Advanced Python Programming (4321602) - Winter 2023 Solution

Milav Dabgar

January 24, 2024

Question 1

Question 1(a) [3 marks]

What is Dictionary? Explain with example.

> **Solution**
>
> **Dictionary** is a collection of key-value pairs in Python which is mutable and ordered.
> **Dictionary Properties:**
>
> | Property | Description |
> |----------|-------------|
> | **Mutable** | Values can be changed |
> | **Ordered** | Insertion order is maintained (Python 3.7+) |
> | **Indexed** | Accessed via keys |
> | **No Duplicates** | Duplicate keys are not allowed |
>
> ```python
> # Dictionary Example
> student = {
>     "name": "Raj",
>     "age": 20,
>     "course": "IT"
> }
> print(student["name"])  # Output: Raj
> ```
>
> - **Key-Value Structure**: Each element has a key and a value
> - **Fast Access**: Data access in O(1) time complexity
> - **Dynamic Size**: Size can be increased or decreased at runtime
>
> > **Mnemonic**
> >
> > Dictionary = Key Value Treasure

Question 1(b) [4 marks]

Explain Tuple Built-in functions and methods.

> **Solution**
>
> Tuple has limited built-in methods because it is immutable.
> **Tuple Methods:**

Method	Description	Example
count()	Returns frequency of element	`t.count(5)`
index()	Returns first index of element	`t.index('a')`
len()	Returns length of tuple	`len(t)`
max()	Returns maximum value	`max(t)`
min()	Returns minimum value	`min(t)`

```python
# 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 do not modify the tuple
- **Return Values**: All methods return new values
- **Type Conversion**: `tuple()` function can convert list to tuple

Mnemonic

Count Index Length Max Min

Question 1(c) [7 marks]

Write a python program to demonstrate set operations.

Solution

Set operations are based on mathematical set theory.
Set Operations:

Operation	Symbol	Method	Description
Union	|	`union()`	Elements of both sets
Intersection	&	`intersection()`	Common elements
Difference	-	`difference()`	Minus second from first
Symmetric Difference	^	`symmetric_difference()`	Unique elements only

```python
# 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
```

```
17  difference_result = set1 - set2
18  print("Difference:", difference_result)
19
20  # Symmetric Difference
21  sym_diff_result = set1 ^ set2
22  print("Symmetric Difference:", sym_diff_result)
23
24  # Subset and Superset
25  set3 = {1, 2}
26  print("Is set3 subset of set1?", set3.issubset(set1))
27  print("Is set1 superset of set3?", set1.issuperset(set3))
28
```

- **Mathematical Operations**: Implements operations of set theory
- **Efficient Processing**: Duplicate elements are automatically removed
- **Boolean Results**: Subset/superset operations return boolean

> **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 provide powerful tools for data manipulation.
> **Dictionary Methods:**
>
> | Method | Description | Example |
> |---|---|---|
> | **keys()** | Returns all keys | `dict.keys()` |
> | **values()** | Returns all values | `dict.values()` |
> | **items()** | Returns key-value pairs | `dict.items()` |
> | **get()** | Safe value retrieval | `dict.get('key')` |
> | **update()** | Merges dictionary | `dict.update()` |
>
> ```
> 1   # Dictionary Operations Program
> 2   student_data = {
> 3       "name": "Amit",
> 4       "age": 21,
> 5       "course": "IT",
> 6       "semester": 2
> 7   }
> 8
> 9   print("Original Dictionary:", student_data)
> 10
> 11  # Accessing values
> 12  print("Student Name:", student_data.get("name"))
> 13  print("Student Age:", student_data["age"])
> 14
> 15  # Adding new key-value pair
> 16  student_data["city"] = "Ahmedabad"
> 17  print("After adding city:", student_data)
> 18
> 19  # Updating existing value
> ```

