Subject Name Solutions

4321602 – Winter 2023

Semester 1 Study Material

Detailed Solutions and Explanations

Question 1(a) [3 marks]

What is Dictionary? Explain with example.

Solution

Dictionary Python key-value pairs collection mutable ordered .

Table 1: Dictionary Properties

Property	Description
Mutable	Values change
Ordered	Python 3.7+ insertion order maintain
Indexed	Keys access
No Duplicates	Duplicate keys allow

```
\# Dictionary Example
student = \{
    "name": "Raj",
    "age": 20,
    "course": "IT"
\}
print(student["name"])  \# Output: Raj
```

- **Key-Value Structure**: element key value
- **Fast Access**: O(1) time complexity data access
- **Dynamic Size**: Runtime size -

Mnemonic

"Dictionary = Key Value Treasure"

Question 1(b) [4 marks]

Explain Tuple Built-in functions and methods.

Solution

Tuple limited built-in methods immutable .

Table 2: Tuple Methods

Method	Description	Example
count()	Element frequency return	`t.count(5)`
index()	Element first index return	`t.index('a')`
len()	Tuple length return	`len(t)`
max()	Maximum value return	`max(t)`
min()	Minimum value return	`min(t)`

```
\# Tuple Methods Example
numbers = (1, 2, 3, 2, 4, 2)
print(numbers.count(2))     \# Output: 3
print(numbers.index(3))     \# Output: 2
print(len(numbers))         \# Output: 6
```

- **Immutable Nature**: Methods tuple modify
- **Return Values**: methods values return
- **Type Conversion**: tuple() function list tuple convert

Mnemonic

"Count Index Length Max Min"

Question 1(c) [7 marks]

Write a python program to demonstrate set operations.

Solution

Set operations mathematics set theory .

Table 3: Set Operations

Operation	Symbol	Method	Description
Union	`|`	`union()`	sets elements
Intersection	`&`	`intersection()`	Common elements
Difference	`-`	`difference()`	First set second minus
Symmetric Difference	`^`	`symmetric_difference()`	Unique elements only

```
\# Set Operations Program
set1 = \{1, 2, 3, 4, 5\}
set2 = \{4, 5, 6, 7, 8\}

print("Set 1:", set1)
print("Set 2:", set2)

\# Union Operation
union\_result = set1 | set2
print("Union:", union\_result)

\# Intersection Operation
intersection\_result = set1 \& set2
print("Intersection:", intersection\_result)

\# Difference Operation
difference\_result = set1 {-} set2
print("Difference:", difference\_result)

\# Symmetric Difference
sym\_diff\_result = set1 \^{} set2
print("Symmetric Difference:", sym\_diff\_result)

\# Subset and Superset
set3 = \{1, 2\}
print("Is set3 subset of set1?", set3.issubset(set1))
print("Is set1 superset of set3?", set1.issuperset(set3))
```

- **Mathematical Operations**: Set theory operations implement
- **Efficient Processing**: Duplicate elements automatically remove
- **Boolean Results**: Subset/superset operations boolean return

Mnemonic

"Union Intersection Difference Symmetric"

Question 1(c OR) [7 marks]

Write a python program to demonstrate the dictionaries functions and operations.

Solution

Dictionary operations data manipulation powerful tools .

Table 4: Dictionary Methods

Method	Description	Example
keys()	keys return	`dict.keys()`
values()	values return	`dict.values()`
items()	Key-value pairs return	`dict.items()`
get()	Safe value retrieval	`dict.get('key')`
update()	Dictionary merge	`dict.update()`

```
\# Dictionary Operations Program
student\_data = \{
    "name": "Amit",
    "age": 21,
    "course": "IT",
    "semester": 2
\}

print("Original Dictionary:", student\_data)

\# Accessing values
print("Student Name:", student\_data.get("name"))
print("Student Age:", student\_data["age"])

\# Adding new key{-value pair}
student\_data["city"] = "Ahmedabad"
print("After adding city:", student\_data)

\# Updating existing value
student\_data.update(\{"age": 22, "semester": 3\})
print("After update:", student\_data)

\# Dictionary methods
print("Keys:", list(student\_data.keys()))
print("Values:", list(student\_data.values()))
print("Items:", list(student\_data.items()))

\# Removing elements
removed\_value = student\_data.pop("semester")
print("Removed value:", removed\_value)
print("Final Dictionary:", student\_data)
```

- **Dynamic Operations**: Runtime keys values add/remove
- **Safe Access**: get() method KeyError prevent
- **Iteration Support**: keys(), values(), items() methods loop useful

Mnemonic

"Get Keys Values Items Update Pop"

Question 2(a) [3 marks]

Distinguish between Tuple and List in Python.

Solution

Table 5: Tuple vs List Comparison

Feature	Tuple	List
Mutability	Immutable (cannot change)	Mutable (can change)
Syntax	Parentheses `()`	Square brackets `[]`
Performance	Faster	Slower
Memory	Less memory	More memory
Methods	Limited (count, index)	Many methods available
Use Case	Fixed data	Dynamic data

- **Immutable Nature**: Tuple create change
- **Performance**: Tuple operations list fast
- **Memory Efficient**: Tuple memory

Mnemonic

"Tuple Tight, List Light"

Question 2(b) [4 marks]

What is the dir() function in python? Explain with example.

Solution

dir() function built-in function object attributes methods list return .

Table 6: dir() Function Features

Feature	Description
Object Inspection	Object attributes show
Method Discovery	Available methods list
Namespace Exploration	Current namespace variables show
Module Analysis	Module contents explore

```
\# dir() Function Example
\# For string object
text = "Hello"
string\_methods = dir(text)
print("String methods:", string\_methods[:5])

\# For list object
my\_list = [1, 2, 3]
list\_methods = dir(my\_list)
print("List methods:", [m for m in list\_methods if not m.startswith({\_})][:5])

\# For current namespace
print("Current namespace:", dir()[:3])

\# For built{-in functions}
import math
print("Math module:", dir(math)[:5])
```

- **Interactive Development**: Objects capabilities useful
- **Debugging Tool**: Available methods quickly identify
- **Learning Aid**: New libraries explore helpful

Mnemonic

"Dir = Directory of Methods"

Question 2(c) [7 marks]

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

Solution

Module approach code reusability organization improve .
Diagram: Module Structure

```
   circle.py              main.py
   (Module)               (Main Program)

 • area()             import circle
 • circumference        use functions
 • PI constant
```

File 1: circle.py (Module)

```
\# circle.py {- Circle calculation module}
import math

\# Constants
PI = math.pi

def area(radius):
    """Calculate area of circle"""
    if radius {} 0:
        return "Radius cannot be negative"
    return PI * radius * radius

def circumference(radius):
    """Calculate circumference of circle"""
    if radius {} 0:
        return "Radius cannot be negative"
    return 2 * PI * radius

def display\_info():
    """Display module information"""
    print("Circle Module {- Version 1.0"})
    print("Functions: area(), circumference()")
```

File 2: main.py (Main Program)

```
\# main.py {- Main program using circle module}
import circle

\# Get radius from user
radius = float(input("Enter radius: "))

\# Calculate using module functions
circle\_area = circle.area(radius)
circle\_circumference = circle.circumference(radius)

\# Display results
print(f"Circle with radius \{radius\}:")
print(f"Area: \{circle\_area:.2f\}")
print(f"Circumference: \{circle\_circumference:.2f\}")

\# Display module info
circle.display\_info()
```

- **Modular Design**: Functions separate file organize
- **Reusability**: Module multiple programs use
- **Namespace Management**: Module prefix function access

Mnemonic

"Import Calculate Display"

Question 2(a OR) [3 marks]

Explain Nested Tuple with example.

Solution

Nested Tuple tuple tuples , hierarchical data structure .

