Problem Solving using Python

Unit 2: Data Structures and Files

Prepared by:

Prof. Mohammed Siraj B

School of Engineering
St Aloysius (Deemed to be University), Mangalore

`siraj_b@staloysius.edu.in`

September 9, 2025

Contents

1. Data Structures

Data structures are fundamental ways to organize and store data in a computer so that it can be accessed and modified efficiently. Python provides several built-in data structures.

1.1 Strings

Definition: A string is a sequence of characters. In Python, strings are immutable, meaning once created, their content cannot be changed.

Explanation: Strings are used to represent text. They can contain letters, numbers, symbols, and spaces. Python strings are enclosed in single quotes (´´), double quotes (¨¨), or triple quotes (´´´´´´ or ¨¨¨¨¨¨) for multi-line strings.

General Syntax:

```
my_string = "Hello, Python!"
another_string = 'Single quotes work too.'
multi_line_string = """This is a
multi-line string.
"""
```

Examples:

Example 1: String Creation

```
str1 = "Hello, World!"
str2 = 'Python Programming'
str3 = """This is a
multi-line
string example.
"""

print(str1)
print(str2)
print(str3)
```

Listing 1: String Creation

Output:

```
Hello, World!
Python Programming
This is a
multi-line
string example.
```

Example 2: String Indexing Accessing individual characters using their position (index). Python uses zero-based indexing.

```python
my_string = "PYTHON"

print(f"First character: {my_string[0]}") # P
print(f"Third character: {my_string[2]}") # T
print(f"Last character: {my_string[-1]}") # N (negative indexing starts from the
    end)
print(f"Second to last character: {my_string[-2]}") # O
```

Listing 2: String Indexing

Output:

```
First character: P
Third character: T
Last character: N
Second to last character: O
```

Example 3: String Slicing Extracting a portion (substring) of a string.

```python
my_string = "Python Programming"

print(f"Slice from index 0 to 5 (exclusive): {my_string[0:6]}") # Python
print(f"Slice from index 7 to end: {my_string[7:]}") # Programming
print(f"Slice from beginning to index 6 (exclusive): {my_string[:6]}") # Python
print(f"Slice with step (every second character): {my_string[::2]}") # Pto rgamn
print(f"Reverse string: {my_string[::-1]}") # gnimmargorP nohtyP
```

Listing 3: String Slicing

Output:

```
Slice from index 0 to 5 (exclusive): Python
Slice from index 7 to end: Programming
Slice from beginning to index 6 (exclusive): Python
Slice with step (every second character): Pto rgamn
Reverse string: gnimmargorP nohtyP
```

Example 4: Common String Methods Strings have many built-in methods for various operations.

```python
text = " Hello, Python World! "

print(f"Original: '{text}'")
print(f"Uppercase: {text.upper()}")
print(f"Lowercase: {text.lower()}")
print(f"Stripped (whitespace removed): '{text.strip()}'")
print(f"Replaced 'World' with 'Universe': {text.replace('World', 'Universe')}")
print(f"Starts with ' Hello': {text.startswith(' Hello')}")
print(f"Ends with 'World! ': {text.endswith('World! ')}")
print(f"Split by space: {text.strip().split(' ')}")
print(f"Find 'Python': {text.find('Python')}") # Returns starting index or -1 if not found
print(f"Count 'o': {text.count('o')}")
print(f"Is alphanumeric? ('Python123'): {'Python123'.isalnum()}")
```

Listing 4: Common String Methods

Output:

```
Original: ' Hello, Python World! '
Uppercase: HELLO, PYTHON WORLD!
Lowercase: hello, python world!
Stripped (whitespace removed): 'Hello, Python World!'
Replaced 'World' with 'Universe': Hello, Python Universe!
Starts with ' Hello': True
Ends with 'World! ': True
Split by space: ['Hello,', 'Python', 'World!']
Find 'Python': 10
Count 'o': 3
Is alphanumeric? ('Python123'): True
```

Exercises:

1. Given the string `s = "Programming is Fun!"`:
   - Print the character at index 4.
   - Print the substring from index 0 up to (but not including) index 11.
   - Print the string in reverse order.
2. Ask the user to input their full name. Then, convert the name to uppercase and print it. Also, count how many times the letter `'a'` (case-insensitive) appears in their name.
3. Write a Python program that checks if a given word is a palindrome (reads the same forwards and backward). Ignore case sensitivity. (e.g., "Madam" is a palindrome).

