# Peoples' Information Technology Program (PITP)

**An Initiative by Information Science & Technology Department, Government of Sindh**

# PYTHON

# Course Manual

**Prepared By:**

**Dr. Muhammad Asif Khan**
**Assistant Professor**
**Sukkur IBA University**

# Table of Contents

## Python Programming Bootcamp – Recommended Resources

---

### Books

1. **Python Crash Course (2nd Edition)** – by Eric Matthes
    - *Description:* A hands-on, project-based introduction to Python. This book is ideal for beginners and covers essential programming concepts using real-world projects.
    - *Publisher:* No Starch Press
    - *ISBN:* 978-1593279288
2. **Automate the Boring Stuff with Python** – by Al Sweigart
    - *Description:* This book focuses on automating everyday tasks using Python. It's suitable for both beginners and intermediate learners looking to apply Python in practical scenarios.
    - *Publisher:* No Starch Press
    - *ISBN:* 978-1593279929
3. **Python for Data Analysis (2nd Edition)** – by Wes McKinney
    - *Description:* A comprehensive guide to using Python for data analysis, including an introduction to libraries such as Pandas, NumPy, and Matplotlib.
    - *Publisher:* O'Reilly Media
    - *ISBN:* 978-1491957660
4. **Fluent Python** – by Luciano Ramalho
    - *Description:* This book is for intermediate-to-advanced Python learners and delves deeply into Python's best practices, including data structures, functions, and object-oriented programming.
    - *Publisher:* O'Reilly Media
    - *ISBN:* 978-1491946008
5. **Python Programming: An Introduction to Computer Science (3rd Edition)** – by John Zelle
    - *Description:* A beginner-friendly introduction to Python programming with a focus on computer science concepts.
    - *Publisher:* Franklin, Beedle & Associates Inc.
    - *ISBN:* 978-1590282755

---

### Online Tutorials

1. **Real Python**
    - *URL:* realpython.com
    - *Description:* Real Python provides high-quality tutorials and courses on a wide range of Python topics, from beginner to advanced levels. It includes step-by-step guides and coding examples.

2. **W3Schools – Python Tutorial**
    - *URL:* w3schools.com/python
    - *Description:* A great beginner-friendly resource that offers interactive tutorials on basic Python concepts, control flow, data structures, and more.
3. **Learn Python – Python.org**
    - *URL:* docs.python.org/3/tutorial
    - *Description:* The official Python documentation provides a thorough tutorial covering everything from Python syntax to advanced concepts such as object-oriented programming.
4. **Code Academy – Learn Python**
    - *URL:* codecademy.com/learn/learn-python-3
    - *Description:* A beginner-friendly, interactive course designed to teach Python from the ground up. It includes hands-on exercises, quizzes, and projects.
5. **SoloLearn – Python for Beginners**
    - *URL:* sololearn.com
    - *Description:* SoloLearn offers bite-sized lessons on Python, making it perfect for learners who want to practice Python on the go.
6. **GeeksforGeeks Python Programming Language**
    - *URL:* geeksforgeeks.org/python-programming-language/
    - *Description:* A comprehensive set of Python tutorials that covers a wide range of topics, including control structures, object-oriented programming, and data structures.

---

**Online Compilers**

1. **Replit**
    - *URL:* replit.com
    - *Description:* An excellent cloud-based IDE that supports Python and many other programming languages. Replit allows you to write, run, and debug code directly in the browser.
2. **PythonAnywhere**
    - *URL:* pythonanywhere.com
    - *Description:* A browser-based Python development and hosting environment that allows you to run Python scripts without the need for local installations.
3. **Programiz – Python Compiler**
    - *URL:* programiz.com/python-programming/online-compiler/
    - *Description:* A simple, online Python compiler that lets you write and execute Python code instantly. Great for quick testing and debugging.
4. **JDoodle**
    - *URL:* jdoodle.com/python3-programming-online/

- *Description:* A versatile online compiler that supports Python 3 and includes features such as code sharing and API integration.

5. **Pycharm Edu**
    - *URL:* jetbrains.com/pycharm-edu/
    - *Description:* A free educational IDE by JetBrains designed for learning and teaching Python. It provides an interactive coding environment with in-built exercises and quizzes.

---

**Note:** Each of the resources listed here is intended to supplement your learning throughout the Python Bootcamp. Feel free to explore different resources based on your learning style and objectives. All of the abovementioned books' pdfs can be downloaded free from the internet.

## Incremental development of a Project

Welcome to the Python Programming Bootcamp! Throughout this course, you will work on a **Task Management System** project, which will be built progressively as you learn Python programming concepts. This project will help you apply everything you learn in a practical, hands-on manner, from basic control flow to more advanced techniques like file handling, object-oriented programming, and working with external libraries.

---

## Project Title: Task Management System

1. **Objective:**

By the end of this course, you will have developed a fully functional **Task Management System** that allows users to:

- Add, update, and delete tasks.
- Check if tasks are overdue, due today, or due in the future.
- Organize tasks by priority or due date.
- Save and load tasks using file handling.
- Extend functionality using object-oriented programming and APIs.

---

## Project Features by Week:

- **Week 1**: Introduction to Python Basics and Control Flow
  - Learn how to use if-else statements, loops, and functions to handle task statuses (overdue, due today, due in the future).
- **Week 2**: Lists, Loops, and Basic Operations
  - Store tasks in a list and iterate over them to manage multiple tasks.
- **Week 3**: Functions, Data Structures, and More
  - Modularize your code by organizing it into reusable functions and use data structures like dictionaries for task attributes.
- **Week 4**: File Handling and Project Structuring
  - Implement file operations to save and load tasks, ensuring users can keep track of their tasks between sessions.
- **Week 5**: Object-Oriented Programming (OOP)
  - Refactor the project to use classes and objects, creating a more scalable and maintainable codebase.
- **Week 6**: API Integration and External Libraries
  - Enhance the task manager by integrating external libraries and APIs, such as using an API to retrieve weather information or setting up task reminders.

- **Week 7**: Data Visualization and Advanced Features
    - Add data visualization to display task completion progress over time using Python libraries like Matplotlib.
- **Week 8**: Project Finalization and Deployment
    - Review and refine your code, deploy the project, and present the final version as your capstone project for this course.

---

## Learning Outcomes:

By completing this project, you will:

- Gain hands-on experience in Python programming.
- Learn to build real-world applications incrementally.
- Understand key concepts like control flow, functions, data structures, object-oriented programming, file handling, and API integration.
- Develop problem-solving skills by debugging and improving your project.

---

This **Task Management System** project will serve as a practical application of the concepts you learn each week, making your learning experience more engaging and rewarding. By the end of the course, you'll have a complete project to showcase as a part of your programming portfolio.

Let's get started!

# Lab Manual for Week 1: Basic Task Input and Output - Day 1

## Day 1: Introduction to Python, Installation, and Basic Concepts

Time: 3 Hours

Learning Outcomes:
- Install Python and set up an Integrated Development Environment (IDE).
- Understand basic programming concepts.
- Write and execute your first Python program.
- Work with basic data types (strings, integers).
- Understand how to declare and use variables.

### Tasks:

1. Install Python and IDE Setup (45 mins):
- Python Installation: Visit https://www.python.org/downloads/ and download the latest version of Python.
- Run the installer, and ensure you check the box that says 'Add Python to PATH' during installation.
- Verify the installation by opening a terminal/command prompt and typing:
`python --version`
- Install an IDE like Visual Studio Code or PyCharm.

- You may also use online compilers, those are mentioned in above resources section.

2. Setting Up the IDE for Python (30 mins):
- Open your IDE and set up a new Python file (e.g., `first_program.py`).
- Run a simple script to ensure Python is working:

```
print('Python is successfully installed!')
```

3. Introduction to Programming (30 mins):
- What is programming?
- Overview of basic programming concepts: variables, data types, input/output.
- Why Python? Talk about its simplicity, readability, and use cases.
4. Writing and Running Your First Python Program (45 mins):

Write a Python program:

```
print('Hello! My name is John Doe.')
print('I am 25 years old.')
print('I love playing basketball.')
```

5. Introduction to Variables (30 mins):
- Understand variables and how they store data.
- Assign values to variables and display them using `print()`.

```
name = 'Alice'
age = 30
favorite_food = 'Pasta'
print('Name:', name)
print('Age:', age)
print('Favorite Food:', favorite_food)
```

6. User Input and String Concatenation (30 mins):
- Learn how to use the `input()` function to get user input.
- Write a program that asks the user for their name and age:

```
name = input('Enter your name: ')
age = input('Enter your age: ')
print('Hello, ' + name + '! You are ' + age + ' years old.')
```

7. Checking Data Types (30 mins):
Write a Python program that creates a string, integer, and float variable, then prints their data types:

```
name = 'Alice'
age = 30
height = 5.5
print('Type of name:', type(name))
print('Type of age:', type(age))
print('Type of height:', type(height))
```

8. Simple Arithmetic Operations (30 mins):
- Learn how to use basic arithmetic operations (addition, subtraction, multiplication, division).
- Write a program that asks the user for two numbers and performs arithmetic operations:

```
num1 = int(input('Enter first number: '))
num2 = int(input('Enter second number: '))
sum = num1 + num2
product = num1 * num2
print('Sum:', sum)
print('Product:', product)
```

9. Combining Strings and Numbers (30 mins):
- Learn how to concatenate numbers and strings in the same output.
- Write a program that asks the user for their name and age, then prints a personalized message that includes arithmetic operations:

```
name = input('Enter your name: ')
age = int(input('Enter your age: '))
years_left = 100 - age
print('Hello ' + name + '! You will turn 100 in ' + str(years_left)
+ ' years.')
```

## Assignment:

Write a detailed Python program that asks the user for the following details:
- Full name
- Age
- Favorite hobby
- Year of birth

Based on the inputs, the program should:
1. Greet the user with their full name.
2. Calculate the number of years remaining until they turn 100 years old.
3. Display their age in dog years (1 human year = 7 dog years).
4. Print a personalized message mentioning their hobby and what year they were born in.

```
name = input('Enter your full name: ')
age = int(input('Enter your age: '))
hobby = input('What is your favorite hobby? ')
year_of_birth = int(input('Enter your year of birth: '))
years_until_100 = 100 - age
age_in_dog_years = age * 7
print('Hello ' + name + '!')
print('You have ' + str(years_until_100) + ' years left until you
turn 100.')
print('In dog years, you are ' + str(age_in_dog_years) + ' years
old.')
print('You were born in ' + str(year_of_birth) + ' and love ' +
hobby + '!')
```

# Lab Manual for Week 1: Basic Task Input and Output - Day 2

## Day 2: Variables, Arithmetic Operations, and Data Types

Time: 3 Hours

Learning Outcomes:
- Understand how variables work in greater depth.
- Perform arithmetic operations in Python.
- Use Python as a calculator to perform real-world calculations.
- Explore more advanced data types and operations.

### Tasks:

1. Review of Variables and Data Types (30 mins):
- Recap variables, data types (strings, integers, floats, booleans).
- Practice variable reassignment and the impact it has on the program.

```
age = 20
print('Initial age:', age)
age = age + 5
print('Updated age:', age)
```

2. Performing Arithmetic Operations (45 mins):
- Learn how to perform basic arithmetic operations in Python: addition, subtraction, multiplication, division.
- Write Python programs to calculate the sum, difference, product, and quotient of two numbers.

```
num1 = int(input('Enter first number: '))
num2 = int(input('Enter second number: '))
sum_result = num1 + num2
difference = num1 - num2
product = num1 * num2
quotient = num1 / num2
print('Sum:', sum_result)
print('Difference:', difference)
print('Product:', product)
print('Quotient:', quotient)
```

3. Working with Floats and Integers (30 mins):
- Understand the difference between integers and floats.
- Perform operations with both integer and float data types.

```
width = float(input('Enter the width of the rectangle: '))
height = float(input('Enter the height of the rectangle: '))
area = width * height
print('The area of the rectangle is:', area)
```

4. Combining User Input and Calculations (45 mins):
- Write programs that ask the user for inputs (like width, height, and other measurements) and perform calculations using these inputs.
- Practice using input() and basic arithmetic to write useful programs.

```
price = float(input('Enter the price of the item: '))
quantity = int(input('Enter the quantity: '))
total_cost = price * quantity
print('The total cost is:', total_cost)
```

5. Complex Calculations (45 mins):
- Learn how to calculate more complex results like averages, percentages, and discounts.
- Write a Python program that calculates the total cost, including a discount, of a purchased item.

```
price = float(input('Enter the price of the item: '))
quantity = int(input('Enter the quantity: '))
discount = 0.1
subtotal = price * quantity
final_price = subtotal * (1 - discount)
print('Final price after discount:', final_price)
```

6. Handling Division and Float Precision (30 mins):
- Explore how Python handles division and float precision.
- Write a program that divides two numbers and limits the result to two decimal places.
- Start teaching them built-in functions, e.g., round().

```
num1 = float(input('Enter first number: '))
num2 = float(input('Enter second number: '))
quotient = num1 / num2
print('Quotient (rounded):', round(quotient, 2))
```

7. Calculating Averages (30 mins):
- Write a program that asks the user for two numbers and calculates the average.

```
num1 = float(input('Enter the first number: '))
num2 = float(input('Enter the second number: '))
average = (num1 + num2) / 2
print('The average is:', average)
```

8. Advanced Input and Output (30 mins):

- Use formatted strings to display results with clarity.
- Write a Python program that takes multiple inputs from the user and displays them in a well-formatted output.

```
name = input('Enter your name: ')
age = int(input('Enter your age: '))
print(f'Hello {name}, you are {age} years old.')
```

9. Exploring Data Types (30 mins):
- Learn how to check data types using the `type()` function.
- Write a Python program that prints the data type of variables.

```
name = 'Alice'
age = 25
height = 5.7
print('Type of name:', type(name))
print('Type of age:', type(age))
print('Type of height:', type(height))
```

10. More Arithmetic Operations (45 mins):
- Write a program that calculates the area and perimeter of a rectangle and circle.
- Use both integers and floats in the same program.

```
length = float(input('Enter the length of the rectangle: '))
width = float(input('Enter the width: '))
area = length * width
perimeter = 2 * (length + width)
print('Area:', area)
print('Perimeter:', perimeter)
```

11. Using the Modulus Operator (30 mins):
- Learn how to use the modulus operator `%` to find remainders in division.
- Write a program that checks if a number is even or odd using the modulus operator.

```
num = int(input('Enter a number: '))
print('The number is even (0) else odd.'+ (num % 2))
```

12. Combining Arithmetic Operations (45 mins):
- Write a more complex program that asks the user for three numbers and performs multiple operations on them (addition, multiplication, division).

```
num1 = float(input('Enter the first number: '))
num2 = float(input('Enter the second number: '))
num3 = float(input('Enter the third number: '))
sum_result = num1 + num2 + num3
product = num1 * num2 * num3
quotient = sum_result / num3
```

```
print('Sum:', sum_result)
print('Product:', product)
print('Quotient:', quotient)
```

**Assignment:**

- Write a Python program that asks the user for the radius of a circle and calculates the area using the formula: `Area = π * radius^2`.

# Lab Manual for Week 1: Basic Task Input and Output - Day 3

## Day 3: Input, Output, and String Formatting

Time: 3 Hours

Learning Outcomes:
- Gain a deeper understanding of user input and output in Python.
- Learn advanced string formatting techniques.
- Build a basic Python program using multiple inputs and outputs.

### Tasks:

1a. Correct Variable Naming in Python:

- Variable names must start with a letter or an underscore (_), but not a number.
- Variable names can only contain alphanumeric characters (A-Z, a-z, 0-9) and underscores (_).
- Variable names are case-sensitive (e.g., age and Age are different variables).
- Variable names should be descriptive and concise, avoiding abbreviations where clarity might be lost.
- Avoid using Python keywords like if, else, for, class, etc., as variable names.

1) 1st_variable = "First variable"
2) @name = "John"
3) my variable = "This is wrong"
4) user-age = 25
5) total$amount = 1000

1b. Basic Input and Output (30 mins):
Review the `input()` and `print()` functions for user interaction.
- Write a Python program that asks the user for their name and age, then displays the message using `print()`.

```
name = input('Enter your name: ')
age = input('Enter your age: ')
print('Hello, ' + name + '! You are ' + age + ' years old.')
```

2. String Concatenation (30 mins):
Practice combining strings using concatenation.
- Write a Python program that accepts a favorite color and food from the user and combines them in a sentence.

```
color = input('Enter your favorite color: ')
food = input('Enter your favorite food: ')
```

```
print('Your favorite color is ' + color + ' and you like to eat '
+ food + '.')
```

3. String Formatting Using f-Strings (30 mins):
Learn how to format strings using f-strings for cleaner code and output.
- Write a Python program that accepts name, age, and hobby, and formats them in a message using f-strings.

```
name = input('Enter your name: ')
age = int(input('Enter your age: '))
hobby = input('Enter your favorite hobby: ')
print(f'Hello {name}, you are {age} years old and love {hobby}.')
```

4. Advanced Input and Output (45 mins):
Learn to take multiple inputs and format them in a structured output.
- Write a program that asks for the user's name, profession, and a fun fact, then prints them in a multi-line format.

```
name = input('Enter your name: ')
profession = input('Enter your profession: ')
fun_fact = input('Tell me a fun fact about yourself: ')
print(f'\nName: {name}\nProfession: {profession}\nFun Fact:
{fun_fact}\n')
```

5. String Formatting with the `format()` Method (45 mins):
Learn how to use the `format()` method for string formatting.
- Write a program that asks for the user's favorite movie and city, then prints a formatted message using `format()`.

```
movie = input('Enter your favorite movie: ')
city = input('Enter your favorite city: ')
print('Your favorite movie is {} and your favorite city is
{}.'.format(movie, city))
```

6. Handling Different Data Types in Input (45 mins):
Learn to handle different data types (integers, floats, strings) when accepting input.
- Write a program that asks for a person's height (float) and weight (float) and calculates their Body Mass Index (BMI).

```
height = float(input('Enter your height in meters: '))
weight = float(input('Enter your weight in kilograms: '))
bmi = weight / (height ** 2)
print(f'Your BMI is {bmi:.2f}')
```

7. String Methods (45 mins):

Explore string methods like `upper()`, `lower()`, and `title()` to manipulate input.
- Write a program that takes a sentence from the user and prints it in uppercase, lowercase, and title case.

```
sentence = input('Enter a sentence: ')
print('Uppercase:', sentence.upper())
print('Lowercase:', sentence.lower())
print('Title Case:', sentence.title())
```

8. Input Validation and Error Handling (45 mins):
Implement input validation and error handling in Python to prevent invalid inputs.
- Write a program that asks for a user's age and ensures that the input is a valid integer. If it's not, display an error message and ask for the input again.

```
age = input('Enter your age: ')
if age.isdigit():
    age = int(age)
    print(f'You are {age} years old.')
    break
else:
    print('Invalid input. Please enter a valid number.')
```

9. Multi-line Input and Output (45 mins):
Learn how to handle multi-line input and output.
- Write a Python program that accepts multiple lines of text from the user (e.g., a short story) and prints it back in a formatted structure.

```
story = input('Tell me a short story: ')
print(f'\nYour Story:\n{story}')
```

10. Combining Input from Multiple Sources (45 mins):
Practice accepting input from multiple sources and combining it into one output.
- Write a program that asks for the user's first name, last name, and age, and combines the inputs into a structured message.

```
first_name = input('Enter your first name: ')
last_name = input('Enter your last name: ')
age = int(input('Enter your age: '))
print(f'Hello {first_name} {last_name}, you are {age} years old.')
```

11. Handling Special Characters in Input (30 mins):
Learn how to handle special characters in user input (such as quotes, slashes, etc.).
- Write a program that asks for a quote from the user and prints it, ensuring proper formatting of special characters.

```
quote = input("Enter your favorite quote: ")
print(f"Your quote is: '{quote}'")
```

12. Looping Through User Input (45 mins):
Practice using loops to iterate through user input.
- Write a Python program that asks the user to enter five hobbies, stores them in a list, and prints the list after each input.

```
hobbies = []
for i in range(5):
    hobby = input(f'Enter hobby {i+1}: ')
    hobbies.append(hobby)
    print('Current hobbies:', hobbies)
```

# Lab Manual for Week 1: Basic Task Input and Output - Day 4

## Day 4: Conditional Statements and Loops

Time: 3 Hours

Learning Outcomes:
- Understand the use of conditional statements in Python (if, elif, else).
- Gain proficiency in loops (for and while).
- Learn how to combine loops and conditions to solve practical problems.

### Tasks:

1. Simple If Statement (15 mins):
Write a Python program that checks if a number is positive and prints a message.

```
num = int(input('Enter a number: '))
if num > 0:
    print('The number is positive.')
```

2. If-Else Statement (15 mins):
Write a Python program that checks if a number is even or odd.

```
num = int(input('Enter a number: '))
if num % 2 == 0:
    print('The number is even.')
else:
    print('The number is odd.')
```

3. If-Elif-Else Statement (15 mins):
Write a program that checks if a number is positive, negative, or zero.

```
num = int(input('Enter a number: '))
if num > 0:
    print('Positive')
elif num < 0:
    print('Negative')
else:
    print('Zero')
```

4. Nested If Statement (15 mins):
Write a Python program to check eligibility for a discount based on age and membership status.

```
age = int(input("Enter your age: "))
membership_status = input("Are you a member (yes/no)? ").lower()

if age >= 18:
    if membership_status == 'yes':
        print("You are eligible for a 20% discount!")
    else:
        print("You are eligible for a 10% discount!")
else:
    if membership_status == 'yes':
        print("You are eligible for a 15% discount!")
    else:
        print("You are eligible for a 5% discount!")
```

4(a). The following code is supposed to check if a number is positive, negative, or zero, but it contains incorrect indentation. Your task is to fix the indentation so the code runs properly.

- Teacher should teach importance of having proper indentation in python. Also, teach about writing user-friendly and readable code.

```
number = int(input("Enter a number: "))
if number > 0:
print("The number is positive.")
elif number == 0:
print("The number is zero.")
else:
print("The number is negative.")
```

5. While Loop (20 mins):
Write a Python program that prints the numbers from 1 to 10 using a while loop.

```
i = 1
while i <= 10:
    print(i)
    i += 1
```

6. For Loop with Range (15 mins):
Write a Python program that prints the numbers from 1 to 10 using a for loop.

```
for i in range(1, 11):
    print(i)
```

7. Breaking Out of a Loop (15 mins):
Write a Python program that asks the user for numbers and prints them. Stop the loop when the user enters '0'.

```
while True:
    num = int(input('Enter a number: '))
    if num == 0:
        break
    print(f'You entered: {num}')
```

8. Using Continue in a Loop (15 mins):
Write a Python program that prints numbers from 1 to 10, but skips printing multiples of 3.

```
for i in range(1, 11):
    if i % 3 == 0:
        continue
    print(i)
```

9. Using Else with Loops (15 mins):
Write a Python program that uses a for loop to find the first number divisible by 7 between 1 and 20. Use else to print a message if no number is found.

```
for i in range(1, 21):
    if i % 7 == 0:
        print(f'{i} is divisible by 7')
        break
else:
    print('No number divisible by 7 found.')
```

10. Checking Prime Numbers (20 mins):
Write a Python program that checks if a number is prime.

```
num = int(input('Enter a number: '))
if num > 1:
    for i in range(2, num):
        if num % i == 0:
            print(f'{num} is not prime')
            break
    else:
        print(f'{num} is prime')
else:
    print(f'{num} is not prime')
```

11. Sum of Numbers Using Loops (20 mins):
Write a Python program that calculates the sum of numbers from 1 to 100 using a loop.

```
total = 0
for i in range(1, 101):
    total += i
print(f'The sum of numbers from 1 to 100 is {total}')
```

12. Factorial Calculation (20 mins):

Write a Python program that calculates the factorial of a number using a loop.

```
num = int(input('Enter a number: '))
factorial = 1
for i in range(1, num + 1):
    factorial *= i
print(f'The factorial of {num} is {factorial}')
```

13. Reverse a String (15 mins):
Write a Python program that asks the user for a string and prints the string in reverse.

```
s = input('Enter a string: ')
print(f'The reverse of the string is {s[::-1]}')
```

14. Palindrome Check (20 mins):
Write a Python program that checks if a given string is a palindrome.

```
s = input('Enter a string: ')
if s == s[::-1]:
    print(f'{s} is a palindrome')
else:
    print(f'{s} is not a palindrome')
```

15. FizzBuzz Problem (20 mins):
Write a Python program that prints the numbers from 1 to 50. For multiples of 3, print 'Fizz', for multiples of 5, print 'Buzz', and for multiples of both, print 'FizzBuzz'.

```
for i in range(1, 51):
    if i % 3 == 0 and i % 5 == 0:
        print('FizzBuzz')
    elif i % 3 == 0:
        print('Fizz')
    elif i % 5 == 0:
        print('Buzz')
    else:
        print(i)
```

16. Multiplication Table (20 mins):
Write a Python program that prints the multiplication table for numbers 1 through 10.

```
for i in range(1, 11):
    for j in range(1, 11):
        print(f'{i} x {j} = {i * j}')
```

17. Sum of Even Numbers (15 mins):
Write a Python program that calculates the sum of all even numbers from 1 to 100.

```
total = 0
for i in range(2, 101, 2):
```

```
    total += i
print(f'The sum of all even numbers from 1 to 100 is {total}')
```

18. Finding the Largest Number (20 mins):
Write a Python program that takes three numbers from the user and prints the largest one. Teach about max() function.

```
a = int(input('Enter first number: '))
b = int(input('Enter second number: '))
c = int(input('Enter third number: '))
print(f'The largest number is {max(a, b, c)}')
```

19. Count Vowels in a String (20 mins):
Write a Python program that counts the number of vowels in a given string.

```
vowels = 'aeiouAEIOU'
string = input('Enter a string: ')
vowel_count = 0
for char in string:
    if char in vowels:
        vowel_count += 1
print(f'There are {vowel_count} vowels in the string.')
```

**Assignment:**

- Add conditional statements to your task manager to check whether tasks are overdue, due today, or due in the future, and print appropriate messages.

```
from datetime import datetime
# Task 1
task1_title = "Complete Python exercises"
task1_due_date = "2024-09-22"
# Task 2
task2_title = "Submit project proposal"
task2_due_date = "2024-09-23"
# Task 3
task3_title = "Prepare for presentation"
task3_due_date = "2024-09-25"
# Get today's date
today = datetime.now().date()
# Task 1 status check
task1_due = datetime.strptime(task1_due_date, "%Y-%m-%d").date()
if task1_due < today:
```

```
    print(f"Task '{task1_title}' is overdue! It was due on
{task1_due}.")
elif task1_due == today:
    print(f"Task '{task1_title}' is due today!")
else:
    print(f"Task '{task1_title}' is due in the future on {task1_due}.")
# Task 2 status check
task2_due = datetime.strptime(task2_due_date, "%Y-%m-%d").date()
if task2_due < today:
    print(f"Task '{task2_title}' is overdue! It was due on
{task2_due}.")
elif task2_due == today:
    print(f"Task '{task2_title}' is due today!")
else:
    print(f"Task '{task2_title}' is due in the future on {task2_due}.")
# Task 3 status check
task3_due = datetime.strptime(task3_due_date, "%Y-%m-%d").date()
if task3_due < today:
    print(f"Task '{task3_title}' is overdue! It was due on
{task3_due}.")
elif task3_due == today:
    print(f"Task '{task3_title}' is due today!")
else:
    print(f"Task '{task3_title}' is due in the future on {task3_due}.")
```

# Lab Manual for Week 1: Project Day - Day 5

## Day 5: Project Day - Week 1

Time: 3 Hours

Learning Outcomes:
- Apply knowledge of basic Python programming (input, output, variables, loops, and conditionals).
- Work on a project that brings together all the skills learned during the week.
- Practice working on a larger, structured problem that simulates real-world coding projects.

## Mega Tasks and Project:

1. Mega Task 1: Building a Simple Calculator (45 mins):
Create a simple calculator that can add, subtract, multiply, and divide two numbers based on the user's choice.
- The calculator should keep asking for new operations until the user decides to quit by entering 'q'.

```
while True:
        print("\nSimple Calculator")
        print("Enter 'q' to quit.")
        num1 = input('Enter first number: ')
        if num1 == 'q':
            break
        num2 = input('Enter second number: ')
        operation = input('Choose operation (+, -, *, /): ')

        if operation == '+':
            print(f'Result: {float(num1) + float(num2)}')
        elif operation == '-':
            print(f'Result: {float(num1) - float(num2)}')
        elif operation == '*':
            print(f'Result: {float(num1) * float(num2)}')
        elif operation == '/':
            if float(num2) != 0:
                print(f'Result: {float(num1) / float(num2)}')
            else:
                print("Error: Division by zero.")
```

```
        else:
            print("Invalid operation.")
```

2. Mega Task 2: Guess the Number Game (45 mins):
Write a Python program that generates a random number between 1 and 100 and asks the user to guess the number.
- Provide hints to the user if their guess is too low or too high.
- The game should keep asking until the user guesses the number correctly.

```
import random

number = random.randint(1, 100)
print("I have selected a number between 1 and 100. Can you guess it?")
while True:
        guess = int(input("Enter your guess: "))
        if guess < number:
            print("Too low! Try again.")
        elif guess > number:
            print("Too high! Try again.")
        else:
            print(f"Congratulations! You guessed the number: {number}.")
            break
```

4. Final Project: Personal Expense Tracker (1 hour):

In this project, you'll build a personal expense tracker that allows the user to track their daily expenses.
- The program should allow the user to add expenses with descriptions, view total expenses, and view expenses by category (e.g., groceries, rent, etc.).
- The user should also be able to save their expense data to a file for future reference.

