SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 2nd semester
PYTHON PROGRAMMING (PCA20C04J)

Lab Manual

Lab 1: Python Numbers, List

Aim

To understand and implement basic operations with Python numbers (integers, floats) and explore fundamental list manipulations including creation, access, modification, and common methods.

Procedure

1. Open a Python interpreter or a Python IDE (like VS Code, PyCharm, or IDLE).
2. Create a new Python file (e.g., `lab1.py`).
3. Write the source code as provided below.
4. Save the file.
5. Run the script from your terminal using `python lab1.py` or execute it directly from your IDE.
6. Observe the output for number operations and list manipulations.

Source Code

```
# --- Python Numbers ---

# Integer operations
num1 = 10
num2 = 3
print(f"Integer Addition: {num1} + {num2} = {num1 + num2}")
print(f"Integer Subtraction: {num1} - {num2} = {num1 - num2}")
print(f"Integer Multiplication: {num1} * {num2} = {num1 * num2}")
print(f"Integer Division (float result): {num1} / {num2} = {num1 / num2}")
print(f"Integer Floor Division: {num1} // {num2} = {num1 // num2}")
print(f"Integer Modulo: {num1} % {num2} = {num1 % num2}")
print(f"Integer Exponentiation: {num1} ** {num2} = {num1 ** num2}")

# Float operations
f_num1 = 10.5
f_num2 = 2.5
print(f"\nFloat Addition: {f_num1} + {f_num2} = {f_num1 + f_num2}")
print(f"Float Multiplication: {f_num1} * {f_num2} = {f_num1 * f_num2}")

# Type conversion
print(f"\nType of num1: {type(num1)}")
print(f"Converting num1 to float: {float(num1)}")
print(f"Type of f_num1: {type(f_num1)}")
print(f"Converting f_num1 to int: {int(f_num1)}")

# --- Python Lists ---
# Creating a list
my_list = [1, 2, 3, "apple", "banana", 3.14]
print(f"\nOriginal List: {my_list}")

# Accessing elements
print(f"First element: {my_list[0]}")
print(f"Last element: {my_list[-1]}")
print(f"Slice from index 2 to 4: {my_list[2:5]}") # End index is exclusive

# Modifying elements
my_list[1] = 20
print(f"List after modifying element at index 1: {my_list}")

# Adding elements
my_list.append("orange") # Adds to the end
print(f"List after append 'orange': {my_list}")
my_list.insert(2, "grape") # Inserts at a specific index
print(f"List after insert 'grape' at index 2: {my_list}")

# Removing elements
my_list.remove("apple") # Removes first occurrence of value
print(f"List after removing 'apple': {my_list}")
popped_element = my_list.pop() # Removes and returns last element
print(f"List after pop (last element): {my_list}, Popped: {popped_element}")
del my_list[0] # Deletes element at specific index
print(f"List after deleting element at index 0: {my_list}")

# List length
print(f"Length of the list: {len(my_list)}")

# Checking if an element exists
print(f"'banana' in list: {'banana' in my_list}")
print(f"'mango' in list: {'mango' in my_list}")

# Sorting a list (if elements are comparable)
num_list = [5, 2, 8, 1, 9]
num_list.sort()
print(f"Sorted numeric list: {num_list}")

# Reversing a list
num_list.reverse()
print(f"Reversed numeric list: {num_list}")

# Concatenating lists
list_a = [1, 2]
list_b = [3, 4]
combined_list = list_a + list_b
print(f"Combined lists: {combined_list}")

# Clearing a list
my_list.clear()
print(f"List after clear: {my_list}")
```

Input

No specific user input is required for this program as all values are hardcoded within the script.

Expected Output

```
Integer Addition: 10 + 3 = 13
Integer Subtraction: 10 - 3 = 7
Integer Multiplication: 10 * 3 = 30
Integer Division (float result): 10 / 3 = 3.3333333333333335
Integer Floor Division: 10 // 3 = 3
Integer Modulo: 10 % 3 = 1
Integer Exponentiation: 10 ** 3 = 1000

Float Addition: 10.5 + 2.5 = 13.0
Float Multiplication: 10.5 * 2.5 = 26.25

Type of num1: <class 'int'>
Converting num1 to float: 10.0
Type of f_num1: <class 'float'>
Converting f_num1 to int: 10

Original List: [1, 2, 3, 'apple', 'banana', 3.14]
First element: 1
Last element: 3.14
Slice from index 2 to 4: [3, 'apple', 'banana']
List after modifying element at index 1: [1, 20, 3, 'apple', 'banana', 3.14]
List after append 'orange': [1, 20, 3, 'apple', 'banana', 3.14, 'orange']
List after insert 'grape' at index 2: [1, 20, 'grape', 3, 'apple', 'banana',
3.14, 'orange']
List after removing 'apple': [1, 20, 'grape', 3, 'banana', 3.14, 'orange']
List after pop (last element): [1, 20, 'grape', 3, 'banana', 3.14], Popped:
orange
List after deleting element at index 0: [20, 'grape', 3, 'banana', 3.14]
Length of the list: 5
'banana' in list: True
'mango' in list: False
Sorted numeric list: [1, 2, 5, 8, 9]
Reversed numeric list: [9, 8, 5, 2, 1]
Combined lists: [1, 2, 3, 4]
List after clear: []
```

Lab 2: Tuple, Strings, Set

Aim

To understand and implement operations with Python tuples, strings, and sets, including their creation, access, modification (where applicable), and common methods.

Procedure

1. Open a Python interpreter or a Python IDE.
2. Create a new Python file (e.g., `lab2.py`).
3. Write the source code as provided below.
4. Save the file.
5. Run the script from your terminal using `python lab2.py` or execute it directly from your IDE.
6. Observe the output for tuple, string, and set manipulations.

Source Code

```
# --- Python Tuples ---

# Creating a tuple
my_tuple = (1, 2, 3, "red", "green", 3.14)
print(f"Original Tuple: {my_tuple}")

# Accessing elements (similar to lists)
print(f"First element: {my_tuple[0]}")
print(f"Last element: {my_tuple[-1]}")
print(f"Slice from index 1 to 3: {my_tuple[1:4]}")

# Tuples are immutable - uncommenting the line below will cause an error
# my_tuple[0] = 10

# Concatenating tuples
tuple_a = (10, 20)
tuple_b = ("A", "B")
combined_tuple = tuple_a + tuple_b
print(f"Combined Tuples: {combined_tuple}")

# Tuple length
print(f"Length of the tuple: {len(my_tuple)}")

# Checking if an element exists
print(f"'red' in tuple: {'red' in my_tuple}")

# --- Python Strings ---

# Creating a string
my_string = "Hello, Python World!"
print(f"\nOriginal String: {my_string}")

# Accessing characters (similar to lists/tuples)
print(f"First character: {my_string[0]}")
print(f"Last character: {my_string[-1]}")
print(f"Slice from index 7 to 13: {my_string[7:14]}")

# Strings are immutable - uncommenting the line below will cause an error
# my_string[0] = 'h'
# String length
print(f"Length of the string: {len(my_string)}")

# String methods
print(f"Uppercase: {my_string.upper()}")
print(f"Lowercase: {my_string.lower()}")
print(f"Starts with 'Hello': {my_string.startswith('Hello')}")
print(f"Ends with 'World!': {my_string.endswith('World!')}")
print(f"Replace 'Python' with 'Programming': {my_string.replace('Python',
'Programming')}")
words = my_string.split(" ")
print(f"Split by space: {words}")
joined_string = "-".join(words)
print(f"Joined with hyphen: {joined_string}")
print(f"Find 'Python': {my_string.find('Python')}") # Returns index of first
occurrence
print(f"Count 'o': {my_string.count('o')}")

# String formatting
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
print("My name is {} and I am {} years old.".format(name, age))

# --- Python Sets ---

# Creating a set
my_set = {1, 2, 3, 2, 1, "apple", "banana"} # Duplicate values are automatically
removed
print(f"\nOriginal Set: {my_set}") # Order is not guaranteed

# Adding elements
my_set.add("orange")
print(f"Set after adding 'orange': {my_set}")

# Removing elements
my_set.remove(3) # Removes a specific element
print(f"Set after removing 3: {my_set}")
my_set.discard("grape") # Removes element if present, no error if not
print(f"Set after discarding 'grape': {my_set}")
popped_item = my_set.pop() # Removes and returns an arbitrary element
print(f"Set after pop: {my_set}, Popped item: {popped_item}")

# Set operations
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print(f"Set A: {set_a}")
print(f"Set B: {set_b}")
print(f"Union (A | B): {set_a.union(set_b)}")
print(f"Intersection (A & B): {set_a.intersection(set_b)}")
print(f"Difference (A - B): {set_a.difference(set_b)}")
print(f"Symmetric Difference (A ^ B): {set_a.symmetric_difference(set_b)}")

# Checking subsets and supersets
print(f"Is {{1, 2}} a subset of A: {{1, 2}}.issubset(set_a) = {{1,
2}}.issubset(set_a)")
print(f"Is A a superset of {{1, 2}}: set_a.issuperset({{1, 2}}) =
set_a.issuperset({{1, 2}})")

# Clearing a set
my_set.clear()
print(f"Set after clear: {my_set}")
```

Input

No specific user input is required for this program as all values are hardcoded within the script.

