Course Overview: Python Programming Course

Aim:

The aim of this course is to provide learners with a comprehensive understanding of Python programming, focusing on both fundamental concepts and practical applications. The course emphasizes project-based learning, allowing students to apply Python concepts to real-world scenarios, thereby enhancing problem-solving skills and programming proficiency.

Objectives:

By the end of the course, students will be able to:

- Understand Python's basic syntax, variables, data types, and operations.
- Develop proficiency in control flow using conditional statements, loops, and functions.
- Work with Python's data structures such as lists, tuples, sets, and dictionaries.
- Utilize Object-Oriented Programming (OOP) principles, including inheritance, polymorphism, encapsulation, and abstraction.
- Build practical projects such as a Simple Calculator, Contact Book, and Library Management System.
- Handle files, read and write data, and manipulate data using Python.

Target Audience:

- **Beginners**: Individuals with no prior programming experience who want to start learning Python.
- **Intermediate Learners**: Those who have some basic understanding of programming but wish to deepen their knowledge of Python and its applications.
- **Students**: College students studying computer science or related fields.
- **Professionals**: Individuals looking to switch careers or enhance their programming skills for professional development.

Course Structure:

- **Module 1**: Python Fundamentals
  - Introduction to Python, Variables, Data Types, Basic Operations, Conditional Statements, Loops, Functions.
  - Project: Simple Calculator.
- **Module 2**: Working with Data Structures
  - Lists, Tuples, Sets, Dictionaries, Iterating through Data Structures, File Handling.
  - Project: Contact Book.
- **Module 3**: Object-Oriented Programming (OOP)
  - Classes, Objects, Inheritance, Polymorphism, Encapsulation, Abstraction.

- Project: Library Management System.

Resources Needed:

- **Software**:
  - Python interpreter (download from python.org).
  - IDLE (Integrated Development and Learning Environment), which comes pre-installed with Python.
  - Text editor (optional) for code editing (e.g., VS Code, Sublime Text).
- **Hardware**:
  - A computer with a minimum of 4GB RAM and a stable internet connection for downloading resources and participating in online activities.
- **Reference Materials**:
  - Python documentation (docs.python.org).
  - Online tutorials and Python-related forums (e.g., Stack Overflow).

This course is designed to be hands-on, with a focus on real-world projects that solidify learning and make Python programming both accessible and engaging for a wide range of learners.

Module 1: Introduction to Python Basics (Using IDLE)

This module focuses on setting up the Python environment, understanding basic data types, and performing basic operations in Python. We will use IDLE, Python's Integrated Development and Learning Environment, which is a simple and beginner-friendly environment that comes pre-installed with Python.

1. Setting Up the Python Environment

a. Installing Python and IDLE

1. **Download Python**:
   - Go to the official Python website.
   - Download the latest version of Python for your operating system (Windows, macOS, or Linux).
   - Ensure the box that says "Add Python to PATH" is checked during installation.
2. **Install Python**:
   - Run the installer and follow the on-screen instructions. Python will automatically install IDLE along with the Python interpreter.
3. **Launching IDLE**:
   - On Windows: Search for "IDLE" in the Start menu.
   - On macOS: Open a terminal and type `idle`.

- On Linux: IDLE is typically included with Python. If not, you can install it via your package manager (e.g., `sudo apt-get install idle3`).

b. Introduction to IDLE

- **Python Shell**: This is an interactive environment where you can type and execute Python commands immediately. It's a great place for testing small pieces of code.
- **Script Mode**: To write and execute longer Python programs, you can open a new file (File > New File) in IDLE and save it with a `.py` extension.

2. Variables and Data Types

a. What are Variables?