Give the concept of list in advance.

```
expenses = []

while True:
        print("\nPersonal Expense Tracker")
        print("1. Add a new expense")
        print("2. View total expenses")
        print("3. View expenses by category")
        print("4. Save expenses to file")
        print("5. Quit")

        choice = input("Choose an option: ")
```

```
        if choice == '1':
            amount = float(input("Enter the expense amount: "))
            category = input("Enter the category (e.g., groceries,
rent): ")
            description = input("Enter a brief description: ")
            expenses.append({"amount": amount, "category": category,
"description": description})
            print("Expense added successfully.")
        elif choice == '2':
            total = sum(expense['amount'] for expense in expenses)
            print(f"Total expenses: ${total:.2f}")
        elif choice == '3':
            category = input("Enter the category to view expenses:
")
            category_expenses = [expense for expense in expenses if
expense['category'] == category]
            if category_expenses:
                for expense in category_expenses:
                    print(f"${expense['amount']:.2f} -
{expense['description']}")
            else:
                print(f"No expenses found for category: {category}")
        elif choice == '4':
            with open("expenses.txt", "w") as f:
                for expense in expenses:

f.write(f"{expense['amount']},{expense['category']},{expense['descri
ption']}\n")
            print("Expenses saved to file.")
        elif choice == '5':
            print("Exiting Expense Tracker.")
            break
        else:
            print("Invalid option. Please choose again.")

expense_tracker()
```

# Lab Manual for Week 2: Control Flow and Looping – Day 1

## Day 1: Loops – Introduction to While Loop

Time: 3 Hours

Learning Outcomes:
- Understanding what loops are and how to use them.
- Learning the syntax and usage of `while` loops in Python.
- Applying control flow statements such as `break` and `continue`.

### Tasks:

1. Introduction to Loops (15 mins):
- What is a loop in programming? Explain with examples how loops allow repeating a block of code multiple times.

```
# Example of a basic while loop
counter = 0
while counter < 5:
    print(f'Counter: {counter}')
    counter += 1
```

2. Creating a Countdown Timer Using `while` (15 mins):
Write a Python program that counts down from a given number to 0 and prints 'Liftoff!' at the end.

```
# Countdown timer
num = 10
while num >= 0:
    print(num)
    num -= 1
print('Liftoff!')
```

3. Using `break` in Loops (15 mins):
Modify the countdown timer to break the loop if the number goes below 5.

```
# Break out of the loop
num = 10
while num >= 0:
    if num < 5:
        break
```

```
    print(num)
    num -= 1
print('Countdown stopped before Liftoff!')
```

4. Using `continue` in Loops (15 mins):
Write a Python program that skips printing the number 5 in a countdown from 10.

```
# Continue to skip a value
num = 10
while num >= 0:
    if num == 5:
        num -= 1
        continue
    print(num)
    num -= 1
```

5. Nested While Loop Example (15 mins):
Create a nested `while` loop to print a simple multiplication table (from 1 to 5).

```
i = 1
while i <= 5:
    j = 1
    while j <= 5:
        print(i * j, end='\t')
        j += 1
    print()
    i += 1
```

6. Writing an Infinite Loop (15 mins):
Write a simple program that creates an infinite loop, and then modify it to break after a certain condition is met.

```
# Infinite loop
counter = 0
while True:
    print(f'Counter: {counter}')
    counter += 1
    if counter > 5:
        break
```

7. Simulating User Input Until Correct Answer is Given (15 mins):
- Create a `while` loop that asks the user to guess a number until they guess it correctly.

```
correct_answer = 7
guess = -1
while guess != correct_answer:
    guess = int(input('Guess the number (1-10): '))
print('You guessed correctly!')
```

8. Using Flags in `while` Loops (15 mins):
Write a Python program that uses a flag variable to terminate the loop.

```
flag = True
counter = 0
while flag:
    print(f'Counter: {counter}')
    counter += 1
    if counter > 5:
        flag = False
```

9. `else` with Loops (15 mins):
Write a Python program that uses the `else` block with a `while` loop to display a message after the loop completes.

```
counter = 0
while counter < 5:
    print(f'Counter: {counter}')
    counter += 1
else:
    print('Loop finished successfully!')
```

10. Combining Control Flow and Loops (15 mins):
Create a number-guessing game where the user has a limited number of attempts to guess a random number.

```
import random
target = random.randint(1, 10)
attempts = 3
while attempts > 0:
    guess = int(input('Guess the number (1-10): '))
    if guess == target:
        print('Correct! You win!')
        break
    else:
        print(f'Wrong! You have {attempts-1} attempts left.')
    attempts -= 1
else:
```

```
    print(f'Sorry, the number was {target}.')
```

## Assignment Task:

- **Building a Countdown Timer with Custom Features** (45 mins):

- Extend the countdown timer by adding these features:

1. Allow the user to input the starting number for the countdown.

2. Allow the user to decide whether to include numbers divisible by 3.

3. Allow the user to terminate the countdown at any point by pressing a specific key (e.g., 'q').

```
# Customized countdown timer
start = int(input('Enter the starting number: '))
skip_multiples_of_three = input('Skip numbers divisible by 3? (y/n):
')

while start >= 0:
    if skip_multiples_of_three == 'y' and start % 3 == 0:
        start -= 1
        continue
    print(start)
    if input("Press 'q' to quit or Enter to continue: ") == 'q':
        break
    start -= 1

print("Countdown ended!")
```

# Lab Manual for Week 2: Control Flow and Looping – Day 2

## Day 2: Loops – Introduction to For Loop

Time: 3 Hours

Learning Outcomes:
- Understand the basics of `for` loops in Python.
- Learn how to use `range()` for looping over numbers.
- Apply `for` loops to iterate over various data structures.

### Tasks:

1. Introduction to `for` Loops (15 mins):
Write a Python program that uses a `for` loop to print numbers from 1 to 10.

```
for i in range(1, 11):
    print(i)
```

2. Using `for` with Strings (15 mins):
Write a Python program that uses a `for` loop to iterate over each character of a string.

```
text = 'Python'
for char in text:
    print(char)
```

3. Using `range()` with Steps (15 mins):
Use the `range()` function to print every second number between 1 and 20.

```
for i in range(1, 21, 2):
    print(i)
```

4. Looping Over a List (15 mins):
Create a list of fruits and use a `for` loop to print each fruit.

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

5. Summing a List of Numbers (15 mins):

Write a Python program that calculates the sum of all numbers in a list using a `for` loop.

```
numbers = [10, 20, 30, 40, 50]
total = 0
for num in numbers:
    total += num
print(f'Total sum: {total}')
```

6. Nested `for` Loops (15 mins):
Use nested `for` loops to print a multiplication table (1 to 5).

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i * j, end='\t')
    print()
```

7. Looping Over a Dictionary (15 mins):
Create a dictionary of employee names and their salaries. Use a `for` loop to print each key-value pair.

```
employees = {'Alice': 50000, 'Bob': 60000, 'Charlie': 55000}
for name, salary in employees.items():
    print(f'{name}: ${salary}')
```

8. Using `for` Loop with `break` (15 mins):
Write a Python program that searches for a specific item in a list and breaks the loop when it is found.

```
numbers = [10, 20, 30, 40, 50]
for num in numbers:
    if num == 30:
        print('Found:', num)
        break
```

9. Using `for` Loop with `continue` (15 mins):
Write a Python program that skips printing the number 3 from a range of 1 to 5.

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

10. Combining `if` and `for` Loops (15 mins):

Create a list of numbers and print only the even numbers using a `for` loop and `if` condition.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    if num % 2 == 0:
        print(f'Even: {num}')
```

11. Looping with `enumerate()` (15 mins):
Use the `enumerate()` function in a `for` loop to print the index and value of each item in a list.

```
colors = ['red', 'green', 'blue']
for index, color in enumerate(colors):
    print(f'{index}: {color}')
```

12. Reverse Looping (15 mins):
Use a `for` loop to print numbers from 10 down to 1.

```
for i in range(10, 0, -1):
    print(i)
```

13. Looping Over Multiple Lists (15 mins):
Use the `zip()` function to loop over two lists and print corresponding items.

```
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 90, 95]
for name, score in zip(names, scores):
    print(f'{name}: {score}')
```

14. Using `else` with `for` Loops (15 mins):
Write a `for` loop that searches for a number in a list, and prints a message if the number is not found.

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 10:
        print('Found 10')
        break
else:
    print('10 not found')
```

15. List Comprehension with `for` Loop (15 mins):

Use list comprehension to create a list of squares of numbers from 1 to 5.

```
squares = [x ** 2 for x in range(1, 6)]
print(squares)
```

## Assignment Task:

- **Creating a Number Pattern Printer** (45 mins):

- Write a Python program that uses `for` loops to print the following pattern:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
# Number pattern printer
n = 5
for i in range(1, n+1):
    for j in range(1, i+1):
        print(j, end=' ')
    print()
```

# Lab Manual for Week 2: Control Flow and Looping – Day 3

## Day 3: Combining Control Flow and Loops

Time: 3 Hours

Learning Outcomes:
- Combine control flow statements with loops.
- Learn to use nested loops and nested conditions.
- Apply loops and control flow to real-world problems like building a number-guessing game.

### Tasks:

1. Using `if` Statements in Loops (15 mins):
Write a Python program that uses a `for` loop to print only even numbers from 1 to 10.

```
for i in range(1, 11):
    if i % 2 == 0:
        print(i)
```

2. Nested `if` and Loops (15 mins):
Write a Python program that prints 'Fizz' for multiples of 3, 'Buzz' for multiples of 5, and 'FizzBuzz' for multiples of both 3 and 5, from 1 to 30.

```
for i in range(1, 31):
    if i % 3 == 0 and i % 5 == 0:
        print('FizzBuzz')
    elif i % 3 == 0:
        print('Fizz')
    elif i % 5 == 0:
        print('Buzz')
    else:
        print(i)
```

3. Nested Loops – Multiplication Table (15 mins):
Use nested loops to generate a multiplication table from 1 to 10.

```
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end='\t')
```

```
    print()
```

4. Using Break in Nested Loops (15 mins):
Write a Python program that exits the inner loop when a certain condition is met, and exits the outer loop if a specific number is found.

```
for i in range(1, 6):
    for j in range(1, 6):
        if j == 3:
            break
        print(i, j)
    if i == 4:
        break
```

5. Using Continue in Nested Loops (15 mins):
Write a Python program that uses the `continue` statement in a nested loop to skip a number in the inner loop while iterating over the outer loop.

```
for i in range(1, 6):
    for j in range(1, 6):
        if j == 3:
            continue
        print(i, j)
```

6. Control Flow and Loops – Guessing Game (15 mins):
Create a number-guessing game where the user has 5 attempts to guess a randomly generated number between 1 and 20.

```
import random
target = random.randint(1, 20)
attempts = 5
while attempts > 0:
    guess = int(input('Guess the number (1-20): '))
    if guess == target:
        print('Correct! You win!')
        break
    elif guess < target:
        print('Too low!')
    else:
        print('Too high!')
    attempts -= 1
else:
    print(f'Sorry, the correct number was {target}.')
```

7. Nested Loops with Conditions (15 mins):
Write a Python program that prints a right-angled triangle pattern using nested loops and conditions.

```
n = 5
for i in range(1, n+1):
    for j in range(1, i+1):
        print('*', end=' ')
    print()
```

8. Control Flow with `while` and `if` (15 mins):
Create a `while` loop that asks the user to guess a secret word, allowing multiple attempts, and exits the loop when the correct word is guessed.

```
secret = 'python'
guess = ''
while guess != secret:
    guess = input('Guess the secret word: ')
print('Correct! You guessed the word.')
```

9. Using `else` with Loops (15 mins):
Write a Python program that uses `for` loop with `else` to search for a specific item in a list, and prints a message if the item is not found.

```
items = ['apple', 'banana', 'cherry']
for item in items:
    if item == 'orange':
        print('Found orange')
        break
else:
    print('Orange not found')
```

10. Simulating Dice Rolls (15 mins):
Simulate rolling a pair of dice 10 times, using nested loops, and print the result of each roll.

```
import random
for i in range(10):
    dice1 = random.randint(1, 6)
    dice2 = random.randint(1, 6)
    print(f'Roll {i+1}: {dice1} and {dice2}')
```

11. Checking for Prime Numbers (15 mins):
Write a Python program that checks if a number between 1 and 20 is prime using a nested loop and `if` condition.

```
for num in range(2, 21):
    is_prime = True
    for i in range(2, num):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(f'{num} is a prime number')
```

12. Factorial Calculation (15 mins):
Write a Python program that calculates the factorial of a given number using a `for` loop.

```
num = 5
factorial = 1
for i in range(1, num + 1):
    factorial *= i
print(f'The factorial of {num} is {factorial}')
```

13. Control Flow with Lists (15 mins):
Create a list of integers and use a `for` loop to find and print the maximum value in the list.

```
numbers = [23, 45, 12, 67, 34]
max_num = numbers[0]
for num in numbers:
    if num > max_num:
        max_num = num
print(f'The maximum number is {max_num}')
```

14. Combining `if` and Loop for Number Patterns (15 mins):
Write a Python program that uses a nested loop and conditions to print a pattern of numbers where each row has the same number repeated.

```
n = 5
for i in range(1, n+1):
    for j in range(1, i+1):
        print(i, end=' ')
    print()
```

15. Reversing a List with a Loop (15 mins):
Write a Python program that reverses a list of numbers using a loop.

```
numbers = [10, 20, 30, 40, 50]
reversed_numbers = []
for num in numbers[::-1]:
```

```
    reversed_numbers.append(num)
print(reversed_numbers)
```

## Assignment Task:

- **Number Guessing Game with Score Tracking** (45 mins):

- Write a Python program that builds upon the number-guessing game by adding score tracking. The program should allow the user to play multiple rounds, keep track of the number of attempts made, and print the score at the end.

```
import random

def play_game():
    target = random.randint(1, 20)
    attempts = 5
    score = 0
    while attempts > 0:
        guess = int(input('Guess the number (1-20): '))
        if guess == target:
            print('Correct! You win!')
            score += 10
            break
        elif guess < target:
            print('Too low!')
        else:
            print('Too high!')
        attempts -= 1
    else:
        print(f'Sorry, the correct number was {target}.')
    return score

total_score = 0
rounds = 3
for _ in range(rounds):
    total_score += play_game()

print(f'Total score after {rounds} rounds: {total_score}')
```

# Lab Manual for Week 2: Control Flow and Looping – Day 4

## Day 4: Lists – Introduction and Basic Operations

Time: 3 Hours

Learning Outcomes:
- Understand how to use lists in Python.
- Learn basic list operations such as adding and removing items.
- Apply loops to iterate over lists and perform common tasks.

### Tasks:

1. Introduction to Lists (15 mins):

Create a list of 5 numbers and print each number using a `for` loop.

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

2. Adding Items to a List (15 mins):

Create an empty list and use a loop to append numbers 1 through 5 to the list.

```
numbers = []
for i in range(1, 6):
    numbers.append(i)
print(numbers)
```

3. Removing Items from a List (15 mins):

Create a list of 5 fruits, and then remove 'apple' from the list.

```
fruits = ['apple', 'banana', 'cherry', 'mango', 'orange']
fruits.remove('apple')
print(fruits)
```

4. List Indexing (15 mins):

Create a list of 5 items and print the first and last items using list indexing.

```
items = ['pen', 'book', 'laptop', 'phone', 'tablet']
print(f'First item: {items[0]}')
print(f'Last item: {items[-1]}')
```

5. Slicing a List (15 mins):

Write a Python program that creates a list of numbers 1 to 10 and prints the first 5 numbers using list slicing.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[:5])
```

6. Iterating Over a List with `for` Loop (15 mins):

Create a list of student names and print each name using a `for` loop.

```
students = ['Alice', 'Bob', 'Charlie', 'David']
for student in students:
    print(student)
```

7. Checking if an Item is in a List (15 mins):

Write a Python program that checks if 'apple' is in the list of fruits.

```
fruits = ['banana', 'cherry', 'orange', 'mango']
if 'apple' in fruits:
    print('Apple is in the list.')
else:
    print('Apple is not in the list.')
```

8. Sorting a List (15 mins):

Create a list of numbers and sort them in ascending order.

```
numbers = [5, 2, 9, 1, 7]
numbers.sort()
print(f'Sorted numbers: {numbers}')
```

9. Reversing a List (15 mins):

Create a list of numbers and reverse the order of the list.

```
numbers = [1, 2, 3, 4, 5]
numbers.reverse()
print(f'Reversed list: {numbers}')
```

10. List Comprehension (15 mins):

Use list comprehension to create a list of squares of numbers from 1 to 5.

```
squares = [x ** 2 for x in range(1, 6)]
print(squares)
```

11. Concatenating Lists (15 mins):

Create two lists of fruits and concatenate them into one list.

```
fruits1 = ['apple', 'banana']
fruits2 = ['cherry', 'mango']
all_fruits = fruits1 + fruits2
print(all_fruits)
```

12. Finding the Length of a List (15 mins):

Create a list of 5 items and print the length of the list.

```
items = ['pen', 'notebook', 'ruler', 'eraser', 'marker']
print(f'The length of the list is {len(items)}')
```

13. Using `max()` and `min()` with Lists (15 mins):

Create a list of numbers and use `max()` and `min()` functions to find the largest and smallest number.

```
numbers = [10, 3, 5, 8, 2]
print(f'Largest number: {max(numbers)}')
print(f'Smallest number: {min(numbers)}')
```

14. Duplicating Items in a List (15 mins):

Create a list with repeated items and count how many times a specific item appears in the list.

```
numbers = [2, 3, 2, 5, 2, 7]
print(f'The number 2 appears {numbers.count(2)} times in the list.')
```

15. Nested Lists (15 mins):

Create a nested list (a list of lists) and use a loop to print each sublist.

```
nested_list = [[1, 2], [3, 4], [5, 6]]
for sublist in nested_list:
    print(sublist)
```

## Assignment Task:

- **To-Do List Program with Multiple Tasks** (45 mins):

- Write a Python program that allows the user to create a to-do list with multiple tasks. The program should allow the user to:
  1. Add tasks to the list.
  2. View all tasks.
  3. Mark tasks as completed.
  4. Remove completed tasks from the list.

```
# To-Do List Program
todo_list = []
```

```
while True:
    print('\nOptions:')
    print('1. Add task')
    print('2. View tasks')
    print('3. Mark task as completed')
    print('4. Remove completed tasks')
    print('5. Exit')

    choice = input('Choose an option: ')

    if choice == '1':
        task = input('Enter a new task: ')
        todo_list.append({'task': task, 'completed': False})
    elif choice == '2':
        for i, task in enumerate(todo_list):
            status = 'Done' if task['completed'] else 'Not Done'
            print(f'{i + 1}. {task["task"]} - {status}')
    elif choice == '3':
        task_number = int(input('Enter task number to mark as
completed: ')) - 1
        todo_list[task_number]['completed'] = True
    elif choice == '4':
        todo_list = [task for task in todo_list if not
task['completed']]
    elif choice == '5':
        break
```

# Lab Manual for Week 2: Mega Tasks and Project Work – Day 5

## Day 5: Mega Tasks and Project Incremental Work

Time: 3 Hours

Learning Outcomes:
- Solve advanced tasks combining knowledge of loops, lists, and control flow.
- Extend and apply knowledge incrementally to build a functional project.

### Mega Tasks:

1. Building a Multi-List To-Do Application (30 mins):

Write a Python program that allows the user to create multiple to-do lists (e.g., work tasks, personal tasks) and manage tasks for each list.

```
# Multi-List To-Do Application
todo_lists = {}

while True:
    print("\nOptions:")
    print("1. Create a new to-do list")
    print("2. Add task to a list")
    print("3. View tasks in a list")
    print("4. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        list_name = input("Enter the name of the new to-do list: ")
        todo_lists[list_name] = []
    elif choice == "2":
        list_name = input("Enter the list name: ")
        if list_name in todo_lists:
            task = input("Enter a new task: ")
            todo_lists[list_name].append(task)
        else:
            print(f"No list named {list_name}")
    elif choice == "3":
        list_name = input("Enter the list name: ")
        if list_name in todo_lists:
            print(f"Tasks in {list_name}: {todo_lists[list_name]}")
        else:
            print(f"No list named {list_name}")
    elif choice == "4":
```

```
        break
```

2. Prime Number Checker with Range (30 mins):

Write a Python program that allows the user to input a range of numbers and checks which numbers are prime within that range.

```
# Prime Number Checker
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, num):
        if num % i == 0:
            return False
    return True

start = int(input("Enter the start of the range: "))
end = int(input("Enter the end of the range: "))

for num in range(start, end+1):
    if is_prime(num):
        print(f"{num} is a prime number")
```

3. Multi-Dimensional List Operations (30 mins):

Write a Python program that creates a 2D list (a list of lists) and performs the following operations:

Print the 2D list.
Find the sum of elements in each row.
Find the sum of elements in each column.

```
# 2D List Operations
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Printing 2D List
print("Matrix:")
for row in matrix:
    print(row)

# Row sums
for row in matrix:
    print(f"Row sum: {sum(row)}")

# Column sums
for col in range(len(matrix[0])):
    col_sum = 0
```

```
    for row in matrix:
        col_sum += row[col]
    print(f"Column {col + 1} sum: {col_sum}")
```

4. Enhanced Guessing Game with Score System (30 mins):

Extend the number-guessing game by adding a score system that tracks how many tries the player needed and calculates a score based on their performance.

```
import random

def play_game():
    target = random.randint(1, 20)
    attempts = 5
    score = 0
    while attempts > 0:
        guess = int(input("Guess the number (1-20): "))
        if guess == target:
            print("Correct! You win!")
            score += 10 * attempts
            break
        elif guess < target:
            print("Too low!")
        else:
            print("Too high!")
        attempts -= 1
    else:
        print(f"Sorry, the correct number was {target}.")
    return score

total_score = 0
rounds = 3
for _ in range(rounds):
    total_score += play_game()

print(f"Total score after {rounds} rounds: {total_score}")
```

## Project Incremental Work:

Continue building the to-do list project by adding the following features:
- Allow the user to mark tasks as completed and view only completed or pending tasks.
- Add a feature to delete a to-do list entirely.

```
# To-Do List Project with Completed Tasks and List Deletion
todo_lists = {}

while True:
    print("\nOptions:")
    print("1. Create a new to-do list")
    print("2. Add task to a list")
    print("3. View tasks in a list")
```

```
    print("4. Mark task as completed")
    print("5. View completed/pending tasks")
    print("6. Delete a list")
    print("7. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        list_name = input("Enter the name of the new to-do list: ")
        todo_lists[list_name] = []
    elif choice == "2":
        list_name = input("Enter the list name: ")
        if list_name in todo_lists:
            task = input("Enter a new task: ")
            todo_lists[list_name].append({'task': task, 'completed':
False})
        else:
            print(f"No list named {list_name}")
    elif choice == "3":
        list_name = input("Enter the list name: ")
        if list_name in todo_lists:
            for task in todo_lists[list_name]:
                status = "Done" if task["completed"] else "Not Done"
                print(f"{task['task']} - {status}")
        else:
            print(f"No list named {list_name}")
    elif choice == "4":
        list_name = input("Enter the list name: ")
        if list_name in todo_lists:
            task_index = int(input("Enter the task number to mark as
completed: ")) - 1
            todo_lists[list_name][task_index]["completed"] = True
    elif choice == "5":
        list_name = input("Enter the list name: ")
        view_option = input("View completed or pending tasks (c/p)?
")
        if list_name in todo_lists:
            for task in todo_lists[list_name]:
                if view_option == "c" and task["completed"]:
                    print(task["task"])
                elif view_option == "p" and not task["completed"]:
                    print(task["task"])
    elif choice == "6":
        list_name = input("Enter the list name to delete: ")
        if list_name in todo_lists:
            del todo_lists[list_name]
            print(f"Deleted list {list_name}")
    elif choice == "7":
        break
```

# Lab Manual for Week 3: Functions and More Data Structures – Day 1

## Day 1: Introduction to Functions

Time: 3 Hours

Learning Outcomes:
- Understand what functions are and why they are useful in programming.
- Learn to define and call functions in Python.
- Explore function parameters and return values.

### Tasks:

1. Defining a Simple Function (15 mins):

Write a Python function `greet()` that prints 'Hello, World!'.

```
def greet():
    print('Hello, World!')
greet()
```

2. Function with Parameters (15 mins):

Write a function `add_numbers(a, b)` that takes two numbers and prints their sum.

```
def add_numbers(a, b):
    print(a + b)
add_numbers(3, 5)
```

3. Function with Return Value (15 mins):

Modify `add_numbers()` to return the sum instead of printing it.

```
def add_numbers(a, b):
    return a + b
result = add_numbers(3, 5)
print(result)
```

4. Function with Multiple Parameters (15 mins):

Write a function `multiply(a, b, c)` that multiplies three numbers and returns the result.

```
def multiply(a, b, c):
    return a * b * c
result = multiply(2, 3, 4)
print(result)
```

5. Function with Default Argument (15 mins):

Write a function `greet_user(name, greeting='Hello')` that prints a greeting with the user's name. The greeting should default to 'Hello' if no other greeting is provided.

```
def greet_user(name, greeting='Hello'):
    print(f'{greeting}, {name}!')
greet_user('Alice')
greet_user('Bob', 'Hi')
```

6. Function with Keyword Arguments (15 mins):

Write a function `describe_pet(animal, name)` that accepts keyword arguments for an animal type and its name.

```
def describe_pet(animal='dog', name='Buddy'):
    print(f'I have a {animal} named {name}.')
describe_pet(animal='cat', name='Whiskers')
```

7. Function Returning Multiple Values (15 mins):

Write a function `math_operations(a, b)` that returns the sum, difference, and product of two numbers.

```
def math_operations(a, b):
    return a + b, a - b, a * b
sum_, diff, prod = math_operations(10, 5)
print(f'Sum: {sum_}, Difference: {diff}, Product: {prod}')
```

8. Function with Variable Number of Arguments (15 mins):

Write a function `sum_all(*numbers)` that sums an arbitrary number of arguments.

```
def sum_all(*numbers):
    return sum(numbers)
print(sum_all(1, 2, 3, 4))
```

9. Nested Functions (15 mins):

Write a function `outer_function()` that contains a nested function `inner_function()`. The inner function should print a message.

```
def outer_function():
    def inner_function():
        print('This is the inner function.')
    inner_function()
outer_function()
```

10. Simple Calculator Function (15 mins):

Write a function `calculator(a, b, operator)` that performs addition, subtraction, multiplication, or division based on the given operator.

```
def calculator(a, b, operator):
    if operator == '+':
        return a + b
    elif operator == '-':
```

```
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        return a / b
    else:
        return 'Invalid operator'
print(calculator(10, 2, '*'))
```

11. Function with a List Argument (15 mins):

Write a function `print_items(items)` that takes a list and prints each item.

```
def print_items(items):
    for item in items:
        print(item)
fruits = ['apple', 'banana', 'cherry']
print_items(fruits)
```

12. Function to Find Maximum (15 mins):

Write a function `find_max(a, b, c)` that returns the largest of three numbers.

```
def find_max(a, b, c):
    return max(a, b, c)
print(find_max(10, 15, 5))
```

13. Recursive Function (15 mins):

Write a recursive function `factorial(n)` that returns the factorial of a number.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

14. Function with Lambda Expression (15 mins):

Write a lambda function that adds two numbers and assign it to a variable `add_lambda`.

```
add_lambda = lambda x, y: x + y
print(add_lambda(3, 4))
```

15. Function to Check Even or Odd (15 mins):

Write a function `is_even(n)` that returns `True` if the number is even and `False` otherwise.

```
def is_even(n):
    return n % 2 == 0
print(is_even(4))
print(is_even(5))
```

## Assignment Task:

- **Simple Banking System** (45 mins):

Write a Python function `bank_system()` that simulates a simple banking system. The user can perform the following actions:

1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Exit the program

The balance should start at $0 and update after each transaction.