1.2 Lists

Definition: A list is an ordered, mutable (changeable) collection of items. Items in a list are enclosed in square brackets `[]` and separated by commas.

Explanation: Lists are one of the most versatile data structures in Python. They can hold items of different data types, and their elements can be added, removed, or modified after creation. Lists are dynamic, meaning their size can change during execution.

General Syntax:

```
my_list = [item1, item2, item3, ...]
```

Examples:

Example 1: List Creation and Access

```python
# Creating a list of integers
numbers = [10, 20, 30, 40, 50]
print(f"Numbers list: {numbers}")

# Creating a list with mixed data types
mixed_list = ["apple", 1, True, 3.14]
print(f"Mixed list: {mixed_list}")

# Accessing elements (indexing)
print(f"First element: {numbers[0]}")
print(f"Last element: {numbers[-1]}")

# Slicing lists
print(f"Elements from index 1 to 3 (exclusive): {numbers[1:4]}")
print(f"All elements from index 2 to end: {numbers[2:]}")
```

Listing 5: List Creation and Access

Output:

```
Numbers list: [10, 20, 30, 40, 50]
Mixed list: ['apple', 1, True, 3.14]
First element: 10
Last element: 50
Elements from index 1 to 3 (exclusive): [20, 30, 40]
All elements from index 2 to end: [30, 40, 50]
```

Example 2: Modifying Lists (Mutability)

```python
fruits = ["apple", "banana", "cherry"]
print(f"Original list: {fruits}")

# Change an element
fruits[1] = "orange"
print(f"After changing element: {fruits}")

# Add an element to the end
fruits.append("grape")
print(f"After appending: {fruits}")

# Insert an element at a specific index
fruits.insert(1, "kiwi")
print(f"After inserting: {fruits}")

# Remove an element by value
fruits.remove("apple")
print(f"After removing 'apple': {fruits}")
```

```python

# Remove and return the last element
popped_fruit = fruits.pop()
print(f"Popped fruit: {popped_fruit}, List after pop: {fruits}")

# Sort the list
fruits.sort()
print(f"Sorted list: {fruits}")

# Reverse the list
fruits.reverse()
print(f"Reversed list: {fruits}")
```

Listing 6: Modifying Lists

Output:

```
Original list: ['apple', 'banana', 'cherry']
After changing element: ['apple', 'orange', 'cherry']
After appending: ['apple', 'orange', 'cherry', 'grape']
After inserting: ['apple', 'kiwi', 'orange', 'cherry', 'grape']
After removing 'apple': ['kiwi', 'orange', 'cherry', 'grape']
Popped fruit: grape, List after pop: ['kiwi', 'orange', 'cherry']
Sorted list: ['cherry', 'kiwi', 'orange']
Reversed list: ['orange', 'kiwi', 'cherry']
```

Exercises:

1. Create a list of your five favorite movies. Add a new movie to the list, then remove one you don't like as much. Print the final list.
2. Given a list of numbers `[1, 5, 2, 8, 3]`, find the maximum and minimum values without using built-in `max()` or `min()` functions.
3. Write a Python program to concatenate two lists: `list1 = [1, 2, 3]` and `list2 = [4, 5, 6]`.

1.3 Tuples

Definition: A tuple is an ordered, immutable (unchangeable) collection of items. Tuples are enclosed in parentheses `()`.

Explanation: Tuples are similar to lists but cannot be modified after creation. This immutability makes them suitable for data that should not change, such as coordinates, database records, or function arguments that should remain constant. They are also generally faster than lists for iteration.