Expected Output

```
Original Tuple: (1, 2, 3, 'red', 'green', 3.14)
First element: 1
Last element: 3.14
Slice from index 1 to 3: (2, 3, 'red')
Combined Tuples: (10, 20, 'A', 'B')
Length of the tuple: 6
'red' in tuple: True

Original String: Hello, Python World!
First character: H
Last character: !
Slice from index 7 to 13: Python
Length of the string: 20
Uppercase: HELLO, PYTHON WORLD!
Lowercase: hello, python world!
Starts with 'Hello': True
Ends with 'World!': True
Replace 'Python' with 'Programming': Hello, Programming World!
Split by space: ['Hello,', 'Python', 'World!']
Joined with hyphen: Hello,-Python-World!
Find 'Python': 7
Count 'o': 3
My name is Alice and I am 30 years old.
My name is Alice and I am 30 years old.

Original Set: {1, 2, 3, 'banana', 'apple'}
Set after adding 'orange': {1, 2, 3, 'banana', 'orange', 'apple'}
Set after removing 3: {1, 2, 'banana', 'orange', 'apple'}
Set after discarding 'grape': {1, 2, 'banana', 'orange', 'apple'}
Set after pop: {2, 'banana', 'orange', 'apple'}, Popped item: 1
Set A: {1, 2, 3, 4}
Set B: {3, 4, 5, 6}
Union (A | B): {1, 2, 3, 4, 5, 6}
Intersection (A & B): {3, 4}
Difference (A - B): {1, 2}
Symmetric Difference (A ^ B): {1, 2, 5, 6}
Is {1, 2} a subset of A: {1, 2}.issubset(set_a) = True
Is A a superset of {1, 2}: set_a.issuperset({1, 2}) = True
Set after clear: set()
```

Lab 3: Lambda & Filter in Python Examples

Aim

To understand and implement anonymous functions using `lambda` and to use the `filter()` function for filtering elements from an iterable based on a given condition.

Procedure

1. Open a Python interpreter or a Python IDE.
2. Create a new Python file (e.g., `lab3.py`).
3. Write the source code as provided below.
4. Save the file.
5. Run the script from your terminal using `python lab3.py` or execute it directly from your IDE.
6. Observe the output for `lambda` function usage and `filter()` examples.

Source Code

```
# --- Lambda Functions ---

# Basic lambda function for addition
add = lambda x, y: x + y
print(f"Lambda Addition (5 + 3): {add(5, 3)}")

# Lambda function to check if a number is even
is_even = lambda num: num % 2 == 0
print(f"Is 4 even? {is_even(4)}")
print(f"Is 7 even? {is_even(7)}")

# Lambda function with if-else (ternary operator)
max_of_two = lambda a, b: a if a > b else b
print(f"Max of 10 and 5: {max_of_two(10, 5)}")
print(f"Max of 3 and 8: {max_of_two(3, 8)}")

# Using lambda with map() (though not explicitly asked, it's a common use case)
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x * x, numbers))
print(f"Original numbers: {numbers}")
print(f"Squared numbers using map and lambda: {squared_numbers}")

# --- Filter Function ---

# Filtering even numbers from a list
numbers_to_filter = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda num: num % 2 == 0, numbers_to_filter))
print(f"\nOriginal list for filter: {numbers_to_filter}")
print(f"Even numbers using filter and lambda: {even_numbers}")

# Filtering strings that start with a specific letter
fruits = ["apple", "banana", "cherry", "date", "grape"]
fruits_starting_with_c = list(filter(lambda fruit: fruit.startswith('c'),
fruits))
print(f"Fruits: {fruits}")
print(f"Fruits starting with 'c': {fruits_starting_with_c}")

# Filtering numbers greater than a certain value
scores = [45, 88, 72, 95, 60, 55]
```

```
passing_scores = list(filter(lambda score: score >= 70, scores))
print(f"Scores: {scores}")
print(f"Passing scores (>=70): {passing_scores}")

# Filtering non-empty strings
words = ["hello", "", "world", None, "python", ""]
# Use `bool` as the function to filter out "falsy" values (empty strings, None,
0, False)
non_empty_words = list(filter(None, words))
print(f"Words list: {words}")
print(f"Non-empty words using filter(None, ...): {non_empty_words}")
```

Input

No specific user input is required for this program as all values are hardcoded within the script.

Expected Output

```
Lambda Addition (5 + 3): 8
Is 4 even? True
Is 7 even? False
Max of 10 and 5: 10
Max of 3 and 8: 8
Original numbers: [1, 2, 3, 4, 5]
Squared numbers using map and lambda: [1, 4, 9, 16, 25]

Original list for filter: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even numbers using filter and lambda: [2, 4, 6, 8, 10]
Fruits: ['apple', 'banana', 'cherry', 'date', 'grape']
Fruits starting with 'c': ['cherry']
Scores: [45, 88, 72, 95, 60, 55]
Passing scores (>=70): [88, 72, 95]
Words list: ['hello', '', 'world', None, 'python', '']
Non-empty words using filter(None, ...): ['hello', 'world', 'python']
```

Lab 4: Creating Class in Python

Aim

To understand the concept of Object-Oriented Programming (OOP) in Python by defining and creating a basic class with attributes.

Procedure

1. Open a Python interpreter or a Python IDE.
2. Create a new Python file (e.g., `lab4.py`).
3. Write the source code as provided below, defining a class.
4. Save the file.
5. Run the script from your terminal using `python lab4.py` or execute it directly from your IDE.
6. Observe how the class is defined and how its attributes are accessed.

Source Code

```
# --- Creating a Class in Python ---

# Define a simple class named 'Dog'
class Dog:
    # Class attribute (shared by all instances of the class)
    species = "Canis familiaris"

    # The __init__ method is the constructor for the class.
    # It is called automatically when a new object (instance) of the class is
created.
    # 'self' refers to the instance of the class itself.
    # 'name' and 'age' are parameters that will be passed when creating an
object.
    def __init__(self, name, age):
        # Instance attributes (unique to each object)
        self.name = name
        self.age = age

    # Another method (function defined inside a class)
    def bark(self):
        return f"{self.name} says Woof!"

# --- Demonstrating Class Creation ---

print("--- Demonstrating Class Creation ---")

# Create an instance (object) of the Dog class
# This calls the __init__ method
my_dog = Dog("Buddy", 3)

# Access instance attributes using dot notation
print(f"My dog's name is: {my_dog.name}")
print(f"My dog's age is: {my_dog.age}")

# Access class attribute
print(f"My dog's species is: {my_dog.species}")

# Create another instance
another_dog = Dog("Lucy", 5)
```

```
print(f"\nAnother dog's name is: {another_dog.name}")
print(f"Another dog's age is: {another_dog.age}")
print(f"Another dog's species is: {another_dog.species}")

# Call a method on the instance
print(my_dog.bark())
print(another_dog.bark())

# You can also set new attributes to an object outside the class definition
my_dog.color = "Golden"
print(f"\nMy dog's color: {my_dog.color}")

# Trying to access a non-existent attribute will cause an AttributeError
# print(another_dog.color) # Uncommenting this will raise an error
```

Input

No specific user input is required for this program as all values are hardcoded within the script.

Expected Output

```
--- Demonstrating Class Creation ---
My dog's name is: Buddy
My dog's age is: 3
My dog's species is: Canis familiaris

Another dog's name is: Lucy
Another dog's age is: 5
Another dog's species is: Canis familiaris
Buddy says Woof!
Lucy says Woof!

My dog's color: Golden
```

Lab 5: Creating Object in Python

Aim

To practically demonstrate the creation of objects (instances) from a defined class in Python and to understand how to access their attributes and call their methods.

Procedure

1. Open a Python interpreter or a Python IDE.
2. Create a new Python file (e.g., `lab5.py`).
3. Define a class first, as shown in Lab 4, and then create multiple objects from it.
4. Write the source code as provided below.
5. Save the file.
6. Run the script from your terminal using `python lab5.py` or execute it directly from your IDE.
7. Observe how different objects of the same class can have distinct attribute values.

Source Code

```
# --- Creating Object in Python ---

# First, define a class (reusing the Dog class from Lab 4 for consistency)
class Dog:
    species = "Canis familiaris" # Class attribute

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

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

    def get_info(self):
        return f"{self.name} is a {self.species} and is {self.age} years old."

# --- Object Creation and Interaction ---

print("--- Object Creation and Interaction ---")

# Creating the first object (instance) of the Dog class
# This process is called instantiation.
dog1 = Dog("Max", 4)
print(f"Created dog1: Name={dog1.name}, Age={dog1.age}, Species={dog1.species}")
print(dog1.bark())
print(dog1.get_info())

# Creating a second object of the Dog class
dog2 = Dog("Bella", 2)
print(f"\nCreated dog2: Name={dog2.name}, Age={dog2.age},
Species={dog2.species}")
print(dog2.bark())
print(dog2.get_info())

# Creating a third object
dog3 = Dog("Charlie", 7)
print(f"\nCreated dog3: Name={dog3.name}, Age={dog3.age},
Species={dog3.species}")
```

```
print(dog3.bark())
print(dog3.get_info())

# Demonstrating that each object has its own distinct instance attributes
print(f"\nAre dog1 and dog2 the same object? {dog1 is dog2}")
print(f"dog1's age: {dog1.age}, dog2's age: {dog2.age}")

# Modifying attributes of an object
dog1.age = 5
print(f"dog1's new age: {dog1.age}")

# Adding a new attribute to a specific object (not part of the class definition)
dog2.breed = "Labrador"
print(f"dog2's breed: {dog2.breed}")
# Note: dog1 does not have a 'breed' attribute unless explicitly added to it.
# print(dog1.breed) # This would cause an AttributeError
```

Input

No specific user input is required for this program as all values are hardcoded within the script.