```
def bank_system():
    balance = 0
    while True:
        print("\nOptions:")
        print("1. Check Balance")
        print("2. Deposit Money")
        print("3. Withdraw Money")
        print("4. Exit")

        choice = input("Choose an option: ")

        if choice == "1":
            print(f"Your balance is: ${balance}")
        elif choice == "2":
            amount = float(input("Enter amount to deposit: "))
            balance += amount
            print(f"${amount} deposited. New balance: ${balance}")
        elif choice == "3":
            amount = float(input("Enter amount to withdraw: "))
            if amount > balance:
                print("Insufficient funds!")
            else:
                balance -= amount
                print(f"${amount} withdrawn. New balance:
${balance}")
        elif choice == "4":
            break
        else:
            print("Invalid choice")

bank_system()
```

# Lab Manual for Week 3: Functions and More Data Structures – Day 2

## Day 2: Lists – Advanced Operations

Time: 3 Hours

Learning Outcomes:
- Learn advanced list operations in Python.
- Access list elements using indexing and slicing.
- Perform common operations such as sorting, reversing, and finding the length of a list.
- Practice using lists in practical scenarios.

### Tasks:

1. Accessing Elements by Index (15 mins):

Create a list of numbers and print the element at index 3.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[3])
```

2. Negative Indexing (15 mins):

Print the last element of a list using negative indexing.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[-1])
```

3. Slicing a List (15 mins):

Write a program to print the first three elements of a list using slicing.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[:3])
```

4. Slicing a List from an Index (15 mins):

Write a program to print all elements from index 2 to the end of the list.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[2:])
```

5. Slicing a List with Step (15 mins):

Write a program to print every second element from the list.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[::2])
```
6. Sorting a List (15 mins):

Write a Python program that sorts a list of numbers in ascending order.

```
numbers = [50, 20, 40, 10, 30]
numbers.sort()
print(numbers)
```
7. Reversing a List (15 mins):

Write a Python program that reverses a list of numbers.

```
numbers = [10, 20, 30, 40, 50]
numbers.reverse()
print(numbers)
```
8. Finding the Length of a List (15 mins):

Write a program that prints the length of a list.

```
numbers = [10, 20, 30, 40, 50]
print(len(numbers))
```
9. Adding Elements to a List (15 mins):

Write a program to add a new element at the end of a list using `append()`.

```
fruits = ['apple', 'banana', 'cherry']
fruits.append('orange')
print(fruits)
```
10. Removing Elements from a List (15 mins):

Write a Python program to remove an element from a list using `remove()`.

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove('banana')
print(fruits)
```
11. Inserting an Element at a Specific Position (15 mins):

Write a program to insert an element at index 2 in a list.

```
numbers = [10, 20, 30, 40]
numbers.insert(2, 25)
print(numbers)
```
12. Clearing a List (15 mins):

Write a Python program to remove all elements from a list using `clear()`.

```
numbers = [10, 20, 30, 40, 50]
numbers.clear()
print(numbers)
```
13. Checking if an Item Exists in a List (15 mins):

Write a program to check if 'banana' is in the list of fruits.

```
fruits = ['apple', 'banana', 'cherry']
if 'banana' in fruits:
    print('Banana is in the list')
```

14. List Comprehension (15 mins):

Write a Python program to create a list of squares of numbers from 1 to 5 using list comprehension.

```
squares = [x ** 2 for x in range(1, 6)]
print(squares)
```

15. Nested Lists (15 mins):

Create a nested list and print the elements of each sublist using a loop.

```
nested_list = [[1, 2], [3, 4], [5, 6]]
for sublist in nested_list:
    print(sublist)
```

## Assignment Task:

- **Shopping List Application** (45 mins):

Write a Python program to create a shopping list. The program should allow the user to:

1. Add items to the list.
2. Remove items from the list.
3. View all items in the list.

```
# Shopping List Application
shopping_list = []

while True:
    print("\nOptions:")
    print("1. Add item")
    print("2. Remove item")
    print("3. View shopping list")
    print("4. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        item = input("Enter the item to add: ")
        shopping_list.append(item)
        print(f"{item} added to the list.")
    elif choice == "2":
        item = input("Enter the item to remove: ")
        if item in shopping_list:
            shopping_list.remove(item)
            print(f"{item} removed from the list.")
        else:
            print("Item not found in the list.")
    elif choice == "3":
        print("Shopping List:", shopping_list)
    elif choice == "4":
```

```
        break
    else:
        print("Invalid choice")
```

# Lab Manual for Week 3: Functions and More Data Structures – Day 3

## Day 3: Tuples and Dictionaries

Time: 3 Hours

Learning Outcomes:
- Understand how to use tuples and dictionaries in Python.
- Learn basic tuple and dictionary operations.
- Explore examples such as creating a contact list using dictionaries.

### Tasks:

1. Creating a Tuple (15 mins):

Write a Python program to create a tuple with different data types (e.g., integers, strings).

```
my_tuple = (1, 'apple', 3.5)
print(my_tuple)
```

2. Accessing Tuple Elements (15 mins):

Write a Python program to access elements of a tuple using indexing.

```
my_tuple = (10, 20, 30, 40)
print(my_tuple[2])
```

3. Unpacking a Tuple (15 mins):

Write a Python program to unpack a tuple into separate variables.

```
my_tuple = ('apple', 'banana', 'cherry')
fruit1, fruit2, fruit3 = my_tuple
print(fruit1, fruit2, fruit3)
```

4. Tuples are Immutable (15 mins):

Try modifying an element in a tuple and observe the result.

```
my_tuple = (10, 20, 30)
# This will raise an error
# my_tuple[0] = 100
```

5. Using Tuples in Functions (15 mins):

Write a function `calculate_stats(numbers)` that takes a tuple of numbers and returns the sum and average.

```
def calculate_stats(numbers):
    total = sum(numbers)
    avg = total / len(numbers)
    return total, avg

result = calculate_stats((10, 20, 30))
print(result)
```

6. Creating a Dictionary (15 mins):

Write a Python program to create a dictionary representing a person with keys `name`, `age`, and `city`.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
print(person)
```

7. Accessing Dictionary Values (15 mins):

Write a Python program to access the value associated with the `name` key in a dictionary.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
print(person['name'])
```

8. Adding Key-Value Pairs to a Dictionary (15 mins):

Write a Python program to add a new key-value pair (`email`) to a dictionary.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
person['email'] = 'alice@example.com'
print(person)
```

9. Removing Key-Value Pairs from a Dictionary (15 mins):

Write a Python program to remove the `city` key from a dictionary using `del`.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
del person['city']
print(person)
```

10. Looping Through a Dictionary (15 mins):

Write a Python program to loop through a dictionary and print each key and value.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
for key, value in person.items():
    print(f'{key}: {value}')
```

11. Dictionary with Multiple Values (15 mins):

Write a Python program to create a dictionary with multiple values for each key (e.g., a list of favorite colors for each person).

```
people = {'Alice': ['red', 'blue'], 'Bob': ['green', 'yellow']}
print(people)
```

12. Checking for a Key in a Dictionary (15 mins):

Write a Python program to check if the key `age` exists in a dictionary.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
if 'age' in person:
    print('Age is in the dictionary')
```

13. Merging Two Dictionaries (15 mins):

Write a Python program to merge two dictionaries.

```
dict1 = {'name': 'Alice', 'age': 30}
dict2 = {'city': 'New York', 'email': 'alice@example.com'}
dict1.update(dict2)
print(dict1)
```

14. Using Dictionaries with Functions (15 mins):

Write a function `describe_person(person)` that takes a dictionary representing a person and prints the information.

```
def describe_person(person):
    for key, value in person.items():
        print(f'{key}: {value}')
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
describe_person(person)
```

15. Nested Dictionaries (15 mins):

Write a Python program to create a dictionary of dictionaries to represent multiple people with their information.

```
people = {
    'Alice': {'age': 30, 'city': 'New York'},
    'Bob': {'age': 25, 'city': 'Chicago'}
}
print(people)
```

## Assignment Task:

- **Contact List Application** (45 mins):

Write a Python program to create a contact list using a dictionary. The program should allow the user to:

1. Add a contact (name, phone number).
2. View all contacts.
3. Search for a contact by name.

```
# Contact List Application
contacts = {}

while True:
    print("\nOptions:")
    print("1. Add contact")
    print("2. View all contacts")
    print("3. Search for a contact by name")
    print("4. Exit")
```

```
    choice = input("Choose an option: ")

    if choice == "1":
        name = input("Enter contact name: ")
        phone = input("Enter phone number: ")
        contacts[name] = phone
        print(f"{name} added to contacts.")
    elif choice == "2":
        for name, phone in contacts.items():
            print(f"{name}: {phone}")
    elif choice == "3":
        name = input("Enter name to search: ")
        if name in contacts:
            print(f"{name}: {contacts[name]}")
        else:
            print("Contact not found.")
    elif choice == "4":
        break
    else:
        print("Invalid choice")
```

# Lab Manual for Week 3: Functions and More Data Structures – Day 4

## Day 4: Error Handling and Debugging

Time: 3 Hours

Learning Outcomes:
- Understand different types of errors in Python.
- Learn to use `try` and `except` blocks for error handling.
- Learn basic debugging techniques and how to handle user input errors.

### Tasks:

1. Simple `try`-`except` Block (15 mins):

Write a Python program to handle division by zero using `try` and `except`.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print('Division by zero is not allowed.')
```

2. Handling Multiple Exceptions (15 mins):

Write a program to handle division by zero and invalid input using `try`-`except`.

```
try:
    num = int(input('Enter a number: '))
    result = 10 / num
except ZeroDivisionError:
    print('Cannot divide by zero.')
except ValueError:
    print('Invalid input. Please enter a number.')
```

3. Using `else` with `try` Block (15 mins):

Write a program that performs division only if no exceptions occur.

```
try:
    num = int(input('Enter a number: '))
    result = 10 / num
except ZeroDivisionError:
    print('Cannot divide by zero.')
else:
    print(f'Result: {result}')
```

4. Using `finally` Block (15 mins):

Write a program with a `finally` block to ensure a message is printed whether an error occurs or not.

```
try:
    num = int(input('Enter a number: '))
    result = 10 / num
except ZeroDivisionError:
    print('Cannot divide by zero.')
finally:
    print('End of program.')
```

5. Raising Exceptions (15 mins):

Write a Python program that raises a custom exception when a specific condition is met.

```
def check_age(age):
    if age < 18:
        raise ValueError('Age must be at least 18.')
    else:
        print('Access granted.')

try:
    check_age(15)
except ValueError as e:
    print(e)
```

6. Debugging with Print Statements (15 mins):

Write a Python program and insert print statements at different points to trace its execution.

```
def multiply(a, b):
    print(f'a = {a}, b = {b}')
    result = a * b
    print(f'Result = {result}')
    return result
multiply(5, 10)
```

7. Handling File Not Found Error (15 mins):

Write a Python program that handles the `FileNotFoundError` when trying to open a non-existent file.

```
try:
    file = open('non_existent_file.txt')
except FileNotFoundError:
    print('File not found.')
```

8. Handling Invalid Input in a Loop (15 mins):

Write a program that continuously asks for a number until the user provides valid input.

```
while True:
    try:
```

```
        num = int(input('Enter a number: '))
        break
    except ValueError:
        print('Invalid input. Please enter a number.')
```

9. Catching Multiple Exception Types (15 mins):

Write a program that catches both `ValueError` and `ZeroDivisionError`.

```
try:
    num = int(input('Enter a number: '))
    result = 10 / num
except (ValueError, ZeroDivisionError):
    print('Error: invalid input or division by zero.')
```

10. Using Assertions (15 mins):

Write a Python program that uses assertions to check if a number is positive.

```
def check_positive(num):
    assert num > 0, 'Number must be positive.'
    print(f'{num} is positive.')

check_positive(10)
# check_positive(-5)  # This will raise an AssertionError
```

11. Custom Exception Handling (15 mins):

Create a custom exception `NegativeNumberError` and raise it if a negative number is entered.

```
class NegativeNumberError(Exception):
    pass

try:
    num = int(input('Enter a positive number: '))
    if num < 0:
        raise NegativeNumberError('Negative number entered.')
except NegativeNumberError as e:
    print(e)
```

12. Using `try`-`except` for Dictionary Lookup (15 mins):

Write a program that handles key errors when looking up a key in a dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
try:
    print(my_dict['city'])
except KeyError:
    print('Key not found in the dictionary.')
```

13. Debugging with the `pdb` Module (15 mins):

Use Python's built-in debugger (`pdb`) to debug a simple program.

```
import pdb

def add(a, b):
```

```
    pdb.set_trace()
    return a + b

result = add(10, 20)
print(result)
```
14. Catching `IndexError` (15 mins):

Write a Python program that handles an `IndexError` when accessing an invalid index in a list.

```
my_list = [1, 2, 3]
try:
    print(my_list[5])
except IndexError:
    print('Index out of range.')
```
15. Logging Errors (15 mins):

Write a Python program that logs error messages to a file using the `logging` module.

```
import logging

logging.basicConfig(filename='error.log', level=logging.ERROR)
try:
    result = 10 / 0
except ZeroDivisionError as e:
    logging.error('Division by zero occurred.')
```

## Assignment Task:

- **Handling User Input Errors Gracefully** (45 mins):

Write a Python program that continuously asks the user for two numbers and an operator (`+`, `-`, `*`, `/`). The program should handle the following errors:
1. Invalid number input (e.g., inputting a string instead of a number).
2. Division by zero.
3. Invalid operator (i.e., operator not in `+`, `-`, `*`, `/`).

The program should display the result of the operation if valid inputs are provided.

```
def calculator():
    while True:
        try:
            num1 = float(input('Enter the first number: '))
            num2 = float(input('Enter the second number: '))
            operator = input('Enter the operator (+, -, *, /): ')

            if operator == '+':
                result = num1 + num2
            elif operator == '-':
                result = num1 - num2
            elif operator == '*':
                result = num1 * num2
            elif operator == '/':
                result = num1 / num2
```

```
            else:
                raise ValueError('Invalid operator')

            print(f'Result: {result}')
            break

        except ValueError as e:
            print(f'Error: {e}. Please try again.')
        except ZeroDivisionError:
            print('Error: Cannot divide by zero.')

calculator()
```

# Lab Manual for Week 3: Mega Tasks and Project Work – Day 5

## Day 5: Mega Tasks and Project Incremental Work

Time: 3 Hours

Learning Outcomes:
- Solve advanced tasks combining knowledge of functions, lists, tuples, and dictionaries.
- Extend and apply knowledge incrementally to build a functional project.

### Mega Tasks:

1. Contact Management System (45 mins):

Write a Python program that uses functions to add, delete, and search for contacts in a dictionary. Each contact should have a name, phone number, and email address.

```
# Contact Management System
contacts = {}

def add_contact(name, phone, email):
    contacts[name] = {'phone': phone, 'email': email}

def delete_contact(name):
    if name in contacts:
        del contacts[name]
        print(f'{name} has been deleted.')
    else:
        print(f'{name} not found.')

def search_contact(name):
    if name in contacts:
        print(f"Name: {name}, Phone: {contacts[name]['phone']},
Email: {contacts[name]['email']}")
    else:
        print(f'{name} not found.')

# Example usage
add_contact('Alice', '123-456', 'alice@example.com')
add_contact('Bob', '789-012', 'bob@example.com')
search_contact('Alice')
```

```
delete_contact('Bob')
```

2. Error Handling with User Input (30 mins):

Write a program that asks the user to enter two numbers and performs division. The program should handle invalid input and division by zero using try-except blocks.

```
def divide_numbers():
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        result = num1 / num2
        print(f"Result: {result}")
    except ValueError:
        print("Invalid input. Please enter numbers.")
    except ZeroDivisionError:
        print("Cannot divide by zero.")

divide_numbers()
```

3. Tuple Operations (30 mins):

Write a function `tuple_operations()` that takes a tuple of numbers, finds the maximum, minimum, and sum, and returns these values.

```
def tuple_operations(numbers):
    max_num = max(numbers)
    min_num = min(numbers)
    total = sum(numbers)
    return max_num, min_num, total

numbers = (10, 20, 30, 40)
max_num, min_num, total = tuple_operations(numbers)
print(f"Max: {max_num}, Min: {min_num}, Sum: {total}")
```

4. Advanced List Operations (30 mins):

Write a function `list_operations()` that takes a list of numbers, sorts the list, reverses it, and returns the modified list.

```
def list_operations(numbers):
    numbers.sort()
    numbers.reverse()
    return numbers

numbers = [50, 20, 40, 10, 30]
modified_list = list_operations(numbers)
print(modified_list)
```

## Project Incremental Work:

Continue refining the contact management system by adding the following features:
- Add error handling for duplicate contact names.
- Allow the user to update an existing contact's phone number and email address.

```
# Contact Management System with Updates and Error Handling
contacts = {}

def add_contact(name, phone, email):
    if name in contacts:
        print(f"{name} already exists. Please use a different
name.")
    else:
        contacts[name] = {'phone': phone, 'email': email}

def update_contact(name, phone, email):
    if name in contacts:
        contacts[name]['phone'] = phone
        contacts[name]['email'] = email
        print(f"{name}'s contact updated.")
    else:
        print(f"{name} not found.")

def delete_contact(name):
    if name in contacts:
        del contacts[name]
        print(f'{name} has been deleted.')
    else:
        print(f'{name} not found.')

def search_contact(name):
    if name in contacts:
        print(f"Name: {name}, Phone: {contacts[name]['phone']},
Email: {contacts[name]['email']}")
    else:
        print(f'{name} not found.')

# Example usage
add_contact('Alice', '123-456', 'alice@example.com')
add_contact('Bob', '789-012', 'bob@example.com')
update_contact('Alice', '555-999', 'alice@newemail.com')
search_contact('Alice')
delete_contact('Bob')
```

# Lab Manual for Week 4: File Handling and Basic Project Structuring – Day 1

## Day 1: Introduction to File Handling

Time: 3 Hours

Learning Outcomes:
- Understand why storing data in files is essential.
- Learn how to open, read, write, and close files in Python.
- Practice file handling through practical exercises.

### Tasks:

1. Creating and Writing to a File (15 mins):

Write a Python program that creates a text file and writes a simple message into it.

```
file = open('example.txt', 'w')
file.write('Hello, this is a file handling example.')
file.close()
```

2. Reading from a File (15 mins):

Write a program that reads the contents of a text file and prints them.

```
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

3. Writing Multiple Lines to a File (15 mins):

Write a program that writes multiple lines to a file using `writelines()`.

```
lines = ['Line 1\n', 'Line 2\n', 'Line 3\n']
file = open('example.txt', 'w')
file.writelines(lines)
file.close()
```

4. Reading File Line by Line (15 mins):

Write a program that reads a file line by line and prints each line.

```
file = open('example.txt', 'r')
for line in file:
```

```
    print(line.strip())
file.close()
```
5. Appending to a File (15 mins):

Write a Python program that appends data to an existing file.

```
file = open('example.txt', 'a')
file.write('This is an appended line.\n')
file.close()
```
6. Using `with` Statement (15 mins):

Write a program to open a file using the `with` statement to ensure the file is properly closed.

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```
7. Writing Numbers to a File (15 mins):

Write a program that writes numbers 1 to 5 into a file, each on a new line.

```
with open('numbers.txt', 'w') as file:
    for num in range(1, 6):
        file.write(f'{num}\n')
```
8. Reading Numbers from a File (15 mins):

Write a program that reads the numbers from `numbers.txt` and prints them.

```
with open('numbers.txt', 'r') as file:
    for line in file:
        print(line.strip())
```
9. Checking if a File Exists (15 mins):

Write a program that checks if a file exists before trying to open it.

```
import os
if os.path.exists('example.txt'):
    with open('example.txt', 'r') as file:
        print(file.read())
else:
    print('File does not exist.')
```
10. Handling FileNotFoundError (15 mins):

Write a program that handles the `FileNotFoundError` when trying to open a non-existent file.

```
try:
    with open('non_existent.txt', 'r') as file:
        print(file.read())
except FileNotFoundError:
    print('File not found.')
```
11. Writing and Reading Lists to a File (15 mins):

Write a program that writes a list of strings to a file and then reads them back.

```
items = ['apple', 'banana', 'cherry']
with open('items.txt', 'w') as file:
    for item in items:
        file.write(f'{item}\n')

with open('items.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

12. Counting Words in a File (15 mins):

Write a program that reads a text file and counts the number of words in it.

```
with open('example.txt', 'r') as file:
    content = file.read()
    words = content.split()
    print(f'Number of words: {len(words)}')
```

13. Copying the Contents of a File (15 mins):

Write a program that copies the contents of one file to another.

```
with open('example.txt', 'r') as file1:
    content = file1.read()
with open('copy.txt', 'w') as file2:
    file2.write(content)
```

14. Writing User Input to a File (15 mins):

Write a program that takes user input and writes it to a file.

```
user_input = input('Enter some text: ')
with open('user_input.txt', 'w') as file:
    file.write(user_input)
```

15. File Seek and Tell (15 mins):

Write a program that demonstrates the use of `seek()` and `tell()` methods while working with a file.

```
with open('example.txt', 'r') as file:
    file.seek(10)
    print(f'Current position: {file.tell()}')
    print(file.read())
```

## Assignment Task:

- **Simple Text Editor** (45 mins):

Write a Python program that simulates a simple text editor with the following functionalities:

1. Create a new text file.
2. Append text to an existing file.
3. Display the contents of a file.
4. Exit the program.

```
def text_editor():
    while True:
        print("\nOptions:")
        print("1. Create a new file")
        print("2. Append to an existing file")
        print("3. Display file contents")
        print("4. Exit")

        choice = input("Choose an option: ")

        if choice == "1":
            filename = input("Enter the filename: ")
            content = input("Enter text to write: ")
            with open(filename, 'w') as file:
                file.write(content)
            print(f"File '{filename}' created.")

        elif choice == "2":
            filename = input("Enter the filename: ")
            content = input("Enter text to append: ")
            with open(filename, 'a') as file:
                file.write(content)
            print(f"Text appended to '{filename}'.")

        elif choice == "3":
            filename = input("Enter the filename: ")
            try:
                with open(filename, 'r') as file:
                    print(file.read())
            except FileNotFoundError:
                print(f"File '{filename}' not found.")

        elif choice == "4":
            break
        else:
            print("Invalid choice")

text_editor()
```

# Lab Manual for Week 4: File Handling and Basic Project Structuring – Day 2

## Day 2: Working with Files (Continued)

Time: 3 Hours

Learning Outcomes:
- Understand how to append data to files in Python.
- Learn to handle common file errors such as `FileNotFoundError`.
- Practice file handling through practical examples, including saving user data to files.

### Tasks:

1. Appending Data to a File (15 mins):

Write a Python program that appends a new line of text to an existing file.

```
with open('example.txt', 'a') as file:
    file.write('This is an appended line.\n')
```

2. Reading After Appending (15 mins):

Write a program to read the contents of a file after appending data to it.

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

3. Writing and Appending User Input (15 mins):

Write a Python program that asks the user for input and writes it to a file. If the file exists, append to it; otherwise, create a new file.

```
user_input = input('Enter text to save: ')
with open('user_data.txt', 'a') as file:
    file.write(user_input + '\n')
```

4. Handling `FileNotFoundError` (15 mins):

Write a Python program that handles `FileNotFoundError` when trying to open a non-existent file.

```
try:
    with open('non_existent_file.txt', 'r') as file:
        print(file.read())
```

```
except FileNotFoundError:
    print('File not found.')
```

5. Checking If a File Exists (15 mins):

Write a program that checks if a file exists before trying to read it.

```
import os
if os.path.exists('example.txt'):
    with open('example.txt', 'r') as file:
        print(file.read())
else:
    print('File does not exist.')
```

6. Deleting a File (15 mins):

Write a program that deletes a file if it exists.

```
import os
if os.path.exists('example.txt'):
    os.remove('example.txt')
    print('File deleted.')
else:
    print('File does not exist.')
```

7. Copying File Contents (15 mins):

Write a Python program that reads data from one file and writes it to another file.

```
with open('source.txt', 'r') as source_file:
    content = source_file.read()
with open('destination.txt', 'w') as dest_file:
    dest_file.write(content)
```

8. Appending List Data to a File (15 mins):

Write a program that appends a list of strings to a file.

```
fruits = ['apple', 'banana', 'cherry']
with open('fruits.txt', 'a') as file:
    for fruit in fruits:
        file.write(fruit + '\n')
```

9. Writing User Data to a File (15 mins):

Write a program that asks the user for their name and age, then saves this information to a file.

```
name = input('Enter your name: ')
age = input('Enter your age: ')
```

```
with open('user_info.txt', 'a') as file:
    file.write(f'Name: {name}, Age: {age}\n')
```

10. Reading and Displaying File Contents (15 mins):

Write a program that reads the contents of a file and displays it on the screen.

```
with open('user_info.txt', 'r') as file:
    content = file.read()
    print(content)
```

11. Handling File Input Errors (15 mins):

Write a program that asks for a filename to read and handles the `FileNotFoundError` if the file doesn't exist.

```
filename = input('Enter the filename to read: ')
try:
    with open(filename, 'r') as file:
        print(file.read())
except FileNotFoundError:
    print(f'File {filename} not found.')
```

12. File Line Count (15 mins):

Write a program that counts the number of lines in a text file.

```
with open('user_info.txt', 'r') as file:
    line_count = len(file.readlines())
    print(f'Total lines: {line_count}')
```

13. Appending Numbers to a File (15 mins):

Write a Python program that appends numbers from 1 to 10 to a file, each on a new line.

```
with open('numbers.txt', 'a') as file:
    for num in range(1, 11):
        file.write(f'{num}\n')
```

14. Appending Multiple Lines of Input (15 mins):

Write a program that asks the user for multiple lines of input and appends each line to a file.

```
lines = []
for _ in range(3):
    line = input('Enter a line of text: ')
    lines.append(line)
with open('user_input.txt', 'a') as file:
    for line in lines:
        file.write(line + '\n')
```

15. Combining Read and Write Operations (15 mins):

Write a program that reads the contents of a file, modifies it, and writes the updated content back to the file.

```
with open('user_input.txt', 'r') as file:
    content = file.read()
updated_content = content.replace('apple', 'orange')
with open('user_input.txt', 'w') as file:
    file.write(updated_content)
```

## Assignment Task:

- **User Data Storage System** (45 mins):

Write a Python program that simulates a basic user data storage system. The program should allow the user to:

1. Add new user data (name, age, and email).
2. View all stored user data.
3. Append new data to an existing user entry.
4. Handle errors if the file does not exist.

```
def user_data_storage():
    while True:
        print("\nOptions:")
        print("1. Add new user data")
        print("2. View all user data")
        print("3. Append to existing user data")
        print("4. Exit")

        choice = input("Choose an option: ")

        if choice == "1":
            name = input("Enter name: ")
            age = input("Enter age: ")
            email = input("Enter email: ")
            with open('user_data.txt', 'a') as file:
                file.write(f'Name: {name}, Age: {age}, Email:
{email}\n')
            print("User data saved.")

        elif choice == "2":
            try:
                with open('user_data.txt', 'r') as file:
                    print(file.read())
```

```
            except FileNotFoundError:
                print("No user data found.")

        elif choice == "3":
            name = input("Enter name: ")
            data = input("Enter data to append (age or email): ")
            try:
                with open('user_data.txt', 'a') as file:
                    file.write(f'Updated data for {name}: {data}\n')
                print(f"Data for {name} updated.")
            except FileNotFoundError:
                print(f"No data found for {name}.")

        elif choice == "4":
            break
        else:
            print("Invalid option.")

user_data_storage()
```

# Lab Manual for Week 4: File Handling and Basic Project Structuring – Day 3

## Day 3: Structuring a Python Project

Time: 3 Hours

Learning Outcomes:
- Learn how to structure a Python project with modules and functions.
- Organize code into reusable components.
- Explore how to import code from other files and create basic Python packages.

### Tasks:

1. Organizing Code into Functions (15 mins):

Write a Python function `greet()` that prints a greeting message. Call the function multiple times in the script.

```
def greet():
    print('Hello, welcome to the Python project!')
# Calling the function multiple times
greet()
greet()
```

2. Saving Functions in a Separate File (15 mins):

Save the `greet()` function from Task 1 in a file named `greetings.py`.

```
def greet():
    print('Hello, welcome to the Python project!')
# Save this code in a file called 'greetings.py'
```

3. Importing a Function from Another File (15 mins):

Write a new Python script that imports the `greet()` function from `greetings.py` and calls it.

```
from greetings import greet
greet()
```

4. Creating a Function to Add Numbers (15 mins):

Write a Python function `add_numbers(a, b)` that takes two numbers as input and returns their sum. Save this function in `math_utils.py`.

```
def add_numbers(a, b):
    return a + b

# Save this code in a file called 'math_utils.py'
```

5. Importing Functions from Multiple Files (15 mins):

Write a script that imports `greet()` from `greetings.py` and `add_numbers()` from `math_utils.py`.

```
from greetings import greet
from math_utils import add_numbers

greet()
result = add_numbers(5, 10)
print(f'The sum is: {result}')
```

6. Using `if __name__ == '__main__'` (15 mins):

Modify the `add_numbers()` function in `math_utils.py` so that it only runs when the file is executed directly.

