Execute the program
if \_\_name\_\_ == "\_\_main\_\_":
    main()

```

#### Table of Binary File Operations:

Operation	Method	Purpose
Write	<code>pickle.dump()</code>	Serialize objects to binary
Read	<code>pickle.load()</code>	Deserialize objects from binary
Append	Read + Add + Write	Add new records
Search	Filter loaded data	Find specific records

#### Binary File Structure:

```

flowchart LR
    A[Student Records] --> B["pickle.dump()"]
    B --> C[Binary File .bin]
    C --> D["pickle.load()"]
    D --> E[Python Objects]

```

- **Binary storage:** Uses pickle for object serialization
- **Efficient storage:** Compact binary format
- **Object preservation:** Maintains data structure integrity
- **Cross-platform:** Works on different operating systems

#### Mnemonic

“Pickle Preserves Python Objects”

### Question 5(a) [3 marks]

Define: GUI, CLI

#### Solution

##### Table of Interface Definitions:

Term	Full Form	Description	Example
GUI	Graphical User Interface	Visual interface with windows, buttons, icons	Windows, Mac desktop
CLI	Command Line Interface	Text-based interface using commands	Terminal, Command Prompt

##### Key Differences:

- **GUI:** User-friendly, mouse-driven, visual elements
- **CLI:** Text-based, keyboard-driven, command syntax
- **Interaction:** GUI uses clicks, CLI uses typed commands

#### Mnemonic

“GUI Graphics, CLI Commands”

### Question 5(b) [4 marks]

Write a Python program to draw square shape using for and while loop using Turtle.

#### Solution

```
import turtle

def draw\_square\_with\_for\_loop():
    """Draw square using for loop"""

    \# Setup turtle
    screen = turtle.Screen()
    screen.bgcolor("white")
    square\_turtle = turtle.Turtle()
    square\_turtle.color("blue")
    square\_turtle.pensize(3)

    \# Draw square using for loop
    print("Drawing square with for loop...")
    side\_length = 100

    for i in range(4):
        square\_turtle.forward(side\_length)
        square\_turtle.right(90)

    square\_turtle.penup()
    square\_turtle.goto(150, 0)
    square\_turtle.pendown()

    return square\_turtle

def draw\_square\_with\_while\_loop(turtle\_obj):
    """Draw square using while loop"""

    \# Change color for second square
    turtle\_obj.color("red")

    \# Draw square using while loop
    print("Drawing square with while loop...")
    side\_length = 100
    sides\_drawn = 0

    while sides\_drawn < 4:
        turtle\_obj.forward(side\_length)
        turtle\_obj.right(90)
        sides\_drawn += 1

    \# Move turtle to center for text
    turtle\_obj.penup()
    turtle\_obj.goto(0, 0)
    turtle\_obj.write("Blue: for loop, Red: while loop",
                    font=("Arial", 12, "normal"))

\# Main execution
def main():
    \# Draw squares
    turtle\_obj = draw\_square\_with\_for\_loop()
    draw\_square\_with\_while\_loop(turtle\_obj)

    \# Keep window open
    turtle.Screen().exitonclick()
```

```
\# Run the program
main()
```

#### Table of Loop Comparison:

Loop Type	Structure	Usage	Control
for loop	for i in range(4):	Known iterations	Counter-based
while loop	while condition:	Conditional iterations	Condition-based

- **for loop:** Best for known number of iterations
- **while loop:** Best for condition-based repetition
- **Both achieve:** Same square drawing result

#### Mnemonic

“For Count, While Condition”

### Question 5(c) [7 marks]

Write a Python program to draw a chessboard using Turtle.

#### Solution

```
import turtle

def setup\_chessboard():
    """Setup turtle screen and properties for chessboard"""

    screen = turtle.Screen()
    screen.bgcolor("white")
    screen.title("Chessboard using Python Turtle")
    screen.setup(width=600, height=600)

    \# Create turtle for drawing
    chess\_turtle = turtle.Turtle()
    chess\_turtle.speed(0) \# Fastest speed
    chess\_turtle.penup()

    return screen, chess\_turtle

def draw\_square(turtle\_obj, size, fill\_color):
    """Draw a single square with given color"""

    turtle\_obj.pendown()
    turtle\_obj.fillcolor(fill\_color)
    turtle\_obj.begin\_fill()

    \# Draw square
    for \_ in range(4):
        turtle\_obj.forward(size)
        turtle\_obj.right(90)

    turtle\_obj.end\_fill()
    turtle\_obj.penup()

def draw\_chessboard():
    """Draw complete 8x8 chessboard"""

    screen, chess\_turtle = setup\_chessboard()
```

```

\# Chessboard parameters
square\_size = 40
board\_size = 8
start\_x = {-}160
start\_y = 160

print("Drawing chessboard...")