Expected Output

```
--- Object Creation and Interaction ---
Created dog1: Name=Max, Age=4, Species=Canis familiaris
Max says Woof!
Max is a Canis familiaris and is 4 years old.

Created dog2: Name=Bella, Age=2, Species=Canis familiaris
Bella says Woof!
Bella is a Canis familiaris and is 2 years old.

Created dog3: Name=Charlie, Age=7, Species=Canis familiaris
Charlie says Woof!
Charlie is a Canis familiaris and is 7 years old.

Are dog1 and dog2 the same object? False
dog1's age: 4, dog2's age: 2
dog1's new age: 5
dog2's breed: Labrador
```

Lab 6: Creating Methods in Python

Aim

To understand and implement methods within Python classes, demonstrating how functions defined inside a class operate on the instance's data (`self`) and encapsulate behavior.

Procedure

1. Open a Python interpreter or a Python IDE.
2. Create a new Python file (e.g., `lab6.py`).
3. Define a class with several methods, including the constructor (`__init__`).
4. Write the source code as provided below.
5. Save the file.
6. Run the script from your terminal using `python lab6.py` or execute it directly from your IDE.
7. Observe how methods are defined and called on objects to perform actions or retrieve information.

Source Code

```
# --- Creating Methods in Python ---

# Define a class named 'Car' with various methods
class Car:
    # Class attribute
    wheels = 4

    # Constructor method
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.speed = 0 # Initial speed of the car

    # Instance method: displays car information
    def display_info(self):
        return f"Car: {self.year} {self.make} {self.model}, Color: {self.color}"

    # Instance method: accelerates the car
    def accelerate(self, increment):
        self.speed += increment
        return f"The {self.model} is now accelerating. Current speed:
{self.speed} km/h."

    # Instance method: brakes the car
    def brake(self, decrement):
        self.speed -= decrement
        if self.speed < 0:
            self.speed = 0
        return f"The {self.model} is braking. Current speed: {self.speed} km/h."

    # Instance method: honks the horn
    def honk(self):
        return "Beep! Beep!"

    # Instance method: changes the color of the car
    def change_color(self, new_color):
        old_color = self.color
        self.color = new_color
        return f"The {self.model}'s color changed from {old_color} to
{self.color}."

# --- Demonstrating Method Usage ---

print("--- Demonstrating Method Usage ---")

# Create a Car object
my_car = Car("Toyota", "Camry", 2020, "Blue")
print(my_car.display_info())
print(f"Number of wheels: {my_car.wheels}")

# Call methods on the object
print(my_car.accelerate(50))
print(my_car.accelerate(20))
print(my_car.brake(30))
print(my_car.honk())
print(my_car.change_color("Red"))
print(my_car.display_info()) # Display info again to see color change

print("\n--- Another Car Object ---")
another_car = Car("Honda", "Civic", 2022, "Silver")
print(another_car.display_info())
print(another_car.accelerate(60))
print(another_car.honk())
```

Input

No specific user input is required for this program as all values are hardcoded within the script.

Expected Output

```
--- Demonstrating Method Usage ---
Car: 2020 Toyota Camry, Color: Blue
Number of wheels: 4
The Camry is now accelerating. Current speed: 50 km/h.
The Camry is now accelerating. Current speed: 70 km/h.
The Camry is braking. Current speed: 40 km/h.
Beep! Beep!
The Camry's color changed from Blue to Red.
Car: 2020 Toyota Camry, Color: Red

--- Another Car Object ---
Car: 2022 Honda Civic, Color: Silver
The Civic is now accelerating. Current speed: 60 km/h.
Beep! Beep!
```

Lab 7: Process Standard Streams

Aim

To understand and implement how to interact with Python's standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) streams for basic console I/O operations.

Procedure

1. Open a Python interpreter or a Python IDE.
2. Create a new Python file (e.g., `lab7.py`).
3. Write the source code as provided below, utilizing `input()`, `print()`, and `sys.stderr.write()`.
4. Save the file.
5. Run the script from your terminal using `python lab7.py` or execute it directly from your IDE.
6. When prompted, provide input from the keyboard. Observe the output on the console and how error messages are directed to `stderr`.

Source Code

```
import sys

# --- Processing Standard Input (stdin) ---

print("--- Standard Input (stdin) ---")
try:
    # Read a line from standard input
    name = input("Enter your name: ")
    print(f"Hello, {name}!")

    # Read an integer from standard input
    age_str = input("Enter your age: ")
    age = int(age_str) # Convert string input to integer
    print(f"You are {age} years old.")

    # Read multiple values separated by spaces
    numbers_str = input("Enter three numbers separated by spaces (e.g., 10 20
30): ")
    num_list_str = numbers_str.split()
    # Convert string list to integer list using map
    numbers = [int(n) for n in num_list_str]
    print(f"You entered numbers: {numbers}")

except ValueError:
    # Handle cases where input conversion fails (e.g., non-integer age)
    sys.stderr.write("Error: Invalid input. Please enter a valid number.\n")
except Exception as e:
    sys.stderr.write(f"An unexpected error occurred: {e}\n")


# --- Processing Standard Output (stdout) ---

print("\n--- Standard Output (stdout) ---")
# Using print() function which writes to stdout by default
print("This message goes to standard output.")
print("Another line to stdout.")
```

```
# You can explicitly write to stdout using sys.stdout.write()
# Note: sys.stdout.write() does not add a newline character automatically,
# so you need to add '\n' manually if you want a new line.
sys.stdout.write("This is written directly to stdout using
sys.stdout.write().\n")
sys.stdout.write("It doesn't add a newline unless specified.\n")

# --- Processing Standard Error (stderr) ---

print("\n--- Standard Error (stderr) ---")
# Example of writing an error message to stderr
sys.stderr.write("This is an error message, typically for diagnostic output.\n")
sys.stderr.write("It might appear in red in some terminals or be redirected
separately.\n")

# Simulate a potential error scenario
def divide(a, b):
    if b == 0:
        # Write error message to stderr before raising an exception
        sys.stderr.write("Error: Division by zero is not allowed.\n")
        return None # Or raise ValueError("Division by zero")
    return a / b

result = divide(10, 0)
if result is None:
    print("Division failed due to an error.")
else:
    print(f"Result of division: {result}")

print("\nProgram finished.")
```

Input

When prompted, provide the following inputs:

```
Enter your name: John Doe
Enter your age: 25
Enter three numbers separated by spaces (e.g., 10 20 30): 100 200 300
```

Expected Output

```
--- Standard Input (stdin) ---
Enter your name: John Doe
Hello, John Doe!
Enter your age: 25
You are 25 years old.
Enter three numbers separated by spaces (e.g., 10 20 30): 100 200 300
You entered numbers: [100, 200, 300]

--- Standard Output (stdout) ---
This message goes to standard output.
Another line to stdout.
This is written directly to stdout using sys.stdout.write().
It doesn't add a newline unless specified.

--- Standard Error (stderr) ---
This is an error message, typically for diagnostic output.
It might appear in red in some terminals or be redirected separately.
Error: Division by zero is not allowed.
Division failed due to an error.

Program finished.
```

(Note: The lines written to `stderr` might appear in a different color or be separated from `stdout` depending on the terminal or IDE configuration.)

Lab 8: Command-line arguments, shell variables

Aim

To understand how to pass and process command-line arguments in Python scripts and to briefly touch upon the concept of shell variables influencing script execution (though direct manipulation of shell variables from Python is limited).

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab8.py`).
3. Write the source code as provided below, using the `sys` module to access command-line arguments.
4. Save the file.
5. Open your system's terminal or command prompt.
6. Navigate to the directory where you saved `lab8.py`.
7. Run the script using `python lab8.py <arg1> <arg2> ...` where `<arg1>`, `<arg2>` are your desired arguments.
8. Observe how the script accesses and processes the arguments you provide.
9. (Optional) Experiment with setting a shell variable before running the script and accessing it via `os.environ`.

Source Code

```
import sys
import os

# --- Command-line Arguments ---

print("--- Command-line Arguments ---")

# sys.argv is a list of command-line arguments.
# sys.argv[0] is always the script name itself.
# Subsequent elements are the arguments passed.

print(f"Number of arguments: {len(sys.argv)}")
print(f"List of arguments: {sys.argv}")

# Check if any arguments (beyond the script name) were provided
if len(sys.argv) > 1:
    print(f"\nScript name: {sys.argv[0]}")
    print(f"First argument (sys.argv[1]): {sys.argv[1]}")

    # Process all arguments (excluding the script name)
    print("\nAll arguments (excluding script name):")
    for i, arg in enumerate(sys.argv[1:]):
        print(f"Argument {i+1}: {arg}")

    # Example: Simple calculator using command-line arguments
    if len(sys.argv) == 4:
        try:
            operand1 = float(sys.argv[1])
            operator = sys.argv[2]
            operand2 = float(sys.argv[3])

            print(f"\n--- Simple Calculator ---")
            print(f"Operation: {operand1} {operator} {operand2}")

            if operator == '+':
                print(f"Result: {operand1 + operand2}")
            elif operator == '-':
                print(f"Result: {operand1 - operand2}")
            elif operator == 'x' or operator == '*':
                print(f"Result: {operand1 * operand2}")
            elif operator == '/':
                if operand2 != 0:
                    print(f"Result: {operand1 / operand2}")
                else:
                    sys.stderr.write("Error: Division by zero.\n")
            else:
                sys.stderr.write("Error: Invalid operator. Use +, -, x, or
/.\n")
        except ValueError:
            sys.stderr.write("Error: Invalid numbers provided for
calculation.\n")
        except Exception as e:
            sys.stderr.write(f"An error occurred during calculation: {e}\n")
    else:
        print("\nTo use the calculator, run: python lab8.py <number1> <operator>
<number2>")
        print("Example: python lab8.py 10 + 5")

else:
    print("\nNo additional command-line arguments provided.")
    print("Try running: python lab8.py hello world 123")


# --- Shell Variables (Environment Variables) ---

print("\n--- Shell Variables (Environment Variables) ---")

# Accessing environment variables using os.environ
# These are variables set in the shell/system environment where the script is
run.

# Example: Accessing common environment variables
print(f"PATH environment variable: {os.environ.get('PATH', 'Not set')}")
print(f"HOME environment variable: {os.environ.get('HOME', 'Not set')}") # On
Windows, it might be USERPROFILE

# Accessing a custom environment variable (if set before running the script)
# To test this, in your terminal BEFORE running the script, type:
# For Linux/macOS: export MY_CUSTOM_VAR="Hello from Shell!"
# For Windows (cmd): set MY_CUSTOM_VAR="Hello from Shell!"
# For Windows (PowerShell): $env:MY_CUSTOM_VAR="Hello from Shell!"
custom_var = os.environ.get('MY_CUSTOM_VAR', 'MY_CUSTOM_VAR not set in
environment.')
print(f"MY_CUSTOM_VAR: {custom_var}")

# You can also set environment variables within Python,
# but they only affect the current Python process and its child processes,
# not the parent shell process.
os.environ['PYTHON_TEST_VAR'] = 'This is a test variable from Python.'
print(f"PYTHON_TEST_VAR (set in script): {os.environ.get('PYTHON_TEST_VAR')}")
```