```
def add_numbers(a, b):
    return a + b

if __name__ == '__main__':
    result = add_numbers(10, 20)
    print(f'The sum is: {result}')
```

7. Creating a Python Package (15 mins):

Create a folder named `utilities`, and place `greetings.py` and `math_utils.py` inside it. Create an empty `__init__.py` file inside the folder to make it a package.

```
# Create a folder structure like this:
# utilities/
# ├── __init__.py
# ├── greetings.py
# └── math_utils.py
```

8. Importing from a Python Package (15 mins):

Write a script that imports the functions `greet()` and `add_numbers()` from the `utilities` package.

```
from utilities.greetings import greet
from utilities.math_utils import add_numbers

greet()
print(add_numbers(10, 15))
```

9. Creating a Module for String Operations (15 mins):

Create a file `string_utils.py` and write a function `capitalize_text(text)` that capitalizes a given string.

```
def capitalize_text(text):
    return text.capitalize()

# Save this code in 'string_utils.py'
```

10. Importing from Multiple Modules (15 mins):

Write a script that imports functions from `greetings.py`, `math_utils.py`, and `string_utils.py` and calls them.

```
from utilities.greetings import greet
from utilities.math_utils import add_numbers
from utilities.string_utils import capitalize_text

greet()
result = add_numbers(7, 8)
print(f'Sum: {result}')
print(capitalize_text('hello world'))
```

11. Using Relative Imports (15 mins):

Modify your package to use relative imports for importing `add_numbers()` inside the `greetings.py` file.

```
from .math_utils import add_numbers

def greet_and_add(a, b):
    print('Hello!')
    print(f'The sum is: {add_numbers(a, b)}')
```

12. Writing a Module for File Operations (15 mins):

Create a new file `file_utils.py` that contains a function `read_file(filename)` to read and print the contents of a file.

```
def read_file(filename):
    with open(filename, 'r') as file:
        content = file.read()
```

```
        print(content)

# Save this code in 'file_utils.py'
```

13. Importing and Using File Operations (15 mins):

Write a script that imports `read_file()` from `file_utils.py` and reads the contents of a file.

```
from utilities.file_utils import read_file

read_file('example.txt')
```

14. Combining Functions in a Project (15 mins):

Write a Python script that combines the functions from all the modules created so far into a single script that performs a greeting, a calculation, and a file read.

```
from utilities.greetings import greet
from utilities.math_utils import add_numbers
from utilities.file_utils import read_file

greet()
print(add_numbers(5, 10))
read_file('example.txt')
```

15. Documenting a Python Project (15 mins):

Write a `README.md` file for the Python project you have created, explaining how to use each module and function.

```
# Example of a simple README.md file content

# Python Project
This project contains modules for greetings, math operations, string
manipulation, and file handling.

## How to Use
1. Import the functions from the respective modules.
2. Call the functions as needed.
```

## Assignment Task:

- **Modularized To-Do List Application** (45 mins):

Write a Python program to modularize the To-Do List application. The application should allow the user to:

1. Add a new task to the list.
2. View all tasks.

3. Delete a task from the list.

The program should be divided into multiple files: one for handling tasks, one for file operations (saving/loading tasks), and one for the main logic.

```
# tasks.py
tasks = []

def add_task(task):
    tasks.append(task)

def view_tasks():
    for i, task in enumerate(tasks, 1):
        print(f'{i}. {task}')

def delete_task(task_number):
    if 0 < task_number <= len(tasks):
        tasks.pop(task_number - 1)

# file_utils.py
def save_tasks(filename):
    with open(filename, 'w') as file:
        for task in tasks:
            file.write(task + '\n')

def load_tasks(filename):
    with open(filename, 'r') as file:
        global tasks
        tasks = file.read().splitlines()

# main.py
from tasks import add_task, view_tasks, delete_task
from file_utils import save_tasks, load_tasks

while True:
    print("\n1. Add Task")
    print("2. View Tasks")
    print("3. Delete Task")
    print("4. Save Tasks")
    print("5. Load Tasks")
    print("6. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        task = input("Enter a task: ")
```

```
            add_task(task)
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            task_number = int(input("Enter task number to delete: "))
            delete_task(task_number)
        elif choice == "4":
            save_tasks('tasks.txt')
            print("Tasks saved.")
        elif choice == "5":
            load_tasks('tasks.txt')
            print("Tasks loaded.")
        elif choice == "6":
            break
        else:
            print("Invalid choice")
```

# Lab Manual for Week 4: File Handling and Basic Project Structuring – Day 4

## Day 4: Project Day Preparation

Time: 3 Hours

Learning Outcomes:
- Review key concepts of file handling and modularization.
- Refactor a Python project for better structure and organization.
- Implement file storage for project data, allowing persistence across sessions.

### Tasks:

1. Reviewing File Handling Basics (15 mins):
Write a simple program that creates a text file and writes a message into it.

```
with open('review.txt', 'w') as file:
    file.write('File handling basics review.')
```

2. Reading from a File (15 mins):
Write a Python program to read the contents of a file and print them to the console.

```
with open('review.txt', 'r') as file:
    print(file.read())
```

3. Appending Data to a File (15 mins):
Write a program to append new data to an existing file.

```
with open('review.txt', 'a') as file:
    file.write('\nAppending new data.')
```

4. Organizing Code into Functions (15 mins):
Write a Python function `save_to_file(filename, data)` that saves data to a file.

```
def save_to_file(filename, data):
    with open(filename, 'w') as file:
        file.write(data)

save_to_file('example.txt', 'This is an example.')
```

5. Reading and Processing File Data (15 mins):
Write a function `read_file(filename)` that reads the file and returns the content.

```
def read_file(filename):
    with open(filename, 'r') as file:
```

```
        return file.read()

print(read_file('example.txt'))
```

6. Creating a Project Folder (15 mins):
Create a folder named `todo_project` and move your Python files into it for better organization.

```
# Create the folder manually or programmatically
import os
os.mkdir('todo_project')
```

7. Saving Task Data to a File (15 mins):
Write a function `save_tasks(tasks)` that saves a list of tasks to a file.

```
def save_tasks(tasks, filename):
    with open(filename, 'w') as file:
        for task in tasks:
            file.write(task + '\n')

tasks = ['Task 1', 'Task 2']
save_tasks(tasks, 'tasks.txt')
```

8. Loading Task Data from a File (15 mins):
Write a function `load_tasks(filename)` that loads tasks from a file and returns them as a list.

```
def load_tasks(filename):
    with open(filename, 'r') as file:
        return file.read().splitlines()

tasks = load_tasks('tasks.txt')
print(tasks)
```

9. Refactoring Code into Modules (15 mins):
Move your task-saving and loading functions into a separate file called `file_utils.py`.

```
# Move the `save_tasks` and `load_tasks` functions into
'file_utils.py'
```

10. Importing Functions from a Module (15 mins):
Write a new script that imports the `save_tasks` and `load_tasks` functions from `file_utils.py` and uses them.

```
from file_utils import save_tasks, load_tasks

tasks = ['Task 3', 'Task 4']
save_tasks(tasks, 'tasks.txt')
print(load_tasks('tasks.txt'))
```

11. Handling File Errors (15 mins):
Write a program that handles `FileNotFoundError` when trying to load tasks from a non-existent file.

```
try:
    tasks = load_tasks('non_existent_file.txt')
except FileNotFoundError:
    print('File not found.')
```

12. Reviewing Modularization Concepts (15 mins):
Refactor your to-do list project by separating file handling, task management, and main logic into separate files.

```
# Refactor project as explained: tasks.py, file_utils.py, main.py
```

13. Using `if __name__ == '__main__'` (15 mins):
Ensure your scripts only execute when run directly, not when imported as modules.

```
if __name__ == '__main__':
    tasks = ['Sample task']
    save_tasks(tasks, 'tasks.txt')
```

14. Debugging with Print Statements (15 mins):
Add print statements to your project to trace the flow of data between modules.

```
def load_tasks(filename):
    print(f'Loading tasks from {filename}')
    with open(filename, 'r') as file:
        return file.read().splitlines()
```

15. Documenting the Refactored Project (15 mins):
Write a `README.md` file explaining how the project is structured and how to use it.

```
# README.md content:

# To-Do List Project
This project allows users to manage a to-do list with file storage
for persistence.
```

## Assignment Task:

- **Refactoring the To-Do List Project** (45 mins):

Refactor the To-Do List project so that:

1. The task management functions (`add_task()`, `view_tasks()`, `delete_task()`) are in a file called `tasks.py`.

2. File handling functions (`save_tasks()`, `load_tasks()`) are in `file_utils.py`.

3. The main program logic is in `main.py`.

```
# main.py
from tasks import add_task, view_tasks, delete_task
from file_utils import save_tasks, load_tasks

if __name__ == '__main__':
    tasks = load_tasks('tasks.txt')
    while True:
        print("\n1. Add Task")
        print("2. View Tasks")
        print("3. Delete Task")
        print("4. Save Tasks")
        print("5. Exit")

        choice = input("Choose an option: ")

        if choice == "1":
            task = input("Enter a task: ")
            add_task(task)
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            task_number = int(input("Enter task number to delete:
"))
            delete_task(task_number)
        elif choice == "4":
            save_tasks(tasks, 'tasks.txt')
        elif choice == "5":
            break
        else:
            print("Invalid choice")
```

# Lab Manual for Week 4: File Handling and Basic Project Structuring – Day 5

## Day 5: Mega Tasks and Project Work

Time: 3 Hours

Learning Outcomes:
- Apply file handling and project structuring concepts learned throughout the week.
- Implement incremental project changes and mega tasks that integrate file storage and modular design.

### Mega Tasks:

1. Enhanced To-Do List with File Storage (45 mins):
Write a Python program that allows users to add, view, and delete tasks. Tasks should be stored in a file so that they persist between program runs.

```
# Enhanced To-Do List with File Storage
tasks = []

def add_task(task):
    tasks.append(task)

def view_tasks():
    for i, task in enumerate(tasks, 1):
        print(f'{i}. {task}')

def delete_task(task_number):
    if 0 < task_number <= len(tasks):
        tasks.pop(task_number - 1)

def save_tasks(filename):
    with open(filename, 'w') as file:
        for task in tasks:
            file.write(task + '\n')

def load_tasks(filename):
    with open(filename, 'r') as file:
        global tasks
        tasks = file.read().splitlines()

if __name__ == '__main__':
```

```
    try:
        load_tasks('tasks.txt')
    except FileNotFoundError:
        print('No existing tasks found.')

    while True:
        print("\n1. Add Task")
        print("2. View Tasks")
        print("3. Delete Task")
        print("4. Save and Exit")

        choice = input("Choose an option: ")
        if choice == "1":
            task = input("Enter a task: ")
            add_task(task)
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            task_number = int(input("Enter task number to delete:
"))
            delete_task(task_number)
        elif choice == "4":
            save_tasks('tasks.txt')
            print("Tasks saved and exiting.")
            break
        else:
            print("Invalid choice")
```

2. Modularizing the To-Do List Project (30 mins):
Refactor the above To-Do List project by separating file handling and task management into separate modules.

```
# File Handling (file_utils.py)
def save_tasks(tasks, filename):
    with open(filename, 'w') as file:
        for task in tasks:
            file.write(task + '\n')

def load_tasks(filename):
    with open(filename, 'r') as file:
        return file.read().splitlines()

# Task Management (tasks.py)
tasks = []
```

```
def add_task(task):
    tasks.append(task)

def view_tasks():
    for i, task in enumerate(tasks, 1):
        print(f'{i}. {task}')

def delete_task(task_number):
    if 0 < task_number <= len(tasks):
        tasks.pop(task_number - 1)

# Main Program (main.py)
from file_utils import save_tasks, load_tasks
from tasks import add_task, view_tasks, delete_task

if __name__ == '__main__':
    tasks = []
    try:
        tasks = load_tasks('tasks.txt')
    except FileNotFoundError:
        print('No existing tasks found.')

    while True:
        print("\n1. Add Task")
        print("2. View Tasks")
        print("3. Delete Task")
        print("4. Save and Exit")

        choice = input("Choose an option: ")
        if choice == "1":
            task = input("Enter a task: ")
            add_task(task)
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            task_number = int(input("Enter task number to delete:
"))
            delete_task(task_number)
        elif choice == "4":
            save_tasks(tasks, 'tasks.txt')
            print("Tasks saved and exiting.")
            break
        else:
            print("Invalid choice")
```

3. Error Handling in File Operations (30 mins):

Add error handling to ensure that the program behaves correctly if the file is not found or cannot be opened.

```
# File Handling with Error Handling (file_utils.py)
def save_tasks(tasks, filename):
    try:
        with open(filename, 'w') as file:
            for task in tasks:
                file.write(task + '\n')
    except IOError as e:
        print(f"An error occurred while saving tasks: {e}")

def load_tasks(filename):
    try:
        with open(filename, 'r') as file:
            return file.read().splitlines()
    except FileNotFoundError:
        print("No tasks found. Starting with an empty list.")
        return []
    except IOError as e:
        print(f"An error occurred while loading tasks: {e}")
        return []
```

4. Adding Search Functionality (30 mins):
Add a search feature to the To-Do List project, allowing users to search for tasks by keyword.

```
# Search Functionality (tasks.py)
def search_task(keyword):
    matches = [task for task in tasks if keyword.lower() in
task.lower()]
    if matches:
        for match in matches:
            print(match)
    else:
        print("No tasks found with that keyword.")

# Modify the Main Program to Include Search (main.py)
from tasks import search_task

if __name__ == '__main__':
    tasks = []
    try:
        tasks = load_tasks('tasks.txt')
```

```
    except FileNotFoundError:
        print('No existing tasks found.')

    while True:
        print("\n1. Add Task")
        print("2. View Tasks")
        print("3. Delete Task")
        print("4. Search Task")
        print("5. Save and Exit")

        choice = input("Choose an option: ")
        if choice == "1":
            task = input("Enter a task: ")
            add_task(task)
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            task_number = int(input("Enter task number to delete:
"))
            delete_task(task_number)
        elif choice == "4":
            keyword = input("Enter keyword to search: ")
            search_task(keyword)
        elif choice == "5":
            save_tasks(tasks, 'tasks.txt')
            print("Tasks saved and exiting.")
            break
        else:
            print("Invalid choice")
```

# Lab Manual for Week 5: Introduction to Object-Oriented Programming – Day 1

## Day 1: Introduction to Classes and Objects

Time: 3 Hours

Learning Outcomes:
- Understand the basic concepts of Object-Oriented Programming (OOP).
- Learn how to define a class and create objects.
- Understand object attributes and methods.

### Tasks:

1. What is Object-Oriented Programming? (15 mins):
Discuss what Object-Oriented Programming (OOP) is and its key concepts such as classes, objects, and methods.

2. Defining a Simple Class (15 mins):
Write a Python class `Car` with two attributes: `brand` and `model`. Then create an object of the class.

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

# Create an object
my_car = Car('Toyota', 'Corolla')
print(f'My car is a {my_car.brand} {my_car.model}')
```

3. Adding Methods to a Class (15 mins):
Add a method `start_engine()` to the `Car` class that prints a message when called.

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        print(f'The {self.brand} {self.model}\'s engine has
started.')

# Create an object and call the method
```

```
my_car = Car('Toyota', 'Corolla')
my_car.start_engine()
```

4. Creating Multiple Objects (15 mins):
Create multiple objects of the `Car` class with different attributes and call their methods.

```
car1 = Car('Honda', 'Civic')
car2 = Car('Ford', 'Mustang')

car1.start_engine()
car2.start_engine()
```

5. Using Object Attributes (15 mins):
Write a class `Student` with attributes `name` and `age`. Add a method `introduce()` that prints a message with the student's name and age.

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f'Hi, I am {self.name} and I am {self.age} years
old.')

# Create a student object and call the method
student = Student('Alice', 20)
student.introduce()
```

6. Modifying Object Attributes (15 mins):
Modify the attributes of the `Student` object after its creation.

```
student.age = 21
student.introduce()  # Now the age should be 21
```

7. Defining a Simple Class for To-Do List (15 mins):
Define a class `Task` with attributes `title` and `completed`. Add a method `mark_complete()` to mark the task as completed.

```
class Task:
    def __init__(self, title):
        self.title = title
        self.completed = False

    def mark_complete(self):
        self.completed = True
        print(f'Task {self.title} is marked as completed.')
```

```
# Create a task object and mark it as complete
task1 = Task('Finish homework')
task1.mark_complete()
```

8. Creating and Managing Multiple Task Objects (15 mins):
Create multiple `Task` objects and store them in a list. Write a function to display all tasks and their statuses.

```
tasks = [Task('Finish homework'), Task('Go shopping'), Task('Read a book')]

def show_tasks(tasks):
    for task in tasks:
        status = 'Completed' if task.completed else 'Incomplete'
        print(f'{task.title}: {status}')

show_tasks(tasks)
```

9. Adding a Method to Display Task Status (15 mins):
Add a method `show_status()` to the `Task` class that prints whether the task is completed or not.

```
class Task:
    def __init__(self, title):
        self.title = title
        self.completed = False

    def mark_complete(self):
        self.completed = True

    def show_status(self):
        status = 'Completed' if self.completed else 'Incomplete'
        print(f'Task {self.title} is {status}.')

# Create and display task status
task1 = Task('Finish homework')
task1.show_status()
task1.mark_complete()
task1.show_status()
```

10. Adding User Input to Create Tasks (15 mins):
Write a Python program that allows users to create `Task` objects by providing a title via user input. Store the tasks in a list.

```
tasks = []

def create_task():
    title = input('Enter task title: ')
```

```
    task = Task(title)
    tasks.append(task)
    print(f'Task {title} created.')

# Create a task from user input
create_task()
```

## Assignment Task:

- **Simple Task Management System** (45 mins):

Write a Python program that simulates a simple task management system using OOP. The program should allow the user to:

1. Add a new task (title only).
2. View all tasks.
3. Mark a task as completed.
4. Exit the program.

```
class Task:
    def __init__(self, title):
        self.title = title
        self.completed = False

    def mark_complete(self):
        self.completed = True

    def show_status(self):
        status = 'Completed' if self.completed else 'Incomplete'
        print(f'Task {self.title} is {status}.')

tasks = []

def create_task():
    title = input('Enter task title: ')
    task = Task(title)
    tasks.append(task)

def view_tasks():
    for i, task in enumerate(tasks, 1):
        task.show_status()

def mark_task_completed():
    task_number = int(input('Enter task number to mark complete: '))
    if 0 < task_number <= len(tasks):
        tasks[task_number - 1].mark_complete()
```

```
while True:
    print("\n1. Add Task")
    print("2. View Tasks")
    print("3. Mark Task as Completed")
    print("4. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        create_task()
    elif choice == "2":
        view_tasks()
    elif choice == "3":
        mark_task_completed()
    elif choice == "4":
        break
    else:
        print("Invalid choice")
```

# Lab Manual for Week 5: Object-Oriented Programming – Day 2

## Day 2: More on Classes and Methods

Time: 3 Hours

Learning Outcomes:

Understand how to define constructor methods (__init__).

Learn how to add methods to a class to perform specific tasks.

Practice managing objects and interacting with object attributes and methods

### Tasks:

2. 1. Defining a Constructor Method (__init__) (15 mins):
- Define a class Person with attributes name and age. Initialize these attributes using a constructor method.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
person1 = Person('John', 30)
print(f'{person1.name} is {person1.age} years old.')
```

3. Adding Methods to a Class (15 mins):
- Add a method greet() to the Person class that prints a greeting.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f'Hello, my name is {self.name}.')

person1.greet()
```

4. Using Class Attributes (15 mins):
- Modify the countdown timer to break the loop if the number goes below 5.

```
class Person:
    species = 'Human'

    def __init__(self, name, age):
        self.name = name
        self.age = age

print(Person.species))
```

5. Creating a Task Class for To-Do List (20 mins):

- Create a Task class with attributes title and completed. Add a method mark_complete().

```
class Task:
    def __init__(self, title):
        self.title = title
        self.completed = False

    def mark_complete(self):
        self.completed = True
        print(f'Task {self.title} is marked as completed.')

task1 = Task('Finish homework')
task1.mark_complete()
```

5. Adding a Method to Display Task Status (15 mins)

- Add a method show_status() to the Task class that prints whether the task is completed

```
class Task:
    def __init__(self, title):
        self.title = title
        self.completed = False

    def show_status(self):
        status = 'Completed' if self.completed else 'Incomplete'
        print(f'Task {self.title} is {status}.')

task1.show_status()
```

6.Creating Multiple Task Objects (15 mins):

- Create multiple Task objects and store them in a list. Write a function to display all tasks and their status.

```
tasks = [Task('Finish homework'), Task('Go shopping'), Task('Read a
book')]

def show_tasks(tasks):
    for task in tasks:
        task.show_status()

show_tasks(tasks)
```

7. Creating a Task with Due Date (20 mins):
- Modify the Task class to include a due date (due_date attribute) and display it.

```
class Task:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
        self.completed = False

    def show_due_date(self):
        print(f'Task {self.title} is due on {self.due_date}.')

task1 = Task('Finish homework', '2024-09-15')
task1.show_due_date()
```

8. Overriding the __str__ Method (15 mins):
- Override the __str__ method to provide a string representation of the task.

```
class Task:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
        self.completed = False

    def __str__(self):
        status = 'Completed' if self.completed else 'Incomplete'
        return f'Task: {self.title}, Due: {self.due_date}, Status:
{status}'

task1 = Task('Finish homework', '2024-09-15')
print(task1)
```

9. Updating Task Title (15 mins):
- Add a method update_title(new_title) to change the title of a task.

```
class Task:
    def __init__(self, title, due_date):
```

```
        self.title = title
        self.due_date = due_date

    def update_title(self, new_title):
        self.title = new_title

task1.update_title('Complete assignment')
print(task1)
```

10. Creating Tasks from User Input (20 mins):
- Write a program that creates Task objects from user input.

```
tasks = []

for _ in range(3):
    title = input('Enter task title: ')
    due_date = input('Enter due date: ')
    task = Task(title, due_date)
    tasks.append(task)
```

11. Marking Task Completion from User Input (20 mins):
- Extend the program to allow users to mark tasks as completed by entering the task number.

```
def mark_task_completed(tasks):
    task_number = int(input('Enter task number to mark complete: '))
    if 0 < task_number <= len(tasks):
        tasks[task_number - 1].mark_complete()

mark_task_completed(tasks)
```

12. Viewing All Tasks (15 mins):
- Write a program that displays the status and due date of all tasks.

```
for task in tasks:
    print(task)
```

13. Adding Task Priority (20 mins):
- Add a priority attribute (Low, Medium, High) to the Task class and display it.

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority

task1 = Task('Complete project', '2024-09-15', 'High')
print(task1)
```

14. Task Search by Keyword (20 mins):
- Add a search function that allows users to search for tasks by keyword.

```
def search_task(keyword):
    for task in tasks:
        if keyword.lower() in task.title.lower():
            print(task)
```

15. Saving and Loading Tasks (30 mins):
- Write functions to save tasks to a file and load them back into the program.

```
def save_tasks(tasks, filename):
    with open(filename, 'w') as file:
        for task in tasks:
            file.write(str(task) + '\n')
def load_tasks(filename):
    with open(filename, 'r') as file:
        tasks = file.readlines()
    return tasks
```

13. Adding Task Priority (20 mins):
- Add a priority attribute (Low, Medium, High) to the Task class and display it.

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority

task1 = Task('Complete project', '2024-09-15', 'High')
print(task1)
```

## Assignment Task:

Task Management System (45 mins):- Extend the countdown timer by adding these features:

- Add a new task (with title, due date, and priority).
- View all tasks with their details.
- Mark a task as completed.
- Save tasks to a file and load tasks from a file.

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority
        self.completed = False
```

```
    def mark_complete(self):
        self.completed = True

    def show_task(self):
        status = 'Completed' if self.completed else 'Incomplete'
        print(f'Task: {self.title}, Due: {self.due_date}, Priority:
{self.priority}, Status: {status}')


tasks = []

def create_task():
    title = input("Enter task title: ")
    due_date = input("Enter due date (YYYY-MM-DD): ")
    priority = input("Enter priority (Low, Medium, High): ")
    task = Task(title, due_date, priority)
    tasks.append(task)

def view_tasks():
    if not tasks:
        print("No tasks available.")
    else:
        for i, task in enumerate(tasks, 1):
            print(f"Task {i}:")
            task.show_task()
            print("")

def mark_task_completed():
    view_tasks()
    task_number = int(input("Enter the task number to mark as
completed: "))
    if 0 < task_number <= len(tasks):
        tasks[task_number - 1].mark_complete()
        print("Task marked as completed.")
    else:
        print("Invalid task number.")

def save_tasks(filename="tasks.txt"):
    with open(filename, 'w') as file:
        for task in tasks:
            task_line =
f"{task.title},{task.due_date},{task.priority},{task.completed}\n"
            file.write(task_line)
    print("Tasks saved successfully.")
```

```
def load_tasks(filename="tasks.txt"):
    try:
        with open(filename, 'r') as file:
            for line in file:
                title, due_date, priority, completed =
line.strip().split(',')
                task = Task(title, due_date, priority)
                task.completed = completed == 'True'
                tasks.append(task)
        print("Tasks loaded successfully.")
    except FileNotFoundError:
        print("No saved tasks found.")

def main():
    while True:
        print("\nTask Management System")
        print("1. Add a new task")
        print("2. View all tasks")
        print("3. Mark a task as completed")
        print("4. Save tasks to file")
        print("5. Load tasks from file")
        print("6. Exit")

        choice = input("Choose an option: ")
        if choice == "1":
            create_task()
        elif choice == "2":
            view_tasks()
        elif choice == "3":
            mark_task_completed()
        elif choice == "4":
            save_tasks()
        elif choice == "5":
            load_tasks()
        elif choice == "6":
            print("Exiting program.")
            break
        else:
            print("Invalid choice. Please try again.")

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

# Lab Manual for Week 5: Object-Oriented Programming – Day 3

## Day 3: Inheritance and Polymorphism

Time: 3 Hours

Learning Outcomes:
- Understand class inheritance and how to create child classes.
- Learn about method overriding and polymorphism.
- Practice creating a task management system with inheritance.

### Tasks:

1. Creating a Base Class (15 mins):
Create a class `Animal` with attributes `name` and `species`. Add a method `make_sound()`.

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        print('Animal sound')

# Create an object
animal1 = Animal('Lion', 'Mammal')
animal1.make_sound()
```

2. Creating a Child Class (15 mins):
Create a class `Dog` that inherits from `Animal` and overrides `make_sound()`.

```
class Dog(Animal):
    def make_sound(self):
        print('Bark')

# Create a Dog object
dog1 = Dog('Buddy', 'Dog')
dog1.make_sound()
```

3. Using `super()` in Child Class (15 mins):
Use the `super()` function to access the parent class's constructor and methods.

```
class Dog(Animal):
    def __init__(self, name, species, breed):
        super().__init__(name, species)
```

```
        self.breed = breed

    def make_sound(self):
        super().make_sound()
        print('Bark')

# Create a Dog object
dog1 = Dog('Buddy', 'Dog', 'Golden Retriever')
dog1.make_sound()
```

4. Creating Another Child Class (15 mins):
Create a class `Cat` that also inherits from `Animal` and has its own `make_sound()`.

```
class Cat(Animal):
    def make_sound(self):
        print('Meow')

# Create a Cat object
cat1 = Cat('Whiskers', 'Cat')
cat1.make_sound()
```

5. Polymorphism Example (15 mins):
Create a function that accepts any `Animal` object and calls its `make_sound()` method.

```
def animal_sound(animal):
    animal.make_sound()

# Test with different animals
animal_sound(dog1)
animal_sound(cat1)
```

6. Adding New Methods in Child Class (15 mins):
Add a method `play()` to the `Dog` class that prints a message.

```
class Dog(Animal):
    def play(self):
        print(f'{self.name} is playing.')

# Test the play method
dog1.play()
```

7. Method Overriding (15 mins):
Override the `make_sound()` method in the `Dog` class and call the parent class method using `super()`.

```
class Dog(Animal):
    def make_sound(self):
        super().make_sound()
        print('Woof')
```

```
# Test method overriding
dog1.make_sound()
```
8. Adding Attributes to Child Class (15 mins):
Add a new attribute `age` to the `Dog` class and initialize it using `__init__()`.

```
class Dog(Animal):
    def __init__(self, name, species, breed, age):
        super().__init__(name, species)
        self.breed = breed
        self.age = age

# Create a Dog object
dog1 = Dog('Buddy', 'Dog', 'Golden Retriever', 3)
print(dog1.age)
```
9. Inheritance with Multiple Levels (15 mins):
Create another class `Puppy` that inherits from `Dog` and adds a method `train()`.

```
class Puppy(Dog):
    def train(self):
        print(f'{self.name} is being trained.')