Table 7: Nested Tuple Features

Feature	Description
Multi-dimensional	2D 3D data structure
Immutable	levels immutable
Indexing	Multiple square brackets access
Heterogeneous	Different data types store

```
\# Nested Tuple Example
student\_records = (
    ("Raj", 20, ("IT", 2)),
    ("Priya", 19, ("CS", 1)),
    ("Amit", 21, ("IT", 3))
)

\# Accessing nested elements
print("First student:", student\_records[0])
print("First student name:", student\_records[0][0])
print("First student course:", student\_records[0][2][0])

\# Iterating through nested tuple
for student in student\_records:
    name, age, (course, semester) = student
    print(f"\{name\} {- Age: }\{age\}, Course: \{course\}, Sem: \{semester\}")
```

- **Data Organization**: Related data group useful
- **Immutable Structure**: create structure change
- **Efficient Access**: Index-based fast access

Mnemonic

"Nested = Tuple Inside Tuple"

Question 2(b OR) [4 marks]

What is PIP? Write the syntax to install and uninstall python packages.

Solution

PIP (Pip Installs Packages) Python package installer PyPI packages download install .

Table 8: PIP Commands

Command	Syntax	Description
Install	`pip install package_name`	Package install

Uninstall	`pip uninstall package_name`	Package remove
List	`pip list`	Installed packages show
Show	`pip show package_name`	Package info display
Upgrade	`pip install --upgrade package_name`	Package update

```
\# PIP Command Examples (Terminal/Command Prompt   run   )

\# Install a package
\# pip install requests

\# Install specific version
\# pip install Django==3.2.0

\# Uninstall a package
\# pip uninstall numpy

\# List all installed packages
\# pip list

\# Show package information
\# pip show matplotlib

\# Upgrade a package
\# pip install {-{-}upgrade pandas}

\# Install from requirements file
\# pip install {-r requirements.txt}
```

- **Package Management**: Third-party libraries easily manage
- **Version Control**: Specific versions install
- **Dependency Resolution**: Required dependencies automatically install

Mnemonic

"PIP = Package Install Python"

Question 2(c OR) [7 marks]

Explain different ways of importing package. How are modules and packages connected to each other?

Solution

Python imports ways code organization namespace management important .
Diagram: Package Structure

```
MyPackage/
   \_\_init\_\_.py
   module1.py
   module2.py
   subpackage/
       \_\_init\_\_.py
       module3.py
```

Table 9: Import Methods

Method	Syntax	Usage
Basic Import	`import module`	Full module name required

From Import	`from module import function`	Direct function access
Alias Import	`import module as alias`	Short name for module
Star Import	`from module import *`	Import all functions
Package Import	`from package import module`	Import from package

```
\# Different Import Ways

\# 1. Basic Import
import math
result = math.sqrt(16)

\# 2. From Import
from math import sqrt, pi
result = sqrt(16)
area = pi * 5 * 5

\# 3. Alias Import
import numpy as np
array = np.array([1, 2, 3])

\# 4. Star Import (not recommended)
from math import *
result = cos(0)

\# 5. Package Import
from mypackage import module1
from mypackage.subpackage import module3

\# 6. Relative Import (within package)
\# from . import module1
\# from ..parent\_module import function
```

Module-Package Connection:

- **Modules**: Single .py files containing Python code
- **Packages**: Directories containing multiple modules with ___init___.py
- **Namespace**: Packages create hierarchical namespace structure
- **init.py**: Makes directory a package and controls imports

Mnemonic

"Import From As Star Package"

Question 3(a) [3 marks]

Describe Runtime Error and Syntax Error. Explain with example.

Solution

Table 10: Error Types Comparison

Error Type	When Occurs	Detection	Example
Syntax Error	Code parsing time	Before execution	Missing colon, brackets
Runtime Error	During execution	While running	Division by zero, file not found
Logic Error	Always	After execution	Wrong calculation logic

```
\# Syntax Error Example
\# print("Hello World"  \# Missing closing parenthesis
\# SyntaxError: unexpected EOF while parsing

\# Runtime Error Examples
try:
    \# ZeroDivisionError
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")

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

- **Syntax Errors**: Code run detect
- **Runtime Errors**: Program execution
- **Prevention**: Exception handling runtime errors handle

Mnemonic

"Syntax Before, Runtime During"

Question 3(b) [4 marks]

What is Exception handling in Python? Explain with example.

Solution

Exception handling mechanism runtime errors gracefully handle program crash prevent .

Table 11: Exception Handling Keywords

Keyword	Purpose	Description
try	Exception code	Risk code block
except	Exception handle	Error handling block
finally	execute	Cleanup code
else	Exception	Success code block
raise	Manual exception raise	Custom error throwing

```
\# Exception Handling Example
def safe\_division(a, b):
    try:
        \# Code that might raise exception
        result = a / b
        print(f"Division successful: \{result\}")

    except ZeroDivisionError:
        \# Handle specific exception
        print("Error: Cannot divide by zero")
        result = None

    except TypeError:
        \# Handle type errors
        print("Error: Invalid data types")
        result = None

    else:
        \# Executes if no exception
        print("Division completed successfully")

    finally:
        \# Always executes
        print("Division operation finished")

    return result

\# Test the function
safe\_division(10, 2)   \# Normal case
safe\_division(10, 0)   \# Zero division
safe\_division(10, "a") \# Type error
```

- **Error Prevention**: Program crash prevent
- **Graceful Handling**: User-friendly error messages provide
- **Resource Management**: finally block cleanup operations

Mnemonic

"Try Except Finally Else Raise"

Question 3(c) [7 marks]

Create a function for division of two numbers, if the value of any argument is non-integer then raise the error or if second argument is 0 then raise the error.

Solution

Custom exception handling function validation error control important .

Diagram: Function Flow

```
flowchart LR
    A[Input Numbers] {-{-} B\{Are both integers?\}}
    B {-{-}|No| C[Raise TypeError]}
    B {-{-}|Yes| D\{Is second number 0?\}}
    D {-{-}|Yes| E[Raise ZeroDivisionError]}
    D {-{-}|No| F[Perform Division]}
    F {-{-} G[Return Result]}
    C {-{-} H[Handle Error]}
    E {-{-} H}
    H {-{-} I[End]}
```

```
    G {-{-} I}

def safe\_integer\_division(num1, num2):
    """
    Divide two numbers with validation
    Raises TypeError if arguments are not integers
    Raises ZeroDivisionError if second argument is 0
    """

    \# Check if both arguments are integers
    if not isinstance(num1, int):
        raise TypeError(f"First argument must be integer, got \{type(num1).\_\_name\_\_\}")

    if not isinstance(num2, int):
        raise TypeError(f"Second argument must be integer, got \{type(num2).\_\_name\_\_\}")

    \# Check for zero division
    if num2 == 0:
        raise ZeroDivisionError("Cannot divide by zero")

    \# Perform division
    result = num1 / num2
    return result

\# Test the function with different cases
def test\_division():
    test\_cases = [
        (10, 2),      \# Valid case
        (15, 3),      \# Valid case
        (10, 0),      \# Zero division error
        (10.5, 2),    \# Non{-integer first argument}
        (10, 2.5),    \# Non{-integer second argument}
        ("10", 2),    \# String argument
    ]

    for num1, num2 in test\_cases:
        try:
            result = safe\_integer\_division(num1, num2)
            print(f"\{num1\}  \{num2\} = \{result\}")

        except TypeError as e:
            print(f"Type Error: \{e\}")

        except ZeroDivisionError as e:
            print(f"Zero Division Error: \{e\}")

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

        print("{-"} * 40)

\# Run tests
test\_division()
```

- **Input Validation**: Arguments type value check
- **Custom Errors**: Specific exceptions raise
- **Error Messages**: Clear descriptive error messages

Mnemonic

"Validate Type, Check Zero, Divide Safe"

Question 3(a OR) [3 marks]

Describe any five built-in exceptions in Python.

Solution

Table 12: Built-in Exceptions

Exception	Cause	Example
ValueError	Invalid value for operation	`int("abc")`
TypeError	Wrong data type	`"hello" + 5`
IndexError	Index out of range	`list[10]` when list has 5 elements
KeyError	Dictionary key not found	`dict["nonexistent"]`
FileNotFoundError	File does not exist	`open("missing.txt")`

```
\# Built{-in Exceptions Examples}

\# 1. ValueError
try:
    number = int("not\_a\_number")
except ValueError:
    print("ValueError: Invalid conversion")

\# 2. TypeError
try:
    result = "Hello" + 5
except TypeError:
    print("TypeError: Cannot add string and integer")