```
20  student_data.update({"age": 22, "semester": 3})
21  print("After update:", student_data)
22
23  # Dictionary methods
24  print("Keys:", list(student_data.keys()))
25  print("Values:", list(student_data.values()))
26  print("Items:", list(student_data.items()))
27
28  # Removing elements
29  removed_value = student_data.pop("semester")
30  print("Removed value:", removed_value)
31  print("Final Dictionary:", student_data)
32
```

- **Dynamic Operations**: Keys and values can be added/removed at runtime
- **Safe Access**: `get()` method prevents KeyError
- **Iteration Support**: `keys()`, `values()`, `items()` methods are useful for loops

Mnemonic

Get Keys Values Items Update Pop

Question 2

Question 2(a) [3 marks]

Distinguish between Tuple and List in Python.

Solution

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 cannot be changed once created
- **Performance**: Tuple operations are faster than list
- **Memory Efficient**: Tuple uses less memory

Mnemonic

Tuple Tight, List Light

Question 2(b) [4 marks]

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

Solution

`dir()` function is a built-in function that returns a list of attributes and methods of an object.
dir() Function Features:

Feature	Description
Object Inspection	Shows attributes of object
Method Discovery	Lists available methods
Namespace Exploration	Shows variables of current namespace
Module Analysis	Explores contents of module

```python
# 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**: Useful for knowing capabilities of objects
- **Debugging Tool**: To quickly identify available methods
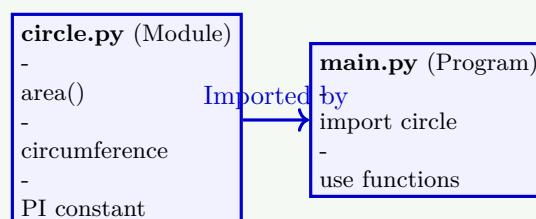- **Learning Aid**: Helpful for exploring new libraries

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 improves code reusability and organization.

File 1: circle.py (Module)

```python
# 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)

```python
# 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**: Organizes functions in a separate file
- **Reusability**: Module can be used in multiple programs
- **Namespace Management**: Functions are accessed via module prefix

> **Mnemonic**
>
> Import Calculate Display

Question 2(a) OR [3 marks]

Explain Nested Tuple with example.

Solution

Nested Tuple contains other tuples inside it, creating a hierarchical data structure.
Nested Tuple Features:

Feature	Description
Multi-dimensional	2D or 3D data structure
Immutable	Immutable at all levels
Indexing	Access using multiple square brackets
Heterogeneous	Can store different data types

```python
# 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**: Useful for grouping related data
- **Immutable Structure**: Structure cannot be changed once created
- **Efficient Access**: Fast index-based 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) is a Python package installer that downloads and installs packages from PyPI.
PIP Commands:

Command	Syntax	Description
Install	`pip install package`	Installs package
Uninstall	`pip uninstall package`	Removes package
List	`pip list`	Shows installed packages
Show	`pip show package`	Displays package info
Upgrade	`pip install --upgrade pkg`	Updates package

```python
# PIP Command Examples (Run in Terminal/Command Prompt)
```

```
 2
 3   # Install a package
 4   # pip install requests
 5
 6   # Install specific version
 7   # pip install Django==3.2.0
 8
 9   # Uninstall a package
10   # pip uninstall numpy
11
12   # List all installed packages
13   # pip list
14
15   # Show package information
16   # pip show matplotlib
17
18   # Upgrade a package
19   # pip install --upgrade pandas
20
21   # Install from requirements file
22   # pip install -r requirements.txt
23
```

- **Package Management**: Can easily manage third-party libraries
- **Version Control**: Specific versions can be installed
- **Dependency Resolution**: Required dependencies are installed automatically

> **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**
>
> Different ways of imports in Python are important for code organization and namespace management.
> **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 |
>
> ```
>  1   # Different Import Ways
>  2
>  3   # 1. Basic Import
>  4   import math
>  5   result = math.sqrt(16)
>  6
>  7   # 2. From Import
>  8   from math import sqrt, pi
> ```

```
 9   result = sqrt(16)
10   area = pi * 5 * 5
11
12   # 3. Alias Import
13   import numpy as np
14   array = np.array([1, 2, 3])
15
16   # 4. Star Import (not recommended)
17   from math import *
18   result = cos(0)
19
20   # 5. Package Import
21   from mypackage import module1
22   from mypackage.subpackage import module3
23
```

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

Question 3(a) [3 marks]

Describe Runtime Error and Syntax Error. Explain with example.

Solution

Error Types Comparison:

Error Type	When Occurs	Detection	Example
Syntax Error	Code parsing time	Before execution	Missing colon
Runtime Error	During execution	While running	Zero division
Logic Error	Always	After execution	Wrong logic