# Test Puppy class
puppy1 = Puppy('Max', 'Dog', 'Labrador', 1)
puppy1.train()
```
10. Creating a Vehicle Base Class (15 mins):
Create a class `Vehicle` with attributes `make`, `model`, and a method `start_engine()`.

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print('Engine started')

# Test Vehicle class
vehicle1 = Vehicle('Toyota', 'Corolla')
vehicle1.start_engine()
```
11. Inheriting from `Vehicle` (15 mins):
Create a class `Car` that inherits from `Vehicle` and adds an attribute `year`.

```
class Car(Vehicle):
    def __init__(self, make, model, year):
        super().__init__(make, model)
        self.year = year

# Create a Car object
```

```
car1 = Car('Honda', 'Civic', 2020)
print(car1.year)
```
12. Overriding a Method in Child Class (15 mins):
Override the `start_engine()` method in the `Car` class.

```
class Car(Vehicle):
    def start_engine(self):
        print('Car engine started')

# Test method override
car1.start_engine()
```
13. Adding Methods to `Car` Class (15 mins):
Add a method `drive()` to the `Car` class.

```
class Car(Vehicle):
    def drive(self):
        print(f'{self.make} {self.model} is driving.')

# Test drive method
car1.drive()
```
14. Multiple Inheritance (15 mins):
Create a class `ElectricCar` that inherits from both `Car` and another class `ElectricVehicle`.

```
class ElectricVehicle:
    def charge_battery(self):
        print('Battery charging...')

class ElectricCar(Car, ElectricVehicle):
    def __init__(self, make, model, year):
        Car.__init__(self, make, model, year)

# Test multiple inheritance
ecar1 = ElectricCar('Tesla', 'Model S', 2022)
ecar1.charge_battery()
```

15. Using Polymorphism (15 mins):
- Write a function that accepts both `Car` and `ElectricCar` objects and calls their methods.

```
def vehicle_info(vehicle):
    vehicle.start_engine()
    vehicle.drive()

# Test with both types of vehicles
vehicle_info(car1)
vehicle_info(ecar1)
```

## Assignment Task:

- **Task Management System with Inheritance** (45 mins):

Extend the Task Management System with the following:

1. Create a base class `Task` with attributes `title`, `due_date`, and `priority`.

2. Create a child class `ProjectTask` that inherits from `Task` and adds an attribute `project_name`.

3. Override the method to display task details in the `ProjectTask` class to include the project name.

4. Write a function that accepts both `Task` and `ProjectTask` objects and displays their details.

# Base class Task

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority

    def display_details(self):
        print(f"Task: {self.title}, Due Date: {self.due_date},
Priority: {self.priority}")

# Child class ProjectTask that inherits from Task
class ProjectTask(Task):
    def __init__(self, title, due_date, priority, project_name):
        super().__init__(title, due_date, priority)
        self.project_name = project_name

    # Overriding the display_details method to include project_name
    def display_details(self):
        print(f"Project: {self.project_name} - Task: {self.title},
Due Date: {self.due_date}, Priority: {self.priority}")

# Function to display details of both Task and ProjectTask objects
def display_task_details(task):
    task.display_details()

# Example usage
task1 = Task("Complete report", "2024-09-15", "High")
```

```
project_task1 = ProjectTask("Design module", "2024-09-20", "Medium",
"Website Redesign")

# Display details
display_task_details(task1)
display_task_details(project_task1)
```

# Lab Manual for Week 5: Object-Oriented Programming – Day 4

## Day 4: Working with Objects in the Project

Time: 3 Hours

Learning Outcomes:
- Refactor code to use object-oriented principles.
- Manage objects as task representations in a project.
- Practice updating, removing, and managing objects in a task management system.

### Tasks:

1. Refactoring Code to Use Classes (15 mins):
Refactor your previous task management system to use the `Task` class for representing individual tasks.

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority

# Refactor your system to create tasks using this class.
```

2. Adding a `mark_complete` Method (15 mins):
Add a method `mark_complete()` to the `Task` class that marks the task as completed.

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority
        self.completed = False

    def mark_complete(self):
        self.completed = True
```

3. Displaying Task Status (15 mins):
Add a method `show_status()` that displays whether a task is completed or not.

```
class Task:
    def show_status(self):
```

```
        status = 'Completed' if self.completed else 'Incomplete'
        print(f'{self.title}: {status}')
```

4. Adding a Task to a List (15 mins):
Write a function that allows users to create a new task and add it to a list of tasks.

```
tasks = []

def add_task():
    title = input('Enter task title: ')
    due_date = input('Enter due date: ')
    priority = input('Enter priority: ')
    task = Task(title, due_date, priority)
    tasks.append(task)
```

5. Viewing All Tasks (15 mins):
Write a function `view_tasks()` to display all tasks in the list, including their completion status.

```
def view_tasks():
    for task in tasks:
        task.show_status()
```

6. Marking a Task as Completed (15 mins):
Extend your program to allow users to mark a task as completed by selecting it from the list.

```
def mark_task_completed():
    task_num = int(input('Enter task number to mark complete: '))
    if 0 < task_num <= len(tasks):
        tasks[task_num - 1].mark_complete()
```

7. Removing a Task (15 mins):
Add a function `remove_task()` that allows users to delete a task from the list.

```
def remove_task():
    task_num = int(input('Enter task number to remove: '))
    if 0 < task_num <= len(tasks):
        tasks.pop(task_num - 1)
```

8. Searching for a Task by Title (15 mins):
Write a function that searches for a task by its title and displays its details.

```
def search_task_by_title(title):
    for task in tasks:
        if task.title.lower() == title.lower():
            task.show_status()
```

9. Sorting Tasks by Due Date (15 mins):
Add functionality to sort tasks by their due date.

```
tasks.sort(key=lambda task: task.due_date)
view_tasks()
```

10. Updating a Task (15 mins):
Write a function `update_task()` to allow users to change the title or due date of a task.

```
def update_task(task_num):
    new_title = input('Enter new title: ')
    new_due_date = input('Enter new due date: ')
    tasks[task_num - 1].title = new_title
    tasks[task_num - 1].due_date = new_due_date
```

11. Saving Tasks to a File (15 mins):
Add functionality to save the list of tasks to a file.

```
def save_tasks_to_file(filename='tasks.txt'):
    with open(filename, 'w') as file:
        for task in tasks:

file.write(f'{task.title},{task.due_date},{task.priority},{task.c
ompleted}\n')
```

12. Loading Tasks from a File (15 mins):
Write a function to load tasks from a file and repopulate the task list.

```
def load_tasks_from_file(filename='tasks.txt'):
    with open(filename, 'r') as file:
        for line in file:
            title, due_date, priority, completed =
line.strip().split(',')
            task = Task(title, due_date, priority)
            task.completed = completed == 'True'
            tasks.append(task)
```

13. Displaying Completed Tasks Only (15 mins):
Write a function to display only tasks that are marked as completed.

```
def show_completed_tasks():
    for task in tasks:
        if task.completed:
            task.show_status()
```

14. Displaying Incomplete Tasks Only (15 mins):
Write a function to display only tasks that are incomplete.

```
def show_incomplete_tasks():
    for task in tasks:
```

```
        if not task.completed:
            task.show_status()
```

15. Exiting the Task Management System (15 mins):
Write a function to exit the program and save tasks before exiting.

```
def exit_program():
    save_tasks_to_file()
    print('Tasks saved. Exiting program.')
    exit()
```

## Assignment Task:

- **Task Management System with Object-Oriented Refactoring** (45 mins):

Refactor the task management system to perform the following:

1. Manage tasks as objects.

2. Allow users to create, update, delete, and search tasks.

3. Implement file saving and loading to store and retrieve tasks between sessions.

4. Sort tasks by due date and filter them based on completion status.

```
import os

# Base Task class
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority
        self.completed = False

    def mark_complete(self):
        self.completed = True

    def update_task(self, title=None, due_date=None,
priority=None):
        if title:
            self.title = title
        if due_date:
            self.due_date = due_date
        if priority:
            self.priority = priority

    def show_task(self):
```

```
        status = "Completed" if self.completed else "Incomplete"
        print(f"Task: {self.title}, Due: {self.due_date},
Priority: {self.priority}, Status: {status}")

# Task Management System class
class TaskManagementSystem:
    def __init__(self):
        self.tasks = []

    def add_task(self):
        title = input("Enter task title: ")
        due_date = input("Enter task due date (YYYY-MM-DD): ")
        priority = input("Enter task priority (Low, Medium,
High): ")
        task = Task(title, due_date, priority)
        self.tasks.append(task)

    def update_task(self):
        self.view_tasks()
        task_num = int(input("Enter task number to update: "))
        if 0 < task_num <= len(self.tasks):
            task = self.tasks[task_num - 1]
            new_title = input(f"Enter new title (leave blank to
keep '{task.title}'): ")
            new_due_date = input(f"Enter new due date (leave
blank to keep '{task.due_date}'): ")
            new_priority = input(f"Enter new priority (leave
blank to keep '{task.priority}'): ")
            task.update_task(new_title or None, new_due_date or
None, new_priority or None)
            print("Task updated successfully.")
        else:
            print("Invalid task number.")

    def delete_task(self):
        self.view_tasks()
        task_num = int(input("Enter task number to delete: "))
        if 0 < task_num <= len(self.tasks):
            self.tasks.pop(task_num - 1)
            print("Task deleted successfully.")
        else:
            print("Invalid task number.")

    def search_task(self):
```

```
        search_term = input("Enter task title to search:
").lower()
        for task in self.tasks:
            if search_term in task.title.lower():
                task.show_task()

    def view_tasks(self):
        if not self.tasks:
            print("No tasks available.")
        else:
            for i, task in enumerate(self.tasks, 1):
                print(f"Task {i}:")
                task.show_task()

    def mark_task_completed(self):
        self.view_tasks()
        task_num = int(input("Enter task number to mark as
completed: "))
        if 0 < task_num <= len(self.tasks):
            self.tasks[task_num - 1].mark_complete()
            print("Task marked as completed.")
        else:
            print("Invalid task number.")

    def show_completed_tasks(self):
        for task in self.tasks:
            if task.completed:
                task.show_task()

    def show_incomplete_tasks(self):
        for task in self.tasks:
            if not task.completed:
                task.show_task()

    def sort_tasks_by_due_date(self):
        self.tasks.sort(key=lambda task: task.due_date)
        print("Tasks sorted by due date.")

    def save_tasks(self, filename="tasks.txt"):
        with open(filename, 'w') as file:
            for task in self.tasks:
```

```
                task_line =
f"{task.title},{task.due_date},{task.priority},{task.completed}\n
"
                file.write(task_line)
        print(f"Tasks saved to {filename}.")

    def load_tasks(self, filename="tasks.txt"):
        if os.path.exists(filename):
            with open(filename, 'r') as file:
                for line in file:
                    title, due_date, priority, completed =
line.strip().split(',')
                    task = Task(title, due_date, priority)
                    task.completed = completed == 'True'
                    self.tasks.append(task)
            print(f"Tasks loaded from {filename}.")
        else:
            print(f"{filename} does not exist.")

    def run(self):
        while True:
            print("\nTask Management System")
            print("1. Add Task")
            print("2. Update Task")
            print("3. Delete Task")
            print("4. Search Task")
            print("5. View All Tasks")
            print("6. Mark Task as Completed")
            print("7. View Completed Tasks")
            print("8. View Incomplete Tasks")
            print("9. Sort Tasks by Due Date")
            print("10. Save Tasks to File")
            print("11. Load Tasks from File")
            print("12. Exit")

            choice = input("Choose an option: ")
            if choice == "1":
                self.add_task()
            elif choice == "2":
                self.update_task()
            elif choice == "3":
                self.delete_task()
            elif choice == "4":
```

```
                self.search_task()
            elif choice == "5":
                self.view_tasks()
            elif choice == "6":
                self.mark_task_completed()
            elif choice == "7":
                self.show_completed_tasks()
            elif choice == "8":
                self.show_incomplete_tasks()
            elif choice == "9":
                self.sort_tasks_by_due_date()
            elif choice == "10":
                self.save_tasks()
            elif choice == "11":
                self.load_tasks()
            elif choice == "12":
                print("Exiting the Task Management System.")
                break
            else:
                print("Invalid choice. Please try again.")

if __name__ == "__main__":
    task_system = TaskManagementSystem()
    task_system.run()
```

# Lab Manual for Week 5: Object-Oriented Programming – Day 5

## Day 5: Project Day – Advanced Task Management System

Time: 3 Hours

Learning Outcomes:
- Apply object-oriented programming concepts to a full-fledged project.
- Implement task and project management systems using classes, inheritance, and file I/O.
- Develop a functional Task Management System with advanced features.

### Mega Tasks:

1. Refactoring Task Management with Inheritance (45 mins):
Refactor the Task Management System to create a base class `Task` and a child class `ProjectTask`. Include the project name in tasks related to a project.

```
class ProjectTask(Task):
    def __init__(self, title, due_date, priority, project_name):
        super().__init__(title, due_date, priority)
        self.project_name = project_name

    def display_details(self):
        print(f'Project: {self.project_name} - Task:
{self.title}, Due Date: {self.due_date}, Priority:
{self.priority}')
```

2. Task Sorting by Multiple Criteria (45 mins):
Enhance the task management system by adding a feature to sort tasks by priority and due date. Allow the user to choose which criteria to sort by.

```
def sort_tasks(criteria):
    if criteria == 'priority':
        tasks.sort(key=lambda task: task.priority)
    elif criteria == 'due_date':
        tasks.sort(key=lambda task: task.due_date)

# Example usage
sort_tasks('priority')
```

3. Project-Based Task Filtering (45 mins):
Add a feature that allows users to filter and display tasks related to a specific project. This is especially useful when managing multiple projects.

```
def filter_by_project(project_name):
    for task in tasks:
        if isinstance(task, ProjectTask) and task.project_name ==
project_name:
            task.show_task()
```

4. Implementing a Comprehensive Save/Load Feature (45 mins):
Improve the file-saving and loading functionality to support both regular tasks and project tasks. Ensure the system can properly handle different task types when saving/loading.

```
def save_tasks_to_file(filename='tasks.txt'):
    with open(filename, 'w') as file:
        for task in tasks:
            if isinstance(task, ProjectTask):

file.write(f'project,{task.title},{task.due_date},{task.priority}
,{task.completed},{task.przoject_name}\n')
            else:

file.write(f'task,{task.title},{task.due_date},{task.priority},{t
ask.completed}\n')

# Load tasks accordingly
# Check if the first value is 'project' or 'task' to decide how
to load each task.
```

## Incremental Project Tasks:

In addition to the mega tasks, continue building the task management system by incrementally improving it with the following features:

1. Add Task Completion Feature (30 mins):
Add a method to mark tasks as completed and filter completed/incomplete tasks.

2. Search Tasks by Title (30 mins):
Implement a feature to search tasks by their title and display results.

3. Update Task Details (30 mins):
Allow users to update task titles, due dates, and priorities.

4. Implement Task Deletion (30 mins):
Enable users to delete tasks from the system.

5. Improve User Interface (30 mins):
Enhance the text-based interface to provide clear prompts and feedback for user actions.

Solution:

**1. Add Task Completion Feature:**

This feature allows marking tasks as completed and filtering completed or incomplete tasks.

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority
        self.completed = False

    def mark_complete(self):
        self.completed = True

    def show_task(self):
        status = "Completed" if self.completed else "Incomplete"
        print(f"Task: {self.title}, Due Date: {self.due_date},
Priority: {self.priority}, Status: {status}")

def show_completed_tasks(tasks):
    for task in tasks:
        if task.completed:
            task.show_task()

def show_incomplete_tasks(tasks):
    for task in tasks:
        if not task.completed:
            task.show_task()
```

**2. Search Tasks by Title:**

This feature allows searching tasks by their title.

```
def search_task_by_title(tasks, title):
    found_tasks = [task for task in tasks if title.lower() in
task.title.lower()]
    if found_tasks:
        for task in found_tasks:
            task.show_task()
    else:
        print(f"No tasks found with title: {title}")
```

**3. Update Task Details:**

This feature allows users to update task titles, due dates, and priorities.

```
def update_task_details(tasks, task_num):
    if 0 < task_num <= len(tasks):
        task = tasks[task_num - 1]
        new_title = input(f"Enter new title (or press Enter to
keep '{task.title}'): ") or task.title
        new_due_date = input(f"Enter new due date (or press Enter
to keep '{task.due_date}'): ") or task.due_date
        new_priority = input(f"Enter new priority (or press Enter
to keep '{task.priority}'): ") or task.priority

        task.title = new_title
        task.due_date = new_due_date
        task.priority = new_priority
        print("Task updated successfully.")
    else:
        print("Invalid task number.")
```

**4. Implement Task Deletion:**

This feature allows users to delete tasks from the system.

```
def delete_task(tasks, task_num):
    if 0 < task_num <= len(tasks):
        deleted_task = tasks.pop(task_num - 1)
        print(f"Task '{deleted_task.title}' deleted
successfully.")
    else:
        print("Invalid task number.")
```

**5. Improve User Interface:**

Enhance the text-based interface with clear prompts and feedback for user actions.

```
def task_management_menu(tasks):
    while True:
        print("\nTask Management System:")
        print("1. Add a new task")
        print("2. View all tasks")
        print("3. Search tasks by title")
        print("4. Mark task as completed")
        print("5. Show completed tasks")
        print("6. Show incomplete tasks")
        print("7. Update task details")
        print("8. Delete a task")
        print("9. Exit")
```

```
        choice = input("Choose an option: ")

        if choice == '1':
            add_task(tasks)
        elif choice == '2':
            view_tasks(tasks)
        elif choice == '3':
            title = input("Enter task title to search: ")
            search_task_by_title(tasks, title)
        elif choice == '4':
            task_num = int(input("Enter task number to mark as
completed: "))
            tasks[task_num - 1].mark_complete()
        elif choice == '5':
            show_completed_tasks(tasks)
        elif choice == '6':
            show_incomplete_tasks(tasks)
        elif choice == '7':
            task_num = int(input("Enter task number to update:
"))
            update_task_details(tasks, task_num)
        elif choice == '8':
            task_num = int(input("Enter task number to delete:
"))
            delete_task(tasks, task_num)
        elif choice == '9':
            print("Exiting Task Management System.")
            break
        else:
            print("Invalid choice, please try again.")
Putting it all together:
# Example of putting it all together:

tasks = []

def add_task(tasks):
    title = input("Enter task title: ")
    due_date = input("Enter task due date (YYYY-MM-DD): ")
    priority = input("Enter task priority (Low, Medium, High): ")
    task = Task(title, due_date, priority)
    tasks.append(task)

def view_tasks(tasks):
```

```
    if tasks:
        for i, task in enumerate(tasks, 1):
            print(f"Task {i}:")
            task.show_task()
    else:
        print("No tasks available.")

# Starting the task management menu
task_management_menu(tasks)
```

**Explanation:**

1. **Add Task Completion Feature:** You can mark tasks as completed and display either completed or incomplete tasks.
2. **Search Tasks by Title:** You can search tasks by their title, and the system will display matching tasks.
3. **Update Task Details:** You can update a task's title, due date, or priority.
4. **Task Deletion:** You can delete a task from the list.
5. **Improved User Interface:** A menu system provides clear options for managing tasks, ensuring a smooth user experience.

This code builds a robust task management system with features for managing tasks efficiently.

# Lab Manual for Week 6: Introduction to APIs – Day 1

## Day 1: Introduction to APIs

Time: 3 Hours

Learning Outcomes:
- Understand what an API is and how it works.
- Make HTTP requests using Python's `requests` library.
- Work with JSON data returned by APIs.
- Fetch data from public APIs.

### Tasks:

1. What is an API? (15 mins):
Write a short description of what an API is.

2. Install the `requests` Library (15 mins):
Use `pip` to install the `requests` library.

```
pip install requests
```

3. Making a Simple HTTP GET Request (15 mins):
Write Python code to make an HTTP GET request to an API endpoint using the `requests` library.

```
import requests

response = requests.get('https://api.github.com')
print(response.status_code)
```

4. Accessing the Response Content (15 mins):
Access and print the content returned by the API request.

```
print(response.content)
```

5. Accessing JSON Data (15 mins):
Convert the API response content to JSON and print it.

```
data = response.json()
print(data)
```

6. Fetching Specific Data from JSON (15 mins):
Extract specific information from the JSON response. (e.g., Extract the 'current_user_url' from GitHub API response).

```
current_user_url = data['current_user_url']
print(current_user_url)
```

7. Handling HTTP Errors (15 mins):
Write code to handle HTTP errors gracefully using `try-except`.

```
try:
    response = requests.get('https://api.github.com')
    response.raise_for_status()
except requests.exceptions.HTTPError as err:
    print(f'HTTP error occurred: {err}')
```

8. Using API Parameters (15 mins):
Make an API request with query parameters. (e.g., Search for a user on GitHub).

```
response = requests.get('https://api.github.com/search/users',
params={'q': 'octocat'})
data = response.json()
print(data['items'][0])
```

9. Using Headers in API Requests (15 mins):
Add headers to your API requests, such as User-Agent.

```
headers = {'User-Agent': 'MyApp'}
response = requests.get('https://api.github.com',
headers=headers)
print(response.status_code)
```

10. Fetching Weather Data from an API (15 mins):
Use a free weather API to fetch and display the current weather of a city.

```
response =
requests.get('https://api.openweathermap.org/data/2.5/weather?q=L
ondon&appid=YOUR_API_KEY')
data = response.json()
print(data['weather'][0]['description'])
```

11. Parsing Complex JSON (15 mins):
Parse and display nested JSON data from the API response.

```
weather_description = data['weather'][0]['description']
print(f'The weather is {weather_description}')
```

12. Rate Limiting in APIs (15 mins):
Research and explain how rate limiting works in APIs. Implement a check for status codes that indicate rate limiting.

13. Fetching Data from Another Public API (15 mins):
Make an API request to a different public API (e.g., a joke or motivational quotes API).

```
response = requests.get('https://api.quotable.io/random')
quote = response.json()
print(quote['content'])
```

14. Using API Keys (15 mins):
Explain what API keys are, and use an API key in an authenticated API request.

15. Wrapping API Requests in a Function (15 mins):
Create a reusable Python function that takes an API URL and returns the JSON response.

```
def fetch_data(url):
    response = requests.get(url)
    return response.json()

# Test the function
print(fetch_data('https://api.github.com'))
```

## Assignment Task:

- **Fetch and Display Weather Information** (45 mins):

Write a Python program that:

1. Fetches the current weather of a city using a public weather API.

2. Displays the temperature, weather description, and humidity.

3. Handles any errors that may occur during the request.

## Solution Code:

```
import requests

def fetch_weather(city):
    api_key = 'YOUR_API_KEY'
    url =
f'https://api.openweathermap.org/data/2.5/weather?q={city}&appid=
{api_key}&units=metric'
    try:
        response = requests.get(url)
        response.raise_for_status()
        data = response.json()
        temperature = data['main']['temp']
        weather_description = data['weather'][0]['description']
        humidity = data['main']['humidity']
        print(f'Temperature: {temperature}°C')
        print(f'Weather: {weather_description}')
```

```
        print(f'Humidity: {humidity}%')
    except requests.exceptions.HTTPError as err:
        print(f'HTTP error occurred: {err}')
    except Exception as err:
        print(f'An error occurred: {err}')

# Fetch weather for London
fetch_weather('London')
```

# Lab Manual for Week 6: Introduction to APIs – Day 2

## Day 2: External Libraries and Packages

Time: 3 Hours

Learning Outcomes:
- Understand how to install and use external libraries in Python.
- Learn how to use `pip` to manage libraries.
- Utilize external libraries for different functionalities.

### Tasks:

1. What are External Libraries? (15 mins):
Write a short description of what external libraries are and why they are useful in Python.

2. Installing a Package with `pip` (15 mins):
Use `pip` to install an external library of your choice (e.g., `requests`).

```
pip install requests
```

3. Listing Installed Packages (15 mins):
Write a Python script to list all installed packages using `pip`.

```
!pip list
```

4. Installing a Specific Version of a Package (15 mins):
Install a specific version of a package using `pip` (e.g., install version 2.26.0 of `requests`).

```
pip install requests==2.26.0
```

5. Importing an External Library (15 mins):
Import an external library into a Python script and print its version.

```
import requests
print(requests.__version__)
```

6. Using `os` Library to List Files (15 mins):
Use the `os` library to list files in the current directory.

```
import os
print(os.listdir())
```

7. Fetching Data with `requests` (15 mins):

Write a Python script that uses the `requests` library to make an HTTP GET request to a public API.

```
import requests

response = requests.get('https://api.github.com')
print(response.status_code)
```

8. Parsing JSON Data (15 mins):
Use `requests` to fetch data from an API and convert the response to JSON.

```
data = response.json()
print(data)
```

9. Using `schedule` Library to Schedule a Task (15 mins):
Install and use the `schedule` library to schedule a task that prints a message every minute.

```
import schedule
import time

def job():
    print('Task running...')

schedule.every(1).minutes.do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

10. Installing `numpy` for Mathematical Operations (15 mins):
Install `numpy` and use it to create and manipulate arrays.

```
pip install numpy

import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

11. Sending Emails with `smtplib` (15 mins):
Use the `smtplib` library to send an email programmatically.

```
import smtplib

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login('youremail@gmail.com', 'yourpassword')
server.sendmail('from@example.com', 'to@example.com', 'Subject:
```

```
Test Email\nThis is a test email.')
server.quit()
```

12. Automating Tasks with `pyautogui` (15 mins):
Install and use the `pyautogui` library to automate a simple task such as opening a browser window.

```
pip install pyautogui
```

```
import pyautogui
pyautogui.hotkey('ctrl', 't')  # Open a new tab in browser
```

13. Working with Dates and Time using `datetime` (15 mins):
Use the `datetime` library to print the current date and time.

```
from datetime import datetime
print(datetime.now())
```

14. Using `matplotlib` for Plotting (15 mins):
Install `matplotlib` and use it to create a simple line plot.

```
pip install matplotlib
```

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 35]
plt.plot(x, y)
plt.show()
```

15. Creating a Virtual Environment (15 mins):
Create a virtual environment using `venv` and activate it.

```
python -m venv myenv
source myenv/bin/activate  # On Windows: myenv\Scripts\activate
```

## Assignment Task:

- **Automate Sending an Email Using `smtplib`** (45 mins):

Write a Python program that:

1. Logs into an email account using `smtplib`.

2. Sends an email with a subject and body to a recipient.

3. Handles any errors that may occur during the process.

## Solution Code:

```
import smtplib
```

```
def send_email(subject, body, to_email):
    smtp_server = 'smtp.gmail.com'
    smtp_port = 587
    email_user = 'youremail@gmail.com'
    email_password = 'yourpassword'

    try:
        server = smtplib.SMTP(smtp_server, smtp_port)
        server.starttls()
        server.login(email_user, email_password)

        message = f'Subject: {subject}\n\n{body}'
        server.sendmail(email_user, to_email, message)
        server.quit()
        print('Email sent successfully!')
    except Exception as e:
        print(f'Error: {e}')

# Usage
send_email('Test Subject', 'This is the body of the test email.',
'recipient@example.com')
```

# Lab Manual for Week 6: Introduction to APIs – Day 3

## Day 3: Integrating APIs into the Project

Time: 3 Hours

Learning Outcomes:
- Learn how to integrate external APIs into a Python project.
- Apply API data to enhance project functionality.
- Work with JSON data returned by APIs and manipulate it.

### Tasks:

1. 1. Review Your To-Do List Project (15 mins):
- Write a short summary of how your current To-Do List project works.

2. 2. Adding an API Integration Plan (15 mins):
- Write a plan to integrate an external API into your To-Do List project. Mention which API you'll use and how you will use the data.

3. 3. Selecting an API (15 mins):
- Choose a public API that you will integrate into the project. Some options include a weather API, quote API, or currency conversion API.

```
For example: https://api.quotable.io/random for a quote API
```

4. 4. Making an API Request (15 mins):
- Make an HTTP GET request to the chosen API using `requests` and print the response status code.

```
import requests

response = requests.get('https://api.quotable.io/random')
print(response.status_code)
```

5. 5. Parsing API Response (15 mins):
- Convert the API response content to JSON and print the data.

```
data = response.json()
print(data)
```

6. 6. Extracting Data from the API (15 mins):

- Extract specific information from the JSON response. For example, extract a quote from a quotes API.

```
quote = data['content']
print(f'Quote of the day: {quote}')
```

7. 7. Adding API Data to the To-Do List (15 mins):
- Integrate the API data into your To-Do List. For example, display a quote or weather information when viewing tasks.

```
# Example integration
print(f'Today's quote: {quote}')
```

8. 8. Handling API Errors (15 mins):
- Handle errors that may occur during the API request using `try-except`.

```
try:
    response = requests.get('https://api.quotable.io/random')
    response.raise_for_status()
except requests.exceptions.HTTPError as err:
    print(f'HTTP error occurred: {err}')
```

9. 9. Fetching Data with Parameters (15 mins):
- Make an API request with parameters (if applicable). For example, request a quote from a specific author.

```
response = requests.get('https://api.quotable.io/quotes',
params={'author': 'Einstein'})
data = response.json()
print(data)
```

10. 10. Using API Keys (15 mins):
- If the API requires authentication, use an API key to authenticate your requests.