\# 3. IndexError
try:
    my\_list = [1, 2, 3]
    value = my\_list[5]
except IndexError:
    print("IndexError: List index out of range")
```

- **Automatic Detection**: Python automatically raises these exceptions
- **Specific Handling**: exception specific purpose
- **Inheritance**: exceptions BaseException class inherit

Mnemonic

"Value Type Index Key File"

Question 3(b OR) [4 marks]

Explain try, except and finally terms with syntax.

Solution

Exception handling blocks specific purpose execution order .

Table 13: Exception Handling Blocks

Block	Purpose	Execution	Mandatory
try	Risky code	First	Yes
except	Error handling	If exception occurs	At least one
else	Success code	If no exception	No

finally	Cleanup code	Always	No

Syntax Structure:

```
try:
    \# Code that might raise exception
    risky\_code()
except ExceptionType1:
    \# Handle specific exception
    handle\_error1()
except ExceptionType2:
    \# Handle another exception
    handle\_error2()
else:
    \# Code runs if no exception
    success\_code()
finally:
    \# Code always runs
    cleanup\_code()
```

Practical Example:

```
def file\_operation(filename):
    file\_handle = None
    try:
        \# Risky operation
        file\_handle = open(filename, {r})
        content = file\_handle.read()
        print("File read successfully")

    except FileNotFoundError:
        print("File not found error")

    except PermissionError:
        print("Permission denied")

    else:
        print("File operation completed successfully")

    finally:
        \# Cleanup {- always executes}
        if file\_handle:
            file\_handle.close()
            print("File closed")

\# Test the function
file\_operation("test.txt")
```

- **Exception Flow**: try $\rightarrow$ *except*/*else* $\rightarrow$ *finally*
- **Multiple Handlers**: Multiple except blocks allowed
- **Guaranteed Execution**: finally block run

Mnemonic

"Try Exception Else Finally"

Question 3(c OR) [7 marks]

Write a user defined exception that could be raised when the text entered by a user consists of less than 10 characters.

Solution

User-defined exceptions custom validation logic implement powerful tool .

Diagram: Custom Exception Flow

```
flowchart LR
    A[User Input] {-{-} B\{Length = 10?\}}
    B {-{-}|No| C[Raise ShortTextError]}
    B {-{-}|Yes| D[Process Text]}
    C {-{-} E[Display Error Message]}
    D {-{-} F[Success Operation]}
    E {-{-} G[Ask for New Input]}
    F {-{-} H[End]}
    G {-{-} A}
```

```
\# User{-defined Exception Class}
class ShortTextError(Exception):
    """Custom exception for text that is too short"""

    def \_\_init\_\_(self, text\_length, minimum\_length=10):
        self.text\_length = text\_length
        self.minimum\_length = minimum\_length
        message = f"Text is too short! Length: \{text\_length\}, Required: \{minimum\_length\}"
        super().\_\_init\_\_(message)

def validate\_text\_length(text):
    """
    Validate text length and raise exception if too short
    """
    if len(text) {} 10:
        raise ShortTextError(len(text))
    return True

def process\_user\_text(text):
    """
    Process text after validation
    """
    try:
        validate\_text\_length(text)
        print(f"  Text accepted: {}\{text\}{"})
        print(f"Text length: \{len(text)\} characters")
        return text.upper()  \# Process the text

    except ShortTextError as e:
        print(f"  \{e\}")
        return None

def interactive\_text\_input():
    """
    Interactive function to get valid text from user
    """
    while True:
        try:
            user\_input = input("Enter text (minimum 10 characters): ")

            \# Validate text length
            if len(user\_input) {} 10:
                raise ShortTextError(len(user\_input))
```

```
            print(f"  Valid text entered: {}\{user\_input\}{"})
            break

        except ShortTextError as e:
            print(f"  Error: \{e\}")
            retry = input("Try again? (y/n): ")
            if retry.lower() != {y}:
                print("Operation cancelled.")
                break

\# Test different scenarios
def test\_custom\_exception():
    test\_texts = [
        "Hi",                    \# Too short (2 chars)
        "Hello",                 \# Too short (5 chars)
        "Short",                 \# Too short (5 chars)
        "This is valid text",    \# Valid (19 chars)
        "Perfect length text"    \# Valid (20 chars)
    ]

    print("Testing Custom Exception:")
    print("=" * 40)

    for text in test\_texts:
        result = process\_user\_text(text)
        if result:
            print(f"Processed: \{result\}")
        print("{-"} * 30)

\# Run tests
test\_custom\_exception()

\# Uncomment to test interactive input
\# interactive\_text\_input()
```

Additional Features:

```
\# Enhanced Custom Exception with more features
class TextValidationError(Exception):
    """Enhanced text validation exception"""

    def \_\_init\_\_(self, text, error\_type, details=None):
        self.text = text
        self.error\_type = error\_type
        self.details = details

        if error\_type == "short":
            message = f"Text too short: \{len(text)\} chars (min: 10)"
        elif error\_type == "empty":
            message = "Text cannot be empty"
        elif error\_type == "spaces":
            message = "Text contains only spaces"
        else:
            message = f"Text validation failed: \{error\_type\}"

        super().\_\_init\_\_(message)

def advanced\_text\_validation(text):
    """Advanced text validation with multiple checks"""
    if not text:
        raise TextValidationError(text, "empty")
```

```
    if text.isspace():
        raise TextValidationError(text, "spaces")

    if len(text.strip()) {} 10:
        raise TextValidationError(text, "short")

    return True
```

- **Custom Logic**: Application-specific validation rules implement
- **Inheritance**: Exception class inherit custom exceptions
- **Detailed Information**: Exception object additional data store

Mnemonic

"Custom Exception = Class Inherit Raise"

Question 4(a) [3 marks]

Write five points on difference between Text File and Binary File.

Solution

Table 14: Text File vs Binary File

Feature	Text File	Binary File
Content	Human-readable characters	Binary data (0s and 1s)
Encoding	Character encoding (UTF-8, ASCII)	No character encoding
Opening Mode	'r', 'w', 'a'	'rb', 'wb', 'ab'
File Size	Generally larger	Generally smaller
Platform	Platform dependent	Platform independent

```
\# Text vs Binary File Examples
\# Text file example
with open("sample.txt", "w") as f:
    f.write("Hello World")

\# Binary file example
with open("sample.bin", "wb") as f:
    f.write(b{}{x48x65x6cx6cx6f}{})
```

- **Readability**: Text files editor read , binary files special software
- **Portability**: Binary files different platforms easily transfer
- **Processing**: Text files string operations , binary files exact data storage

Mnemonic

"Text Human, Binary Machine"

Question 4(b) [4 marks]

Write a program to read the data from a file and separate the uppercase character and lowercase character into two separate files.

Solution

File processing character-based operations common requirements .

Table 15: File Operations

Operation	Method	Purpose
Read	`read()`	Complete file content
Write	`write()`	Write string to file
Character Check	`isupper()`, `islower()`	Character case detection
File Handling	`with open()`	Safe file operations

```
def separate\_case\_characters(input\_file, upper\_file, lower\_file):
    """
    Read file and separate uppercase/lowercase characters
    """
    try:
        \# Read from input file
        with open(input\_file, {r}) as infile:
            content = infile.read()

        \# Separate characters
        uppercase\_chars = ""
        lowercase\_chars = ""

        for char in content:
            if char.isupper():
                uppercase\_chars += char
            elif char.islower():
                lowercase\_chars += char

        \# Write to uppercase file
        with open(upper\_file, {w}) as upfile:
            upfile.write(uppercase\_chars)

        \# Write to lowercase file
        with open(lower\_file, {w}) as lowfile:
            lowfile.write(lowercase\_chars)

        print(f"  Characters separated successfully!")
        print(f"Uppercase characters: \{len(uppercase\_chars)\}")
        print(f"Lowercase characters: \{len(lowercase\_chars)\}")

    except FileNotFoundError:
        print(f"Error: File {}\{input\_file\}{ not found"})
    except Exception as e:
        print(f"Error: \{e\}")

\# Create sample input file
def create\_sample\_file():
    sample\_text = """Hello World! This is a SAMPLE Text file.
It contains UPPERCASE and lowercase Characters.
Python Programming is FUN and Educational."""