```
 1   # Syntax Error Example
 2   # print("Hello World"  # Missing closing parenthesis
 3   # SyntaxError: unexpected EOF while parsing
 4
 5   # Runtime Error Examples
 6   try:
 7       # ZeroDivisionError
 8       result = 10 / 0
 9   except ZeroDivisionError:
10       print("Cannot divide by zero")
11
12   try:
13       # FileNotFoundError
14       file = open("nonexistent.txt", "r")
15   except FileNotFoundError:
```

```
16        print("File not found")
17
```

- **Syntax Errors**: Detected before code runs
- **Runtime Errors**: occur during program execution
- **Prevention**: Exception handling handles runtime errors

> **Mnemonic**
>
> Syntax Before, Runtime During

Question 3(b) [4 marks]

What is Exception handling in Python? Explain with example.

> **Solution**
>
> **Exception handling** is a mechanism that gracefully handles runtime errors and prevents program crashes.
> **Exception Handling Keywords:**
>
> | Keyword | Purpose | Description |
> |---------|---------|-------------|
> | **try** | Code prone to exception | Risk code block |
> | **except** | To handle exception | Error handling block |
> | **finally** | Always executed | Cleanup code |
> | **else** | If no exception | Success code block |
> | **raise** | To raise manual exception | Custom error throwing |

```python
1   # Exception Handling Example
2   def safe_division(a, b):
3       try:
4           # Code that might raise exception
5           result = a / b
6           print(f"Division successful: {result}")
7
8       except ZeroDivisionError:
9           # Handle specific exception
10          print("Error: Cannot divide by zero")
11          result = None
12
13      except TypeError:
14          # Handle type errors
15          print("Error: Invalid data types")
16          result = None
17
18      else:
19          # Executes if no exception
20          print("Division completed successfully")
21
22      finally:
23          # Always executes
24          print("Division operation finished")
25
26      return result
27
28  # Test the function
29  safe_division(10, 2)    # Normal case
```

```
30   safe_division(10, 0)    # Zero division
31   safe_division(10, "a")  # Type error
32
```

- **Error Prevention**: Prevents program from crashing
- **Graceful Handling**: Provides user-friendly error messages
- **Resource Management**: Cleanup operations in finally block

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

Creating custom exception handling function is important for validation and error control.

```python
1    def safe_integer_division(num1, num2):
2        """
3        Divide two numbers with validation
4        Raises TypeError if arguments are not integers
5        Raises ZeroDivisionError if second argument is 0
6        """
7
8        # Check if both arguments are integers
9        if not isinstance(num1, int):
10           raise TypeError(f"First argument must be integer, got {type(num1).__name__}")
11
12       if not isinstance(num2, int):
13           raise TypeError(f"Second argument must be integer, got {type(num2).__name__}")
14
15       # Check for zero division
16       if num2 == 0:
17           raise ZeroDivisionError("Cannot divide by zero")
18
19       # Perform division
20       result = num1 / num2
21       return result
22
23   # Test the function with different cases
24   def test_division():
25       test_cases = [
26           (10, 2),      # Valid case
27           (15, 3),      # Valid case
28           (10, 0),      # Zero division error
29           (10.5, 2),    # Non-integer first argument
30           (10, 2.5),    # Non-integer second argument
31           ("10", 2),    # String argument
32       ]
33
34       for num1, num2 in test_cases:
35           try:
36               result = safe_integer_division(num1, num2)
37               print(f"{num1} / {num2} = {result}")
38
```