\# Draw the board
for row in range(board\_size):
    for col in range(board\_size):

        \# Calculate position
        x = start\_x + (col * square\_size)
        y = start\_y {-} (row * square\_size)

        \# Move turtle to position
        chess\_turtle.goto(x, y)

        \# Determine square color (alternating pattern)
        if (row + col) \% 2 == 0:
            color = "white"
        else:
            color = "black"

        \# Draw the square
        draw\_square(chess\_turtle, square\_size, color)

\# Add border around chessboard
draw\_border(chess\_turtle, start\_x, start\_y, square\_size, board\_size)

\# Add labels
add\_chessboard\_labels(chess\_turtle, start\_x, start\_y, square\_size, board\_size)

return screen

def draw\_border(turtle\_obj, start\_x, start\_y, square\_size, board\_size):
    """Draw border around the chessboard"""

    turtle\_obj.goto(start\_x {-} 5, start\_y + 5)
    turtle\_obj.pencolor("brown")
    turtle\_obj.pensize(3)
    turtle\_obj.pendown()

    \# Draw border rectangle
    border\_width = board\_size * square\_size + 10
    border\_height = board\_size * square\_size + 10

    for \_ in range(2):
        turtle\_obj.forward(border\_width)
        turtle\_obj.right(90)
        turtle\_obj.forward(border\_height)
        turtle\_obj.right(90)

    turtle\_obj.penup()
    turtle\_obj.pensize(1)
    turtle\_obj.pencolor("black")

def add\_chessboard\_labels(turtle\_obj, start\_x, start\_y, square\_size, board\_size):
    """Add row and column labels to chessboard"""

```



```

turtle\_obj.color("blue")

\# Column labels (A{-H})
columns = [{A}, {B}, {C}, {D}, {E}, {F}, {G}, {H}]
for i, letter in enumerate(columns):
    x = start\_x + (i * square\_size) + (square\_size // 2)
    y = start\_y {-} (board\_size * square\_size) {-} 20
    turtle\_obj.goto(x, y)
    turtle\_obj.write(letter, align="center", font=("Arial", 12, "bold"))

\# Row labels (1{-8})
for i in range(board\_size):
    x = start\_x {-} 20
    y = start\_y {-} (i * square\_size) {-} (square\_size // 2)
    turtle\_obj.goto(x, y)
    turtle\_obj.write(str(8{-}i), align="center", font=("Arial", 12, "bold"))

\# Title
turtle\_obj.goto(0, start\_y + 30)
turtle\_obj.write("Python Turtle Chessboard", align="center",
                 font=("Arial", 16, "bold"))

def draw\_enhanced\_chessboard\_with\_pieces():
    """Enhanced version with some chess piece indicators"""

    screen = draw\_chessboard()

    \# Create new turtle for pieces
    piece\_turtle = turtle.Turtle()
    piece\_turtle.speed(0)
    piece\_turtle.penup()

    \# Add some piece symbols (simplified)
    pieces = [
        ({-}120, 120, " "), ({-}80, 120, " "), ({-}40, 120, " "), (0, 120, " "),
        ({-}120, {-}120, " "), ({-}80, {-}120, " "), ({-}40, {-}120, " "), (0, {-}120, " ")
    ]

    piece\_turtle.color("red")
    for x, y, symbol in pieces:
        piece\_turtle.goto(x, y)
        piece\_turtle.write(symbol, align="center", font=("Arial", 20, "normal"))

    piece\_turtle.hideturtle()

    \# Display statistics
    total\_squares = 64
    black\_squares = 32
    white\_squares = 32

    piece\_turtle.goto(0, {-}200)
    piece\_turtle.color("green")
    piece\_turtle.write(f"Chessboard: \{total\_squares\} squares (\{black\_squares\} black, \{white\_squares\} white)",
                      align="center", font=("Arial", 12, "normal"))

    return screen

\# Main execution
def main():
    """Main function to create chessboard"""

```

```

print("Creating chessboard with Python Turtle...")
screen = draw\_enhanced\_chessboard\_with\_pieces()

print("Chessboard created successfully!")
print("Click on the screen to close the window.")