General Syntax:

```python
my_tuple = (item1, item2, item3, ...)
```

Examples:

Example 1: Tuple Creation and Access

```python
# Creating a tuple
coordinates = (10.0, 20.5)
print(f"Coordinates tuple: {coordinates}")

# Tuple with mixed data types
person_info = ("Alice", 30, "New York")
print(f"Person info tuple: {person_info}")

# Accessing elements (indexing)
print(f"First coordinate: {coordinates[0]}")
print(f"Person's name: {person_info[0]}")

# Slicing tuples
print(f"Slice of person info: {person_info[0:2]}")
```

Listing 7: Tuple Creation and Access

Output:

```
Coordinates tuple: (10.0, 20.5)
Person info tuple: ('Alice', 30, 'New York')
First coordinate: 10.0
Person's name: Alice
Slice of person info: ('Alice', 30)
```

Example 2: Immutability of Tuples

```python
my_tuple = (1, 2, 3)
# my_tuple[0] = 5 # This will raise a TypeError
# print(my_tuple)

# You can reassign the entire tuple, but not modify its elements
my_tuple = (4, 5, 6)
print(f"Reassigned tuple: {my_tuple}")
```

Listing 8: Immutability of Tuples

Output:

```
Reassigned tuple: (4, 5, 6)
```

Example 3: Tuple Packing and Unpacking

```python
# Tuple packing
packed_data = 10, "hello", True
print(f"Packed data: {packed_data}, Type: {type(packed_data)}")

# Tuple unpacking
```

```python
x, y, z = packed_data
print(f"Unpacked x: {x}, y: {y}, z: {z}")

# Swapping variables using tuple unpacking
a = 100
b = 200
print(f"Before swap: a={a}, b={b}")
a, b = b, a
print(f"After swap: a={a}, b={b}")
```

Listing 9: Tuple Packing and Unpacking

Output:

```
Packed data: (10, 'hello', True), Type: <class 'tuple'>
Unpacked x: 10, y: hello, z: True
Before swap: a=100, b=200
After swap: a=200, b=100
```

Exercises:

1. Create a tuple `rgb_color` with three integer values representing an RGB color (e.g., `(255, 0, 0)` for red). Try to change one of the values and observe the error.
2. Given a tuple `student = ("John Doe", 20, "Computer Science")`, unpack its elements into three separate variables: `name`, `age`, and `major`. Print these variables.
3. Explain a scenario where using a tuple would be more appropriate than using a list.

1.4 Sets

Definition: A set is an unordered collection of unique items. Sets are mutable, but their elements must be immutable (hashable).

Explanation: Sets are primarily used to perform mathematical set operations like union, intersection, difference, and symmetric difference. They are also useful for quickly checking for membership and removing duplicate elements from a collection.

General Syntax:

```python
my_set = {item1, item2, item3, ...} # For non-empty sets
empty_set = set() # To create an empty set
```

Examples:

Example 1: Set Creation and Basic Operations

```python
# Creating a set
my_set = {1, 2, 3, 4, 5}
print(f"My set: {my_set}")
```

```python

# Sets automatically remove duplicates
duplicate_set = {1, 2, 2, 3, 4, 4, 5}
print(f"Set with duplicates removed: {duplicate_set}")

# Adding elements
my_set.add(6)
print(f"After adding 6: {my_set}")

# Removing elements
my_set.remove(3) # Raises KeyError if item not found
print(f"After removing 3: {my_set}")

my_set.discard(10) # Does not raise error if item not found
print(f"After discarding 10: {my_set}")

# Checking for membership
print(f"Is 4 in my_set? {4 in my_set}")
print(f"Is 3 in my_set? {3 in my_set}")
```

Listing 10: Set Creation and Basic Operations

Output:

```
My set: {1, 2, 3, 4, 5}
Set with duplicates removed: {1, 2, 3, 4, 5}
After adding 6: {1, 2, 4, 5, 6}
After removing 3: {1, 2, 4, 5, 6} # Note: 3 was removed
After discarding 10: {1, 2, 4, 5, 6}
Is 4 in my_set? True
Is 3 in my_set? False
```