```
39          except TypeError as e:
40              print(f"Type Error: {e}")
41
42          except ZeroDivisionError as e:
43              print(f"Zero Division Error: {e}")
44
45          except Exception as e:
46              print(f"Unexpected Error: {e}")
47
48          print("-" * 40)
49
50  # Run tests
51  test_division()
52
```

- **Input Validation**: Checks type and value of arguments
- **Custom Errors**: Raises specific exceptions
- **Error Messages**: Clear and 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**
>
> **Built-in Exceptions:**
>
> | Exception | Cause | Example |
> |---|---|---|
> | **ValueError** | Invalid value for operation | `int("abc")` |
> | **TypeError** | Wrong data type | `"5" + 5` |
> | **IndexError** | Index out of range | `list[10]` |
> | **KeyError** | Dictionary key not found | `dict["missing"]` |
> | **FileNotFoundError** | File does not exist | `open("missing.txt")` |
>
> - **Automatic Detection**: Python automatically raises these exceptions
> - **Specific Handling**: Each exception has a specific purpose
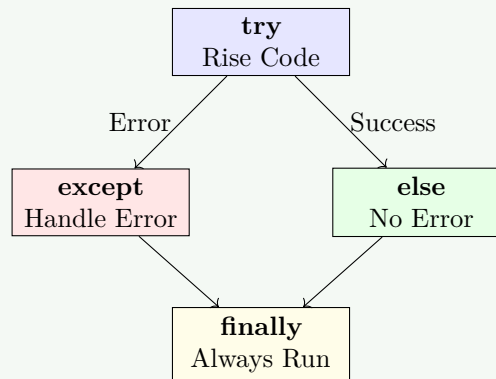> - **Inheritance**: All exceptions inherit from BaseException class
>
> **Mnemonic**
>
> Value Type Index Key File

Question 3(b) OR [4 marks]

Explain try...except...else...finally block with example.

Solution

Exception Block Structure:


```python
1   # Complete Exception Block Example
2   def file_operation(filename):
3       try:
4           # Try to open file
5           f = open(filename, "r")
6           print("File opened successfully")
7
8       except FileNotFoundError:
9           # Handle if file missing
10          print("Error: File not found")
11
12      else:
13          # Run if no error occurred
14          print("Reading file content...")
15          print(f.read())
16          f.close()
17
18      finally:
19          # Always runs
20          print("File operation attempt finished")
21
22   # Test with existing and missing files
23   file_operation("existing.txt")
24   file_operation("missing.txt")
25
```

- **Else Block**: Executes only if try block raises no exception
- **Finally Block**: Executes regardless of error (cleanup code)
- **Complete Flow**: Covers all possibilities of execution

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 allow custom validation logic.
>
> ```python
> class ShortTextError(Exception):
>     """Custom exception for text validation"""
>     def __init__(self, length):
>         self.length = length
>         self.message = f"Text too short ({length} chars). Minimum 10 required."
>         super().__init__(self.message)
>
> def validate_text():
>     while True:
>         try:
>             # Get user input
>             text = input("Enter text (min 10 chars): ")
>
>             # Check length
>             if len(text) < 10:
>                 raise ShortTextError(len(text))
>
>             print(f"Valid text accepted: {text}")
>             break
>
>         except ShortTextError as e:
>             print(f"Error: {e}")
>             print("Please try again.\n")
>
>         except Exception as e:
>             print(f"Unexpected error: {e}")
>
> # Run the validation
> print("--- Text Validation Program ---")
> # validate_text() # Uncomment to run
>
> ```
>
> - **Inheritance**: Custom exception class inherits from `Exception`
> - **Raising**: `raise` keyword used to trigger exception
> - **Usage**: Helps in implementing domain-specific constraints
>
> > **Mnemonic**
> >
> > Class Inherit Raise Catch

Question 4

Question 4(a) [3 marks]

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

> **Solution**
>
> **Text vs Binary File:**

Feature	Text File	Binary File
Content	Human readable characters	Machine readable (0s and 1s)
Encoding	Uses encoding (ASCII/UTF-8)	No encoding used
Extensions	.txt, .py, .csv	.bin, .jpg, .exe
EOL	Handles Newline translation	No translation
Processing	Slower processing	Faster processing

- **Readability**: Text files can be opened in simple editors
- **Portability**: Binary files are continuous stream of bytes
- **Usage**: Text for documents, Binary for images/videos/data

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 involves reading, analyzing, and writing to multiple files.