    with open("input.txt", "w") as f:
        f.write(sample\_text)
    print("Sample input file created: input.txt")

\# Main execution
create\_sample\_file()
separate\_case\_characters("input.txt", "uppercase.txt", "lowercase.txt")

\# Display results
print("{n}File Contents:")
print("{-"} * 30)

try:
    with open("uppercase.txt", "r") as f:
        print(f"Uppercase file: \{f.read()\}")

    with open("lowercase.txt", "r") as f:
        print(f"Lowercase file: \{f.read()\}")
except FileNotFoundError:
```

```
print("Output files not found")
```

- **Character Processing**: character case individually check
- **File Safety**: with statement automatic file closing ensure
- **Error Handling**: File operations proper exception handling

Mnemonic

"Read Separate Write"

Question 4(c) [7 marks]

Describe dump() and load() method. Explain with example.

Solution

dump() load() methods pickle module part object serialization .

Table 16: Pickle Methods

Method	Purpose	File Mode	Description
dump()	Serialize object to file	'wb'	Object binary file store
load()	Deserialize object from file	'rb'	File object retrieve
dumps()	Serialize to bytes	N/A	Object bytes convert
loads()	Deserialize from bytes	N/A	Bytes object

Diagram: Serialization Process

```
flowchart LR
    A[Python Object] {-{-}|"dump()"| B[Binary File]}
    B {-{-}|"load()"| C[Python Object]}
    A {-{-}|"dumps()"| D[Bytes String]}
    D {-{-}|"loads()"| C}

import pickle

\# Example with different data types
def demonstrate\_pickle():
    \# Sample data to serialize
    student\_data = \{
        {name}: {Raj Patel},
        {age}: 20,
        {grades}: [85, 92, 78, 96],
        {subjects}: ({Math}, {Python}, {Database}),
        {is\_active}: True
    \}

    class Student:
        def \_\_init\_\_(self, name, roll\_no):
            self.name = name
            self.roll\_no = roll\_no

        def \_\_str\_\_(self):
            return f"Student: \{self.name\} (Roll: \{self.roll\_no\})"

    \# Create objects
    student\_obj = Student("Priya Shah", 101)
    data\_list = [student\_data, student\_obj, [1, 2, 3, 4, 5]]

    \# DUMP {- Serialize objects to file}
    print("=== DUMP Operation ===")
    try:
        with open({student\_data.pkl}, {wb}) as f:
            pickle.dump(data\_list, f)
        print("  Data successfully dumped to student\_data.pkl")

        \# Also demonstrate dumps()
        serialized\_bytes = pickle.dumps(student\_data)
        print(f"  Data serialized to bytes: \{len(serialized\_bytes)\} bytes")

    except Exception as e:
        print(f"  Dump error: \{e\}")

    \# LOAD {- Deserialize objects from file  }
    print("{n}=== LOAD Operation ===")
    try:
        with open({student\_data.pkl}, {rb}) as f:
            loaded\_data = pickle.load(f)

        print("  Data successfully loaded from student\_data.pkl")
        print("{n}Loaded Data:")
        print("{-"} * 20)

        \# Display loaded data
        for i, item in enumerate(loaded\_data):
            print(f"Item \{i+1\}: \{item\}")
            print(f"Type: \{type(item)\}")
            print()
```

```
        \# Also demonstrate loads()
        deserialized\_data = pickle.loads(serialized\_bytes)
        print(f"  Data deserialized from bytes: \{deserialized\_data\}")

    except FileNotFoundError:
        print("  Pickle file not found")
    except Exception as e:
        print(f"  Load error: \{e\}")

\# Advanced example with custom class
def advanced\_pickle\_example():
    class BankAccount:
        def \_\_init\_\_(self, account\_no, holder\_name, balance):
            self.account\_no = account\_no
            self.holder\_name = holder\_name
            self.balance = balance
            self.transactions = []

        def deposit(self, amount):
            self.balance += amount
            self.transactions.append(f"Deposit: +\{amount\}")

        def withdraw(self, amount):
            if self.balance {=} amount:
                self.balance {-=} amount
                self.transactions.append(f"Withdraw: {-}\{amount\}")
            else:
                print("Insufficient balance")

        def \_\_str\_\_(self):
            return f"Account \{self.account\_no\}: \{self.holder\_name\} {- Balance:  }\{self.balance\}

    \# Create and use account
    account = BankAccount("12345", "Amit Kumar", 5000)
    account.deposit(1500)
    account.withdraw(800)

    print("=== Advanced Pickle Example ===")
    print(f"Original: \{account\}")
    print(f"Transactions: \{account.transactions\}")

    \# Serialize account object
    with open({bank\_account.pkl}, {wb}) as f:
        pickle.dump(account, f)

    \# Load account object
    with open({bank\_account.pkl}, {rb}) as f:
        loaded\_account = pickle.load(f)

    print(f"Loaded: \{loaded\_account\}")
    print(f"Loaded transactions: \{loaded\_account.transactions\}")

    \# Verify object functionality
    loaded\_account.deposit(200)
    print(f"After new deposit: \{loaded\_account\}")

\# Run demonstrations
demonstrate\_pickle()
print("{n}" + "="*50 + "{n}")
advanced\_pickle\_example()
```

Benefits and Limitations:

```
\# Benefits
benefits = [
    "Complete object state preservation",
    "Works with complex nested objects",
    "Maintains object relationships",
    "Fast serialization/deserialization"
]

\# Limitations
limitations = [
    "Python{-specific format"},
    "Security risks with untrusted data",
    "Version compatibility issues",
    "Not human{-readable"}
]

print("Benefits:", benefits)
print("Limitations:", limitations)
```

- **Object Persistence**: Python objects file permanently store
- **Complete State**: Object complete state including methods preserve
- **Binary Format**: Efficient storage human-readable

Mnemonic

"Dump Store, Load Restore"

Question 4(a OR) [3 marks]

List different types of file modes provided by python for file operations and explain their uses.

Solution

Table 17: Python File Modes

Mode	Type	Description	Pointer Position
'r'	Text Read	Read only, file must exist	Beginning
'w'	Text Write	Write only, creates/overwrites	Beginning
'a'	Text Append	Write only, creates if not exist	End
'x'	Text Create	Create new file, fails if exists	Beginning
'rb'	Binary Read	Read binary data	Beginning
'wb'	Binary Write	Write binary data	Beginning
'ab'	Binary Append	Append binary data	End
'r+'	Text Read/Write	Read and write, file must exist	Beginning
'w+'	Text Write/Read	Write and read, creates/overwrites	Beginning

```
\# File Modes Examples
import os

\# Create sample file for demonstration
with open({demo.txt}, {w}) as f:
    f.write("Original content{n}Line 2{n}Line 3")

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

\# Append mode ({a)  }
with open({demo.txt}, {a}) as f:
    f.write("{n}Appended line")

\# Read+Write mode ({r+)}
with open({demo.txt}, {r+}) as f:
    f.seek(0)  \# Go to beginning
    f.write("Modified")

print("File modes demonstrated successfully")
```

- **Safety**: 'x' mode prevents accidental file overwriting
- **Efficiency**: Binary modes faster for non-text data
- **Flexibility**: Combined modes allow both read and write operations

Mnemonic

"Read Write Append Create Binary Plus"

Question 4(b OR) [4 marks]

Describe readline() and writeline() functions of the file.

Solution

Note: Python `writeline()` function exist . Correct function `writelines()` .

Table 18: Line-based File Functions

Function	Purpose	Return Type	Usage
readline()	Read single line	String	Sequential line reading
readlines()	Read all lines	List of strings	Complete file as list
writelines()	Write multiple lines	None	Write list of strings
write()	Write single string	Number of chars	Basic writing