- **Definition**: A variable in Python is a name that refers to a value stored in memory.
- **Usage**: You can assign values to variables using the = operator.

b. Basic Data Types

- **Integers (`int`)**: Whole numbers, e.g., `5`, `-3`.
- **Floating-point numbers (`float`)**: Decimal numbers, e.g., `3.14`, `-0.001`.
- **Strings (`str`)**: Text, enclosed in single or double quotes, e.g., `'Hello'`, `"World"`.
- **Booleans (`bool`)**: True or False values, e.g., `True`, `False`.
- **Lists**: Ordered, mutable collections, e.g., `[1, 2, 3]`.
- **Tuples**: Ordered, immutable collections, e.g., `(1, 2, 3)`.
- **Dictionaries (`dict`)**: Key-value pairs, e.g., `{'name': 'Alice', 'age': 25}`.

c. Variable Naming Rules

- Must start with a letter or an underscore (_).
- Cannot contain spaces or special characters (except _).
- Should be descriptive, e.g., `user_age` instead of `u`.

d. Declaring Variables in IDLE

- Open IDLE and type the following code in the Python Shell:

```
# Integer variable
age = 25

# Float variable
height = 5.9
```

```
# String variable
name = "Alice"

# Boolean variable
is_student = True

# List variable
colors = ['red', 'green', 'blue']

# Tuple variable
coordinates = (10, 20)

# Dictionary variable
person = {'name': 'Bob', 'age': 30}

# Print variables to see their values
print(age)
print(height)
print(name)
print(is_student)
print(colors)
print(coordinates)
print(person)
```

3. Basic Operations in Python

a. Arithmetic Operations

Python supports basic arithmetic operations like addition, subtraction, multiplication, and division.

- **Addition**: `+`
- **Subtraction**: `-`
- **Multiplication**: `*`
- **Division**: `/`
- **Exponentiation**: `**`
- **Modulus (remainder)**: `%`

b. Example: Performing Arithmetic Operations in IDLE

```
# Addition
a = 10
b = 5
sum_result = a + b
```

```
print("Addition:", sum_result)

# Subtraction
sub_result = a - b
print("Subtraction:", sub_result)

# Multiplication
mul_result = a * b
print("Multiplication:", mul_result)

# Division
div_result = a / b
print("Division:", div_result)

# Exponentiation
exp_result = a ** 2
print("Exponentiation:", exp_result)

# Modulus
mod_result = a % b
print("Modulus:", mod_result)
```

c. String Operations

Strings in Python can be concatenated, sliced, and manipulated using various methods.

- **Concatenation**: + (joins two strings together).
- **Repetition**: `*` (repeats the string multiple times).
- **Indexing**: Accessing individual characters, e.g., `name[0]`.
- **Slicing**: Extracting a portion of a string, e.g., `name[1:3]`.

d. Example: String Operations in IDLE

```
# String concatenation
greeting = "Hello" + " " + "World"
print(greeting)

# String repetition
echo = "Hello" * 3
print(echo)

# String indexing
first_letter = greeting[0]
print("First letter:", first_letter)
```

```
# String slicing
substring = greeting[1:5]
print("Substring:", substring)
```

Python Basics - Conditional Statements, Loops, and Functions (Using IDLE)

In this part of the module, we'll explore how to control the flow of a Python program using conditional statements and loops, and how to create reusable blocks of code with functions. Finally, we'll put it all together in a project: creating a simple command-line calculator.

---

1. Conditional Statements (`if`, `else`, `elif`)

a. What are Conditional Statements?

Conditional statements allow your program to make decisions based on certain conditions. Python provides three main keywords for this purpose: `if`, `elif` (else if), and `else`.

b. Syntax and Usage

- **`if` statement**: Executes a block of code if the condition is true.
- **`elif` statement**: Checks another condition if the previous `if` or `elif` condition is false.
- **`else` statement**: Executes a block of code if none of the previous conditions are true.

c. Example: Using Conditional Statements in IDLE

```
# Open IDLE and type the following code in the Python Shell or a new
script file

age = int(input("Enter your age: "))

if age < 18:
    print("You are a minor.")
elif 18 <= age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

- **Explanation**: The program checks the value of `age` and prints a message based on the condition.

d. Exercise:

1. Write a Python program that asks the user for a number and prints whether it is positive, negative, or zero.

---

2. Loops (`for`, `while`)

a. What are Loops?

Loops allow you to repeat a block of code multiple times. Python has two main types of loops: `for` loops and `while` loops.

b. `for` Loop

- **Definition**: A `for` loop iterates over a sequence (like a list, tuple, or string) and executes a block of code for each item in the sequence.
- **Example:**