Input

Run the script from your terminal with different arguments:

Scenario 1: No arguments

```
python lab8.py
```

Scenario 2: Multiple arguments

```
python lab8.py apple banana 123 test
```

Scenario 3: Calculator example (addition)

```
python lab8.py 10 + 5
```

Scenario 4: Calculator example (multiplication)

```
python lab8.py 2.5 x 4
```

Scenario 5: Calculator example (division by zero)

```
python lab8.py 10 / 0
```

Scenario 6: Accessing a custom shell variable (First, set the variable in your terminal, then run the script)

- **Linux/macOS:**
- `export MY_CUSTOM_VAR="This came from the shell!"`
- `python lab8.py`

- **Windows (Command Prompt):**
- `set MY_CUSTOM_VAR="This came from the shell!"`
- `python lab8.py`

- **Windows (PowerShell):**
- `$env:MY_CUSTOM_VAR="This came from the shell!"`
- `python lab8.py`

Expected Output

Scenario 1: No arguments

```
--- Command-line Arguments ---
Number of arguments: 1
List of arguments: ['lab8.py']

No additional command-line arguments provided.
Try running: python lab8.py hello world 123

--- Shell Variables (Environment Variables) ---
PATH environment variable: /usr/local/bin:/usr/bin:... (your system's PATH)
HOME environment variable: /home/youruser (your system's HOME)
MY_CUSTOM_VAR: MY_CUSTOM_VAR not set in environment.
PYTHON_TEST_VAR (set in script): This is a test variable from Python.
```

Scenario 2: Multiple arguments

```
--- Command-line Arguments ---
Number of arguments: 5
List of arguments: ['lab8.py', 'apple', 'banana', '123', 'test']

Script name: lab8.py
First argument (sys.argv[1]): apple

All arguments (excluding script name):
Argument 1: apple
Argument 2: banana
Argument 3: 123
Argument 4: test

To use the calculator, run: python lab8.py <number1> <operator> <number2>
Example: python lab8.py 10 + 5

--- Shell Variables (Environment Variables) ---
PATH environment variable: ...
HOME environment variable: ...
MY_CUSTOM_VAR: MY_CUSTOM_VAR not set in environment.
PYTHON_TEST_VAR (set in script): This is a test variable from Python.
```

Scenario 3: Calculator example (addition)

```
--- Command-line Arguments ---
Number of arguments: 4
List of arguments: ['lab8.py', '10', '+', '5']

Script name: lab8.py
First argument (sys.argv[1]): 10

All arguments (excluding script name):
Argument 1: 10
Argument 2: +
Argument 3: 5

--- Simple Calculator ---
Operation: 10.0 + 5.0
Result: 15.0

--- Shell Variables (Environment Variables) ---
PATH environment variable: ...
HOME environment variable: ...
MY_CUSTOM_VAR: MY_CUSTOM_VAR not set in environment.
PYTHON_TEST_VAR (set in script): This is a test variable from Python.
```

Scenario 4: Calculator example (multiplication)

```
--- Command-line Arguments ---
Number of arguments: 4
List of arguments: ['lab8.py', '2.5', 'x', '4']

Script name: lab8.py
First argument (sys.argv[1]): 2.5

All arguments (excluding script name):
Argument 1: 2.5
Argument 2: x
Argument 3: 4

--- Simple Calculator ---
```

```
Operation: 2.5 x 4.0
Result: 10.0

--- Shell Variables (Environment Variables) ---
PATH environment variable: ...
HOME environment variable: ...
MY_CUSTOM_VAR: MY_CUSTOM_VAR not set in environment.
PYTHON_TEST_VAR (set in script): This is a test variable from Python.
```

Scenario 5: Calculator example (division by zero)

```
--- Command-line Arguments ---
Number of arguments: 4
List of arguments: ['lab8.py', '10', '/', '0']

Script name: lab8.py
First argument (sys.argv[1]): 10

All arguments (excluding script name):
Argument 1: 10
Argument 2: /
Argument 3: 0

--- Simple Calculator ---
Operation: 10.0 / 0.0
Error: Division by zero.

--- Shell Variables (Environment Variables) ---
PATH environment variable: ...
HOME environment variable: ...
MY_CUSTOM_VAR: MY_CUSTOM_VAR not set in environment.
PYTHON_TEST_VAR (set in script): This is a test variable from Python.
```

(Note: The "Error: Division by zero." line will be printed to `stderr`.)

Scenario 6: Accessing a custom shell variable (Assuming `MY_CUSTOM_VAR` was set in the shell)

```
--- Command-line Arguments ---
Number of arguments: 1
List of arguments: ['lab8.py']

No additional command-line arguments provided.
Try running: python lab8.py hello world 123

--- Shell Variables (Environment Variables) ---
PATH environment variable: ...
HOME environment variable: ...
MY_CUSTOM_VAR: This came from the shell!
PYTHON_TEST_VAR (set in script): This is a test variable from Python.
```

Lab 9: Python scripts here perform real tasks

Aim

To demonstrate how Python scripts can perform practical "real tasks" such as file operations (creating, reading, writing), basic data processing, and interacting with the operating system.

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab9.py`).
3. Write the source code as provided below, which includes file I/O and basic text processing.
4. Save the file.
5. Run the script from your terminal using `python lab9.py`.
6. Observe the creation of a new file, its content, and the processed output on the console. You can also manually check the created file (`my_notes.txt`) in the same directory.

Source Code

```
import os

# --- Task 1: File Operations (Create, Write, Read) ---

print("--- Task 1: File Operations ---")

file_name = "my_notes.txt"
content_to_write = """
This is a sample note for Lab 9.
It demonstrates writing multiple lines to a file.
Python makes file handling easy!
Line 4: Data processing example.
Line 5: Another line.
"""

# 1. Writing to a file
try:
    # 'w' mode: write (creates new file or overwrites existing)
    with open(file_name, 'w') as file:
        file.write(content_to_write.strip()) # .strip() removes leading/trailing
whitespace
    print(f"Successfully wrote content to '{file_name}'")
except IOError as e:
    print(f"Error writing to file: {e}")

# 2. Reading from a file
print(f"\n--- Reading content from '{file_name}' ---")
try:
    # 'r' mode: read
    with open(file_name, 'r') as file:
        read_content = file.read()
        print(read_content)
except FileNotFoundError:
    print(f"Error: File '{file_name}' not found.")
except IOError as e:
    print(f"Error reading from file: {e}")

# 3. Appending to a file
print(f"\n--- Appending content to '{file_name}' ---")
```

```
append_content = "\nThis line was appended later."
try:
    # 'a' mode: append (adds to the end of the file)
    with open(file_name, 'a') as file:
        file.write(append_content)
    print(f"Successfully appended content to '{file_name}'")
except IOError as e:
    print(f"Error appending to file: {e}")

# Verify appended content by reading again
print(f"\n--- Reading content after appending ---")
try:
    with open(file_name, 'r') as file:
        print(file.read())
except Exception as e:
    print(f"Error reading appended content: {e}")


# --- Task 2: Basic Data Processing (Counting words/lines) ---

print("\n--- Task 2: Basic Data Processing ---")
try:
    with open(file_name, 'r') as file:
        lines = file.readlines() # Reads all lines into a list
        num_lines = len(lines)
        print(f"Number of lines in '{file_name}': {num_lines}")

        word_count = 0
        for line in lines:
            words = line.strip().split() # Split line into words
            word_count += len(words)
        print(f"Total word count in '{file_name}': {word_count}")

except FileNotFoundError:
    print(f"Error: File '{file_name}' not found for processing.")
except Exception as e:
    print(f"An error occurred during data processing: {e}")


# --- Task 3: Basic OS Interaction (Listing directory contents) ---

print("\n--- Task 3: Basic OS Interaction ---")
current_directory = os.getcwd() # Get current working directory
print(f"Current working directory: {current_directory}")

print("\nFiles and directories in the current directory:")
try:
    for item in os.listdir(current_directory):
        print(f"- {item}")
except OSError as e:
    print(f"Error listing directory contents: {e}")

# --- Clean up: Remove the created file (optional, but good practice) ---
# Uncomment the following lines if you want the script to delete the file after
execution
# print(f"\n--- Cleaning up: Deleting '{file_name}' ---")
# try:
#     os.remove(file_name)
#     print(f"Successfully deleted '{file_name}'.")
# except OSError as e:
#     print(f"Error deleting file: {e}")
```

Input

No specific user input is required for this program. The script performs file operations and data processing automatically.