\# Keep window open until clicked
screen.exitonclick()

\# Run the program
if \_\_name\_\_ == "\_\_main\_\_":
    main()

```

#### Table of Chessboard Components:

Component	Implementation	Purpose
<b>Squares</b>	8x8 grid alternating colors	Main board pattern
<b>Colors</b>	Black and white alternating	Traditional chess pattern
<b>Border</b>	Brown rectangle outline	Frame the board
<b>Labels</b>	A-H columns, 1-8 rows	Chess notation
<b>Pieces</b>	Unicode chess symbols	Sample piece placement

#### Chessboard Pattern Logic:

```

flowchart LR
    A[Start Row 0] --> B[For each row]
    B --> C[For each column]
    C --> D{row + col even?}
    D -- Yes --> E[White Square]
    D -- No --> F[Black Square]
    E --> G[Next Column]
    F --> G
    G --> C
    C -- Row Complete --> H[Next Row]
    H --> B
    B -- All Rows Done --> I[Complete Board]

```

- **Alternating pattern:**  $(\text{row} + \text{col}) \% 2$  determines color
- **Grid system:** 8x8 squares with precise positioning
- **Visual enhancements:** Border, labels, and sample pieces
- **Scalable design:** Easy to modify square size

#### Mnemonic

“Alternate Colors in Grid Pattern”

### Question 5(a OR) [3 marks]

How many types of shapes in turtle? Explain any one shape with suitable example.

#### Solution

##### Table of Turtle Shapes:

Shape Type	Examples	Method
<b>Basic Shapes</b>	Circle, Square, Triangle	Built-in functions
<b>Line Patterns</b>	Straight lines, Curves	forward(), backward()
<b>Polygons</b>	Pentagon, Hexagon, Octagon	Loop with angles
<b>Complex Shapes</b>	Stars, Spirals, Fractals	Mathematical patterns
<b>Custom Shapes</b>	User-defined patterns	Combination of moves

### Circle Shape Example:

```
import turtle

def draw\_circle\_example():
    screen = turtle.Screen()
    circle\_turtle = turtle.Turtle()

    \# Draw circle with radius 50
    circle\_turtle.circle(50)

    screen.exitonclick()

draw\_circle\_example()
```

- **Built-in shapes:** Circle, square, triangle readily available
- **Custom shapes:** Created using movement combinations
- **Mathematical shapes:** Use geometry for precise drawing

### Mnemonic

“Turtle Draws Many Shape Types”

### Question 5(b OR) [4 marks]

Explain about four basic methods of Turtle module.

### Solution

Table of Basic Turtle Methods:

Method	Purpose	Parameters	Example
<b>forward(distance)</b>	Move turtle forward	distance in pixels	<code>turtle.forward(100)</code>
<b>backward(distance)</b>	Move turtle backward	distance in pixels	<code>turtle.backward(50)</code>
<b>right(angle)</b>	Turn turtle right	angle in degrees	<code>turtle.right(90)</code>
<b>left(angle)</b>	Turn turtle left	angle in degrees	<code>turtle.left(45)</code>

```
import turtle

def demonstrate\_basic\_methods():
    \# Create turtle
    demo\_turtle = turtle.Turtle()

    \# 1. Forward movement
    demo\_turtle.forward(100) \# Move 100 pixels forward

    \# 2. Right turn
    demo\_turtle.right(90) \# Turn 90 degrees right

    \# 3. Backward movement
    demo\_turtle.backward(50) \# Move 50 pixels backward

    \# 4. Left turn
    demo\_turtle.left(135) \# Turn 135 degrees left

    turtle.done()

demonstrate\_basic\_methods()
```

- **Movement methods:** forward() and backward() for distance
- **Rotation methods:** right() and left() for direction changes
- **Coordinate system:** Based on current turtle position and heading
- **Angle measurement:** Degrees (0-360)

### Mnemonic

“Forward, Backward, Right, Left Basics”

## Question 5(c OR) [7 marks]

Write a Python program to draw square, rectangle, and circle using Turtle.

### Solution