```
# Open IDLE and type the following code

# Loop through a list of numbers
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```

c. `while` Loop

- **Definition**: A `while` loop continues to execute a block of code as long as the condition is true.
- **Example:**

```
# Open IDLE and type the following code

# Loop until the condition becomes False
counter = 0
while counter < 5:
    print(counter)
    counter += 1  # Increment counter by 1
```

d. Breaking and Continuing in Loops

- **`break` statement**: Exits the loop entirely.
- **`continue` statement**: Skips the current iteration and continues with the next one.

e. Example: Using `break` and `continue`

```
# Open IDLE and type the following code

for i in range(10):
    if i == 5:
        break  # Exit the loop when i is 5
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)
```

f. Exercise:

1. Write a Python program that prints the numbers from 1 to 10 but skips the number 5 using a loop.

---

3. Functions (Defining and Calling Functions)

a. What are Functions?

Functions are reusable blocks of code that perform a specific task. They help in organizing code and reducing repetition.

b. Defining a Function

- **Syntax**: Functions are defined using the `def` keyword, followed by the function name and parentheses `()`.
- **Example:**

```
# Open IDLE and type the following code

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

c. Calling a Function

- **Syntax**: To execute the code inside a function, you "call" the function by writing its name followed by parentheses.
- **Example:**

```
# Call the function
greet()
```

d. Functions with Parameters

- **Definition**: You can pass values to a function through parameters.
- **Example:**

```
def greet(name):
    print(f"Hello, {name}!")

# Call the function with an argument
greet("Alice")
```

e. Functions with Return Values

- **Definition**: Functions can return values using the `return` statement.
- **Example:**

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

# Store the result in a variable and print it
result = add(3, 5)
print(result)
```

f. Exercise:

1. Write a Python function that takes two numbers as input and returns their product.

---

4. Project: Simple Calculator

Now that we have covered conditional statements, loops, and functions, let's build a simple command-line calculator.

a. Project Overview

- The calculator should allow the user to perform basic arithmetic operations: addition, subtraction, multiplication, and division.
- The user should be able to input two numbers and choose an operation.
- The calculator should continue to run until the user chooses to exit.

b. Example Code:

```
# Open IDLE and type the following code

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b != 0:
        return a / b
    else:
        return "Error: Division by zero"

while True:
    print("\nSimple Calculator")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Exit")

    choice = input("Choose an operation (1-5): ")

    if choice == '5':
        print("Exiting the calculator. Goodbye!")
        break

    if choice in ['1', '2', '3', '4']:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        if choice == '1':
```

```
            print("Result:", add(num1, num2))
        elif choice == '2':
            print("Result:", subtract(num1, num2))
        elif choice == '3':
            print("Result:", multiply(num1, num2))
        elif choice == '4':
            print("Result:", divide(num1, num2))
    else:
        print("Invalid input. Please select a valid option.")
```

c. Project Explanation:

- **Functions**: We define functions for each arithmetic operation (add, subtract, multiply, divide).
- **Loop**: The `while` loop allows the program to run continuously until the user chooses to exit.
- **Conditionals**: We use `if`, `elif`, and `else` statements to handle the user's choice and execute the corresponding operation.
- **Error Handling**: The division function checks if the denominator is zero to avoid division errors.

d. Exercise:

1. Extend the calculator by adding a new operation (e.g., modulus `%`).
2. Add error handling to ensure that the user enters valid numbers.

Module 2: Working with Data Structures (Using IDLE)

In this module, we will explore Python's data structures: Lists, Tuples, Sets, and Dictionaries. These structures are essential for storing and managing collections of data. We'll also cover list comprehensions, a concise way to create lists, and perform basic operations like adding and removing elements. All exercises and examples will be done using IDLE as the development environment.

---

1. Lists and List Comprehensions

a. What are Lists?

- **Definition**: A list in Python is an ordered, mutable (changeable) collection of elements. Lists can store elements of different data types.
- **Syntax**: Lists are defined using square brackets `[]`.