```python
def separate_case_chars(source_file):
    try:
        # Strings to store separated characters
        upper_chars = ""
        lower_chars = ""

        # Read source file
        with open(source_file, "r") as f:
            content = f.read()

            # Process each character
            for char in content:
                if char.isupper():
                    upper_chars += char
                elif char.islower():
                    lower_chars += char

        # Write uppercase characters
        with open("uppercase.txt", "w") as f:
            f.write(upper_chars)

        # Write lowercase characters
        with open("lowercase.txt", "w") as f:
            f.write(lower_chars)

        print("Characters separated successfully!")

    except FileNotFoundError:
        print("Source file not found")

# Create a sample file first
with open("input.txt", "w") as f:
```

```
33       f.write("Hello World! Python Programming.")
34
35  # Run separation
36  separate_case_chars("input.txt")
37
```

- **String Methods**: `isupper()` and `islower()` check case
- **File Handling**: Uses `with` statement for safe file operations
- **Data Separation**: Content is filtered into different streams

Mnemonic

Read Loop Check Write

Question 4(c) [7 marks]

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

Solution

`dump()` and `load()` are part of **pickle** module used for object serialization.
Pickle Methods:

Method	Purpose	Mode
dump(obj, file)	Serializes object to file	Write Binary ('wb')
load(file)	Deserializes object from file	Read Binary ('rb')

```
1   import pickle
2
3   # Data to serialize
4   student = {
5       "name": "Rahul",
6       "roll": 101,
7       "marks": [85, 90, 88]
8   }
9
10  # DUMP Example
11  try:
12      with open("data.pkl", "wb") as f:
13          pickle.dump(student, f)
14      print("Data dumped successfully")
15  except Exception as e:
16      print(f"Error: {e}")
17
18  # LOAD Example
19  try:
20      with open("data.pkl", "rb") as f:
21          loaded_data = pickle.load(f)
22      print("Data loaded successfully:")
23      print(loaded_data)
24  except Exception as e:
25      print(f"Error: {e}")
26
```

- **Serialization**: Converting object to byte stream (Pickling)
- **Deserialization**: Converting byte stream back to object (Unpickling)
- **Persistence**: Allows saving state of objects to disk

> **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**
>
> **Python File Modes:**
>
> | Mode | Name | Description |
> |------|------|-------------|
> | 'r' | Read | Default mode. Opens for reading. |
> | 'w' | Write | Opens for writing. Overwrites file. |
> | 'a' | Append | Opens for writing. Appends to end. |
> | 'x' | Create | Creates new file. Fails if exists. |
> | 'b' | Binary | Binary mode (e.g., 'rb', 'wb'). |
> | '+' | Update | Read and Write (e.g., 'r+', 'w+'). |
>
> - **Safety**: 'x' prevents accidental overwriting
> - **Binary**: 'b' must be used for non-text files
> - **Combination**: '+ can be combined with other modes
>
> > **Mnemonic**
> >
> > Read Write Append Create

Question 4(b) OR [4 marks]

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

> **Solution**
>
> **Note**: Python has `writelines()` but no `writeline()`.
> **Line Operations:**
>
> | Function | Purpose | Example |
> |----------|---------|---------|
> | **readline()** | Reads single line | `line = f.readline()` |
> | **readlines()** | Reads all lines into list | `lines = f.readlines()` |
> | **writelines()** | Writes list of strings | `f.writelines(list)` |
>
> ```python
> # Write lines
> lines = ["First Line\n", "Second Line\n"]
> with open("demo.txt", "w") as f:
>     f.writelines(lines)
>
> # Read lines
> with open("demo.txt", "r") as f:
>     line1 = f.readline()
> ```

```
9        print(f"Read 1: {line1.strip()}")
10
11       line2 = f.readline()
12       print(f"Read 2: {line2.strip()}")
13
```

- **Sequential**: readline moves pointer line by line
- **List Support**: writelines accepts iterable (list/tuple)
- **Memory**: readline is efficient for large files

> **Mnemonic**
>
> Read One, Write Many

Question 4(c) OR [7 marks]

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

> **Solution**
>
> seek() moves file pointer, tell() returns current position.
>
> ```python
> 1  # Seek and Tell Demonstration
> 2  filename = "seek_demo.txt"
> 3
> 4  # Create file
> 5  with open(filename, "w") as f:
> 6      f.write("Hello Python World")
> 7
> 8  # Read operations
> 9  with open(filename, "r") as f:
> 10     # Initial position
> 11     print(f"Start Position: {f.tell()}")  # 0
> 12
> 13     # Read first 5 chars
> 14     print(f"Read: {f.read(5)}")          # Hello
> 15     print(f"Current Position: {f.tell()}") # 5
> 16
> 17     # Seek to start
> 18     f.seek(0)
> 19     print(f"After seek(0): {f.tell()}")   # 0
> 20
> 21     # Seek to specific position
> 22     f.seek(6)
> 23     print(f"After seek(6): {f.read(6)}")  # Python
> 24
> 25     # Seek from end (requires binary mode usually, or specific syntax)
> 26     # f.seek(0, 2) moves to end
> 27
> ```
>
> | Method | Syntax |
> |--------|--------|
> | **tell()** | `pos = f.tell()` |
> | **seek()** | `f.seek(offset, whence)` |
>
> - **Navigation**: Allows random access in files
> - **Whence**: 0=Start, 1=Current, 2=End

- **Binary**: Relative seeking works best in binary mode

Mnemonic

Tell Position, Seek Location

Question 5

Question 5(a) [3 marks]

Draw the shape of circle and rectangle using turtle and fill with red color.

Solution