Example 2: Set Operations

```python
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}

print(f"Set A: {set_a}")
print(f"Set B: {set_b}")

# Union (all unique elements from both sets)
print(f"Union (A | B): {set_a | set_b}")
print(f"Union (A.union(B)): {set_a.union(set_b)}")

# Intersection (common elements)
print(f"Intersection (A & B): {set_a & set_b}")
print(f"Intersection (A.intersection(B)): {set_a.intersection(set_b)}")

# Difference (elements in A but not in B)
print(f"Difference (A - B): {set_a - set_b}")
print(f"Difference (A.difference(B)): {set_a.difference(set_b)}")

# Symmetric Difference (elements in A or B but not both)
print(f"Symmetric Difference (A ^ B): {set_a ^ set_b}")
print(f"Symmetric Difference (A.symmetric_difference(B)): {set_a.
    symmetric_difference(set_b)}")
```

Listing 11: Set Operations

Output:

```
Set A: {1, 2, 3, 4, 5}
Set B: {4, 5, 6, 7, 8}
Union (A | B): {1, 2, 3, 4, 5, 6, 7, 8}
Union (A.union(B)): {1, 2, 3, 4, 5, 6, 7, 8}
Intersection (A & B): {4, 5}
Intersection (A.intersection(B)): {4, 5}
Difference (A - B): {1, 2, 3}
Difference (A.difference(B)): {1, 2, 3}
Symmetric Difference (A ^ B): {1, 2, 3, 6, 7, 8}
Symmetric Difference (A.symmetric_difference(B)): {1, 2, 3, 6, 7, 8}
```

Exercises:

1. Create a set of unique numbers from the following list: `[1, 2, 2, 3, 4, 4, 5, 5, 5]`.
2. Given two sets of students enrolled in different courses: `math_students = Älice; Böb; Charlie`

   and `physics_students = Charlie; David; Eve`

   . Find:

   - Students enrolled in both Math and Physics.
   - Students enrolled in Math but not Physics.
   - All unique students enrolled in either Math or Physics.
3. Explain why lists cannot be elements of a set, but tuples can.

1.5 Dictionaries

Definition: A dictionary is an unordered, mutable collection of key-value pairs. Each key must be unique and immutable, while values can be of any data type and can be duplicated.

Explanation: Dictionaries are used to store data values in `key:value` pairs. They are optimized for retrieving values when the key is known. Think of a real-world dictionary where you look up a word (key) to find its definition (value).

General Syntax:

```
my_dict = {
    key1: value1,
    key2: value2,
    # ...
}
```

Examples:

Example 1: Dictionary Creation and Access

```python
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 20,
    "major": "Computer Science",
    "is_enrolled": True
}
print(f"Student dictionary: {student}")

# Accessing values using keys
print(f"Student's name: {student["name"]}")
print(f"Student's major: {student.get("major")}") # Using .get() is safer,
    returns None if key not found

# Trying to access a non-existent key (will raise KeyError)
# print(student["gpa"])
print(f"GPA (using .get()): {student.get("gpa")}") # Returns None
print(f"GPA (using .get() with default value): {student.get("gpa", "N/A")}")
```

Listing 12: Dictionary Creation and Access

Output:

```
Student dictionary: {'name': 'Alice', 'age': 20, 'major': 'Computer Science', '
    is_enrolled': True}
Student's name: Alice
Student's major: Computer Science
GPA (using .get()): None
GPA (using .get() with default value): N/A
```

Example 2: Modifying Dictionaries

```python
student = {"name": "Alice", "age": 20}
print(f"Original dictionary: {student}")

# Adding a new key-value pair
student["city"] = "New York"
print(f"After adding city: {student}")