b. Creating and Accessing Lists

```
# Open IDLE and type the following code

# Create a list of numbers
numbers = [1, 2, 3, 4, 5]

# Access the first element
print("First element:", numbers[0])

# Access the last element
print("Last element:", numbers[-1])

# Slice the list (first 3 elements)
print("First 3 elements:", numbers[:3])
```

c. Modifying Lists

- **Adding elements**: Use `append()` to add an element to the end of the list.
- **Removing elements**: Use `remove()` to remove the first occurrence of a value, and `pop()` to remove an element at a specific index.

```
# Open IDLE and type the following code

# Append an element to the list
numbers.append(6)
```

```
print("After appending:", numbers)

# Remove an element from the list
numbers.remove(2)
print("After removing 2:", numbers)

# Pop the last element
popped_element = numbers.pop()
print("Popped element:", popped_element)
print("After popping:", numbers)
```

d. List Comprehensions

- **Definition**: List comprehensions provide a concise way to create lists. The syntax is `[expression for item in iterable if condition]`.
- **Example**: Creating a list of squares of numbers.

```
# Open IDLE and type the following code

# List comprehension to create a list of squares
squares = [x**2 for x in range(1, 6)]
print("List of squares:", squares)
```

e. Exercise:

1. Create a list of the first 10 even numbers using list comprehension.
2. Write a Python program to remove all instances of a specific value from a list.

---

2. Tuples, Sets, and Dictionaries

a. What are Tuples?

- **Definition**: A tuple is an ordered, immutable collection of elements. Tuples are similar to lists, but once created, their elements cannot be changed.
- **Syntax**: Tuples are defined using parentheses `()`.

b. Creating and Accessing Tuples

```
# Open IDLE and type the following code

# Create a tuple
coordinates = (10, 20, 30)

# Access the first element
print("First coordinate:", coordinates[0])

# Access the last element
print("Last coordinate:", coordinates[-1])
```

c. What are Sets?

- **Definition**: A set is an unordered collection of unique elements. Sets do not allow duplicate values.
- **Syntax**: Sets are defined using curly braces `{}` or the `set()` function.

d. Creating and Modifying Sets

```
# Open IDLE and type the following code

# Create a set of colors
colors = {"red", "green", "blue"}

# Add an element to the set
colors.add("yellow")
print("After adding yellow:", colors)

# Remove an element from the set
colors.remove("green")
print("After removing green:", colors)
```

- **Note**: Since sets are unordered, the elements may not appear in the same order as they were added.

e. What are Dictionaries?

- **Definition**: A dictionary is an unordered collection of key-value pairs. Each key is unique, and it maps to a value.
- **Syntax**: Dictionaries are defined using curly braces `{}` with key-value pairs separated by colons `:`.

f. Creating and Accessing Dictionaries

```
import 'package:flutter/material.dart';

# Open IDLE and type the following code

# Create a dictionary representing a person
person = {"name": "Alice", "age": 30, "city": "New York"}

# Access the value associated with the key "name"
print("Name:", person["name"])

# Access the value associated with the key "age"
print("Age:", person["age"])
```

g. Modifying Dictionaries

- **Adding/Updating elements**: Assign a value to a key.
- **Removing elements**: Use `pop()` to remove a key-value pair by key.

```
# Open IDLE and type the following code

# Update the person's age
person["age"] = 31
print("Updated age:", person["age"])

# Add a new key-value pair
person["job"] = "Engineer"
print("After adding job:", person)

# Remove a key-value pair
person.pop("city")
print("After removing city:", person)
```

h. Exercise:

1. Create a dictionary to store information about a book (title, author, year). Add a new key-value pair for the genre, and remove the year.
2. Write a Python program that converts a list of tuples into a dictionary.

---

3. Basic Operations on Data Structures

a. Adding Elements

- **Lists**: Use `append()`, `insert()` to add elements.
- **Sets**: Use `add()` to add elements.
- **Dictionaries**: Use assignment = to add key-value pairs.

b. Removing Elements

- **Lists**: Use `remove()`, `pop()`, or `del` to remove elements.
- **Sets**: Use `remove()` or `discard()` to remove elements.
- **Dictionaries**: Use `pop()` or `del` to remove key-value pairs.