```
api_key = 'YOUR_API_KEY'
response =
requests.get(f'https://api.example.com/data?apikey={api_key}')
```

11. 11. Automating API Calls (15 mins):
- Use a loop or scheduler to make API requests at regular intervals.

```
import schedule
import time

schedule.every(10).minutes.do(fetch_quote)

while True:
```

```
    schedule.run_pending()
    time.sleep(1)
```

12. 12. Storing API Data in Files (15 mins):
- Store the API data in a file for future use.

```
with open('quote.txt', 'w') as file:
    file.write(quote)
```

13. 13. Displaying API Data in the Project (15 mins):
- Display the API data within your To-Do List project (e.g., show a motivational quote when displaying tasks).

```
def display_task_with_quote(task, quote):
    print(f'Task: {task} - {quote}')
```

14. 14. API Data Validation (15 mins):
- Check if the API data meets the expected structure or values before using it in your project.

```
if 'content' in data:
    print('Valid response')
else:
    print('Invalid response')
```

15. 15. Refactoring API Calls into Functions (15 mins):
- Refactor your API request code into reusable functions.

```
def fetch_quote():
    response = requests.get('https://api.quotable.io/random')
    return response.json()
```

## Assignment Task:

- **Integrate an API into the To-Do List Project** (45 mins):

Write a Python program that:

1. Fetches data from a public API.

2. Displays the data alongside your To-Do List tasks.

3. Handles errors that may occur during the API request.

## Solution Code:

```
import requests

# Function to fetch a motivational quote from the Quotable API
def fetch_quote():
    try:
```

```
        response = requests.get('https://api.quotable.io/random')
        response.raise_for_status()
        data = response.json()
        return data['content']
    except requests.exceptions.RequestException as err:
        print(f"Error fetching quote: {err}")
        return "No quote available."

# Example To-Do List
tasks = ['Finish project', 'Go shopping', 'Read a book']

# Display tasks with a quote
quote = fetch_quote()
for task in tasks:
    print(f"Task: {task} - Quote: {quote}")
```

# Lab Manual for Week 6: Introduction to APIs – Day 4

## Day 4: Project Enhancement with Libraries

Time: 3 Hours

Learning Outcomes:
- Understand how to enhance your project using external libraries.
- Learn how to schedule tasks using the `schedule` library.
- Use `datetime` to handle dates in tasks.

### Tasks:

1. Install the `schedule` Library (15 mins):
Use `pip` to install the `schedule` library.

```
pip install schedule
```

2. Scheduling a Simple Task (15 mins):
Write a Python script to schedule a task that prints 'Hello World' every minute.

```
import schedule
import time

schedule.every(1).minutes.do(lambda: print('Hello World'))

while True:
    schedule.run_pending()
    time.sleep(1)
```

3. Adding a `datetime` Library (15 mins):
Import the `datetime` library and print the current date and time.

```
from datetime import datetime

print(datetime.now())
```

4. Scheduling a Task at a Specific Time (15 mins):
Use `schedule` to schedule a task at a specific time (e.g., every day at 8:00 AM).

```
schedule.every().day.at('08:00').do(lambda: print('Good
morning!'))
```

5. Adding a Deadline to a Task (15 mins):
Use `datetime` to add a deadline to a task and print how many days remain until the deadline.

```
from datetime import datetime

due_date = datetime(2024, 12, 31)
current_date = datetime.now()
days_left = (due_date - current_date).days
print(f'Days until deadline: {days_left}')
```

6. Scheduling a Task with a Function (15 mins):
Create a function that prints a custom message and schedule it to run every 5 minutes.

```
def my_task():
    print('This is my custom task')

schedule.every(5).minutes.do(my_task)
```

7. Handling Task Scheduling in a Loop (15 mins):
Modify the previous code to ensure the task runs inside an infinite loop.

```
while True:
    schedule.run_pending()
    time.sleep(1)
```

8. Handling Dates and Deadlines in To-Do List (15 mins):
Use the `datetime` library to manage task deadlines in your To-Do List project.

```
# Example:
task_deadline = datetime(2024, 10, 25)
print(f'Task deadline: {task_deadline}')
```

9. Displaying Upcoming Tasks (15 mins):
Write code to display tasks that have upcoming deadlines within the next 7 days.

```
def show_upcoming_tasks(tasks):
    current_date = datetime.now()
    for task in tasks:
        if (task['deadline'] - current_date).days <= 7:
            print(task)
```

10. Scheduling a Reminder Task (15 mins):
Use the `schedule` library to create a daily reminder to check the To-Do List.

```
schedule.every().day.at('09:00').do(lambda: print('Check your To-Do List'))
```

11. Storing Scheduled Tasks in a File (15 mins):
Write a Python script that saves all scheduled tasks in a text file.

```
with open('scheduled_tasks.txt', 'w') as file:
    for task in schedule.jobs:
        file.write(str(task))
```

12. Loading Scheduled Tasks from a File (15 mins):
Write a Python script that reads scheduled tasks from a file and schedules them.

```
with open('scheduled_tasks.txt', 'r') as file:
    tasks = file.readlines()
    for task in tasks:
        # Schedule tasks accordingly
```

13. Creating a Deadline Notification (15 mins):
Use `datetime` to notify when a task's deadline is approaching within the next 2 days.

```
if (task_deadline - datetime.now()).days <= 2:
    print('Deadline approaching!')
```

14. Combining Scheduling with To-Do List (15 mins):
Enhance your To-Do List project by scheduling reminders to complete tasks using `schedule`.

```
schedule.every().day.at('12:00').do(lambda: print('Complete your tasks!'))
```

15. Refactoring Code for Reusability (15 mins):
Refactor the scheduling and `datetime` code into reusable functions.

```
def schedule_task(task, time):
    schedule.every().day.at(time).do(lambda: print(f'Remember to do: {task}'))
```

## Assignment Task:

- **Schedule Task Reminders for Your To-Do List** (45 mins):

Write a Python program that:

1. Schedules a reminder to check the To-Do List every day at 9:00 AM.

2. Tracks task deadlines and notifies when a deadline is within the next 2 days.

3. Saves the scheduled reminders to a file for future use.

## Solution Code:

```
import schedule
```

```
import time
from datetime import datetime

# Example To-Do List with deadlines
tasks = [
    {'task': 'Submit project', 'deadline': datetime(2024, 10,
25)},
    {'task': 'Prepare for meeting', 'deadline': datetime(2024,
10, 23)}
]

# Function to check deadlines
def check_deadlines():
    current_date = datetime.now()
    for task in tasks:
        days_left = (task['deadline'] - current_date).days
        if days_left <= 2:
            print(f"Deadline approaching for task: {task['task']}
- {days_left} days left.")

# Schedule daily reminder to check To-Do List
schedule.every().day.at('09:00').do(lambda: print('Check your To-
Do List'))

# Schedule to check deadlines
schedule.every().day.at('12:00').do(check_deadlines)

# Run the scheduled tasks
while True:
    schedule.run_pending()
    time.sleep(1)
```

# Lab Manual for Week 6: Introduction to APIs – Day 5

## Day 5: Project Day – API Integration and Task Scheduling

Time: 3 Hours

Learning Outcomes:
- Apply the knowledge of APIs and task scheduling to enhance the To-Do List project.
- Integrate external APIs and automate tasks using the `schedule` library.
- Use `datetime` for managing task deadlines and reminders.

### Mega Tasks:

1. Full API Integration into To-Do List (45 mins):
Integrate an external API (e.g., weather or motivational quotes) into your To-Do List project. Display the fetched data every time the user views their tasks.

```
import requests

# Example of fetching a quote from an API and displaying it
alongside tasks
quote =
requests.get('https://api.quotable.io/random').json()['content']
tasks = ['Finish project', 'Read a book']
for task in tasks:
    print(f'Task: {task} - Quote: {quote}')
```

2. Implement Deadline Notifications (45 mins):
Use the `datetime` library to notify the user when a task deadline is approaching (e.g., within the next 2 days).

```
from datetime import datetime

tasks = [{'name': 'Submit report', 'deadline': datetime(2024, 10,
25)}]
for task in tasks:
    days_left = (task['deadline'] - datetime.now()).days
    if days_left <= 2:
        print(f'Deadline approaching: {task['name']} -
{days_left} days left.')
```

3. Schedule Daily Task Reminders (45 mins):

Use the `schedule` library to schedule a daily reminder for users to check their tasks.

```
import schedule
import time

schedule.every().day.at('09:00').do(lambda: print('Check your To-
Do List!'))
while True:
    schedule.run_pending()
    time.sleep(1)
```

4. Save and Load Scheduled Reminders (45 mins):
Save the scheduled task reminders to a file and load them when the program starts to ensure persistence.

```
schedule_file = 'schedule.txt'

# Save schedule to file
with open(schedule_file, 'w') as file:
    for job in schedule.jobs:
        file.write(str(job))

# Load schedule from file
with open(schedule_file, 'r') as file:
    saved_jobs = file.readlines()
    for job in saved_jobs:
        # Re-create schedule (depends on saved job details)
```

## Incremental Project Tasks:

In addition to the mega tasks, incrementally improve the To-Do List project by adding the following features:

1. Task Completion Status (30 mins):
Add a feature to mark tasks as completed and filter completed/incomplete tasks.

```
tasks = [{'task': 'Finish project', 'completed': False}, {'task':
'Read a book', 'completed': True}]

def mark_task_completed(task):
    task['completed'] = True

def show_completed_tasks(tasks):
    for task in tasks:
        if task['completed']:
            print(f"Completed: {task['task']}")
```

```
mark_task_completed(tasks[0])
show_completed_tasks(tasks)
```

2. Task Search by Title (30 mins):
Implement a search feature to find tasks by their title.

```
def search_task_by_title(tasks, title):
    found_tasks = [task for task in tasks if title.lower() in
task['task'].lower()]
    for task in found_tasks:
        print(task)

tasks = [{'task': 'Finish project'}, {'task': 'Read a book'}]
search_task_by_title(tasks, 'finish')
```

3. Task Sorting (30 mins):
Sort tasks by due date or priority using `sorted()` in Python.

```
tasks = [{'task': 'Finish project', 'due_date': '2024-10-25'},
{'task': 'Read a book', 'due_date': '2024-10-20'}]

sorted_tasks = sorted(tasks, key=lambda x: x['due_date'])
for task in sorted_tasks:
    print(task)
```

4. Task Deletion (30 mins):
Allow users to delete tasks from the list.

```
def delete_task(tasks, task_name):
    tasks = [task for task in tasks if task['task'] != task_name]
    return tasks

tasks = [{'task': 'Finish project'}, {'task': 'Read a book'}]
tasks = delete_task(tasks, 'Read a book')
print(tasks)
```

5. Display Upcoming Tasks (30 mins):
Display only the tasks with deadlines within the next 7 days.

```
from datetime import datetime, timedelta

tasks = [{'task': 'Submit report', 'deadline': datetime(2024, 10,
25)},
         {'task': 'Prepare for meeting', 'deadline':
datetime(2024, 10, 20)}]

def show_upcoming_tasks(tasks):
    current_date = datetime.now()
    upcoming_tasks = [task for task in tasks if (task['deadline']
- current_date).days <= 7]
    for task in upcoming_tasks:
        print(f"Upcoming task: {task['task']} - Due in
{(task['deadline'] - current_date).days} days")

show_upcoming_tasks(tasks)
```

# Lab Manual for Week 7: Advanced Python Concepts – Day 1

## Day 1: Advanced Functions and Recursion

Time: 3 Hours

Learning Outcomes:
- Understand higher-order functions and how to use functions as arguments.
- Learn about lambda functions and their applications.
- Understand recursion and how it can be applied to solve problems.

### Tasks:

1. What are Higher-Order Functions? (15 mins):
Write a short description of what higher-order functions are in Python.

2. Writing a Higher-Order Function (15 mins):
Write a Python function that takes another function as an argument and applies it to a list of numbers.

```
def apply_function(numbers, func):
    return [func(n) for n in numbers]

def square(n):
    return n * n

numbers = [1, 2, 3, 4]
print(apply_function(numbers, square))
```

3. Using Lambda Functions (15 mins):
Write a lambda function to square numbers and use it within a higher-order function.

```
numbers = [1, 2, 3, 4]
print(apply_function(numbers, lambda n: n * n))
```

4. Lambda Function for Sorting (15 mins):
Write a Python script to sort a list of tuples (name, age) by age using a lambda function.

```
people = [('Alice', 25), ('Bob', 20), ('Charlie', 30)]
sorted_people = sorted(people, key=lambda person: person[1])
print(sorted_people)
```

5. What is Recursion? (15 mins):
Write a brief explanation of recursion and how it works.

6. Writing a Simple Recursive Function (15 mins):
Write a Python function that uses recursion to calculate the factorial of a number.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

7. Recursive Countdown (15 mins):
Write a recursive function that counts down from a given number to zero.

```
def countdown(n):
    if n == 0:
        print('Blast off!')
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

8. Understanding Base Case in Recursion (15 mins):
Explain the concept of the base case in recursion and why it is important.

9. Recursive Fibonacci Sequence (15 mins):
Write a recursive function to calculate the nth number in the Fibonacci sequence.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```
print(fibonacci(10))
```

10. Using `map()` with Lambda Functions (15 mins):
Use `map()` with a lambda function to double all numbers in a list.

```
numbers = [1, 2, 3, 4]
doubled = list(map(lambda n: n * 2, numbers))
print(doubled)
```

11. Writing a Function that Returns Another Function (15 mins):
Write a higher-order function that returns another function.

```
def make_multiplier(factor):
    return lambda x: x * factor

double = make_multiplier(2)
print(double(5))
```

12. Using `filter()` with Lambda Functions (15 mins):
Use `filter()` with a lambda function to filter out even numbers from a list.

```
numbers = [1, 2, 3, 4, 5, 6]
filtered = list(filter(lambda n: n % 2 != 0, numbers))
print(filtered)
```

13. Recursion vs Iteration (15 mins):
Write a short explanation comparing recursion and iteration, highlighting their pros and cons.

14. Factorial with Iteration (15 mins):
Rewrite the factorial function using iteration instead of recursion.

```
def iterative_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

```
print(iterative_factorial(5))
```

15. Recursion and Stack Overflow (15 mins):
Explain why recursion can lead to stack overflow in Python and how to prevent it.

## Assignment Task:

- **Recursive Sum of a List** (45 mins):

Write a Python program that uses recursion to:

1. Find the sum of all numbers in a list.

2. Ensure that the program can handle both empty and non-empty lists.

## Solution Code:

```
def recursive_sum(lst):
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + recursive_sum(lst[1:])

# Test the recursive sum function
numbers = [1, 2, 3, 4, 5]
print(f'Sum of the list: {recursive_sum(numbers)}')
```

# Lab Manual for Week 7: Advanced Python Concepts – Day 2

## Day 2: Introduction to Data Visualization

Time: 3 Hours

Learning Outcomes:
- Understand how to create charts using the `matplotlib` library.
- Learn to create line and bar charts.
- Customize charts by adding labels, titles, and legends.
- Apply data visualization to the To-Do List project.

### Tasks:

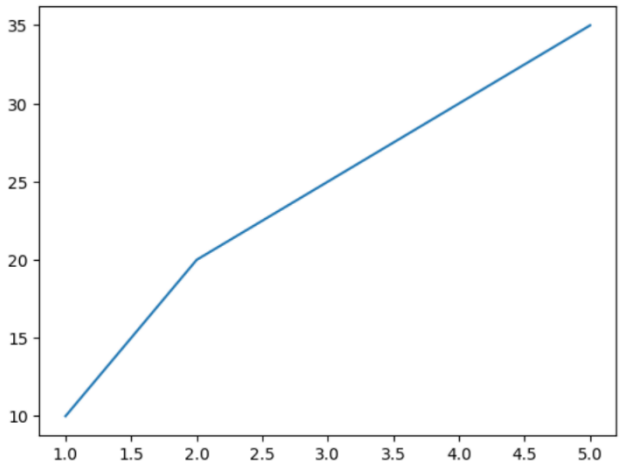
1. Installing Matplotlib (15 mins):
Use `pip` to install the `matplotlib` library.

```
pip install matplotlib
```

2. Importing Matplotlib (15 mins):
Import `matplotlib.pyplot` and create a simple plot of a list of numbers.

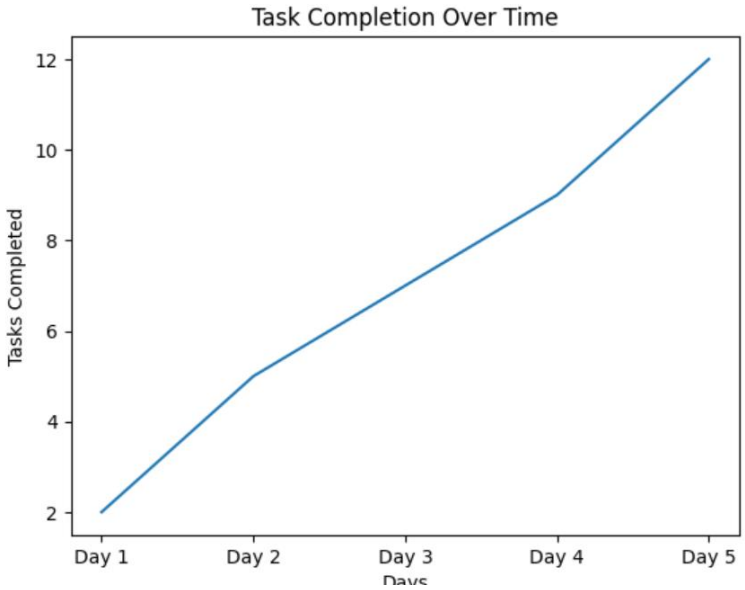
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 35]
plt.plot(x, y)
plt.show()
```



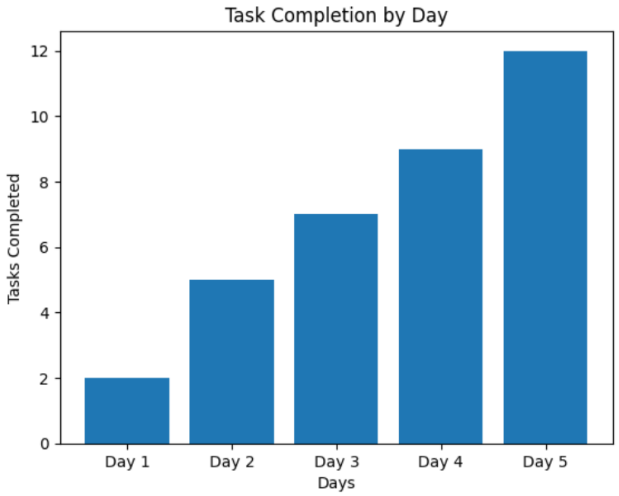
3. Creating a Line Chart (15 mins):
Use `matplotlib` to create a line chart that represents the growth of task completion over time.

```
x = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
y = [2, 5, 7, 9, 12]
plt.plot(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.show()
```



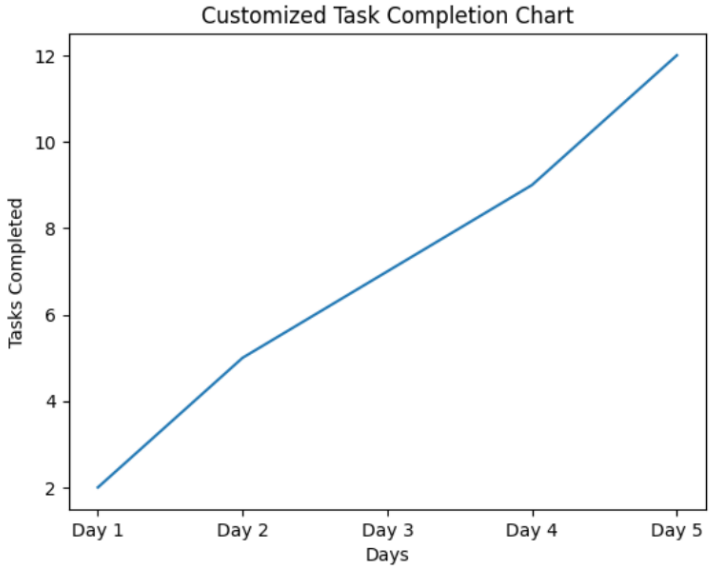

4. Creating a Bar Chart (15 mins):
Create a bar chart to represent the number of tasks completed each day.

```
plt.bar(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion by Day')
plt.show()
```



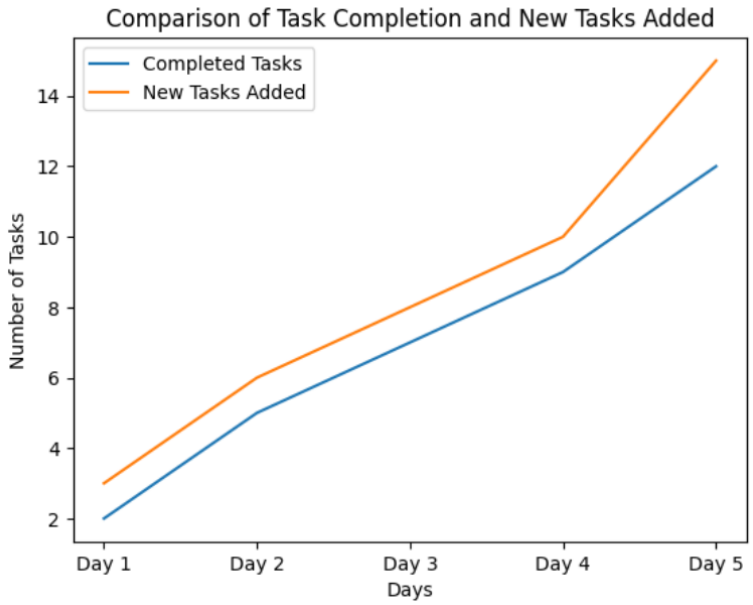
5. Adding Labels and Titles (15 mins):
Customize your chart by adding labels to the axes and a title to the chart.

```
plt.plot(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Customized Task Completion Chart')
plt.show()
```



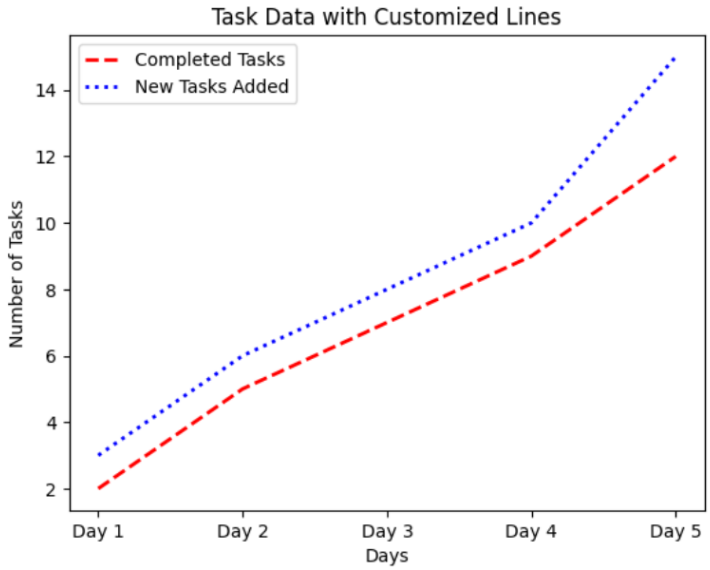

6. Plotting Multiple Lines (15 mins):
Plot multiple lines on the same chart to compare the number of tasks completed and the number of new tasks added over time.

```
y2 = [3, 6, 8, 10, 15]
plt.plot(x, y, label='Completed Tasks')
plt.plot(x, y2, label='New Tasks Added')
plt.xlabel('Days')
plt.ylabel('Number of Tasks')
plt.title('Comparison of Task Completion and New Tasks Added')
plt.legend()
plt.show()
```




7. Using Different Line Styles (15 mins):
Customize the appearance of your lines by changing their color, width, and style (e.g., dashed or dotted).

```
plt.plot(x, y, label='Completed Tasks', linestyle='--',
color='r', linewidth=2)
plt.plot(x, y2, label='New Tasks Added', linestyle=':',
color='b', linewidth=2)
plt.xlabel('Days')
plt.ylabel('Number of Tasks')
plt.title('Task Data with Customized Lines')
plt.legend()
plt.show()
```

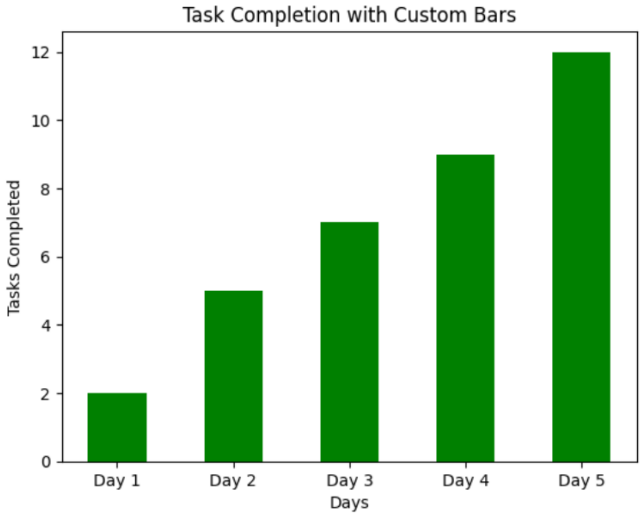


8. Creating a Pie Chart (15 mins):
Create a pie chart that shows the percentage of tasks completed versus tasks pending.

```
labels = 'Completed', 'Pending'
sizes = [75, 25]
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.title('Task Completion Status')
plt.show()
```


9. Customizing a Bar Chart (15 mins):
Customize the bar chart by changing the color and width of the bars.

```
plt.bar(x, y, color='green', width=0.5)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion with Custom Bars')
plt.show()
```

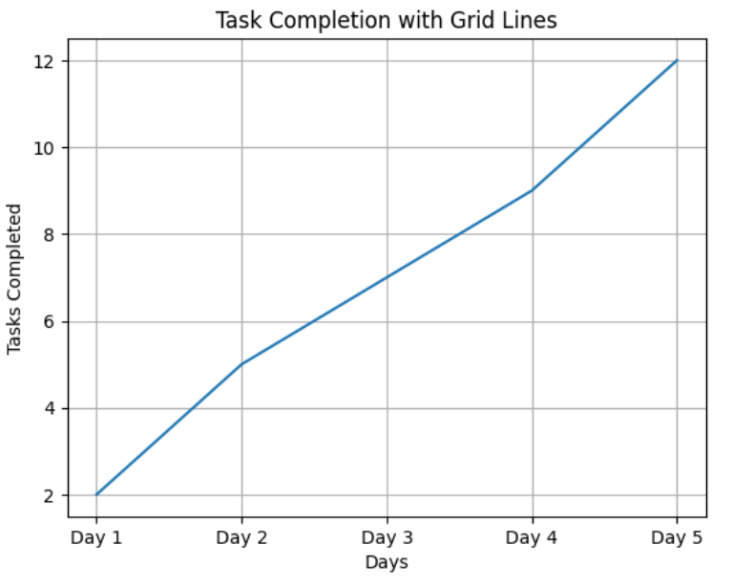


10. Saving a Chart to a File (15 mins):
Save a chart to a file (e.g., PNG or PDF format) using `savefig()`.

```
plt.plot(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.savefig('task_completion_chart.png')
plt.show()
```

11. Adding Grid Lines (15 mins):
Add grid lines to your chart for better readability.

```
plt.plot(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion with Grid Lines')
plt.grid(True)
plt.show()
```




12. Changing Figure Size (15 mins):
Change the size of the figure to make it larger or smaller.

```
plt.figure(figsize=(10, 6))
plt.plot(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
```

```
plt.title('Task Completion with Custom Figure Size')
plt.show()
```

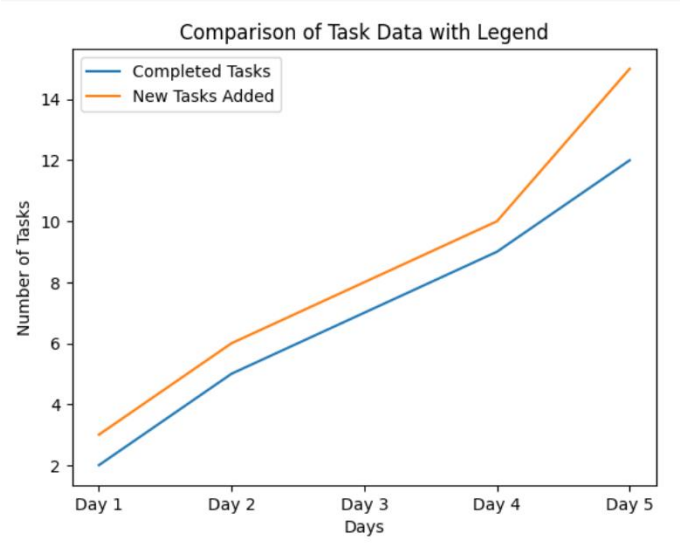
13. Adding Annotations (15 mins):
Add annotations to your chart to highlight specific points or data.