```
def demonstrate\_line\_functions():
    \# Create sample file with multiple lines
    lines\_to\_write = [
        "First line of text{n}",
        "Second line of text{n}",
        "Third line of text{n}",
        "Fourth line without newline"
    ]

    print("=== WRITELINES() Demonstration ===")
    \# Write multiple lines using writelines()
    with open({sample\_lines.txt}, {w}) as f:
        f.writelines(lines\_to\_write)
    print("  Multiple lines written using writelines()")

    print("{n}=== READLINE() Demonstration ===")
    \# Read lines one by one using readline()
    with open({sample\_lines.txt}, {r}) as f:
        line\_count = 0
        while True:
            line = f.readline()
            if not line:  \# End of file
                break
            line\_count += 1
            print(f"Line \{line\_count\}: \{line.strip()\}")

    print(f"Total lines read: \{line\_count\}")

    print("{n}=== READLINES() Demonstration ===")
    \# Read all lines at once using readlines()
    with open({sample\_lines.txt}, {r}) as f:
        all\_lines = f.readlines()

    print("All lines as list:")
    for i, line in enumerate(all\_lines, 1):
        print(f"  [\{i\}] \{repr(line)\}")

    \# Practical example: Processing file line by line
    print("{n}=== Practical Example ===")
    student\_data = [
        "Raj,20,IT{n}",
        "Priya,19,CS{n}",
        "Amit,21,EC{n}",
        "Sneha,20,IT{n}"
    ]

    \# Write student data
    with open({students.txt}, {w}) as f:
        f.writelines(student\_data)

    \# Read and process line by line
    print("Student Information:")
    with open({students.txt}, {r}) as f:
        while True:
            line = f.readline()
            if not line:
                break

            \# Process each line
            parts = line.strip().split({,})
            if len(parts) == 3:
```

```
                name, age, course = parts
                print(f"  \{name\} (Age: \{age\}, Course: \{course\})")

\# Run demonstration
demonstrate\_line\_functions()

\# File pointer behavior example
def file\_pointer\_demo():
    print("{n}=== File Pointer Behavior ===")

    with open({sample\_lines.txt}, {r}) as f:
        print(f"Initial position: \{f.tell()\}")

        line1 = f.readline()
        print(f"After readline(): position \{f.tell()\}")
        print(f"Read: \{repr(line1)\}")

        line2 = f.readline()
        print(f"After second readline(): position \{f.tell()\}")
        print(f"Read: \{repr(line2)\}")

file\_pointer\_demo()
```

- **Sequential Access**: readline() sequential manner lines read
- **Memory Efficient**: Large files readline() memory-efficient
- **List Operations**: writelines() list of strings efficiently write

Mnemonic

"Read Line, Write Lines"

Question 4(c OR) [7 marks]

Write a python program to demonstrate seek() and tell() methods.

Solution

seek() tell() methods file pointer manipulation .

Table 19: File Pointer Methods

Method	Purpose	Parameters	Return Value
tell()	Current position	None	Integer (byte position)
seek()	Move pointer	offset, whence	New position
whence=0	From beginning	Default	Absolute position
whence=1	From current	Relative	Current + offset
whence=2	From end	End relative	End + offset

Diagram: File Pointer Movement

```
File: "Hello World"
      \^{             \^{}         \^{}}
   Position 0   Position 6  Position 11 (EOF)

seek(0)    {- Move to beginning}
seek(6)    {- Move to position 6  }
seek({-5,2) {-} Move 5 positions before end}

def demonstrate\_seek\_tell():
    \# Create sample file with known content
    sample\_text = "Hello Python Programming World!"

    with open({pointer\_demo.txt}, {w}) as f:
        f.write(sample\_text)

    print("=== File Pointer Demonstration ===")
    print(f"File content: {}\{sample\_text\}{"})
    print(f"File length: \{len(sample\_text)\} characters")
    print()

    with open({pointer\_demo.txt}, {r}) as f:
        \# Initial position
        print(f"1. Initial position: \{f.tell()\}")

        \# Read some characters
        first\_part = f.read(5)  \# Read "Hello"
        print(f"2. After reading {}\{first\_part\}{: position }\{f.tell()\}")

        \# Move to specific position
        f.seek(6)  \# Move to position 6 (start of "Python")
        print(f"3. After seek(6): position \{f.tell()\}")

        \# Read from new position
        next\_part = f.read(6)  \# Read "Python"
        print(f"4. Read {}\{next\_part\}{: position }\{f.tell()\}")

        \# Move relative to current position (only in binary mode for positive offset)
        \# Let{s demonstrate absolute positioning}
        f.seek(0)  \# Go to beginning
        print(f"5. After seek(0): position \{f.tell()\}")

        \# Move to end of file
        f.seek(0, 2)  \# 0 offset from end (position 2 = end)
        print(f"6. After seek(0,2) {- end of file: position }\{f.tell()\}")

        \# Move backwards from end
        f.seek({-}6, 2)  \# 6 positions before end
        print(f"7. After seek({-6,2): position }\{f.tell()\}")

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

def practical\_seek\_tell\_example():
    print("{n}=== Practical Example: File Editor Simulation ===")

    \# Create a file with structured data
    data\_lines = [
        "NAME:John Doe{n}",
        "AGE:25{n}",
```

```
        "CITY:Mumbai{n}",
        "PHONE:9876543210{n}",
        "EMAIL:john@example.com{n}"
    ]

    with open({person\_data.txt}, {w}) as f:
        f.writelines(data\_lines)

    \# Demonstrate finding and modifying specific data
    with open({person\_data.txt}, {r+}) as f:  \# Read+Write mode
        \# Find and display all positions
        positions = \{\}

        while True:
            pos = f.tell()
            line = f.readline()
            if not line:
                break

            field = line.split({:})[0]
            positions[field] = pos
            print(f"Field {}\{field\}{ starts at position }\{pos\}")

        print(f"{n}File positions: \{positions\}")

        \# Modify specific field (AGE)
        if {AGE} in positions:
            f.seek(positions[{AGE}])
            print(f"{n}Moving to AGE field at position \{f.tell()\}")

            \# Read current line
            current\_line = f.readline()
            print(f"Current line: \{current\_line.strip()\}")

            \# Calculate position to overwrite
            f.seek(positions[{AGE}])
            new\_age\_line = "AGE:26{n}"  \# Same length as original
            f.write(new\_age\_line)
            print(f"Updated AGE field")

    \# Verify changes
    print("{n}Updated file content:")
    with open({person\_data.txt}, {r}) as f:
        print(f.read())

def binary\_seek\_tell\_demo():
    print("{n}=== Binary File Seek/Tell Demo ===")

    \# Create binary file
    binary\_data = b{}{x48x65x6cx6cx6fx20x57x6fx72x6cx64}{}  \# "Hello World"

    with open({binary\_demo.bin}, {wb}) as f:
        f.write(binary\_data)

    with open({binary\_demo.bin}, {rb}) as f:
        print(f"Binary file size: \{len(binary\_data)\} bytes")

        \# Demonstrate all seek modes in binary
        print(f"Initial position: \{f.tell()\}")

        \# Read first 5 bytes
```

```
        first\_bytes = f.read(5)
        print(f"Read first 5 bytes: \{first\_bytes\} at position \{f.tell()\}")

        \# Seek relative to current position (works in binary mode)
        f.seek(1, 1)  \# Move 1 byte forward from current
        print(f"After seek(1,1): position \{f.tell()\}")

        \# Seek from end
        f.seek({-}3, 2)  \# 3 bytes before end
        print(f"After seek({-3,2): position }\{f.tell()\}")

        \# Read remaining
        remaining\_bytes = f.read()
        print(f"Remaining bytes: \{remaining\_bytes\}")

\# Run all demonstrations
demonstrate\_seek\_tell()
practical\_seek\_tell\_example()
binary\_seek\_tell\_demo()

\# Cleanup
import os
try:
    os.remove({pointer\_demo.txt})
    os.remove({person\_data.txt})
    os.remove({binary\_demo.bin})
    print("{n}Demo files cleaned up")
except:
    pass
```

- **File Navigation**: seek() arbitrary position move
- **Position Tracking**: tell() current position track useful
- **File Editing**: Specific locations data modify

Mnemonic

"Tell Position, Seek Destination"

Question 5(a) [3 marks]

Draw Circle and rectangle shapes using Turtle and fill them with red color.

Solution

Turtle graphics module shapes draw fill specific methods .

Table 20: Turtle Shape Methods

Method	Purpose	Example
circle()	Draw circle	`turtle.circle(50)`
forward()	Move forward	`turtle.forward(100)`
right()	Turn right	`turtle.right(90)`
begin_fill()	Start filling	`turtle.begin_fill()`
end_fill()	End filling	`turtle.end_fill()`
fillcolor()	Set fill color	`turtle.fillcolor("red")`