```

```python
# Updating an existing value
student["age"] = 21
print(f"After updating age: {student}")

# Removing a key-value pair
removed_age = student.pop("age")
print(f"Removed age: {removed_age}, Dictionary after pop: {student}")

# Removing the last inserted item (Python 3.7+)
# student.popitem()
# print(f"After popitem: {student}")

# Clear all items
# student.clear()
# print(f"After clear: {student}")
```

Listing 13: Modifying Dictionaries

Output:

```
Original dictionary: {'name': 'Alice', 'age': 20}
After adding city: {'name': 'Alice', 'age': 20, 'city': 'New York'}
After updating age: {'name': 'Alice', 'age': 21, 'city': 'New York'}
Removed age: 21, Dictionary after pop: {'name': 'Alice', 'city': 'New York'}
```

Example 3: Iterating Through Dictionaries

```python
student = {"name": "Bob", "age": 22, "course": "Physics"}

print("Iterating through keys:")
for key in student:
    print(key)

print("\nIterating through values:")
for value in student.values():
    print(value)

print("\nIterating through key-value pairs:")
for key, value in student.items():
    print(f"{key}: {value}")
```

Listing 14: Iterating Through Dictionaries

Output:

```
Iterating through keys:
name
age
course

Iterating through values:
```

```
Bob
22
Physics

Iterating through key-value pairs:
name: Bob
age: 22
course: Physics
```

Exercises:

1. Create a dictionary representing a book with keys like `"title"`, `"author"`, `"year"`, and `"genre"`. Add a new key `"pages"` with an appropriate value. Print the updated dictionary.

2. Given the dictionary `inventory = äpple:̈  50, b̈anana:̈  30, örange:̈  20`, write a program to:

   - Check if "banana" is in the inventory.
   - Update the quantity of "apple" to 60.
   - Remove "orange" from the inventory.

   Print the inventory after each operation.

3. Write a Python program that counts the frequency of each character in a given string using a dictionary.

2. Object-Oriented Concepts

2.1 Introduction to Class and Object

Definition:

- **Object-Oriented Programming (OOP):** A programming paradigm based on the concept of "objects", which can contain data (attributes) and code (methods).
- **Class:** A blueprint or a template for creating objects. It defines the common attributes and methods that all objects of that type will have.
- **Object (Instance):** A real-world entity or an instance of a class. When a class is defined, no memory is allocated until an object is created from it.

Explanation: OOP helps in structuring programs in a way that makes them more modular, reusable, and easier to manage. It mimics real-world entities where objects have characteristics (attributes) and behaviors (methods).

General Syntax:

```python
# Class Definition
class ClassName:
    # Class attributes (optional)
    class_attribute = "I belong to the class"

    # Constructor method (optional, for initializing objects)
    def __init__(self, param1, param2):
        self.param1 = param1 # Instance attribute
        self.param2 = param2 # Instance attribute

    # Instance method
    def method_name(self):
        # Code that operates on the object's data
        print("This is a method.")

# Object Creation (Instantiation)
object_name = ClassName(arg1, arg2)

# Accessing attributes and calling methods
print(object_name.param1)
object_name.method_name()
```

Examples:

Example 1: Defining a Simple Class and Creating Objects Let's create a Dog class.

```python
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # The constructor method
    def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age # Instance attribute

    # An instance method
    def bark(self):
        return f"{self.name} says Woof!"

    # Another instance method
    def description(self):
        return f"{self.name} is {self.age} years old."

# Creating objects (instances) of the Dog class
my_dog = Dog("Buddy", 3)
your_dog = Dog("Lucy", 5)

```

```python
# Accessing attributes
print(f"My dog's name: {my_dog.name}")
print(f"Your dog's age: {your_dog.age}")

# Accessing class attribute
print(f"My dog's species: {my_dog.species}")
print(f"Your dog's species: {your_dog.species}")

# Calling methods
print(my_dog.bark())
print(your_dog.description())
```