```
import turtle
import math

def setup\_drawing\_environment():
    """Setup turtle screen and drawing environment"""

    screen = turtle.Screen()
    screen.bgcolor("lightblue")
    screen.title("Drawing Shapes: Square, Rectangle, Circle")
    screen.setup(width=800, height=600)

    \# Create main drawing turtle
    shape\_turtle = turtle.Turtle()
    shape\_turtle.speed(3)
    shape\_turtle.pensize(2)

    return screen, shape\_turtle

def draw\_square(turtle\_obj, size, color, position):
    """Draw a square with given size and color"""

    x, y = position
    turtle\_obj.penup()
    turtle\_obj.goto(x, y)
```

```

turtle\_obj.pendown()

turtle\_obj.color(color)
turtle\_obj.fillcolor(color)
turtle\_obj.begin\_fill()

\# Draw square using 4 equal sides
for \_ in range(4):
    turtle\_obj.forward(size)
    turtle\_obj.right(90)

turtle\_obj.end\_fill()

\# Add label
turtle\_obj.penup()
turtle\_obj.goto(x + size//2, y {-} 30)
turtle\_obj.color("black")
turtle\_obj.write(f"Square (\{size\}x\{size\})", align="center",
                 font=("Arial", 10, "bold"))

def draw\_rectangle(turtle\_obj, width, height, color, position):
    """Draw a rectangle with given dimensions and color"""

    x, y = position
    turtle\_obj.penup()
    turtle\_obj.goto(x, y)
    turtle\_obj.pendown()

    turtle\_obj.color(color)
    turtle\_obj.fillcolor(color)
    turtle\_obj.begin\_fill()

    \# Draw rectangle with alternating width and height
    for \_ in range(2):
        turtle\_obj.forward(width)
        turtle\_obj.right(90)
        turtle\_obj.forward(height)
        turtle\_obj.right(90)

    turtle\_obj.end\_fill()

    \# Add label
    turtle\_obj.penup()
    turtle\_obj.goto(x + width//2, y {-} height {-} 20)
    turtle\_obj.color("black")
    turtle\_obj.write(f"Rectangle (\{width\}x\{height\})", align="center",
                     font=("Arial", 10, "bold"))

def draw\_circle(turtle\_obj, radius, color, position):
    """Draw a circle with given radius and color"""

    x, y = position
    turtle\_obj.penup()
    turtle\_obj.goto(x, y {-} radius) \# Position at bottom of circle
    turtle\_obj.pendown()

    turtle\_obj.color(color)
    turtle\_obj.fillcolor(color)
    turtle\_obj.begin\_fill()

    \# Draw circle

```

```

turtle\_obj.circle(radius)

turtle\_obj.end\_fill()

\# Add label with area calculation
area = math.pi * radius * radius
turtle\_obj.penup()
turtle\_obj.goto(x, y {-} radius {-} 30)
turtle\_obj.color("black")
turtle\_obj.write(f"Circle (r={radius}, area={area:.1f})", align="center",
                  font=("Arial", 10, "bold"))

def draw\_all\_shapes():
    """Main function to draw all three shapes"""

    screen, shape\_turtle = setup\_drawing\_environment()

    print("Drawing geometric shapes...")