```
plt.plot(x, y)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion with Annotations')
plt.annotate('Highest Point', xy=('Day 5', 12), xytext=('Day 3',
15),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.show()
```

4. Displaying Legends (15 mins):

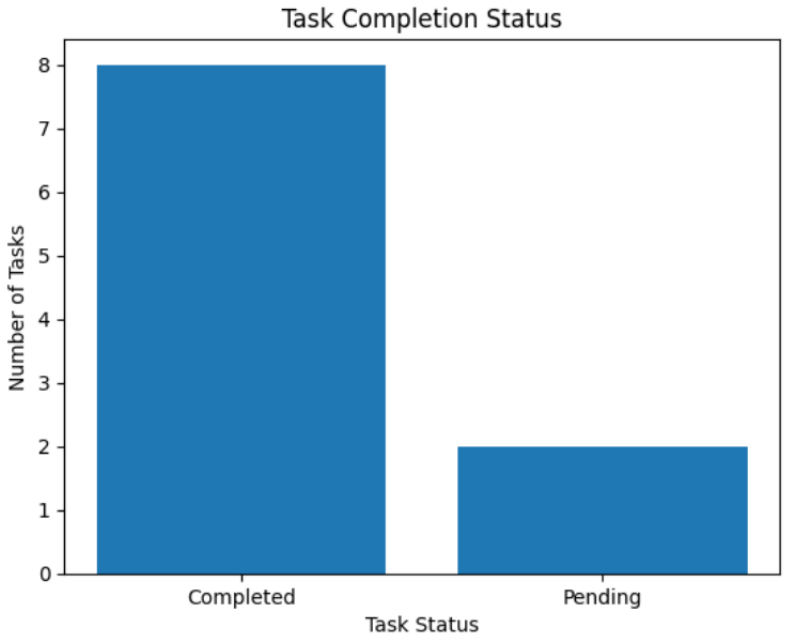
Use the `legend()` function to display a legend for your chart.

```
plt.plot(x, y, label='Completed Tasks')
plt.plot(x, y2, label='New Tasks Added')
plt.xlabel('Days')
plt.ylabel('Number of Tasks')
plt.title('Comparison of Task Data with Legend')
plt.legend()
plt.show()
```




15. Visualizing Data from the To-Do List (15 mins):
Create a bar chart that shows the number of completed and pending tasks from the To-Do List.

```
tasks = ['Completed', 'Pending']
counts = [8, 2]
plt.bar(tasks, counts)
plt.xlabel('Task Status')
plt.ylabel('Number of Tasks')
plt.title('Task Completion Status')
plt.show()
```




## Assignment Task:

- **Task Completion Visualization** (45 mins):

Write a Python program that uses `matplotlib` to:

1. Plot a line chart showing task completion over time.

2. Create a bar chart showing the number of tasks completed versus the number of tasks pending.

3. Customize the charts with labels, titles, and legends.

## Solution Code:

```
import matplotlib.pyplot as plt

# Line Chart - Task Completion Over Time
x = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
y = [2, 5, 7, 9, 12]
plt.plot(x, y)
plt.xlabel('Days')
```

```
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.show()

# Bar Chart - Completed vs Pending Tasks
tasks = ['Completed', 'Pending']
counts = [8, 2]
plt.bar(tasks, counts)
plt.xlabel('Task Status')
plt.ylabel('Number of Tasks')
plt.title('Task Completion Status')
plt.show()
```

# Lab Manual for Week 7: Advanced Python Concepts – Day 3

## Day 3: Data Manipulation with Pandas

Time: 3 Hours

Learning Outcomes:
- Learn how to use the `pandas` library for data manipulation.
- Perform operations like sorting, filtering, and grouping using DataFrames.
- Read and write data from different file formats.
- Apply data analysis to enhance your To-Do List project.

### Tasks:

1. Installing Pandas (15 mins):
Use `pip` to install the `pandas` library.

```
pip install pandas
```

2. Importing Pandas (15 mins):
Import the `pandas` library and create a simple DataFrame from a dictionary.

```
import pandas as pd

data = {'Task': ['Finish project', 'Read book', 'Go shopping'],
        'Status': ['Completed', 'Pending', 'Pending']}
df = pd.DataFrame(data)
print(df)
```

```
             Task     Status
0  Finish project  Completed
1       Read book    Pending
2     Go shopping    Pending
```

2a. more into dataframe of pandas.

```
import pandas as pd
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [24, 27, 22, 32, 29],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston',
'Phoenix'],
    'Salary': [50000, 60000, 55000, 62000, 58000]
}

df = pd.DataFrame(data)
print(df)
```

```
      Name  Age         City  Salary
0    Alice   24     New York   50000
1      Bob   27  Los Angeles   60000
2  Charlie   22      Chicago   55000
3    David   32      Houston   62000
4      Eva   29      Phoenix   58000
```

2b. Write code to calculate and add the bonus column to the DataFrame.

```
df['Bonus'] = df['Salary'] * 0.10
```

2c. Write code to:

- Compute the average salary.
- Find the minimum and maximum ages.

```
avg_salary = df['Salary'].mean()
min_age = df['Age'].min()
max_age = df['Age'].max()
```

2d. Sort the DataFrame based on **Salary** in descending order.

**Task**: Write code to sort the DataFrame by salary in descending order.

```
df_sorted = df.sort_values(by='Salary', ascending=False)
```

2e. Detailed code.

```
import pandas as pd

# Step 1: Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
```

```
    'Age': [24, 27, 22, 32, 29],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston',
'Phoenix'],
    'Salary': [50000, 60000, 55000, 62000, 58000]
}

df = pd.DataFrame(data)
print("DataFrame:\n", df)

# Step 2: Access Data from the DataFrame
print("\nName and City columns:\n", df[['Name', 'City']])
print("\nFirst three rows:\n", df.head(3))

# Step 3: Add a New Column (Bonus)
df['Bonus'] = df['Salary'] * 0.10
print("\nDataFrame with Bonus column:\n", df)

# Step 4: Filter Data (Age > 25)
df_filtered = df[df['Age'] > 25]
print("\nFiltered DataFrame (Age > 25):\n", df_filtered)

# Step 5: Perform Basic Statistics
avg_salary = df['Salary'].mean()
min_age = df['Age'].min()
max_age = df['Age'].max()
print(f"\nAverage Salary: {avg_salary}")
print(f"Minimum Age: {min_age}, Maximum Age: {max_age}")

# Step 6: Sort the Data (by Salary)
df_sorted = df.sort_values(by='Salary', ascending=False)
print("\nDataFrame sorted by Salary:\n", df_sorted)

# Step 7: Export to CSV
df.to_csv('employees.csv', index=False)
```

3. Reading Data from CSV (15 mins):

Read data from a CSV file into a pandas DataFrame.

```
df = pd.read_csv('tasks.csv')
print(df.head())
```

4. Writing Data to CSV (15 mins):

Write the DataFrame to a CSV file.

```
df.to_csv('tasks_output.csv', index=False)
```

5. Sorting Data (15 mins):
Sort the DataFrame by a column (e.g., sort tasks by their status).

```
sorted_df = df.sort_values(by='Status')
print(sorted_df)
```

6. Filtering Data (15 mins):
Filter the DataFrame to show only tasks that are pending.

```
pending_tasks = df[df['Status'] == 'Pending']
print(pending_tasks)
```

7. Adding a New Column (15 mins):
Add a new column to the DataFrame to show task priorities.

```
df['Priority'] = [1, 2, 3]
print(df)
```

8. Removing a Column (15 mins):
Remove a column from the DataFrame (e.g., remove the Priority column).

```
df = df.drop('Priority', axis=1)
print(df)
```

9. Grouping Data (15 mins):
Group the tasks by their status and calculate the number of tasks in each status group.

```
grouped = df.groupby('Status').size()
print(grouped)
```

10. Reading and Writing Excel Files (15 mins):
Read and write data from an Excel file using `pandas`.

```
df = pd.read_excel('tasks.xlsx')
df.to_excel('tasks_output.xlsx', index=False)
```

11. Handling Missing Data (15 mins):
Handle missing data in the DataFrame by filling or dropping NaN values.

```
df.fillna('No Status', inplace=True)
print(df)
```

12. DataFrame Statistics (15 mins):
Get basic statistics of the numeric columns in the DataFrame.

```
print(df.describe())
```

13. Renaming Columns (15 mins):
Rename the columns of the DataFrame for better readability.

```
df.rename(columns={'Task': 'Task Name', 'Status': 'Task Status'},
inplace=True)
print(df)
```

14. Merging DataFrames (15 mins):
Merge two DataFrames based on a common column (e.g., merge task details with task priorities).

```
priority_df = pd.DataFrame({'Task Name': ['Finish project', 'Read
book', 'Go shopping'],
                            'Priority': [1, 2, 3]})
merged_df = pd.merge(df, priority_df, on='Task Name')
print(merged_df)
```

15. Plotting Data with Pandas (15 mins):
Use the `plot()` function to visualize the number of tasks by their status.

```
df['Status'].value_counts().plot(kind='bar')
plt.xlabel('Task Status')
plt.ylabel('Number of Tasks')
```

```
plt.title('Task Status Count')
plt.show()
```

16. Pivot Tables (15 mins):
Create a pivot table to summarize task status by priority.

```
pivot_table = pd.pivot_table(merged_df, values='Priority',
index='Task Status', aggfunc='count')
print(pivot_table)
```

17. Filtering Rows by Condition (15 mins):
Filter rows where the priority is higher than 1.

```
high_priority_tasks = merged_df[merged_df['Priority'] > 1]
print(high_priority_tasks)
```

18. Applying Functions to Columns (15 mins):
Apply a custom function to a column in the DataFrame.

```
def mark_high_priority(priority):
    return 'High' if priority > 1 else 'Low'

merged_df['Priority Label'] =
merged_df['Priority'].apply(mark_high_priority)
print(merged_df)
```

19. Resetting Index (15 mins):
Reset the index of the DataFrame after modifying it.

```
df_reset = merged_df.reset_index(drop=True)
print(df_reset)
```

20. Exporting Filtered Data (15 mins):
Export only the filtered rows (e.g., high priority tasks) to a new CSV file.

```
high_priority_tasks.to_csv('high_priority_tasks.csv', index=False)
```

## Assignment Task:

- **Task Data Analysis** (45 mins):

Write a Python program that uses `pandas` to:

1. Read tasks from a CSV file into a DataFrame.

2. Filter the tasks to show only high priority tasks.

3. Create a bar chart to visualize the number of tasks by their status.

## Solution Code:

```
import pandas as pd
import matplotlib.pyplot as plt

# Read tasks from CSV
df = pd.read_csv('tasks.csv')

# Filter high priority tasks
high_priority_tasks = df[df['Priority'] > 1]
print(high_priority_tasks)

# Create a bar chart to visualize tasks by status
df['Status'].value_counts().plot(kind='bar')
plt.xlabel('Task Status')
plt.ylabel('Number of Tasks')
plt.title('Task Status Count')
plt.show()
```

# Lab Manual for Week 7: Advanced Python Concepts – Day 4

## Day 4: Enhancing the To-Do List with Data Visualization

Time: 3 Hours

Learning Outcomes:
- Understand how to enhance the To-Do List project using data visualization.
- Learn to apply visualizations to represent task progress and status.
- Integrate `matplotlib` with your project to create insightful visualizations.

### Tasks:

1. Importing Matplotlib for Visualization (15 mins):
- Start by importing `matplotlib.pyplot` to use for data visualization.

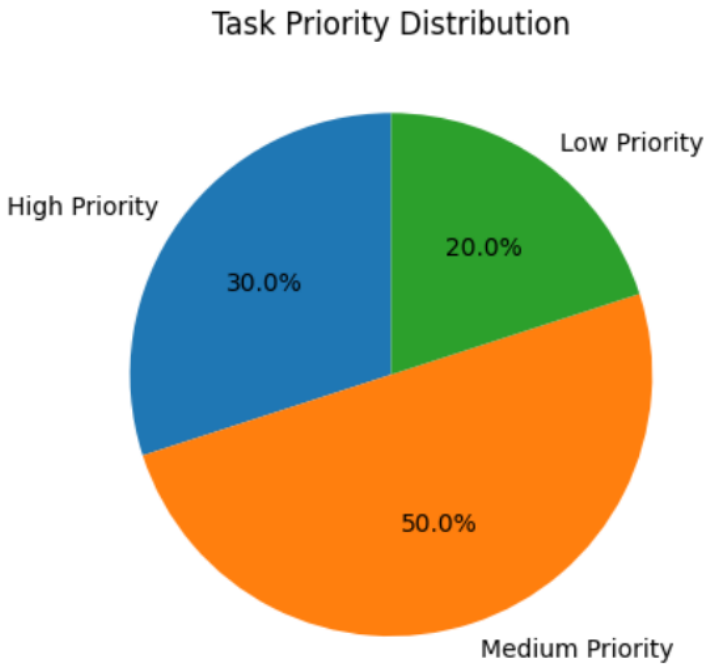
import matplotlib.pyplot as plt

2. Creating a Simple Bar Chart (15 mins):
- Create a bar chart to show the number of completed and pending tasks.

```
tasks = ['Completed', 'Pending']
counts = [8, 2]
plt.bar(tasks, counts)
plt.xlabel('Task Status')
plt.ylabel('Number of Tasks')
plt.title('Task Completion Status')
plt.show()
```

3. Visualizing Task Priorities (15 mins):
- Create a pie chart to visualize task priorities (High, Medium, Low).

```
labels = 'High Priority', 'Medium Priority', 'Low Priority'
sizes = [3, 5, 2]
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.title('Task Priority Distribution')
plt.show()
```




4. Line Chart for Task Completion Over Time (15 mins):
- Create a line chart to show task completion over time.

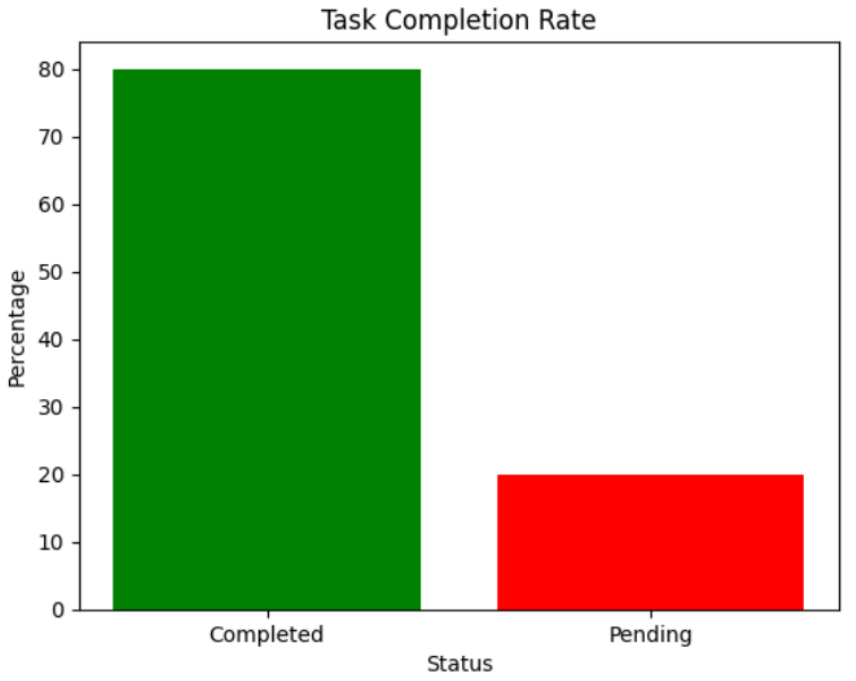
```
days = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
tasks_completed = [2, 5, 7, 9, 12]
plt.plot(days, tasks_completed)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.show()
```

5. Customizing Charts (15 mins):
- Add labels to the axes and a title to make your chart more informative.

```
plt.plot(days, tasks_completed)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Customized Task Completion Chart')
plt.show()
```

6. Task Completion Rate (15 mins):
- Create a bar chart to represent the task completion rate as a percentage.

```
completion_rate = [80, 20]
labels = ['Completed', 'Pending']
plt.bar(labels, completion_rate, color=['green', 'red'])
plt.xlabel('Status')
plt.ylabel('Percentage')
plt.title('Task Completion Rate')
plt.show()
```

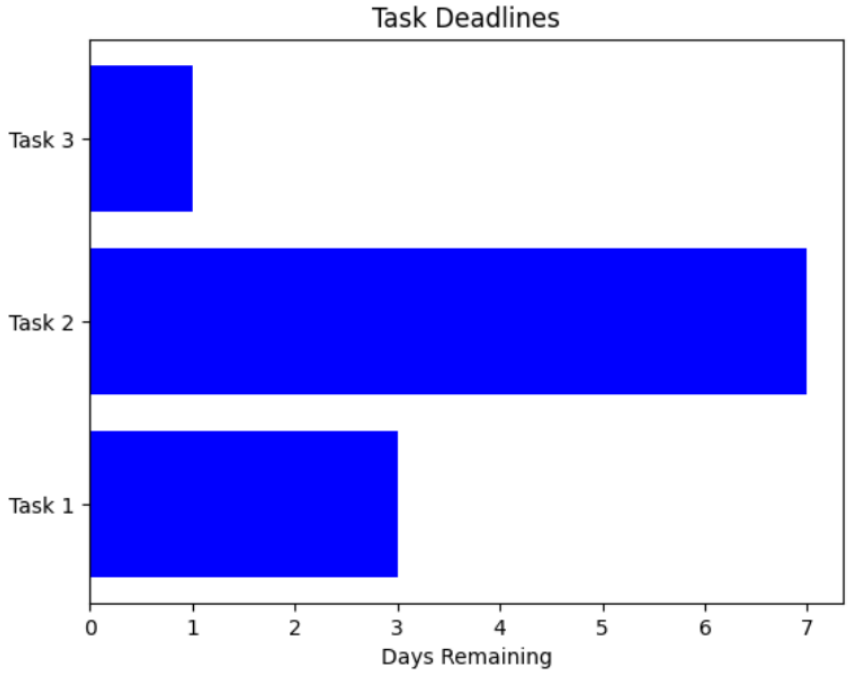



7. Adding Legends (15 mins):
- Use the `legend()` function to add a legend to the chart.

```
tasks = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
completed = [2, 5, 7, 9, 12]
new_tasks = [3, 4, 6, 7, 9]
plt.plot(tasks, completed, label='Completed Tasks')
plt.plot(tasks, new_tasks, label='New Tasks')
plt.xlabel('Days')
plt.ylabel('Tasks')
plt.title('Task Progress with Legends')
plt.legend()
plt.show()
```

8. Horizontal Bar Chart for Task Deadlines (15 mins):
- Create a horizontal bar chart to represent how close tasks are to their deadlines.

```
tasks = ['Task 1', 'Task 2', 'Task 3']
days_remaining = [3, 7, 1]
plt.barh(tasks, days_remaining, color='blue')
plt.xlabel('Days Remaining')
plt.title('Task Deadlines')
plt.show()
```




9. Annotating Charts (15 mins):
- Add annotations to the chart to highlight important data points.

```
plt.plot(days, tasks_completed)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion with Annotations')
plt.annotate('Highest Completion', xy=('Day 5', 12), xytext=('Day
3', 10),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.show()
```

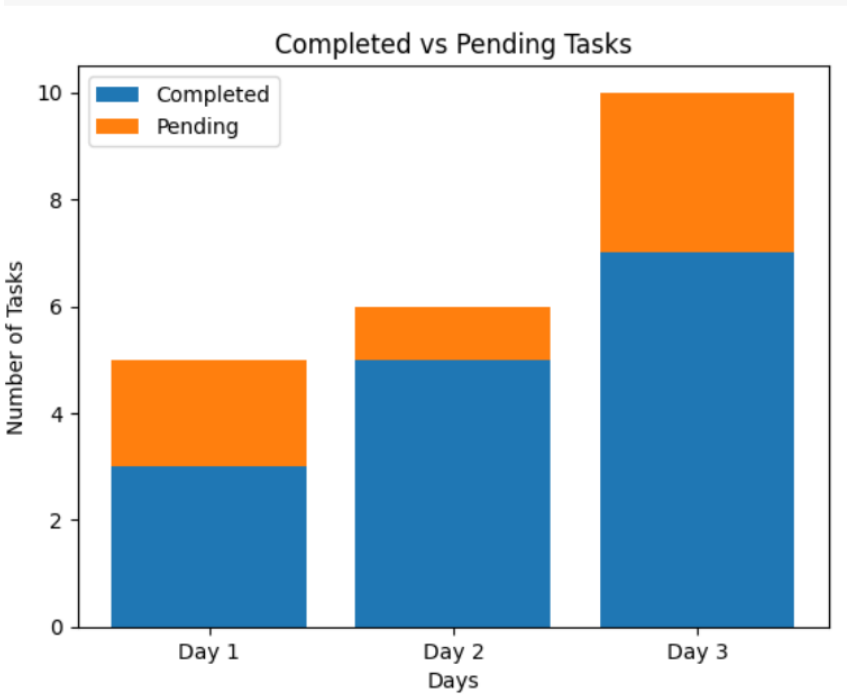
10. Saving Visualizations (15 mins):
- Use the `savefig()` function to save your chart as a PNG or PDF file.

```
plt.plot(days, tasks_completed)
```

```
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.savefig('task_completion_chart.png')
plt.show()
```

11. Creating Stacked Bar Charts (15 mins):
- Use a stacked bar chart to compare completed and pending tasks.

```
tasks = ['Day 1', 'Day 2', 'Day 3']
completed = [3, 5, 7]
pending = [2, 1, 3]
plt.bar(tasks, completed, label='Completed')
plt.bar(tasks, pending, bottom=completed, label='Pending')
plt.xlabel('Days')
plt.ylabel('Number of Tasks')
plt.title('Completed vs Pending Tasks')
plt.legend()
plt.show()
```




12. Creating Subplots (15 mins):
- Create a subplot to show multiple charts side by side.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
```

```
# First plot: Task completion
ax1.bar(['Completed', 'Pending'], [8, 2])
ax1.set_title('Task Completion')

# Second plot: Task priorities
ax2.pie([3, 5], labels=['High', 'Low'], autopct='%1.1f%%')
ax2.set_title('Task Priority Distribution')

plt.tight_layout()
plt.show()
```

13. Task Breakdown by Category (15 mins):
- Use a pie chart to show the percentage of tasks for different categories.

```
categories = ['Work', 'Personal', 'Shopping']
tasks = [4, 3, 2]
plt.pie(tasks, labels=categories, autopct='%1.1f%%',
startangle=140)
plt.title('Task Breakdown by Category')
plt.show()
```

14. Creating Line Charts for Multiple Data Sets (15 mins):
- Create a line chart to show task completion for multiple teams over time.

```
team_a = [1, 4, 6, 8, 10]
team_b = [2, 3, 5, 7, 11]
plt.plot(days, team_a, label='Team A')
plt.plot(days, team_b, label='Team B')
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Team Progress Over Time')
plt.legend()
plt.show()
```

15. Creating Interactive Visualizations (15 mins):
- Explore how to make visualizations interactive using tools like Plotly or Matplotlib's interactive mode.

## Assignment Task:

- **Task Completion Visualization** (45 mins):

Write a Python program that uses `matplotlib` to:

1. Create a bar chart to visualize the task completion rate.

2. Use a pie chart to visualize task priorities.

3. Create a line chart to show task completion over time.

### Solution Code:

```
import matplotlib.pyplot as plt

# Bar Chart - Task Completion Rate
completion_rate = [80, 20]
labels = ['Completed', 'Pending']
plt.bar(labels, completion_rate, color=['green', 'red'])
plt.xlabel('Status')
plt.ylabel('Percentage')
plt.title('Task Completion Rate')
plt.show()

# Pie Chart - Task Priorities
labels = 'High Priority', 'Medium Priority', 'Low Priority'
sizes = [3, 5, 2]
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.title('Task Priority Distribution')
plt.show()

# Line Chart - Task Completion Over Time
days = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
tasks_completed = [2, 5, 7, 9, 12]
plt.plot(days, tasks_completed)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.show()
```

# Lab Manual for Week 7: Advanced Python Concepts – Day 5

## Day 5: Project Day – Data Visualization Enhancement

Time: 3 Hours

Learning Outcomes:
- Apply the knowledge of data visualization to enhance the To-Do List project.
- Implement various types of charts and visual representations for task data.
- Integrate all previous tasks to build a final project component.

### Mega Tasks:

1. Visualizing Task Completion Over Time (45 mins):
Create a line chart that shows how many tasks have been completed each day of the week. This should represent the progress made over time.

```
days = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
tasks_completed = [2, 5, 7, 9, 12]
plt.plot(days, tasks_completed)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.show()
```

2. Task Priority Visualization (45 mins):
Create a pie chart that displays the distribution of task priorities (High, Medium, Low). Ensure that the chart includes percentages.

```
labels = ['High Priority', 'Medium Priority', 'Low Priority']
sizes = [3, 5, 2]
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.title('Task Priority Distribution')
plt.show()
```

3. Task Completion Rate by Category (45 mins):
Using the categories (e.g., Work, Personal, Shopping), create a bar chart to show the task completion rate for each category.

```
categories = ['Work', 'Personal', 'Shopping']
completion_rate = [80, 50, 60]
plt.bar(categories, completion_rate, color=['blue', 'green',
'orange'])
plt.xlabel('Categories')
plt.ylabel('Completion Rate (%)')
plt.title('Task Completion by Category')
plt.show()
```

4. Task Deadlines Visualization (45 mins):
Create a horizontal bar chart that shows how close each task is to its deadline, highlighting tasks that are approaching their deadlines.

```
tasks = ['Task 1', 'Task 2', 'Task 3']
days_left = [3, 7, 1]
plt.barh(tasks, days_left, color=['red' if days <= 2 else 'green'
for days in days_left])
plt.xlabel('Days Left')
plt.title('Tasks with Approaching Deadlines')
plt.show()
```

## Incremental Project Tasks:

In addition to the mega tasks, incrementally improve the To-Do List project by adding the following features:

1. Task Completion Visualization (30 mins):
Enhance the To-Do List project by creating a line chart to visualize task completion over time.

```
import matplotlib.pyplot as plt

days = ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5']
tasks_completed = [2, 5, 7, 9, 12]
plt.plot(days, tasks_completed)
plt.xlabel('Days')
plt.ylabel('Tasks Completed')
plt.title('Task Completion Over Time')
plt.show()
```

2. Task Priority Breakdown (30 mins):

Create a pie chart that displays the task priorities (High, Medium, Low) for the tasks in your To-Do List.

```
labels = ['High Priority', 'Medium Priority', 'Low Priority']
sizes = [3, 5, 2]
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.title('Task Priority Distribution')
plt.show()
```

3. Task Progress vs New Tasks (30 mins):
Create a line chart that compares task completion progress with the number of new tasks added each day.

```
completed_tasks = [2, 4, 6, 8, 10]
new_tasks = [1, 3, 5, 7, 9]
plt.plot(days, completed_tasks, label='Completed Tasks')
plt.plot(days, new_tasks, label='New Tasks')
plt.xlabel('Days')
plt.ylabel('Number of Tasks')
plt.title('Task Progress vs New Tasks')
plt.legend()
plt.show()
```

4. Task Deadline Status (30 mins):
Create a horizontal bar chart that visualizes how close tasks are to their deadlines, using different colors for tasks that are near their deadlines.

```
tasks = ['Task A', 'Task B', 'Task C']
days_left = [1, 5, 10]
colors = ['red' if days <= 2 else 'green' for days in days_left]
plt.barh(tasks, days_left, color=colors)
plt.xlabel('Days Left')
plt.title('Task Deadline Status')
plt.show()
```

# Lab Manual for Week 8: Project Finalization – Day 1

## Day 1: Review of All Key Concepts

Time: 3 Hours

Learning Outcomes:
- Review the main Python concepts learned during the course.
- Refactor and optimize the codebase for the final project.
- Perform code clean-up and ensure the project is well-structured.

**Task: Writing User-Friendly Python Code**

- In this task, you will focus on writing **user-friendly Python code** by following best practices such as proper use of input prompts, clear and informative error messages, and making code more readable and maintainable.

### Instructions:

**Step 1: Create a User-Friendly Input System**

Write a Python program that asks the user to enter their **name**, **age**, and **favorite color**. Ensure that the input prompts are clear and easy to understand.

**Task**: Write a Python program that:

- Prompts the user with simple, friendly messages.
- Uses variables with clear, descriptive names.

**Example**:

```
name = input("Please enter your name: ")
age = input("How old are you? ")
favorite_color = input("What's your favorite color? ")
print(f"Hi {name}! You are {age} years old and your favorite
color is {favorite_color}.")
```

**Step 2: Handle User Errors Gracefully**

Extend the previous program to handle errors in user input. For example, if the user enters an invalid number for age, show a friendly error message and ask them to enter a valid number.

**Task**: Modify your program to:

- Catch any errors (e.g., entering non-numeric input for age).
- Display clear and polite error messages if the user makes a mistake.