Listing 15: Defining a Simple Class

Output:

```
My dog's name: Buddy
Your dog's age: 5
My dog's species: Canis familiaris
Your dog's species: Canis familiaris
Buddy says Woof!
Lucy is 5 years old.
```

Example 2: Attributes and Methods in a `Car` Class

```python
class Car:
    def __init__(self, brand, model, year, color):
        self.brand = brand
        self.model = model
        self.year = year
        self.color = color
        self.is_running = False # Default attribute

    def start_engine(self):
        if not self.is_running:
            self.is_running = True
            return f"The {self.color} {self.brand} {self.model}'s engine started."
        else:
            return f"The {self.brand} {self.model}'s engine is already running."

    def stop_engine(self):
        if self.is_running:
            self.is_running = False
            return f"The {self.brand} {self.model}'s engine stopped."
        else:
            return f"The {self.brand} {self.model}'s engine is already off."

    def display_info(self):
        return f"This is a {self.year} {self.brand} {self.model} ({self.color}).
            Engine running: {self.is_running}"
```

```python

# Create car objects
car1 = Car("Toyota", "Camry", 2020, "Blue")
car2 = Car("Honda", "Civic", 2022, "Red")

print(car1.display_info())
print(car1.start_engine())
print(car1.display_info())
print(car1.stop_engine())
print(car1.display_info())

print("\n---")

print(car2.display_info())
print(car2.start_engine())
print(car2.start_engine()) # Try starting again
```

Listing 16: Car Class Attributes and Methods

Output:

```
This is a 2020 Toyota Camry (Blue). Engine running: False
The Blue Toyota Camry's engine started.
This is a 2020 Toyota Camry (Blue). Engine running: True
The Toyota Camry's engine stopped.
This is a 2020 Toyota Camry (Blue). Engine running: False

---
This is a 2022 Honda Civic (Red). Engine running: False
The Red Honda Civic's engine started.
The Honda Civic's engine is already running.
```

Exercises:

1. Create a class `Book` with attributes `title`, `author`, and `pages`. Include a method `get_info()` that returns a string like "Title by Author, Pages pages.". Create two `Book` objects and print their information.

2. Define a class `Student` with attributes `name`, `student_id`, and `grades` (a list of numbers). Add a method `calculate_average_grade()` that returns the average of the student's grades.

3. Explain the difference between a class attribute and an instance attribute in Python.

3. File Handling

Definition: File handling refers to the operations involved in working with files on a computer system, such as creating, reading, writing, and deleting files.

Explanation: Programs often need to interact with external files to store data persistently (so it's not lost when the program ends) or to read existing data. Python provides built-in functions and modules to handle various file operations.

General Syntax:

```python
# Opening a file
file_object = open("filename.txt", "mode")

# Reading from a file
content = file_object.read()

# Writing to a file
file_object.write("Some text")

# Closing a file (important!)
file_object.close()

# Using 'with' statement (recommended for automatic closing)
with open("filename.txt", "mode") as file_object:
    # file operations
    pass
```

Common File Modes:

- `"r"`: Read mode (default). Opens a file for reading. Error if the file does not exist.
- `"w"`: Write mode. Opens a file for writing. Creates the file if it does not exist, or truncates (empties) the file if it exists.
- `"a"`: Append mode. Opens a file for appending. Creates the file if it does not exist. If the file exists, new content is added to the end.
- `"x"`: Exclusive creation mode. Creates a new file. Error if the file already exists.
- `"r+"`: Read and Write mode. Opens a file for both reading and writing.
- `"w+"`: Write and Read mode. Opens a file for both writing and reading. Creates the file if it does not exist, or truncates the file if it exists.
- `"a+"`: Append and Read mode. Opens a file for both appending and reading. Creates the file if it does not exist. If the file exists, new content is added to the end.