Expected Output

```
--- Task 1: File Operations ---
Successfully wrote content to 'my_notes.txt'

--- Reading content from 'my_notes.txt' ---

This is a sample note for Lab 9.
It demonstrates writing multiple lines to a file.
Python makes file handling easy!
Line 4: Data processing example.
Line 5: Another line.

--- Appending content to 'my_notes.txt' ---
Successfully appended content to 'my_notes.txt'

--- Reading content after appending ---

This is a sample note for Lab 9.
It demonstrates writing multiple lines to a file.
Python makes file handling easy!
Line 4: Data processing example.
Line 5: Another line.
This line was appended later.

--- Task 2: Basic Data Processing ---
Number of lines in 'my_notes.txt': 7
Total word count in 'my_notes.txt': 28

--- Task 3: Basic OS Interaction ---
Current working directory: /path/to/your/script (will vary based on your system)

Files and directories in the current directory:
- lab9.py
- my_notes.txt
- ... (other files/folders in your directory)
```

(Note: The exact output for "Files and directories" will depend on the contents of your current directory.)

Lab 10: Client Socket Methods

Aim

To understand and implement the basic methods used by a client socket to establish a connection with a server, send data, and receive data over a network.

Procedure

First, create a simple server script. You will need a server running to test the client. Create a file named `simple_server.py` with the following content:

```
# simple_server.py
import socket
import threading

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)

def handle_client(conn, addr):
    print(f"Connected by {addr}")
    try:
        while True:
            data = conn.recv(1024) # Receive up to 1024 bytes
            if not data:
                break # Client disconnected
            message = data.decode('utf-8')
            print(f"Received from {addr}: {message}")
            response = f"Server received: {message.upper()}"
            conn.sendall(response.encode('utf-8'))
    except ConnectionResetError:
        print(f"Client {addr} disconnected unexpectedly.")
    finally:
        conn.close()
        print(f"Connection with {addr} closed.")

def start_server():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        print(f"Server listening on {HOST}:{PORT}")
        while True:
            conn, addr = s.accept() # Accept incoming connection
            thread = threading.Thread(target=handle_client, args=(conn,
addr))
            thread.start()

if __name__ == "__main__":
    start_server()
```

1. **Run the server script:** Open a separate terminal/command prompt and run `python simple_server.py`. Keep this terminal open.
2. **Create the client script (`lab10.py`):** Open another terminal/command prompt or IDE.
3. Write the source code for `lab10.py` as provided below.
4. Save the file.
5. Run the client script from your terminal using `python lab10.py`.

6. Observe the client connecting to the server, sending messages, and receiving responses. Check both the client and server terminal outputs.

Source Code

```
# lab10.py - Client Socket Methods

import socket
import time

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

print("--- Client Socket Methods Demonstration ---")

try:
    # 1. Create a socket object
    # socket.AF_INET: Address Family - IPv4
    # socket.SOCK_STREAM: Socket Type - TCP (reliable, connection-oriented)
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
        print(f"Client socket created: {client_socket}")

        # 2. Connect to the server
        print(f"Attempting to connect to {HOST}:{PORT}...")
        client_socket.connect((HOST, PORT))
        print(f"Successfully connected to server at {HOST}:{PORT}")

        # 3. Send data to the server
        message1 = "Hello from client!"
        client_socket.sendall(message1.encode('utf-8')) # Encode string to bytes
        print(f"Sent: '{message1}'")

        # 4. Receive data from the server
        data1 = client_socket.recv(1024) # Receive up to 1024 bytes
        response1 = data1.decode('utf-8') # Decode bytes to string
        print(f"Received: '{response1}'")

        time.sleep(1) # Wait a bit before sending next message

        message2 = "How are you, server?"
        client_socket.sendall(message2.encode('utf-8'))
        print(f"\nSent: '{message2}'")
        data2 = client_socket.recv(1024)
        response2 = data2.decode('utf-8')
        print(f"Received: '{response2}'")

        time.sleep(1)

        message3 = "Goodbye!"
        client_socket.sendall(message3.encode('utf-8'))
        print(f"\nSent: '{message3}'")
        data3 = client_socket.recv(1024)
        response3 = data3.decode('utf-8')
        print(f"Received: '{response3}'")

    print("\nClient connection closed automatically by 'with' statement.")

except ConnectionRefusedError:
    print(f"Error: Connection refused. Make sure the server is running on
{HOST}:{PORT}.")
except socket.timeout:
    print("Error: Connection timed out.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

print("Client script finished.")
```

Input

No direct user input for the client script. The client sends predefined messages.

Expected Output

`simple_server.py` (Server Terminal Output):

```
Server listening on 127.0.0.1:65432
Connected by ('127.0.0.1', <client_port_number>)
Received from ('127.0.0.1', <client_port_number>): Hello from client!
Received from ('127.0.0.1', <client_port_number>): How are you, server?
Received from ('127.0.0.1', <client_port_number>): Goodbye!
Connection with ('127.0.0.1', <client_port_number>) closed.
```

(Note: `<client_port_number>` will be a random port number assigned by the OS.)

`lab10.py` (Client Terminal Output):

```
--- Client Socket Methods Demonstration ---
Client socket created: <socket.socket fd=3, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('0.0.0.0', 0)>
Attempting to connect to 127.0.0.1:65432...
Successfully connected to server at 127.0.0.1:65432
Sent: 'Hello from client!'
Received: 'Server received: HELLO FROM CLIENT!'

Sent: 'How are you, server?'
Received: 'Server received: HOW ARE YOU, SERVER?'

Sent: 'Goodbye!'
Received: 'Server received: GOODBYE!'

Client connection closed automatically by 'with' statement.
Client script finished.
```

Lab 11: General Socket Methods

Aim

To understand and implement general methods available for socket objects in Python, including setting socket options, getting socket information, and managing socket states beyond basic client/server communication.

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab11.py`).
3. Write the source code as provided below, demonstrating various socket methods.
4. Save the file.
5. Run the script from your terminal using `python lab11.py`.
6. Observe the output, which will show different socket properties and the results of various method calls.

Source Code

```
# lab11.py - General Socket Methods

import socket
import sys
import time

print("--- General Socket Methods Demonstration ---")

try:
    # 1. Create a socket object
    # Default: AF_INET (IPv4), SOCK_STREAM (TCP)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print(f"\n1. Socket created: {s}")

    # 2. Get socket name (local address and port)
    # This will be ('0.0.0.0', 0) or similar before binding/connecting
    print(f"2. Socket's local address (before bind/connect): {s.getsockname()}")

    # 3. Set socket options (e.g., SO_REUSEADDR)
    # This allows the socket to be bound to an address that is already in use
    # (e.g., after a server crashes and restarts quickly).
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    print("3. Set SO_REUSEADDR option to 1.")

    # 4. Bind the socket to a specific address and port (typical for servers)
    HOST = '127.0.0.1'
    PORT = 50000 # A high port number to avoid conflicts
    s.bind((HOST, PORT))
    print(f"4. Socket bound to {HOST}:{PORT}")
    print(f"   Socket's local address (after bind): {s.getsockname()}")

    # 5. Get socket name (peer address - remote address and port)
    # This will raise an error if not connected
    # print(f"Socket's peer address (before connect): {s.getpeername()}") #
Uncomment to see error

    # 6. Listen for incoming connections (typical for servers)
    s.listen(5) # Allow up to 5 pending connections
    print("6. Socket is now listening for connections (max 5 backlog).")

    # 7. Set a timeout for blocking socket operations
    s.settimeout(1.0) # 1 second timeout
    print("7. Socket timeout set to 1.0 seconds.")

    # 8. Get socket options (e.g., SO_TYPE, SO_RCVBUF)
    print(f"8. Socket type (SO_TYPE): {s.getsockopt(socket.SOL_SOCKET,
socket.SO_TYPE)}")
    # socket.SOCK_STREAM (1) for TCP, socket.SOCK_DGRAM (2) for UDP
    print(f"   Receive buffer size (SO_RCVBUF): {s.getsockopt(socket.SOL_SOCKET,
socket.SO_RCVBUF)} bytes")

    # 9. Get the default buffer sizes (SO_SNDBUF, SO_RCVBUF)
    print(f"9. Default send buffer size: {s.getsockopt(socket.SOL_SOCKET,
socket.SO_SNDBUF)}")
    print(f"   Default receive buffer size: {s.getsockopt(socket.SOL_SOCKET,
socket.SO_RCVBUF)}")

    # 10. Shutdown the socket (can shut down read, write, or both)
    # This is typically done before closing, to indicate no more data will be
sent/received.
    # We'll simulate a client connection to demonstrate this properly.
    print("\n10. Demonstrating shutdown with a temporary client...")
    client_temp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client_temp_socket.connect((HOST, PORT))
        print(f"    Temporary client connected to {HOST}:{PORT}")

        # Accept the connection on the server socket
        conn, addr = s.accept()
        print(f"    Server accepted connection from {addr}")

        # Client sends data
        client_temp_socket.sendall(b"Hello from temp client!")
        print("    Temp client sent data.")
        time.sleep(0.1) # Give time for data to arrive

        # Server receives data
        received_data = conn.recv(1024)
        print(f"    Server received: {received_data.decode()}")

        # Client shuts down sending
        client_temp_socket.shutdown(socket.SHUT_WR)
        print("    Temp client shut down writing (SHUT_WR).")

        # Server tries to send data back (will succeed initially)
        conn.sendall(b"Server response after client SHUT_WR.")
        print("    Server sent response after client SHUT_WR.")

        # Client tries to receive (will still work)
        data_after_shutdown = client_temp_socket.recv(1024)
        print(f"    Temp client received after SHUT_WR:
{data_after_shutdown.decode()}")

        # Server shuts down sending
        conn.shutdown(socket.SHUT_WR)
        print("    Server shut down writing (SHUT_WR).")

        # Client tries to send (will fail/get error)
        try:
            client_temp_socket.sendall(b"Client trying to send after server
SHUT_WR.")
        except BrokenPipeError:
            print("    Client got BrokenPipeError when sending after server
SHUT_WR.")
```

```
        # Server tries to receive (will still work)
        try:
            conn.recv(1024) # Will receive empty bytes if client also shut down
read or closed
        except Exception as e:
            print(f"    Server error receiving after client SHUT_WR: {e}")


    except socket.timeout:
        print("    Temporary client connection timed out (expected if no server
response).")
    except Exception as e:
        print(f"    Error during temporary client/server interaction: {e}")
    finally:
        if 'conn' in locals() and conn:
            conn.close()
            print("    Server's accepted connection closed.")
        if client_temp_socket:
            client_temp_socket.close()
            print("    Temporary client socket closed.")


    # 11. Close the socket
    # This releases the socket's resources.
    s.close()
    print("\n11. Main server socket closed.")

except socket.timeout:
    print("Error: Socket operation timed out. Ensure no other process is using
the port.")
except OSError as e:
    print(f"Operating System Error: {e}")
    print("This might mean the port is already in use or permissions are an
issue.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

print("General socket methods demonstration finished.")
```

Input

No direct user input is required. The script demonstrates various socket methods internally.