```
import turtle

def draw\_filled\_shapes():
    \# Create screen and turtle
    screen = turtle.Screen()
    screen.title("Filled Shapes with Turtle")
    screen.bgcolor("white")
    screen.setup(800, 600)

    \# Create turtle object
    painter = turtle.Turtle()
    painter.speed(3)

    \# Draw filled circle
    print("Drawing filled circle...")
    painter.penup()
    painter.goto({-}150, 0)  \# Move to left side
    painter.pendown()

    painter.fillcolor("red")
    painter.begin\_fill()
    painter.circle(80)  \# Radius = 80
    painter.end\_fill()

    \# Draw filled rectangle
    print("Drawing filled rectangle...")
    painter.penup()
    painter.goto(50, 50)  \# Move to right side
    painter.pendown()

    painter.fillcolor("red")
    painter.begin\_fill()

    \# Draw rectangle (100x80)
    for \_ in range(2):
        painter.forward(100)
        painter.right(90)
        painter.forward(80)
        painter.right(90)

    painter.end\_fill()

    \# Add labels
    painter.penup()
    painter.goto({-}150, {-}120)
    painter.write("Red Circle", align="center", font=("Arial", 14, "normal"))

    painter.goto(100, {-}50)
    painter.write("Red Rectangle", align="center", font=("Arial", 14, "normal"))

    \# Hide turtle and display result
    painter.hideturtle()
    print("Shapes drawn successfully!")

    \# Keep window open
    screen.exitonclick()

\# Run the program
draw\_filled\_shapes()
```

- **Fill Process**: begin_fill() end_fill() drawn shape automatically fill
- **Color Setting**: fillcolor() method fill color set

- **Shape Drawing**: Geometric shapes specific turtle movements

Mnemonic

"Begin Fill Draw End"

Question 5(b) [4 marks]

Explain the various inbuilt methods to change the direction of the Turtle.

Solution

Table 21: Turtle Direction Methods

Method	Parameters	Description	Example
right()	angle	Turn right by degrees	`turtle.right(90)`
left()	angle	Turn left by degrees	`turtle.left(45)`
setheading()	angle	Set absolute direction	`turtle.setheading(0)`
towards()	x, y	Point towards coordinates	`turtle.towards(100, 50)`
home()	none	Return to center, face east	`turtle.home()`

```
import turtle

def demonstrate\_direction\_methods():
    screen = turtle.Screen()
    screen.setup(600, 600)
    screen.title("Turtle Direction Methods")

    t = turtle.Turtle()
    t.speed(2)
    t.shape("turtle")

    \# 1. right() method
    t.write("1. right(90)", font=("Arial", 10, "normal"))
    t.forward(50)
    t.right(90)
    t.forward(50)

    \# 2. left() method
    t.penup()
    t.goto({-}100, 100)
    t.pendown()
    t.write("2. left(45)", font=("Arial", 10, "normal"))
    t.forward(50)
    t.left(45)
    t.forward(50)

    \# 3. setheading() method
    t.penup()
    t.goto(100, 100)
    t.pendown()
    t.write("3. setheading(180)", font=("Arial", 10, "normal"))
    t.setheading(180)  \# Face west
    t.forward(50)

    \# 4. towards() method
    t.penup()
    t.goto({-}100, {-}100)
    t.pendown()
    target\_x, target\_y = 100, {-}100
    t.write("4. towards(100,{-100)"}, font=("Arial", 10, "normal"))
    angle = t.towards(target\_x, target\_y)
    t.setheading(angle)
    t.goto(target\_x, target\_y)

    \# 5. home() method
    t.write("5. home()", font=("Arial", 10, "normal"))
    t.home()  \# Return to center and face east

    t.hideturtle()
    screen.exitonclick()

demonstrate\_direction\_methods()
```

- **Relative Turns**: right() left() current direction relative turn
- **Absolute Direction**: setheading() absolute compass direction set
- **Smart Pointing**: towards() specific coordinates point

Mnemonic

"Right Left Set Towards Home"

Question 5(c) [7 marks]

Write a python program to draw a rainbow using Turtle.

Solution

Rainbow drawing multiple colored arcs proper positioning .

Diagram: Rainbow Structure

```
    Red (outer)
   Orange
  Yellow
 Green
Blue
Indigo
Violet (inner)
```

```
import turtle

def draw\_rainbow():
    \# Screen setup
    screen = turtle.Screen()
    screen.title("Beautiful Rainbow")
    screen.bgcolor("lightblue")
    screen.setup(800, 600)

    \# Turtle setup
    rainbow\_turtle = turtle.Turtle()
    rainbow\_turtle.speed(8)
    rainbow\_turtle.pensize(8)

    \# Rainbow colors (ROYGBIV)
    colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]

    \# Rainbow parameters
    start\_radius = 200
    radius\_decrease = 15
    start\_y = {-}150

    print("Drawing rainbow...")

    \# Draw each color arc
    for i, color in enumerate(colors):
        \# Set color
        rainbow\_turtle.pencolor(color)

        \# Calculate current radius
        current\_radius = start\_radius {-} (i * radius\_decrease)

        \# Position turtle for semi{-circle}
        rainbow\_turtle.penup()
        rainbow\_turtle.goto({-}current\_radius, start\_y)
        rainbow\_turtle.pendown()
        rainbow\_turtle.setheading(0)  \# Face east

        \# Draw semi{-circle (180 degrees)}
        rainbow\_turtle.circle(current\_radius, 180)

        print(f"Drew \{color\} arc with radius \{current\_radius\}")

    \# Add clouds at ends
    draw\_clouds(rainbow\_turtle)

    \# Add sun
```

```
    draw\_sun(rainbow\_turtle)

    \# Add text
    rainbow\_turtle.penup()
    rainbow\_turtle.goto(0, {-}250)
    rainbow\_turtle.pencolor("black")
    rainbow\_turtle.write("  Beautiful Rainbow  ", align="center",
                         font=("Arial", 16, "bold"))

    rainbow\_turtle.hideturtle()
    print("Rainbow completed!")

    screen.exitonclick()

def draw\_clouds(turtle\_obj):
    """Draw clouds at both ends of rainbow"""
    turtle\_obj.pensize(3)
    turtle\_obj.pencolor("white")
    turtle\_obj.fillcolor("lightgray")

    \# Left cloud
    cloud\_positions = [({-}250, {-}100), (250, {-}100)]

    for x, y in cloud\_positions:
        turtle\_obj.penup()
        turtle\_obj.goto(x, y)
        turtle\_obj.pendown()
        \# Draw cloud using multiple circles
        turtle\_obj.begin\_fill()
        for i in range(3):
            turtle\_obj.circle(20)
            turtle\_obj.left(120)
        turtle\_obj.end\_fill()

def draw\_sun(turtle\_obj):
    """Draw sun in corner"""
    turtle\_obj.penup()
    turtle\_obj.goto(300, 200)
    turtle\_obj.pendown()
    turtle\_obj.pencolor("orange")
    turtle\_obj.fillcolor("yellow")

    \# Draw sun body
    turtle\_obj.begin\_fill()
    turtle\_obj.circle(30)
    turtle\_obj.end\_fill()

    \# Draw sun rays
    turtle\_obj.pensize(2)
    for angle in range(0, 360, 45):
        turtle\_obj.setheading(angle)
        turtle\_obj.forward(45)
        turtle\_obj.backward(45)

\# Alternative rainbow with gradient effect
def draw\_gradient\_rainbow():
    screen = turtle.Screen()
    screen.title("Gradient Rainbow")
    screen.bgcolor("skyblue")
    screen.setup(800, 600)
```

```
    t = turtle.Turtle()
    t.speed(0)
    t.pensize(5)

    \# Color variations for gradient effect
    rainbow\_colors = [
        "\#FF0000", "\#FF4500", "\#FFD700", "\#32CD32",
        "\#0000FF", "\#4B0082", "\#8B00FF"
    ]