3.1 Creating files, reading and writing to text files

Examples:

Example 1: Creating and Writing to a Text File (`"w"` mode)

```python
# Open the file in write mode ('w'). If it exists, its content will be
    overwritten.
with open("my_first_file.txt", "w") as file:
    file.write("Hello, Python file handling!\n")
    file.write("This is the second line.\n")
    file.write("And this is the third.\n")

print("Content written to my_first_file.txt")

# Verify content by reading it back
with open("my_first_file.txt", "r") as file:
    content = file.read()
    print("\nContent of my_first_file.txt:")
    print(content)
```

Listing 17: Writing to Text File

Output:

```
Content written to my_first_file.txt

Content of my_first_file.txt:
Hello, Python file handling!
This is the second line.
And this is the third.
```

Example 2: Appending to a Text File ("a" mode)

```python
# Open the file in append mode ('a'). Content will be added to the end.
with open("my_first_file.txt", "a") as file:
    file.write("\n--- Appended Content ---\n")
    file.write("This line was appended.\n")

print("Content appended to my_first_file.txt")

# Verify content by reading it back
with open("my_first_file.txt", "r") as file:
    content = file.read()
    print("\nContent of my_first_file.txt (after append):")
    print(content)
```

Listing 18: Appending to Text File

Output:

```
Content appended to my_first_file.txt

Content of my_first_file.txt (after append):
Hello, Python file handling!
This is the second line.
```

```
And this is the third.

--- Appended Content ---
This line was appended.
```

Example 3: Reading from a Text File ("r" mode)

```python
# Create a file for reading demonstration
with open("read_example.txt", "w") as file:
    file.write("Line 1: The quick brown fox\n")
    file.write("Line 2: jumps over the lazy dog.\n")
    file.write("Line 3: This is the end.")

print("read_example.txt created for demonstration.")

# Read the entire file
with open("read_example.txt", "r") as file:
    full_content = file.read()
    print("\nFull content:")
    print(full_content)

# Read line by line using readline()
with open("read_example.txt", "r") as file:
    print("\nReading line by line:")
    line1 = file.readline()
    line2 = file.readline()
    print(f"Line 1: {line1.strip()}") # .strip() removes newline character
    print(f"Line 2: {line2.strip()}")

# Read all lines into a list using readlines()
with open("read_example.txt", "r") as file:
    all_lines = file.readlines()
    print("\nReading all lines into a list:")
    for i, line in enumerate(all_lines):
        print(f"List element {i}: {line.strip()}")

# Iterate over file object (most memory efficient for large files)
with open("read_example.txt", "r") as file:
    print("\nIterating over file object:")
    for line in file:
        print(f"Iterated line: {line.strip()}")
```

Listing 19: Reading from Text File

Output:

```
read_example.txt created for demonstration.

Full content:
Line 1: The quick brown fox
Line 2: jumps over the lazy dog.
Line 3: This is the end.

Reading line by line:
Line 1: Line 1: The quick brown fox
Line 2: Line 2: jumps over the lazy dog.

Reading all lines into a list:
List element 0: Line 1: The quick brown fox
List element 1: Line 2: jumps over the lazy dog.
List element 2: Line 3: This is the end.

Iterating over file object:
Iterated line: Line 1: The quick brown fox
Iterated line: Line 2: jumps over the lazy dog.
Iterated line: Line 3: This is the end.
```

Exercises:

1. Write a Python program that creates a file named `my_diary.txt`. Ask the user to enter a diary entry, and then write that entry to the file. If the file already exists, append the new entry to it.
2. Create a text file named `numbers.txt` and write numbers from 1 to 10, each on a new line. Then, write a Python program to read this file, calculate the sum of all numbers, and print the result.
3. Explain the importance of closing files and why the `with` statement is recommended for file handling.

3.2 Working with CSV files

Definition: CSV (Comma Separated Values) is a simple file format used to store tabular data, such as a spreadsheet or database. Each line in a CSV file represents a row, and commas separate the values within each row.

Explanation: Python's built-in `csv` module provides functionality to easily read from and write to CSV files, handling the complexities of quoting and delimiters.