**Hint**: Use `try-except` blocks to handle invalid input.

**Example**:

```
while True:
    try:
        age = int(input("How old are you? Please enter a number: "))
        break  # Exit the loop if the input is valid
    except ValueError:
        print("Oops! That doesn't seem to be a number. Please try again.")
```

**Step 3: Provide Feedback to the User**

After collecting the user's details (name, age, and favorite color), print a message that summarizes the information in a friendly and polite manner.

**Task**: Write code that provides positive feedback to the user, thanking them for their input and acknowledging the data they have entered.

**Example**:

```
print(f"Thank you, {name}! It's great to know that you are {age} years old and love the color {favorite_color}.")
```

**Step 4: Validate User Input**

Modify the code to validate the user's input for both the **age** and **favorite color** fields:

- Ensure the **age** is a positive number.
- Ensure the **favorite color** is a non-empty string.

**Task**: Write code that:

- Ensures age is positive.
- Ensures the favorite color is not an empty string.

**Example**:

```
while True:
    try:
        age = int(input("How old are you? Please enter a number:
"))
        if age <= 0:
            print("Age cannot be zero or negative. Please enter a
valid age.")
            continue
        break
    except ValueError:
        print("Oops! That doesn't seem to be a number. Please try
again.")
```

```
while True:
    favorite_color = input("What's your favorite color? ")
    if favorite_color.strip() == "":
        print("You didn't enter anything! Please enter your
favorite color.")
    else:
        break
```

**Step 5: Add Helpful Comments**

Make your code more readable by adding comments that explain what each part of the code does. This makes it easier for others (and your future self) to understand.

**Task**: Add comments throughout your program to explain the purpose of each section.

**Example**:

```
# Ask the user for their name
name = input("Please enter your name: ")

# Loop to ensure valid age input
while True:
    try:
        age = int(input("How old are you? Please enter a number:
"))
        if age <= 0:
            print("Age cannot be zero or negative. Please enter a
valid age.")
            continue
```

```
            break
        except ValueError:
            print("Oops! That doesn't seem to be a number. Please try
again.")

# Loop to ensure valid favorite color input
while True:
    favorite_color = input("What's your favorite color? ")
    if favorite_color.strip() == "":
        print("You didn't enter anything! Please enter your
favorite color.")
    else:
        break

# Provide feedback to the user
print(f"Thank you, {name}! It's great to know that you are {age}
years old and love the color {favorite_color}.")
```

**Step 6: Modularize Your Code**

Break your code into smaller, reusable functions to improve readability and maintainability. This is important for writing user-friendly code that can be reused or updated more easily.

**Task**: Write a function to get the user's name, age, and favorite color. Return the values and print the final message using a separate function.

**Example**:

```
def get_user_info():
    name = input("Please enter your name: ")

    # Get valid age
    while True:
        try:
            age = int(input("How old are you? Please enter a number: "))
            if age <= 0:
                print("Age cannot be zero or negative. Please enter a valid
age.")
                continue
            break
        except ValueError:
            print("Oops! That doesn't seem to be a number. Please try again.")

    # Get valid favorite color
    while True:
        favorite_color = input("What's your favorite color? ")
        if favorite_color.strip() == "":
```

```
            print("You didn't enter anything! Please enter your favorite
color.")
        else:
            break

    return name, age, favorite_color

def print_user_info(name, age, favorite_color):
    print(f"Thank you, {name}! It's great to know that you are {age} years old
and love the color {favorite_color}.")

# Main program flow
name, age, favorite_color = get_user_info()
print_user_info(name, age, favorite_color)
```

## Key Learning Points:

- **Clear User Prompts**: Ensure input prompts are user-friendly and easy to understand.
- **Error Handling**: Gracefully handle invalid input by providing helpful error messages.
- **Input Validation**: Validate user input to avoid errors and unexpected behavior.
- **Code Modularity**: Break the code into reusable functions for better readability and maintainability.
- **Comments**: Use comments to explain the purpose of each section of the code.

This task will help you practice writing Python code that is **user-friendly**, easy to understand, and maintainable. Let me know if you need further clarifications or help!

### Tasks:

1. Reviewing Variables and Data Types (15 mins):
Write a brief explanation of different data types in Python and provide examples of each.

```
# Examples of data types in Python
int_var = 10  # Integer
float_var = 10.5  # Float
str_var = "Hello World"  # String
bool_var = True  # Boolean
list_var = [1, 2, 3]  # List
tuple_var = (1, 2, 3)  # Tuple
dict_var = {'key': 'value'}  # Dictionary
```

2. Review of Conditionals (15 mins):
Write a Python function that takes a number as input and checks if it's positive, negative, or zero.

```
def check_number(num):
    if num > 0:
        return "Positive"
    elif num < 0:
        return "Negative"
    else:
        return "Zero"

print(check_number(5))
```

3. Reviewing Loops (15 mins):
Create a loop that prints all numbers from 1 to 10 using a `for` loop.

```
for i in range(1, 11):
    print(i)
```

4. Reviewing Functions (15 mins):
Define a Python function that calculates the factorial of a number.

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

print(factorial(5))
```

5. Reviewing Lists (15 mins):
Create a list of 5 tasks and print them using a `for` loop.

```
tasks = ['Task 1', 'Task 2', 'Task 3', 'Task 4', 'Task 5']
for task in tasks:
    print(task)
```

6. Refactoring Code (15 mins):
Refactor your To-Do List project by creating a function to add tasks and one to display all tasks.

```
def add_task(task_list, task):
```

```
    task_list.append(task)

def display_tasks(task_list):
    for task in task_list:
        print(task)

tasks = []
add_task(tasks, 'Finish Project')
add_task(tasks, 'Read Book')
display_tasks(tasks)
```

7. Reviewing Dictionaries (15 mins):
Create a dictionary to store task details (title, priority, and status) and display it.

```
task_details = {
    'title': 'Finish Project',
    'priority': 'High',
    'status': 'Incomplete'
}

print(task_details)
```

8. Reviewing Classes (15 mins):
Write a class `Task` with attributes `title`, `priority`, and `status`. Define a method to display task details.

```
class Task:
    def __init__(self, title, priority, status):
        self.title = title
        self.priority = priority
        self.status = status

    def display(self):
        print(f"Title: {self.title}, Priority: {self.priority},
Status: {self.status}")

task1 = Task('Finish Project', 'High', 'Incomplete')
task1.display()
```

9. Reviewing File Handling (15 mins):
Write a Python function that reads tasks from a file and prints them.

```
def read_tasks_from_file(filename):
    with open(filename, 'r') as file:
        tasks = file.readlines()
    for task in tasks:
        print(task.strip())

read_tasks_from_file('tasks.txt')
```

10. Reviewing Error Handling (15 mins):
Write a Python function that handles potential file not found errors.

```
def read_file_with_error_handling(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
            print(data)
    except FileNotFoundError:
        print(f"File '{filename}' not found!")

read_file_with_error_handling('tasks.txt')
```

11. Reviewing Modules (15 mins):
Import the `math` module and use it to calculate the square root of a number.

```
import math
print(math.sqrt(16))
```

12. Refactoring the To-Do List Project (15 mins):
Break the project code into multiple functions to improve readability and maintainability.

```
def add_task(task_list, task):
    task_list.append(task)

def remove_task(task_list, task):
    if task in task_list:
        task_list.remove(task)

def display_tasks(task_list):
    for task in task_list:
```

```
        print(task)

tasks = []
add_task(tasks, 'Finish Project')
display_tasks(tasks)
remove_task(tasks, 'Finish Project')
display_tasks(tasks)
```

13. Reviewing Recursion (15 mins):
Write a recursive function to calculate the sum of a list of numbers.

```
def recursive_sum(lst):
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + recursive_sum(lst[1:])

numbers = [1, 2, 3, 4, 5]
print(recursive_sum(numbers))
```

14. Reviewing Git Concepts (15 mins):
Explain the purpose of version control and why Git is important for collaborative projects.

15. Reviewing Code Optimization (15 mins):
Review the code in your To-Do List project and remove any redundant code or functions.

## Assignment Task:

- **Project Refactoring Assignment** (45 mins):

Refactor your To-Do List project by doing the following:

1. Create separate functions for each major action (e.g., adding, removing, displaying tasks).

2. Organize the code to make it modular and easier to maintain.

3. Ensure the code follows clean coding principles, including proper naming conventions, comments, and error handling.

## Solution Code:

```
def add_task(task_list, task):
    task_list.append(task)

def remove_task(task_list, task):
```

```
    if task in task_list:
        task_list.remove(task)

def display_tasks(task_list):
    for task in task_list:
        print(task)

def main():
    tasks = []
    add_task(tasks, 'Finish Project')
    display_tasks(tasks)
    remove_task(tasks, 'Finish Project')
    display_tasks(tasks)

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

# Lab Manual for Week 8: Project Finalization – Day 2

## Day 2: Final Project Adjustments

Time: 3 Hours

Learning Outcomes:
- Apply final touches to the To-Do List project.
- Test the project to ensure it functions as expected.
- Improve the user interface for better user experience.

### Tasks:

1. Review of Project (15 mins):
Write a brief review of the current state of your To-Do List project. Identify areas where improvements can be made.

2. Enhancing the User Interface (15 mins):
Add prompts and feedback to the project to make it more user-friendly.

```
def add_task(task_list, task):
    task_list.append(task)
    print(f"Task '{task}' added successfully!")

tasks = []
task = input("Enter a task: ")
add_task(tasks, task)
```

3. Error Handling (15 mins):
Add error handling to the project to manage invalid inputs or actions.

```
def remove_task(task_list, task):
    try:
        task_list.remove(task)
        print(f"Task '{task}' removed successfully!")
    except ValueError:
        print(f"Task '{task}' not found!")

tasks = ['Finish Project', 'Read Book']
```

```
remove_task(tasks, 'Complete Homework')
```

4. Adding Task Categories (15 mins):
Enhance the project by allowing users to categorize their tasks (e.g., Work, Personal).

```
def add_task(task_list, task, category):
    task_list.append({'task': task, 'category': category})
    print(f"Task '{task}' added under '{category}' category.")

tasks = []
add_task(tasks, 'Finish Project', 'Work')
print(tasks)
```

5. Displaying Tasks by Category (15 mins):
Add functionality to display tasks by category.

```
def display_tasks_by_category(task_list, category):
    print(f"Tasks in '{category}' category:")
    for task in task_list:
        if task['category'] == category:
            print(f"- {task['task']}")

tasks = [{'task': 'Finish Project', 'category': 'Work'}, {'task':
'Read Book', 'category': 'Personal'}]
display_tasks_by_category(tasks, 'Work')
```

6. Adding Task Status (15 mins):
Modify the project to allow users to mark tasks as completed or pending.

```
def add_task(task_list, task, status='Pending'):
    task_list.append({'task': task, 'status': status})
    print(f"Task '{task}' added with status '{status}'.")

tasks = []
add_task(tasks, 'Finish Project', 'Completed')
print(tasks)
```

7. Displaying Task Status (15 mins):
Add functionality to display the status of each task (Completed or Pending).

```
def display_tasks_with_status(task_list):
    for task in task_list:
        print(f"Task: {task['task']} - Status: {task['status']}")

tasks = [{'task': 'Finish Project', 'status': 'Completed'},
{'task': 'Read Book', 'status': 'Pending'}]
display_tasks_with_status(tasks)
```

8. Editing Tasks (15 mins):
Allow users to edit the title or category of a task.

```
def edit_task(task_list, old_task, new_task):
    for task in task_list:
        if task['task'] == old_task:
            task['task'] = new_task
            print(f"Task '{old_task}' has been renamed to
'{new_task}'")
            break

tasks = [{'task': 'Finish Project', 'status': 'Completed'},
{'task': 'Read Book', 'status': 'Pending'}]
edit_task(tasks, 'Read Book', 'Write Book')
print(tasks)
```

9. Task Search (15 mins):
Add a search feature that allows users to find tasks by title.

```
def search_task(task_list, search_term):
    for task in task_list:
        if search_term.lower() in task['task'].lower():
            print(f"Found Task: {task['task']} - Status:
{task['status']}")

tasks = [{'task': 'Finish Project', 'status': 'Completed'},
{'task': 'Write Book', 'status': 'Pending'}]
search_task(tasks, 'write')
```

10. Task Completion Feature (15 mins):
Allow users to mark tasks as completed.

```
def mark_task_as_completed(task_list, task):
    for t in task_list:
        if t['task'] == task:
            t['status'] = 'Completed'
            print(f"Task '{task}' marked as completed!")
            break

tasks = [{'task': 'Finish Project', 'status': 'Pending'},
{'task': 'Write Book', 'status': 'Pending'}]
mark_task_as_completed(tasks, 'Finish Project')
print(tasks)
```

11. Sorting Tasks by Status (15 mins):
Add a sorting feature to display tasks by their status (Completed or Pending).

```
def sort_tasks_by_status(task_list):
    completed = [task for task in task_list if task['status'] ==
'Completed']
    pending = [task for task in task_list if task['status'] ==
'Pending']
    print("Completed Tasks:", completed)
    print("Pending Tasks:", pending)

tasks = [{'task': 'Finish Project', 'status': 'Completed'},
{'task': 'Write Book', 'status': 'Pending'}]
sort_tasks_by_status(tasks)
```

12. Task Deletion (15 mins):
Allow users to delete tasks from the list.

```
def delete_task(task_list, task):
    for t in task_list:
        if t['task'] == task:
            task_list.remove(t)
            print(f"Task '{task}' deleted.")
            break

tasks = [{'task': 'Finish Project', 'status': 'Completed'},
{'task': 'Write Book', 'status': 'Pending'}]
delete_task(tasks, 'Write Book')
```

```
print(tasks)
```

13. Testing the Project (15 mins):
Test the project by adding, editing, deleting, and displaying tasks to ensure all functionalities work.

14. Debugging the Project (15 mins):
Identify any bugs or errors in the project and fix them.

15. User Feedback (15 mins):
Enhance the user feedback in the project by adding confirmation messages for all actions.

## Assignment Task:

- **Final Adjustments Assignment** (45 mins):

Write a Python program that allows the following tasks:

1. Add a new task with a category and status.

2. Edit an existing task (title and category).

3. Search for a task by title.

4. Mark a task as completed.

## Solution Code:

```
def add_task(task_list, task, category, status='Pending'):
    task_list.append({'task': task, 'category': category,
'status': status})
    print(f"Task '{task}' added under '{category}' category with
status '{status}'.")

def edit_task(task_list, old_task, new_task):
    for task in task_list:
        if task['task'] == old_task:
            task['task'] = new_task
            print(f"Task '{old_task}' has been renamed to
'{new_task}'")

def search_task(task_list, search_term):
    for task in task_list:
        if search_term.lower() in task['task'].lower():
            print(f"Found Task: {task['task']} - Status:
{task['status']}")
```

```
def mark_task_as_completed(task_list, task):
    for t in task_list:
        if t['task'] == task:
            t['status'] = 'Completed'
            print(f"Task '{task}' marked as completed!")

# Example usage:
tasks = []
add_task(tasks, 'Finish Project', 'Work')
edit_task(tasks, 'Finish Project', 'Complete Project')
search_task(tasks, 'complete')
mark_task_as_completed(tasks, 'Complete Project')
print(tasks)
```

# Lab Manual for Week 8: Project Finalization – Day 3

## Day 3: Introduction to Git and Version Control

Time: 3 Hours

Learning Outcomes:
- Understand the importance of version control systems like Git.
- Learn basic Git commands and use them to manage your project.
- Collaborate on projects using GitHub and manage project updates with branches and commits.

### Tasks:

1. 1. What is Git? (15 mins):
- Write a brief explanation of what Git is and why it's important for version control.

2. 2. Installing Git (15 mins):
- Install Git on your machine and verify the installation by running the command:

```
git --version
```

3. 3. Initializing a Git Repository (15 mins):
- Navigate to your project folder and initialize a new Git repository.

```
cd path/to/your/project
git init
```

4. 4. Checking Repository Status (15 mins):
- Use the `git status` command to check the current state of your repository.

```
git status
```

5. 5. Staging Changes (15 mins):
- Stage your project files for the next commit using the `git add` command.

```
git add .
```

6. 6. Committing Changes (15 mins):
- Commit your staged changes to the repository with a meaningful commit message.

```
git commit -m "Initial commit of To-Do List project"
```

7. 7. Viewing Commit History (15 mins):
- Use the `git log` command to view the commit history of your project.

```
git log
```

8. 8. Creating a Remote Repository (15 mins):
- Create a new repository on GitHub and link it to your local project using the following command:

```
git remote add origin https://github.com/yourusername/your-repo-name.git
```

9. 9. Pushing Changes to GitHub (15 mins):
- Push your committed changes to the remote repository on GitHub.

```
git push -u origin master
```

10. 10. Making Changes and Committing Again (15 mins):
- Make some changes to your project files, then add and commit the changes.

```
# Make some changes to your project files
git add .
```
git commit -m "Updated the To-Do List project with task categories"

11. 11. Pulling Changes from GitHub (15 mins):
- If working on a team, use `git pull` to pull changes made by others from the remote repository.

```
git pull origin master
```

12. 12. Branching in Git (15 mins):
- Create a new branch for feature development using the `git branch` command.

```
git branch new-feature
git checkout new-feature
```

13. 13. Merging Branches (15 mins):
- Merge your feature branch back into the master branch.

```
git checkout master
git merge new-feature
```

14. 14. Resolving Merge Conflicts (15 mins):
- If there are merge conflicts, Git will alert you. Resolve the conflicts manually and commit the changes.

```
# Open the conflicted files and resolve the conflicts
git add .
git commit -m "Resolved merge conflicts"
```

15. 15. Creating a New File (15 mins):
- Create a new file in your project, stage it, and commit the changes.

```
echo "print('Hello World')" > new_script.py
git add new_script.py
git commit -m "Added new_script.py"
```

16. 16. Removing a File (15 mins):
- Remove a file from the Git repository and commit the change.

```
git rm new_script.py
git commit -m "Removed new_script.py"
```

17. 17. Viewing Differences Between Commits (15 mins):
- Use the `git diff` command to view the differences between the current working directory and the last commit.

```
git diff
```

18. 18. Tagging a Commit (15 mins):
- Tag a specific commit in the repository to mark a release or important version.

```
git tag -a v1.0 -m "Version 1.0 release"
git push origin v1.0
```

19. 19. Cloning a Repository (15 mins):

- Clone an existing repository from GitHub to your local machine.

```
git clone https://github.com/username/repository.git
```

20. 20. Scenario: Using GitHub for Project Updates (15 mins):
- In this task, you will simulate a code update scenario. Follow these steps to demonstrate how to update your project using GitHub:
- Create a branch for a new feature (e.g., adding task deadlines).
- Implement the feature and commit the changes.
- Push the branch to GitHub and open a pull request.
- Merge the pull request into the master branch.
- Pull the changes to your local machine.
- Test the updated project code.

```
# Step 1: Create a new feature branch
git branch task-deadlines
git checkout task-deadlines

# Step 2: Implement the feature and commit the changes
echo "def add_deadline(task, deadline):" > task_deadlines.py
echo "    task['deadline'] = deadline" >> task_deadlines.py
git add task_deadlines.py
git commit -m "Added task deadlines feature"

# Step 3: Push the branch to GitHub and open a pull request
git push origin task-deadlines

# Step 4: Merge the pull request on GitHub, then pull the changes locally
git checkout master
git pull origin master
```

## Assignment Task:

- **Git and Version Control Assignment** (45 mins):

Perform the following tasks using Git and GitHub:

1. Initialize a Git repository for your To-Do List project.

2. Commit the project files to the repository.

3. Create a new branch for a feature and merge it back into the master branch.

4. Push the changes to GitHub and test the updated code.

### Solution Steps:

```
# Step 1: Initialize a Git repository
git init

# Step 2: Stage and commit project files
git add .
git commit -m "Initial commit of To-Do List project"

# Step 3: Create a new branch, add a feature, and merge it
git branch new-feature
git checkout new-feature
# Make changes to the project files
git add .
git commit -m "Added task priority feature"
git checkout master
git merge new-feature

# Step 4: Push the changes to GitHub
git remote add origin https://github.com/yourusername/your-repo-
name.git
git push -u origin master
```

# Lab Manual for Week 8: Project Finalization – Day 4

## Day 4: Project Deployment

Time: 3 Hours

Learning Outcomes:
- Learn how to deploy Python applications.
- Understand the basics of Python packaging and deployment platforms.
- Deploy a Python project using a platform like Heroku or similar services.

### Tasks:

1. Introduction to Deployment (15 mins):
Write a brief explanation of what deployment is and why it's important for Python projects.

2. Python Virtual Environments (15 mins):
Set up a virtual environment for your project using the following command:

```
python -m venv myenv
source myenv/bin/activate  # On Windows: myenv\Scripts\activate
```

3. Installing Dependencies (15 mins):
Use `pip` to install the necessary dependencies for your project and save them to a `requirements.txt` file.

```
pip install flask
pip freeze > requirements.txt
```

4. Creating a Flask App (15 mins):
Write a simple Flask app to demonstrate deployment. Save the following code to `app.py`.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
```

```
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(debug=True)
```

5. Testing the Flask App Locally (15 mins):
Run the Flask app locally to ensure it's working as expected.

```
python app.py
```

6. Preparing for Deployment (15 mins):
Make sure you have a `Procfile` in your project directory for Heroku deployment.

```
# Create a Procfile
echo "web: gunicorn app:app" > Procfile
```

7. Adding Gunicorn (15 mins):
Install `gunicorn`, which is required for deploying a Flask app on Heroku.

```
pip install gunicorn
pip freeze > requirements.txt
```

8. Heroku CLI Installation (15 mins):
Install the Heroku CLI on your local machine. Verify the installation with:

```
heroku --version
```

9. Logging Into Heroku (15 mins):
Log into your Heroku account using the CLI by running:

```
heroku login
```

10. Creating a Heroku App (15 mins):
Create a new app on Heroku using the CLI.

```
heroku create my-flask-app
```

11. Configuring Heroku (15 mins):
Set the buildpack for Python and ensure all configurations are in place for deployment.

```
heroku buildpacks:set heroku/python
```

12. Deploying the App (15 mins):
Push your local code to the Heroku app for deployment.

```
git add .
git commit -m "Initial deployment"
git push heroku master
```

13. Opening the Deployed App (15 mins):
Use the Heroku CLI to open your deployed app in a web browser.

```
heroku open
```

14. Viewing Logs (15 mins):
Check the logs of your deployed application using the following command to debug any issues.

```
heroku logs --tail
```

15. Scaling the App (15 mins):
Scale your app using the Heroku CLI to ensure it runs smoothly under traffic.

```
heroku ps:scale web=1
```

16. Environment Variables (15 mins):
Set environment variables for your application on Heroku using the following command:

```
heroku config:set SECRET_KEY=mysecretkey
```

17. Updating the App (15 mins):
Make changes to your app code, commit the changes, and push them to Heroku to update the deployment.

```
# Make changes to app.py
git add .
git commit -m "Updated app"
git push heroku master
```

18. Monitoring App Performance (15 mins):
Use the Heroku dashboard or CLI to monitor the performance of your application.

19. Dealing with Deployment Issues (15 mins):
Identify common issues during deployment and use Heroku logs or online resources to resolve them.

20. Best Practices for Deployment (15 mins):
Write about some best practices for deploying Python applications, including security, scalability, and monitoring.

## Assignment Task:

- **Deploy a Python Project** (45 mins):

Follow these steps to deploy your To-Do List project on Heroku:

1. Set up a Flask application for the To-Do List project.

2. Prepare the project for deployment by adding a `Procfile` and necessary dependencies.

3. Deploy the project to Heroku.

4. Test the deployed app to ensure it works.

## Solution Steps:

# Step 1: Set up a Flask application for the To-Do List project
from flask import Flask, render_template

```
app = Flask(__name__)

@app.route('/')
def home():
    return 'This is the To-Do List project!'

if __name__ == '__main__':
    app.run(debug=True)

# Step 2: Prepare for deployment
echo "web: gunicorn app:app" > Procfile
pip install gunicorn
pip freeze > requirements.txt

# Step 3: Deploy to Heroku
heroku create to-do-list-project
git add .
git commit -m "Initial deployment"
git push heroku master

# Step 4: Test the app
heroku open
```

# Lab Manual for Week 8: Project Finalization – Day 5

## Day 5: Final Project Presentation and Assessment

Time: 3 Hours

Learning Outcomes:
- Review the final To-Do List project and make any last adjustments.
- Present the final project, explaining the process, features, and challenges.
- Perform a self-assessment of the project and receive feedback from peers or instructors.

### Tasks with Examples:

1. 1. Final Code Review (15 mins):

- Example: Review your To-Do List project for redundant code and ensure each function is documented properly. Here's an example of cleaning up a function:

```
def add_task(task_list, task):
    if task and task not in task_list:
        task_list.append(task)
        print(f"Task '{task}' added!")
# Ensure proper function naming and comments
```

2. 2. Testing All Features (15 mins):

- Example: Test features like adding, editing, deleting, and viewing tasks. Ensure no errors occur during user interaction.

```
# Adding a task
add_task(tasks, 'Complete the project')
# Viewing tasks
view_tasks(tasks)
# Deleting a task
delete_task(tasks, 'Complete the project')
```

3. 3. Final Touches on User Interface (15 mins):

- Example: Improve feedback messages and user prompts. Here's an example of a more user-friendly prompt:

```
def add_task(task_list, task):
    if task and task not in task_list:
        task_list.append(task)
        print(f"Task '{task}' added successfully! You have
{len(task_list)} tasks in total.")
```

4. 4. Preparing for Presentation (15 mins):
- Example: Prepare a project presentation. Include slides covering the project's goals, features, and the process of building the application.

5. 5. Create a README File (15 mins):
- Example: Write a README file explaining your project. Here's an example:

# To-Do List Project
This is a simple command-line To-Do List application built with Python.
## Features:
- Add, edit, delete, and view tasks.
- Mark tasks as complete or pending.
## Installation:
1. Clone the repository.
2. Run `pip install -r requirements.txt`.
3. Start the app by running `python app.py`.

6. 6. Deploying the Final Project (15 mins):
- Example: Deploy your project to Heroku using Git. Run the following commands to push your final project to Heroku:

```
git add .
git commit -m "Final deployment of To-Do List project"
git push heroku master
```

7. 7. Gathering Feedback (15 mins):
- Example: After presenting your project, ask for feedback on code quality, features, and user experience. Note suggestions for improvements.

8. 8. Self-Assessment (15 mins):

- Example: Reflect on the project. Write a brief paragraph about what you learned and how you overcame challenges. Here's an example:

I learned how to manage a project using Git and GitHub. The biggest challenge I faced was deploying the app, but I resolved it by following Heroku's documentation.

9. 9. Project Presentation (15 mins):
- Example: During your presentation, demonstrate key features of the To-Do List project by running the app and showing how tasks are added, edited, and deleted.

10. 10. Preparing a Demonstration (15 mins):
- Example: Prepare a live demonstration. Have the project running on your machine and showcase adding tasks, marking them complete, and displaying tasks.

11. 11. Project Walkthrough (15 mins):
- Example: Walk through the code, explaining each function and its purpose. Use comments and docstrings to guide the audience through the code.

12. 12. Addressing Feedback (15 mins):
- Example: After receiving feedback, make necessary changes. For instance, if feedback suggests improving user feedback, enhance the confirmation messages when tasks are added or removed.

13. 13. Versioning the Final Project (15 mins):
- Example: Create a Git tag for the final version of your project using the following commands:

```
git tag -a v1.0 -m "Final version of To-Do List project"
git push origin v1.0
```

14. 14. Documentation Review (15 mins):
- Example: Review all code and ensure every function and class is properly documented. Here's an example of a documented function:

```
def delete_task(task_list, task):
    '''
    This function deletes a task from the task_list.
    Args:
```

```
    - task_list: list of tasks
    - task: the task to be deleted
    '''
    if task in task_list:
        task_list.remove(task)
        print(f"Task '{task}' deleted.")
    else:
        print(f"Task '{task}' not found.")
```

15. 15. Final GitHub Push (15 mins):
- Example: Ensure all final changes are pushed to GitHub. Use the following commands to push the final version:

```
git add .
git commit -m "Final version"
git push origin master
```

16. 16. Peer Review (15 mins):
- Example: Work with a peer to review each other's code. Provide feedback on functionality, structure, and user interface design.

17. 17. Backup the Project (15 mins):
- Example: Backup the final project by saving it on an external drive or uploading it to cloud storage like Google Drive or GitHub.

18. 18. Reflecting on the Learning Process (15 mins):
- Example: Write a reflection paragraph. What have you learned about Python, Git, and project development?

During this course, I learned how to break down a project into smaller tasks and manage them using Git. I also gained hands-on experience with deployment using Heroku.

19. 19. Project Submission (15 mins):
- Example: Submit your final project by providing a link to your GitHub repository or submitting it as a zip file.

20. 20. Preparing for Future Projects (15 mins):

- Example: Write down ideas for future improvements or new projects you'd like to work on. For example, you could expand the To-Do List project to include deadlines or notifications.

*The End*