Expected Output

```
--- General Socket Methods Demonstration ---

   Socket created: <socket.socket fd=3, family=AddressFamily.AF_INET,
   type=SocketKind.SOCK_STREAM, proto=0, laddr=('0.0.0.0', 0)>
   Socket's local address (before bind/connect): ('0.0.0.0', 0)
   Set SO_REUSEADDR option to 1.
   Socket bound to 127.0.0.1:50000
   Socket's local address (after bind): ('127.0.0.1', 50000)
   Socket is now listening for connections (max 5 backlog).
   Socket timeout set to 1.0 seconds.
   Socket type (SO_TYPE): 1
   Receive buffer size (SO_RCVBUF): 262144 bytes (This value can vary)
   Default send buffer size: 262144 (This value can vary)
   Default receive buffer size: 262144 (This value can vary)

   Demonstrating shutdown with a temporary client...
   Temporary client connected to 127.0.0.1:50000
   Server accepted connection from ('127.0.0.1', <client_port_number>)
   Temp client sent data.
```

```
Server received: Hello from temp client!
Temp client shut down writing (SHUT_WR).
Server sent response after client SHUT_WR.
Temp client received after SHUT_WR: Server response after client SHUT_WR.
Server shut down writing (SHUT_WR).
Client got BrokenPipeError when sending after server SHUT_WR.
Server error receiving after client SHUT_WR: [Errno 107] Transport endpoint
is not connected (or similar, depending on OS)
Server's accepted connection closed.
Temporary client socket closed.

Main server socket closed.
General socket methods demonstration finished.
```

(Note: `fd` and port numbers will vary. The exact error message for `recv` after `SHUT_WR` might differ slightly across operating systems, but it indicates the connection is no longer fully open for receiving.)

Lab 12: Creating Thread Using Threading Module

Aim

To understand the concept of multithreading in Python and to implement concurrent execution of tasks using the `threading` module.

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab12.py`).
3. Write the source code as provided below, defining functions to be executed by threads and using `threading.Thread` to create and start them.
4. Save the file.
5. Run the script from your terminal using `python lab12.py`.
6. Observe how the different tasks (functions) run seemingly simultaneously, and how their execution order might interleave, demonstrating concurrency.

Source Code

```
# lab12.py - Creating Thread Using Threading Module

import threading
import time
import random

# Function to be executed by the first thread
def task1(name, duration):
    print(f"Thread {name}: Starting task for {duration} seconds...")
    for i in range(duration):
        time.sleep(1) # Simulate work
        print(f"Thread {name}: Working... {i+1}s passed")
    print(f"Thread {name}: Task finished.")

# Function to be executed by the second thread
def task2(name, count):
    print(f"Thread {name}: Starting to count to {count}...")
    for i in range(1, count + 1):
        print(f"Thread {name}: Count {i}")
        time.sleep(0.5) # Shorter sleep to show interleaving
    print(f"Thread {name}: Counting finished.")

# Function to demonstrate shared resource (with potential issues without locks)
shared_counter = 0
def increment_counter(thread_id, num_increments):
    global shared_counter
    print(f"Thread {thread_id}: Starting to increment counter.")
    for _ in range(num_increments):
        current_value = shared_counter
        time.sleep(0.01) # Simulate some delay
        shared_counter = current_value + 1
    print(f"Thread {thread_id}: Finished incrementing. Final local value:
{shared_counter}")


print("--- Creating Thread Using Threading Module ---")

# --- Example 1: Basic Thread Creation and Execution ---
```

```
print("\n--- Example 1: Basic Threading ---")

# Create Thread objects
# target: the function to be executed by the thread
# args: a tuple of arguments to pass to the target function
thread1 = threading.Thread(target=task1, args=("Thread-A", 3))
thread2 = threading.Thread(target=task2, args=("Thread-B", 5))

# Start the threads
# This calls the target function in a new thread of execution
thread1.start()
thread2.start()

# Wait for both threads to complete
# .join() blocks the calling thread (main thread) until the thread it's called
on terminates.
print("\nMain: Waiting for Thread-A to finish...")
thread1.join()
print("Main: Thread-A finished.")

print("\nMain: Waiting for Thread-B to finish...")
thread2.join()
print("Main: Thread-B finished.")

print("\nAll basic threads have completed.")

# --- Example 2: Demonstrating Shared Resource (without explicit locking) ---
# This example is intended to show potential race conditions.
# The final shared_counter value might not be 2000 due to race conditions.
print("\n--- Example 2: Shared Resource (Potential Race Condition) ---")
shared_counter = 0 # Reset counter for this example
num_increments_per_thread = 1000

thread_inc1 = threading.Thread(target=increment_counter, args=(1,
num_increments_per_thread))
thread_inc2 = threading.Thread(target=increment_counter, args=(2,
num_increments_per_thread))

thread_inc1.start()
thread_inc2.start()

thread_inc1.join()
thread_inc2.join()

print(f"Main: Final shared_counter value (might be less than 2000 due to race
condition): {shared_counter}")
print("Note: For proper shared resource management, use threading.Lock or other
synchronization primitives.")

print("\nProgram finished.")
```

Input

No specific user input is required. The script demonstrates thread creation and execution.

Expected Output

```
--- Creating Thread Using Threading Module ---

--- Example 1: Basic Threading ---
Thread Thread-A: Starting task for 3 seconds...
Thread Thread-B: Starting to count to 5...
Thread Thread-B: Count 1
Thread Thread-A: Working... 1s passed
Thread Thread-B: Count 2
Thread Thread-B: Count 3
Thread Thread-A: Working... 2s passed
Thread Thread-B: Count 4
Thread Thread-B: Count 5
Thread Thread-B: Counting finished.
Thread Thread-A: Working... 3s passed
Thread Thread-A: Task finished.

Main: Waiting for Thread-A to finish...
Main: Thread-A finished.

Main: Waiting for Thread-B to finish...
Main: Thread-B finished.

All basic threads have completed.

--- Example 2: Shared Resource (Potential Race Condition) ---
Thread 1: Starting to increment counter.
Thread 2: Starting to increment counter.
Thread 1: Finished incrementing. Final local value: 1999 (This value can vary)
Thread 2: Finished incrementing. Final local value: 2000 (This value can vary)
Main: Final shared_counter value (might be less than 2000 due to race
condition): 1999 (This value can vary, typically less than 2000)
Note: For proper shared resource management, use threading.Lock or other
synchronization primitives.

Program finished.
```

(Note: The exact interleaving of "Working..." and "Count" messages will vary each time you run the script due to the nature of thread scheduling. The `Final shared_counter value` in Example 2 will almost certainly be less than 2000, demonstrating a race condition.)

Lab 13: Represent compound data using Python

Aim

To understand and implement various ways to represent compound data structures in Python, focusing on combinations of built-in types like lists of lists, lists of dictionaries, and dictionaries containing lists or other dictionaries.

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab13.py`).
3. Write the source code as provided below, demonstrating different compound data structures.
4. Save the file.
5. Run the script from your terminal using `python lab13.py`.
6. Observe how complex data can be structured and accessed using nested Python data types.