    \# Draw rainbow with varying thickness
    for i, color in enumerate(rainbow\_colors):
        t.pencolor(color)
        t.pensize(12 {-} i)  \# Decreasing thickness

        radius = 150 {-} (i * 10)
        t.penup()
        t.goto({-}radius, {-}100)
        t.pendown()
        t.setheading(0)
        t.circle(radius, 180)

    t.hideturtle()
    screen.exitonclick()

\# Run the rainbow programs
print("Choose rainbow type:")
print("1. Standard Rainbow")
print("2. Gradient Rainbow")

choice = input("Enter choice (1 or 2): ")
if choice == "2":
    draw\_gradient\_rainbow()
else:
    draw\_rainbow()
```

- **Color Sequence**: ROYGBIV (Red Orange Yellow Green Blue Indigo Violet) proper order
- **Radius Management**: arc radius gradually decrease
- **Positioning**: Proper positioning penup/pendown goto methods

Mnemonic

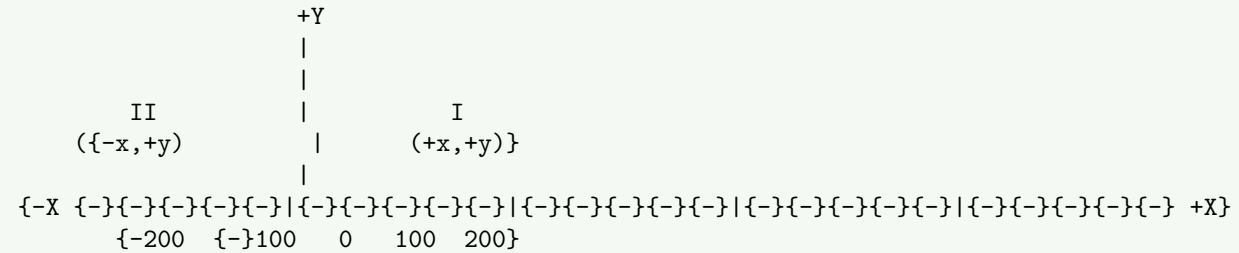
"ROYGBIV Arc Radius Position"

Question 5(a OR) [3 marks]

Draw a diagram of turtle screen and explain all 4 quadrants of x and y coordinates.

Solution

Diagram: Turtle Coordinate System



```
                    |
        III         |          IV
    ({-x,{-}y)          |      (+x,{-}y)}
                    |
                    |
                   {-Y}

Default Screen: 400x300 pixels
Center: (0, 0)
```

Table 22: Coordinate Quadrants

Quadrant	X Value	Y Value	Description	Example
I	Positive (+)	Positive (+)	Top-right	(100, 50)
II	Negative (-)	Positive (+)	Top-left	(-100, 50)
III	Negative (-)	Negative (-)	Bottom-left	(-100, -50)
IV	Positive (+)	Negative (-)	Bottom-right	(100, -50)

```
import turtle

def demonstrate\_coordinate\_system():
    screen = turtle.Screen()
    screen.title("Turtle Coordinate System")
    screen.setup(600, 500)
    screen.bgcolor("white")

    t = turtle.Turtle()
    t.speed(3)
    t.shape("turtle")

    \# Draw coordinate axes
    t.pencolor("gray")
    t.pensize(2)

    \# X{-axis}
    t.penup()
    t.goto({-}250, 0)
    t.pendown()
    t.goto(250, 0)

    \# Y{-axis}
    t.penup()
    t.goto(0, {-}200)
    t.pendown()
    t.goto(0, 200)

    \# Mark center
    t.penup()
    t.goto(0, 0)
    t.dot(8, "red")
    t.write("(0,0)", font=("Arial", 12, "normal"))

    \# Demonstrate each quadrant
    quadrants = [
        (100, 100, "I", "red"),     \# Quadrant I
        ({-}100, 100, "II", "blue"),  \# Quadrant II
        ({-}100, {-}100, "III", "green"), \# Quadrant III
        (100, {-}100, "IV", "orange")   \# Quadrant IV
    ]

    for x, y, quad\_name, color in quadrants:
        t.pencolor(color)
        t.penup()
        t.goto(x, y)
        t.pendown()
        t.dot(10, color)
        t.write(f"Q\{quad\_name\}{n}(\{x\},\{y\})", align="center",
                font=("Arial", 10, "bold"))

    t.hideturtle()
    screen.exitonclick()

demonstrate\_coordinate\_system()
```

- **Origin**: (0,0) screen center
- **Positive Direction**: X-axis right , Y-axis up positive
- **Navigation**: goto(x, y) method specific coordinates move

Mnemonic

"Right Up Positive, Left Down Negative"

Question 5(b OR) [4 marks]

Describe various turtle screen methods to change the background color, title, screensize and shapesize.

Solution

Table 23: Turtle Screen Methods

Method	Purpose	Parameters	Example
bgcolor()	Set background color	color name/hex	`screen.bgcolor("blue")`
title()	Set window title	string	`screen.title("My Program")`
setup()	Set screen size	width, height	`screen.setup(800, 600)`
screensize()	Set canvas size	width, height	`screen.screensize(400, 300)`
shapesize()	Set turtle size	stretch_wid, stretch_len	`turtle.shapesize(2, 3)`

```
import turtle

def demonstrate\_screen\_methods():
    \# Create screen object
    screen = turtle.Screen()

    \# 1. Title Method
    screen.title("Screen Methods Demonstration")
    print("  Title set to: {Screen Methods Demonstration"})

    \# 2. Background Color Method
    screen.bgcolor("lightgreen")
    print("  Background color set to: lightgreen")

    \# 3. Setup Method (window size)
    screen.setup(width=800, height=600)
    print("  Window size set to: 800x600 pixels")

    \# 4. Screen Size Method (canvas size)
    screen.screensize(canvwidth=400, canvheight=300)
    print("  Canvas size set to: 400x300")

    \# Create turtle to demonstrate shapesize
    demo\_turtle = turtle.Turtle()
    demo\_turtle.speed(3)

    \# 5. Shape Size Method
    demo\_turtle.shape("turtle")
    demo\_turtle.shapesize(stretch\_wid=3, stretch\_len=2, outline=3)
    print("  Turtle shape size: width=3x, length=2x, outline=3")

    \# Demonstrate different background colors
    colors = ["lightblue", "lightyellow", "lightpink", "lightcoral"]

    for i, color in enumerate(colors):
        screen.bgcolor(color)
        demo\_turtle.write(f"Background: \{color\}",
                          font=("Arial", 14, "normal"))
        demo\_turtle.forward(50)
        demo\_turtle.right(90)
        screen.ontimer(lambda: None, 1000)  \# Wait 1 second

    \# Reset to final state
    screen.bgcolor("white")
    demo\_turtle.penup()
    demo\_turtle.goto(0, {-}50)
    demo\_turtle.write("Screen Methods Demo Complete!",
                      align="center", font=("Arial", 16, "bold"))

    demo\_turtle.hideturtle()
    screen.exitonclick()

def advanced\_screen\_customization():
    """Advanced screen customization example"""
    screen = turtle.Screen()

    \# Advanced setup with all parameters
    screen.setup(width=0.8, height=0.8, startx=100, starty=50)
    screen.title("  Advanced Turtle Graphics  ")
    screen.bgcolor("\#2E8B57")  \# Sea Green
```

```
    \# Custom color palette
    screen.colormode(255)  \# Enable RGB mode

    \# Create multiple turtles with different sizes
    turtles = []
    shapes = ["turtle", "circle", "square", "triangle"]
    sizes = [(1, 1), (2, 1), (1, 2), (3, 3)]
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]

    for i in range(4):
        t = turtle.Turtle()
        t.shape(shapes[i])
        t.shapesize(sizes[i][0], sizes[i][1])
        t.color(colors[i])
        t.penup()
        t.goto({-}150 + i*100, 0)
        turtles.append(t)

        \# Label each turtle
        t.write(f"\{shapes[i]\}{n}\{sizes[i]\}",
               align="center", font=("Arial", 10, "normal"))