    \# Draw square
    print("1. Drawing square...")
    draw\_square(shape\_turtle, 80, "red", ({-}300, 100))

    \# Draw rectangle
    print("2. Drawing rectangle...")
    draw\_rectangle(shape\_turtle, 120, 80, "green", ({-}50, 100))

    \# Draw circle
    print("3. Drawing circle...")
    draw\_circle(shape\_turtle, 60, "blue", (200, 100))

    \# Add title and information
    add\_shape\_information(shape\_turtle)

    print("All shapes drawn successfully!")
    return screen

def add\_shape\_information(turtle\_obj):
    """Add title and shape information"""

    \# Title
    turtle\_obj.penup()
    turtle\_obj.goto(0, 200)
    turtle\_obj.color("purple")
    turtle\_obj.write("Python Turtle Shapes", align="center",
                     font=("Arial", 18, "bold"))

    \# Shape properties table
    turtle\_obj.goto({-}350, {-}50)
    turtle\_obj.color("black")
    turtle\_obj.write("Shape Properties:", font=("Arial", 12, "bold"))

    properties = [
        "• Square: 4 equal sides, 4 right angles",
        "• Rectangle: 4 sides, opposite sides equal",
        "• Circle: All points equidistant from center"
    ]

    for i, prop in enumerate(properties):
        turtle\_obj.goto({-}350, {-}80 {-} (i * 20))
        turtle\_obj.write(prop, font=("Arial", 10, "normal"))

```

```

\# Mathematical formulas
turtle\_obj.goto({-}350, {-}170)
turtle\_obj.color("blue")
turtle\_obj.write("Formulas:", font=("Arial", 12, "bold"))

formulas = [
    "• Square area: side{2}",
    "• Rectangle area: length width",
    "• Circle area: radius{2}"
]

for i, formula in enumerate(formulas):
    turtle\_obj.goto({-}350, {-}200 {-} (i * 20))
    turtle\_obj.write(formula, font=("Arial", 10, "normal"))

def interactive\_shape\_drawer():
    """Enhanced version with user interaction"""

    screen, shape\_turtle = setup\_drawing\_environment()

    \# Get user preferences
    print("{n}=== Interactive Shape Drawer ===")

    try:
        \# Square parameters
        square\_size = int(input("Enter square size (50{-}100): ") or "80")
        square\_color = input("Enter square color (red/blue/green): ") or "red"

        \# Rectangle parameters
        rect\_width = int(input("Enter rectangle width (80{-}150): ") or "120")
        rect\_height = int(input("Enter rectangle height (60{-}100): ") or "80")
        rect\_color = input("Enter rectangle color: ") or "green"

        \# Circle parameters
        circle\_radius = int(input("Enter circle radius (40{-}80): ") or "60")
        circle\_color = input("Enter circle color: ") or "blue"

        \# Draw shapes with user parameters
        draw\_square(shape\_turtle, square\_size, square\_color, ({-}300, 100))
        draw\_rectangle(shape\_turtle, rect\_width, rect\_height, rect\_color, ({-}50, 100))
        draw\_circle(shape\_turtle, circle\_radius, circle\_color, (200, 100))

        add\_shape\_information(shape\_turtle)

    except ValueError:
        print("Invalid input! Using default values...")
        return draw\_all\_shapes()

    return screen

\# Main execution
def main():
    """Main function with options"""

    print("Choose drawing mode:")
    print("1. Default shapes")
    print("2. Interactive mode")

    choice = input("Enter choice (1 or 2): ").strip()

    if choice == "2":

```

```

        screen = interactive\_shape\_drawer()
    else:
        screen = draw\_all\_shapes()

    print("\nClick on the screen to close the window.")
    screen.exitonclick()

\# Run the program
if \_\_name\_\_ == "\_\_main\_\_":
    main()

```

#### Table of Shape Characteristics:

Shape	Sides	Properties	Area Formula
<b>Square</b>	4 equal	All angles 90°	side <sup>2</sup>
<b>Rectangle</b>	4 (2 pairs)	Opposite sides equal	length $\times$ width
<b>Circle</b>	0 (curved)	All points equidistant	$\times radius^2$

#### Shape Drawing Process:

```

flowchart LR
    A[Start Drawing] --> B[Setup Turtle]
    B --> C[Draw Square]
    C --> D[Draw Rectangle]
    D --> E[Draw Circle]
    E --> F[Add Labels]
    F --> G[Display Information]
    G --> H[Complete]

```

- **Geometric accuracy:** Precise angle and distance measurements
- **Visual appeal:** Different colors and filled shapes
- **Educational value:** Shows formulas
- **Mathematical calculations:** Area formulas included
- **Interactive features:** User can customize parameters

#### Mnemonic

“Square Equal, Rectangle Opposite, Circle Round”