Source Code

```
# lab13.py - Represent compound data using Python

print("--- Representing Compound Data in Python ---")

# --- 1. List of Lists (Matrix/2D Array) ---
print("\n--- 1. List of Lists (Matrix) ---")
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(f"Matrix: {matrix}")
print(f"Element at [0][0]: {matrix[0][0]}") # First row, first column
print(f"Element at [1][2]: {matrix[1][2]}") # Second row, third column

# Iterating through a matrix
print("Iterating through matrix:")
for row in matrix:
    for element in row:
        print(element, end=" ")
    print() # New line after each row

# --- 2. List of Dictionaries (Table-like Data) ---
print("\n--- 2. List of Dictionaries (Records) ---")
students = [
    {"id": 101, "name": "Alice", "age": 20, "major": "Computer Science"},
    {"id": 102, "name": "Bob", "age": 22, "major": "Electrical Engineering"},
    {"id": 103, "name": "Charlie", "age": 21, "major": "Computer Science"}
]
print(f"Students data: {students}")

# Accessing data
print(f"Second student's name: {students[1]['name']}")
print(f"First student's major: {students[0]['major']}")

# Iterating and filtering
print("Students in Computer Science:")
for student in students:
    if student["major"] == "Computer Science":
        print(f"- ID: {student['id']}, Name: {student['name']}")

# --- 3. Dictionary of Lists ---
print("\n--- 3. Dictionary of Lists ---")
# Example: Grouping items by category
inventory = {
    "fruits": ["apple", "banana", "cherry"],
    "vegetables": ["carrot", "spinach", "potato"],
    "dairy": ["milk", "cheese"]
}
print(f"Inventory: {inventory}")

print(f"Fruits available: {inventory['fruits']}")
inventory['fruits'].append("grape")
print(f"Fruits after adding grape: {inventory['fruits']}")

# --- 4. Dictionary of Dictionaries (Nested Objects) ---
print("\n--- 4. Dictionary of Dictionaries (Nested Objects) ---")
# Example: User profiles
user_profiles = {
    "user123": {
        "name": "John Doe",
        "email": "john.doe@example.com",
        "address": {
            "street": "123 Main St",
            "city": "Anytown",
            "zip": "12345"
        }
    },
    "jane_smith": {
        "name": "Jane Smith",
        "email": "jane.smith@example.com",
        "address": {
            "street": "456 Oak Ave",
            "city": "Otherville",
            "zip": "67890"
        }
    }
}
print(f"User Profiles: {user_profiles}")

print(f"John Doe's email: {user_profiles['user123']['email']}")
print(f"Jane Smith's city: {user_profiles['jane_smith']['address']['city']}")

# Modifying nested data
user_profiles['user123']['address']['zip'] = "54321"
print(f"John Doe's updated zip: {user_profiles['user123']['address']['zip']}")

# --- 5. Mixed Compound Data (e.g., Tuple of Dictionaries) ---
print("\n--- 5. Mixed Compound Data (Tuple of Dictionaries) ---")
# Example: Immutable set of configuration settings
config_settings = (
    {"setting_name": "debug_mode", "value": True},
    {"setting_name": "log_level", "value": "INFO"},
    {"setting_name": "max_connections", "value": 100}
)
print(f"Config Settings: {config_settings}")
print(f"Log level: {config_settings[1]['value']}")

# Note: While the tuple itself is immutable, the dictionaries inside it are
mutable.
config_settings[0]['value'] = False
print(f"Debug mode updated: {config_settings[0]['value']}")

print("\nCompound data representation demonstration finished.")
```

Input

No specific user input is required. The script demonstrates the creation and manipulation of various compound data structures.

Expected Output

```
--- Representing Compound Data in Python ---

--- 1. List of Lists (Matrix) ---
Matrix: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Element at [0][0]: 1
Element at [1][2]: 6
Iterating through matrix:
1 2 3
4 5 6
7 8 9

--- 2. List of Dictionaries (Records) ---
Students data: [{'id': 101, 'name': 'Alice', 'age': 20, 'major': 'Computer
Science'}, {'id': 102, 'name': 'Bob', 'age': 22, 'major': 'Electrical
Engineering'}, {'id': 103, 'name': 'Charlie', 'age': 21, 'major': 'Computer
Science'}]
Second student's name: Bob
First student's major: Computer Science
Students in Computer Science:
- ID: 101, Name: Alice
- ID: 103, Name: Charlie

--- 3. Dictionary of Lists ---
Inventory: {'fruits': ['apple', 'banana', 'cherry'], 'vegetables': ['carrot',
'spinach', 'potato'], 'dairy': ['milk', 'cheese']}
Fruits available: ['apple', 'banana', 'cherry']
Fruits after adding grape: ['apple', 'banana', 'cherry', 'grape']

--- 4. Dictionary of Dictionaries (Nested Objects) ---
User Profiles: {'user123': {'name': 'John Doe', 'email': 'john.doe@example.com',
'address': {'street': '123 Main St', 'city': 'Anytown', 'zip': '12345'}},
'jane_smith': {'name': 'Jane Smith', 'email': 'jane.smith@example.com',
'address': {'street': '456 Oak Ave', 'city': 'Otherville', 'zip': '67890'}}}
John Doe's email: john.doe@example.com
Jane Smith's city: Otherville
John Doe's updated zip: 54321

--- 5. Mixed Compound Data (Tuple of Dictionaries) ---
Config Settings: ({'setting_name': 'debug_mode', 'value': True},
{'setting_name': 'log_level', 'value': 'INFO'}, {'setting_name':
'max_connections', 'value': 100})
Log level: INFO
Debug mode updated: False

Compound data representation demonstration finished.
```

Lab 14: Lists, tuples, dictionaries

Aim

To provide a consolidated understanding and practical implementation of Python's fundamental collection data types: lists, tuples, and dictionaries, highlighting their key characteristics and common operations.

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab14.py`).
3. Write the source code as provided below, demonstrating the creation, manipulation, and key differences of lists, tuples, and dictionaries.
4. Save the file.
5. Run the script from your terminal using `python lab14.py`.
6. Observe the output to reinforce the understanding of each data type's behavior.

Source Code

```
# lab14.py - Lists, Tuples, Dictionaries

print("--- Lists, Tuples, and Dictionaries in Python ---")

# --- 1. Lists ---
# Characteristics: Ordered, Changeable (Mutable), Allows duplicate members
print("\n--- 1. Lists ---")
my_list = ["apple", "banana", "cherry", "apple", 10, 20.5]
print(f"Original List: {my_list}")
print(f"Type of my_list: {type(my_list)}")

# Accessing elements
print(f"First element: {my_list[0]}")
print(f"Last element: {my_list[-1]}")
print(f"Slice (index 1 to 3): {my_list[1:4]}")

# Modifying elements
my_list[1] = "blueberry"
print(f"List after modifying index 1: {my_list}")

# Adding elements
my_list.append("date")
print(f"List after append: {my_list}")
my_list.insert(2, "grape")
print(f"List after insert at index 2: {my_list}")

# Removing elements
my_list.remove("apple") # Removes first occurrence
print(f"List after removing 'apple': {my_list}")
popped_item = my_list.pop() # Removes and returns last item
print(f"List after pop: {my_list}, Popped: {popped_item}")

# List length
print(f"Length of list: {len(my_list)}")

# Iterating a list
print("Iterating through list:")
for item in my_list:
    print(f"- {item}")

# --- 2. Tuples ---
# Characteristics: Ordered, Unchangeable (Immutable), Allows duplicate members
print("\n--- 2. Tuples ---")
my_tuple = ("red", "green", "blue", "red", 100, 3.14)
print(f"Original Tuple: {my_tuple}")
print(f"Type of my_tuple: {type(my_tuple)}")

# Accessing elements (same as lists)
print(f"First element: {my_tuple[0]}")
print(f"Last element: {my_tuple[-1]}")
print(f"Slice (index 1 to 3): {my_tuple[1:4]}")

# Attempting to modify (will cause an error if uncommented)
# my_tuple[0] = "purple" # TypeError: 'tuple' object does not support item
assignment

# Tuple length
print(f"Length of tuple: {len(my_tuple)}")

# Concatenating tuples
another_tuple = (1, 2, 3)
combined_tuple = my_tuple + another_tuple
print(f"Combined tuple: {combined_tuple}")

# Counting occurrences
print(f"Count of 'red' in tuple: {my_tuple.count('red')}")
print(f"Index of 'blue' in tuple: {my_tuple.index('blue')}")

# Iterating a tuple
print("Iterating through tuple:")
for item in my_tuple:
    print(f"- {item}")

# --- 3. Dictionaries ---
# Characteristics: Unordered (as of Python 3.7+ insertion order is preserved),
#                  Changeable (Mutable), No duplicate keys
print("\n--- 3. Dictionaries ---")
my_dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964,
    "colors": ["red", "white", "blue"]
}
print(f"Original Dictionary: {my_dict}")
print(f"Type of my_dict: {type(my_dict)}")

# Accessing values
print(f"Brand: {my_dict['brand']}")
print(f"Model (using .get()): {my_dict.get('model')}")
print(f"Colors: {my_dict['colors']}")

# Modifying values
my_dict["year"] = 2020
print(f"Dictionary after modifying year: {my_dict}")

# Adding new key-value pairs
my_dict["engine"] = "V8"
print(f"Dictionary after adding 'engine': {my_dict}")

# Removing key-value pairs
removed_value = my_dict.pop("colors")
print(f"Dictionary after pop 'colors': {my_dict}, Removed value:
{removed_value}")
del my_dict["brand"]
print(f"Dictionary after del 'brand': {my_dict}")
```

```
# Dictionary length
print(f"Length of dictionary: {len(my_dict)}")

# Getting keys, values, items
print(f"Keys: {my_dict.keys()}")
print(f"Values: {my_dict.values()}")
print(f"Items (key-value pairs): {my_dict.items()}")

# Iterating a dictionary
print("Iterating through dictionary items:")
for key, value in my_dict.items():
    print(f"- {key}: {value}")

print("\nDemonstration of Lists, Tuples, and Dictionaries finished.")
```

Input

No specific user input is required. The script demonstrates the characteristics and operations of lists, tuples, and dictionaries.