c. Iterating Over Data Structures

- **Lists and Tuples**: Use `for` loops.
- **Sets**: Use `for` loops (unordered iteration).
- **Dictionaries**: Use `for` loops to iterate over keys, values, or both.

```
# Open IDLE and type the following code

# List iteration
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print("List item:", number)

# Set iteration
colors = {"red", "green", "blue"}
for color in colors:
    print("Set item:", color)

# Dictionary iteration (keys)
person = {"name": "Alice", "age": 30, "job": "Engineer"}
for key in person:
    print(f"Key: {key}, Value: {person[key]}")
```

d. Checking Membership

- **Lists, Tuples, Sets**: Use the `in` keyword to check if an element exists in the data structure.
- **Dictionaries**: Use `in` to check if a key exists in the dictionary.

```
# Open IDLE and type the following code
```

```
# Check if 3 is in the list
print(3 in numbers)  # True

# Check if "yellow" is in the set
print("yellow" in colors)  # False

# Check if "name" is a key in the dictionary
print("name" in person)  # True
```

e. Exercise:

1. Write a Python program that removes all duplicates from a list using a set.
2. Create a dictionary with student names as keys and grades as values. Write a program that checks if a particular student is in the dictionary.

Iterating Through Data Structures and Introduction to File Handling (Using IDLE)

In this module, we will focus on iterating through different data structures like lists, tuples, sets, and dictionaries, and then move on to file handling in Python. We'll learn how to read from and write to files, which is essential for many real-world applications. To put everything together, we'll build a Contact Book application where contacts are stored in a file and can be added, deleted, updated, and searched.

---

1. Iterating Through Data Structures

a. Iterating Through Lists

Lists are ordered collections, and you can iterate through them using a `for` loop or a `while` loop.

Example:

```
# Open IDLE and type the following code

# Define a list of fruits
fruits = ["apple", "banana", "cherry", "date"]

# Iterate through the list using a for loop
for fruit in fruits:
    print(fruit)
```

Example with Index:

```
# Open IDLE and type the following code

# Use enumerate() to get both index and value
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

b. Iterating Through Tuples

Tuples are similar to lists, but they are immutable. Iterating through tuples is the same as iterating through lists.

Example:

```
import 'package:flutter/material.dart';
```

```
# Open IDLE and type the following code

# Define a tuple of numbers
numbers = (1, 2, 3, 4, 5)

# Iterate through the tuple
for number in numbers:
    print(number)
```

c. Iterating Through Sets

Sets are unordered collections of unique elements. When you iterate through a set, the order of elements may not be the same as their insertion order.

```
# Open IDLE and type the following code

# Define a set of colors
colors = {"red", "green", "blue", "yellow"}

# Iterate through the set
for color in colors:
    print(color)
```

d. Iterating Through Dictionaries

Dictionaries store key-value pairs. You can iterate through the keys, values, or both.

Example:

```
# Open IDLE and type the following code

# Define a dictionary of student grades
grades = {"Alice": 85, "Bob": 92, "Charlie": 78}

# Iterate through the keys
for student in grades:
    print(f"{student} scored {grades[student]}")

# Iterate through keys and values using items()
for student, grade in grades.items():
    print(f"{student} scored {grade}")
```

e. Exercise:

1. Create a list of integers and print each element multiplied by 2.
2. Create a dictionary of products and their prices. Write a Python program that iterates through the dictionary and prints only the products that cost more than $10.

---

2. Introduction to File Handling (Reading/Writing Files)

File handling allows you to store and retrieve data from files, which is essential for persistent storage in applications.

a. Opening and Closing Files

Files need to be opened before you can read or write to them. Python provides the `open()` function to open files and the `close()` method to close them.

Example:

```
# Open IDLE and type the following code

# Open a file in write mode
file = open("example.txt", "w")

# Write something to the file
file.write("Hello, world!")

# Close the file
file.close()
```

b. Writing to a File

You can write data to a file using the `write()` method. Always ensure to close the file after writing.

Example:

```
# Open IDLE and type the following code

# Open a file in write mode
with open("example.txt", "w") as file:
    file.write("This is a new line of text.\n")
    file.write("Another line of text.\n")
```

- **Note**: The `with` statement is used here for better file handling. It automatically closes the file after the block of code is executed.

c. Reading from a File

To read data from a file, you can use the `read()` or `readlines()` method.

Example:

```
# Open IDLE and type the following code

# Open a file in read mode
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

- **`read()`**: Reads the entire file as a single string.
- **`readlines()`**: Reads the file line by line and returns a list of strings.

d. Appending to a File

To add new data to the end of a file without overwriting the existing content, use the `append` mode (`'a'`).