General Syntax (using `csv` module):

```python
import csv

# Writing to CSV
with open("output.csv", "w", newline="") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["header1", "header2"])
    writer.writerow(["value1", "value2"])
```

```python
# Reading from CSV
with open("input.csv", "r", newline="") as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(row)
```

Note: `newline=""` is crucial when opening CSV files to prevent extra blank rows.

Examples:

Example 1: Writing Data to a CSV File

```python
import csv

data = [
    ["Name", "Age", "City"],
    ["Alice", 30, "New York"],
    ["Bob", 24, "London"],
    ["Charlie", 35, "Paris"]
]

with open("students.csv", "w", newline="") as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerows(data)

print("Data written to students.csv")

# Verify content by reading it back
with open("students.csv", "r", newline="") as csvfile:
    csv_reader = csv.reader(csvfile)
    print("\nContent of students.csv:")
    for row in csv_reader:
        print(row)
```

Listing 20: Writing to CSV File

Output:

```
Data written to students.csv

Content of students.csv:
['Name', 'Age', 'City']
['Alice', '30', 'New York']
['Bob', '24', 'London']
['Charlie', '35', 'Paris']
```

Example 2: Reading Data from a CSV File

```python
import csv

```

```python
# First, let's ensure the file exists for reading
csv_content = """
product,price,quantity
Laptop,1200,10
Mouse,25,50
Keyboard,75,20
"""
with open("products.csv", "w", newline="") as f:
    f.write(csv_content)
print("products.csv created for demonstration.")

# Reading CSV data
with open("products.csv", "r", newline="") as csvfile:
    csv_reader = csv.reader(csvfile)

    # Skip header row if present
    header = next(csv_reader)
    print(f"\nHeader: {header}")

    print("Product Inventory:")
    for row in csv_reader:
        product_name = row[0]
        price = float(row[1])
        quantity = int(row[2])
        print(f" {product_name}: Price=${price:.2f}, Quantity={quantity}")
```

Listing 21: Reading from CSV File

Output:

```
products.csv created for demonstration.

Header: ['product', 'price', 'quantity']
Product Inventory:
  Laptop: Price=$1200.00, Quantity=10
  Mouse: Price=$25.00, Quantity=50
  Keyboard: Price=$75.00, Quantity=20
```

Example 3: Reading/Writing CSV with Dictionaries (`DictReader`, `DictWriter`)
This is often more convenient as it treats each row as a dictionary.

```python
import csv

# Writing CSV using DictWriter
fieldnames = ["id", "name", "email"]
users_data = [
    {"id": 1, "name": "John Doe", "email": "john@example.com"},
    {"id": 2, "name": "Jane Smith", "email": "jane@example.com"}
]

```

```python
with open("users.csv", "w", newline="") as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader() # Write the header row
    writer.writerows(users_data)

print("Data written to users.csv using DictWriter")

# Reading CSV using DictReader
with open("users.csv", "r", newline="") as csvfile:
    reader = csv.DictReader(csvfile)
    print("\nUsers from CSV (DictReader):")
    for row in reader:
        print(row) # Each row is an OrderedDict
        print(f" User ID: {row["id"]}, Name: {row["name"]}")
```

Listing 22: CSV with Dictionaries

Output:

```
Data written to users.csv using DictWriter

Users from CSV (DictReader):
OrderedDict([('id', '1'), ('name', 'John Doe'), ('email', 'john@example.com')])
  User ID: 1, Name: John Doe
OrderedDict([('id', '2'), ('name', 'Jane Smith'), ('email', 'jane@example.com')])
  User ID: 2, Name: Jane Smith
```

Exercises:

1. Create a CSV file named `grades.csv` with columns `StudentName`, `Math`, `Science`, `English`. Write at least three rows of sample student data into it.
2. Write a Python program to read the `grades.csv` file you just created. Calculate the average score for each student across all subjects and print it.
3. Imagine you have a CSV file with product information (Product, Price, Stock). Write a program that reads this file, identifies products with stock less than 5, and writes these low-stock products to a new CSV file named `low_stock_alerts.csv`.