```
1   import turtle
2
3   t = turtle.Turtle()
4
5   # Fill color set to red
6   t.fillcolor("red")
7
8   # Draw Circle
9   t.begin_fill()
10  t.circle(50)        # Draw circle with radius 50
11  t.end_fill()
12
13  # Move to new position
14  t.penup()
15  t.goto(100, 0)
16  t.pendown()
17
18  # Draw Rectangle
19  t.begin_fill()
20  for _ in range(2):
21      t.forward(100) # Length
22      t.right(90)
23      t.forward(50)  # Width
24      t.right(90)
25  t.end_fill()
26
27  turtle.done()
28
```

- **begin_fill()**: Starts filling shape
- **end_fill()**: Completes filling shape
- **Geometry**: Circle and loop-based rectangle drawing

Mnemonic

Begin Fill End

Question 5(b) [4 marks]

Explain various inbuilt methods for changing the direction of turtle.

> **Solution**
>
> **Turtle Direction Methods:**
>
> | Method | Description | Example |
> |--------|-------------|---------|
> | **right(angle)** | Turn right by angle | `t.right(90)` |
> | **left(angle)** | Turn left by angle | `t.left(45)` |
> | **setheading(angle)** | Set absolute angle | `t.setheading(0)` |
> | **towards(x,y)** | Point towards coords | `t.towards(0,0)` |
>
> ```python
> # Direction Examples
> import turtle
> t = turtle.Turtle()
>
> t.forward(100)
> t.right(90)        # Turn 90 degrees right
> t.forward(50)
> t.left(45)         # Turn 45 degrees left
> t.setheading(180) # Face West (180 degrees)
>
> ```
>
> - **Relative**: right/left are relative to current direction
> - **Absolute**: setheading uses 0-360 degree system
> - **Navigation**: Essential for drawing complex shapes
>
> > **Mnemonic**
> >
> > Right Left Heading Towards

Question 5(c) [7 marks]

Write a python program to draw rainbow using turtle.

> **Solution**
>
> Rainbow drawing utilizes concentric semi-circles with VIBGYOR colors.
>
> ```python
> import turtle
>
> def draw_rainbow():
>     # Setup screen
>     wn = turtle.Screen()
>     wn.bgcolor("skyblue")
>
>     # Setup turtle
>     t = turtle.Turtle()
>     t.speed(5)
>     t.pensize(10)
>
>     # Rainbow colors (VIBGYOR reversed)
>     colors = ['violet', 'indigo', 'blue', 'green',
>               'yellow', 'orange', 'red']
>
>     # Initial position parameters
>     radius = 180
>
>     # Draw arcs
> ```

```
21      for color in colors:
22          t.penup()
23          t.goto(0, -50)       # Center bottom
24          t.setheading(90)     # Face up
25          t.right(90)          # Face right (0 deg)
26          t.forward(radius)    # Move to start of arc
27          t.left(90)           # Face up again
28          t.pendown()
29
30          t.pencolor(color)
31          t.circle(radius, 180) # Draw semi-circle
32
33          radius -= 20        # Decrease radius for next color
34
35      t.hideturtle()
36      turtle.done()
37
38  draw_rainbow()
39
```

- **Looping**: Iterates through list of colors
- **Concentric**: Radius decreases for each inner arc
- **Arc**: `circle(radius, 180)` draws half circle

Mnemonic

Colors Loop Radius Circle