    \# Add instructions
    instruction\_turtle = turtle.Turtle()
    instruction\_turtle.hideturtle()
    instruction\_turtle.penup()
    instruction\_turtle.goto(0, {-}100)
    instruction\_turtle.color("white")
    instruction\_turtle.write("Different turtle shapes and sizes",
                          align="center", font=("Arial", 16, "bold"))

    screen.exitonclick()

\# Run demonstrations
print("Running Screen Methods Demo...")
demonstrate\_screen\_methods()

print("{n}Running Advanced Customization...")
advanced\_screen\_customization()
```

- **Window vs Canvas**: setup() window size, screensize() canvas size control
- **Color Modes**: bgcolor() color names hex values accept
- **Shape Scaling**: shapesize() turtle appearance scale

Mnemonic

"Title Background Setup Size Shape"

Question 5(c OR) [7 marks]

Write a python program to draw a star, triangle and octagon using turtle.

Solution

Geometric shapes drawing angles sides proper calculation .

Table 24: Shape Properties

Shape	Sides	External Angle	Internal Angle	Turn Angle
Triangle	3	120°	60°	120°

Star (5-point)	5	144°	36°	144°
Octagon	8	45°	135°	45°

Diagram: Shape Construction

```
Triangle:        Star:           Octagon:
   /{             *              /  }
  /  {           /             /     }
 /\_\_\_\_{        /             |       |}
               *     *          {     /}
                {   /            \_\_\_/}
                 { /}
                  *
```

```
import turtle
import math

def draw\_geometric\_shapes():
    \# Screen setup
    screen = turtle.Screen()
    screen.title("Geometric Shapes: Star, Triangle, Octagon")
    screen.bgcolor("black")
    screen.setup(900, 600)

    \# Turtle setup
    artist = turtle.Turtle()
    artist.speed(6)
    artist.pensize(3)

    \# Shape 1: Triangle
    draw\_triangle(artist, {-}250, 100, 80, "cyan")

    \# Shape 2: Five{-pointed Star  }
    draw\_star(artist, 0, 100, 80, "yellow")

    \# Shape 3: Octagon
    draw\_octagon(artist, 250, 100, 60, "magenta")

    \# Add labels
    add\_labels(artist)

    artist.hideturtle()
    print("All shapes drawn successfully!")
    screen.exitonclick()

def draw\_triangle(turtle\_obj, x, y, size, color):
    """Draw an equilateral triangle"""
    print(f"Drawing triangle at (\{x\}, \{y\})")

    turtle\_obj.penup()
    turtle\_obj.goto(x, y)
    turtle\_obj.pendown()
    turtle\_obj.color(color)
    turtle\_obj.fillcolor(color)

    turtle\_obj.begin\_fill()
    for \_ in range(3):
        turtle\_obj.forward(size)
        turtle\_obj.left(120)  \# External angle for triangle
    turtle\_obj.end\_fill()

def draw\_star(turtle\_obj, x, y, size, color):
    """Draw a five{-pointed star"""}
    print(f"Drawing star at (\{x\}, \{y\})")
```

```
    turtle\_obj.penup()
    turtle\_obj.goto(x, y)
    turtle\_obj.pendown()
    turtle\_obj.color(color)
    turtle\_obj.fillcolor(color)

    turtle\_obj.begin\_fill()
    for \_ in range(5):
        turtle\_obj.forward(size)
        turtle\_obj.right(144)  \# 144^ turn for 5{-pointed star}
    turtle\_obj.end\_fill()

def draw\_octagon(turtle\_obj, x, y, size, color):
    """Draw a regular octagon"""
    print(f"Drawing octagon at (\{x\}, \{y\})")

    turtle\_obj.penup()
    turtle\_obj.goto(x, y)
    turtle\_obj.pendown()
    turtle\_obj.color(color)
    turtle\_obj.fillcolor(color)

    turtle\_obj.begin\_fill()
    for \_ in range(8):
        turtle\_obj.forward(size)
        turtle\_obj.right(45)  \# 360^/8 = 45^ for octagon
    turtle\_obj.end\_fill()

def add\_labels(turtle\_obj):
    """Add labels for each shape"""
    turtle\_obj.color("white")

    labels = [
        ({-}250, 30, "Triangle{n}3 sides{n}120^ turns"),
        (0, 30, "Star{n}5 points{n}144^ turns"),
        (250, 30, "Octagon{n}8 sides{n}45^ turns")
    ]

    for x, y, text in labels:
        turtle\_obj.penup()
        turtle\_obj.goto(x, y)
        turtle\_obj.write(text, align="center", font=("Arial", 12, "normal"))

def draw\_advanced\_shapes():
    """Advanced version with animations and multiple variations"""
    screen = turtle.Screen()
    screen.title("Advanced Geometric Shapes")
    screen.bgcolor("navy")
    screen.setup(1000, 700)

    artist = turtle.Turtle()
    artist.speed(8)
    artist.pensize(2)

    \# Animated triangle variations
    triangle\_sizes = [40, 60, 80]
    triangle\_colors = ["red", "orange", "yellow"]

    for i, (size, color) in enumerate(zip(triangle\_sizes, triangle\_colors)):
        x = {-}300 + i * 30
        y = 200 {-} i * 20
```

```
        draw\_triangle(artist, x, y, size, color)

    \# Animated star variations
    star\_sizes = [30, 50, 70, 90]
    star\_colors = ["pink", "lightblue", "lightgreen", "gold"]

    for i, (size, color) in enumerate(zip(star\_sizes, star\_colors)):
        angle = i * 90
        x = 150 + math.cos(math.radians(angle)) * 80
        y = 100 + math.sin(math.radians(angle)) * 80

        artist.penup()
        artist.goto(x, y)
        artist.setheading(angle)
        artist.pendown()
        artist.color(color)
        artist.fillcolor(color)

        artist.begin\_fill()
        for \_ in range(5):
            artist.forward(size)
            artist.right(144)
        artist.end\_fill()

    \# Octagon pattern
    for i in range(3):
        size = 40 + i * 15
        color\_intensity = 0.3 + i * 0.2
        draw\_octagon(artist, {-}100, {-}100 + i * 80, size,
                     (color\_intensity, 0, color\_intensity))

    \# Mathematical information
    artist.penup()
    artist.goto(0, {-}250)
    artist.color("white")
    artist.write("Geometric Shapes {- Mathematical Properties"},
                 align="center", font=("Arial", 16, "bold"))

    artist.goto(0, {-}280)
    artist.write("Triangle: Sum of angles = 180^, Star: 36^ points, Octagon: Sum = 1080^",
                 align="center", font=("Arial", 12, "normal"))

    artist.hideturtle()
    screen.exitonclick()

def calculate\_shape\_properties():
    """Calculate and display mathematical properties"""
    shapes\_info = \{
        "Triangle": \{
            "sides": 3,
            "internal\_angle": 180 * (3{-}2) / 3,
            "external\_angle": 360 / 3,
            "sum\_of\_angles": 180 * (3{-}2)
        \,}
        "Star (5{-point)"}: \{
            "points": 5,
            "point\_angle": 36,
            "turn\_angle": 144,
            "total\_rotation": 720
        \,}
        "Octagon": \{
```

```
            "sides": 8,
            "internal\_angle": 180 * (8{-}2) / 8,
            "external\_angle": 360 / 8,
            "sum\_of\_angles": 180 * (8{-}2)
        \}
    \}

    print("{n}" + "="*50)
    print("GEOMETRIC SHAPES {- MATHEMATICAL PROPERTIES"})
    print("="*50)

    for shape, props in shapes\_info.items():
        print(f"{n}\{shape\}:")
        for prop, value in props.items():
            print(f"  \{prop.replace({\_}, { }).title()\}: \{value\}^" if {angle} in prop else f"  \{pr

\# Run the programs
print("Choose drawing mode:")
print("1. Basic Shapes")
print("2. Advanced Shapes with Variations")

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

if choice == "2":
    draw\_advanced\_shapes()
else:
    draw\_geometric\_shapes()

\# Display mathematical properties
calculate\_shape\_properties()
```

- **Angle Calculation**: shape correct turn angles calculation
- **Fill Technique**: begin_fill() end_fill() shape automatically fill
- **Mathematical Foundation**: Geometry principles shapes construct

Mnemonic

“Triangle 120, Star 144, Octagon 45”