Expected Output

```
--- Lists, Tuples, and Dictionaries in Python ---

--- 1. Lists ---
Original List: ['apple', 'banana', 'cherry', 'apple', 10, 20.5]
Type of my_list: <class 'list'>
First element: apple
Last element: 20.5
Slice (index 1 to 3): ['banana', 'cherry', 'apple']
List after modifying index 1: ['apple', 'blueberry', 'cherry', 'apple', 10,
20.5]
List after append: ['apple', 'blueberry', 'cherry', 'apple', 10, 20.5, 'date']
List after insert at index 2: ['apple', 'blueberry', 'grape', 'cherry', 'apple',
10, 20.5, 'date']
List after removing 'apple': ['blueberry', 'grape', 'cherry', 'apple', 10, 20.5,
'date']
List after pop: ['blueberry', 'grape', 'cherry', 'apple', 10, 20.5], Popped:
date
Length of list: 6
Iterating through list:
- blueberry
- grape
- cherry
- apple
- 10
- 20.5

--- 2. Tuples ---
Original Tuple: ('red', 'green', 'blue', 'red', 100, 3.14)
Type of my_tuple: <class 'tuple'>
First element: red
Last element: 3.14
Slice (index 1 to 3): ('green', 'blue', 'red')
Length of tuple: 6
Combined tuple: ('red', 'green', 'blue', 'red', 100, 3.14, 1, 2, 3)
Count of 'red' in tuple: 2
Index of 'blue' in tuple: 2
Iterating through tuple:
- red
- green
- blue
- red
- 100
- 3.14

--- 3. Dictionaries ---
Original Dictionary: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964,
'colors': ['red', 'white', 'blue']}
Type of my_dict: <class 'dict'>
Brand: Ford
Model (using .get()): Mustang
Colors: ['red', 'white', 'blue']
Dictionary after modifying year: {'brand': 'Ford', 'model': 'Mustang', 'year':
2020, 'colors': ['red', 'white', 'blue']}
Dictionary after adding 'engine': {'brand': 'Ford', 'model': 'Mustang', 'year':
2020, 'colors': ['red', 'white', 'blue'], 'engine': 'V8'}
Dictionary after pop 'colors': {'brand': 'Ford', 'model': 'Mustang', 'year':
2020, 'engine': 'V8'}, Removed value: ['red', 'white', 'blue']
Dictionary after del 'brand': {'model': 'Mustang', 'year': 2020, 'engine': 'V8'}
Length of dictionary: 3
Keys: dict_keys(['model', 'year', 'engine'])
Values: dict_values(['Mustang', 2020, 'V8'])
Items (key-value pairs): dict_items([('model', 'Mustang'), ('year', 2020),
('engine', 'V8')])
Iterating through dictionary items:
- model: Mustang
- year: 2020
- engine: V8

Demonstration of Lists, Tuples, and Dictionaries finished.
```

Lab 15: Read and write data from/to files in Python Programs

Aim

To understand and implement various methods for reading and writing data to text files in Python, including different modes of file opening and best practices for file handling.

Procedure

1. Open a text editor or IDE.
2. Create a new Python file (e.g., `lab15.py`).
3. Write the source code as provided below, demonstrating file writing and reading operations.
4. Save the file.
5. Run the script from your terminal using `python lab15.py`.
6. Observe the creation of new files, their contents, and the output of reading operations on the console. You can also manually check the created files (`example.txt`, `lines.txt`, `append_example.txt`) in the same directory.

Source Code

```python
# lab15.py - Read and write data from/to files in Python Programs

import os

print("--- Reading and Writing Data from/to Files ---")

# --- 1. Writing to a file (Write Mode: 'w') ---
# 'w' mode:
#   - Creates the file if it does not exist.
#   - If the file exists, it truncates (empties) the file before writing.
print("\n--- 1. Writing to 'example.txt' (mode 'w') ---")
file_name_w = "example.txt"
content_w = "Hello, Python file handling!\n" \
            "This is the first line.\n" \
            "This is the second line."

try:
    with open(file_name_w, 'w') as file:
        file.write(content_w)
    print(f"Successfully wrote content to '{file_name_w}'.")
except IOError as e:
    print(f"Error writing to '{file_name_w}': {e}")

# --- 2. Reading from a file (Read Mode: 'r') ---
# 'r' mode:
#   - Opens the file for reading.
#   - Raises FileNotFoundError if the file does not exist.
print(f"\n--- 2. Reading from '{file_name_w}' (mode 'r') ---")
try:
    with open(file_name_w, 'r') as file:
        read_data = file.read() # Reads the entire content of the file
        print("Content of the file:")
        print(read_data)
except FileNotFoundError:
    print(f"Error: File '{file_name_w}' not found.")
except IOError as e:
    print(f"Error reading from '{file_name_w}': {e}")
```

```
# --- 3. Reading line by line (Read Mode: 'r' with readline() or readlines()) --
-
print("\n--- 3. Reading 'lines.txt' line by line ---")
file_name_lines = "lines.txt"
lines_to_write = [
    "Line 1: This is the first line.\n",
    "Line 2: This is the second line.\n",
    "Line 3: And this is the third line.\n"
]

# First, create the file with multiple lines
try:
    with open(file_name_lines, 'w') as file:
        file.writelines(lines_to_write) # Writes a list of strings
    print(f"Successfully wrote lines to '{file_name_lines}'.")
except IOError as e:
    print(f"Error writing lines to '{file_name_lines}': {e}")

# Now, read line by line
print(f"\nReading '{file_name_lines}' using readlines():")
try:
    with open(file_name_lines, 'r') as file:
        all_lines = file.readlines() # Reads all lines into a list of strings
        for i, line in enumerate(all_lines):
            print(f"Line {i+1}: {line.strip()}") # .strip() to remove trailing
newline
except FileNotFoundError:
    print(f"Error: File '{file_name_lines}' not found.")
except IOError as e:
    print(f"Error reading lines from '{file_name_lines}': {e}")

print(f"\nReading '{file_name_lines}' using a loop (more memory efficient for
large files):")
try:
    with open(file_name_lines, 'r') as file:
        for i, line in enumerate(file): # Iterates over lines directly
            print(f"Line {i+1}: {line.strip()}")
except FileNotFoundError:
    print(f"Error: File '{file_name_lines}' not found.")
except IOError as e:
    print(f"Error reading lines from '{file_name_lines}': {e}")


# --- 4. Appending to a file (Append Mode: 'a') ---
# 'a' mode:
#   - Creates the file if it does not exist.
#   - If the file exists, it appends content to the end of the file.
print("\n--- 4. Appending to 'append_example.txt' (mode 'a') ---")
file_name_a = "append_example.txt"
append_content1 = "This is the initial content.\n"
append_content2 = "This line is appended later.\n"
append_content3 = "And another line is appended."

# Write initial content
try:
    with open(file_name_a, 'w') as file: # Use 'w' first to ensure a clean start
        file.write(append_content1)
    print(f"Successfully wrote initial content to '{file_name_a}'.")
except IOError as e:
    print(f"Error writing initial content to '{file_name_a}': {e}")

# Append content
try:
    with open(file_name_a, 'a') as file:
        file.write(append_content2)
        file.write(append_content3)
    print(f"Successfully appended content to '{file_name_a}'.")
```

```
except IOError as e:
    print(f"Error appending to '{file_name_a}': {e}")

# Read to verify appended content
print(f"\nContent of '{file_name_a}' after appending:")
try:
    with open(file_name_a, 'r') as file:
        print(file.read())
except FileNotFoundError:
    print(f"Error: File '{file_name_a}' not found.")
except IOError as e:
    print(f"Error reading from '{file_name_a}': {e}")


# --- 5. Binary Mode (e.g., 'wb', 'rb') ---
# For handling non-text files (images, audio, etc.) or when explicit byte
handling is needed.
print("\n--- 5. Binary Mode (Writing and Reading Bytes) ---")
file_name_b = "binary_data.bin"
binary_data = b'\x00\x01\x02\x03\xff\xfe\xfd\xfc' # Byte string

try:
    with open(file_name_b, 'wb') as file: # 'wb' for write binary
        file.write(binary_data)
    print(f"Successfully wrote binary data to '{file_name_b}'.")

    with open(file_name_b, 'rb') as file: # 'rb' for read binary
        read_binary_data = file.read()
    print(f"Read binary data from '{file_name_b}': {read_binary_data}")
    print(f"Type of read binary data: {type(read_binary_data)}")
except IOError as e:
    print(f"Error handling binary file '{file_name_b}': {e}")


# --- Clean up: Remove created files (optional, but good practice) ---
print("\n--- Cleaning up created files ---")
files_to_clean = [file_name_w, file_name_lines, file_name_a, file_name_b]
for f_name in files_to_clean:
    if os.path.exists(f_name):
        try:
            os.remove(f_name)
            print(f"Deleted '{f_name}'.")
        except OSError as e:
            print(f"Error deleting '{f_name}': {e}")
    else:
        print(f"'{f_name}' does not exist, skipping deletion.")

print("\nFile reading and writing demonstration finished.")
```

Input

No specific user input is required. The script performs file operations automatically.

Expected Output

```
--- Reading and Writing Data from/to Files ---

--- 1. Writing to 'example.txt' (mode 'w') ---
Successfully wrote content to 'example.txt'.

--- 2. Reading from 'example.txt' (mode 'r') ---
Content of the file:
Hello, Python file handling!
This is the first line.
This is the second line.

--- 3. Reading 'lines.txt' line by line ---
Successfully wrote lines to 'lines.txt'.

Reading 'lines.txt' using readlines():
Line 1: Line 1: This is the first line.
Line 2: Line 2: This is the second line.
Line 3: And this is the third line.

Reading 'lines.txt' using a loop (more memory efficient for large files):
Line 1: Line 1: This is the first line.
Line 2: Line 2: This is the second line.
Line 3: And this is the third line.

--- 4. Appending to 'append_example.txt' (mode 'a') ---
Successfully wrote initial content to 'append_example.txt'.
Successfully appended content to 'append_example.txt'.

Content of 'append_example.txt' after appending:
This is the initial content.
This line is appended later.
And another line is appended.

--- 5. Binary Mode (Writing and Reading Bytes) ---
Successfully wrote binary data to 'binary_data.bin'.
Read binary data from 'binary_data.bin': b'\x00\x01\x02\x03\xff\xfe\xfd\xfc'
Type of read binary data: <class 'bytes'>

--- Cleaning up created files ---
Deleted 'example.txt'.
Deleted 'lines.txt'.
Deleted 'append_example.txt'.
Deleted 'binary_data.bin'.

File reading and writing demonstration finished.
```