Example:

```
# Open IDLE and type the following code

# Open a file in append mode
with open("example.txt", "a") as file:
    file.write("Appending a new line to the file.\n")
```

e. Exercise:

1. Create a text file and write five lines of text into it. Then, read and print the content of the file.
2. Write a Python program that appends a new line to an existing file every time it runs.

---

3. Project: Contact Book Application

Now, we'll build a Contact Book application where users can add, delete, update, and search for contacts. The contacts will be stored in a file for persistent storage.

a. Project Overview

The Contact Book will support the following functionalities:

1. **Add Contact**: Add a new contact with a name and phone number.
2. **Delete Contact**: Delete a contact by name.
3. **Update Contact**: Update the phone number of an existing contact.
4. **Search Contact**: Search for a contact by name.
5. **View All Contacts**: Display all contacts stored in the file.

b. Project Code

Step 1: Define Functions for Each Operation

```
# Open IDLE and type the following code

# File to store contacts
CONTACT_FILE = "contacts.txt"

def add_contact(name, phone):
    with open(CONTACT_FILE, "a") as file:
        file.write(f"{name},{phone}\n")
    print(f"Contact {name} added.")

def delete_contact(name):
    contacts = []
    with open(CONTACT_FILE, "r") as file:
        contacts = file.readlines()

    with open(CONTACT_FILE, "w") as file:
        for contact in contacts:
            if not contact.startswith(name + ","):
                file.write(contact)
            else:
                print(f"Contact {name} deleted.")

def update_contact(name, new_phone):
    contacts = []
    updated = False
    with open(CONTACT_FILE, "r") as file:
        contacts = file.readlines()
```

```
    with open(CONTACT_FILE, "w") as file:
        for contact in contacts:
            if contact.startswith(name + ","):
                file.write(f"{name},{new_phone}\n")
                updated = True
                print(f"Contact {name} updated.")
            else:
                file.write(contact)

    if not updated:
        print(f"Contact {name} not found.")

def search_contact(name):
    with open(CONTACT_FILE, "r") as file:
        contacts = file.readlines()
        for contact in contacts:
            if contact.startswith(name + ","):
                print(f"Found contact: {contact.strip()}")
                return
        print(f"Contact {name} not found.")

def view_contacts():
    print("Contact List:")
    with open(CONTACT_FILE, "r") as file:
        contacts = file.readlines()
        for contact in contacts:
            print(contact.strip())
```

Step 2: Implement the User Interface

```
# Open IDLE and type the following code

def main():
    while True:
        print("\nContact Book")
        print("1. Add Contact")
        print("2. Delete Contact")
        print("3. Update Contact")
        print("4. Search Contact")
        print("5. View All Contacts")
        print("6. Exit")

        choice = input("Choose an option (1-6): ")

        if choice == '1':
```

```
            name = input("Enter contact name: ")
            phone = input("Enter contact phone number: ")
            add_contact(name, phone)
        elif choice == '2':
            name = input("Enter contact name to delete: ")
            delete_contact(name)
        elif choice == '3':
            name = input("Enter contact name to update: ")
            new_phone = input("Enter new phone number: ")
            update_contact(name, new_phone)
        elif choice == '4':
            name = input("Enter contact name to search: ")
            search_contact(name)
        elif choice == '5':
            view_contacts()
        elif choice == '6':
            print("Exiting Contact Book. Goodbye!")
            break
        else:
            print("Invalid choice. Please choose a valid option.")

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

c. Project Explanation

- **File Handling**: The contacts are stored in a file (`contacts.txt`). Each contact is stored as a line with the format `name,phone_number`.
- **Adding a Contact**: Appends a new contact to the file.
- **Deleting a Contact**: Reads all contacts, removes the specified contact, and writes the remaining contacts back to the file.
- **Updating a Contact**: Searches for the contact, updates the phone number, and rewrites the file.
- **Searching for a Contact**: Searches for the contact by name and prints it if found.
- **Viewing All Contacts**: Reads and displays all contacts in the file.

d. Exercise:

1. Modify the Contact Book to allow storing and managing additional details, like email and address.
2. Add functionality to export contacts to a CSV file.

Module 3: Object-Oriented Programming (OOP) (Using IDLE)

In this module, you will learn the fundamentals of Object-Oriented Programming (OOP) in Python. OOP is a programming paradigm that allows you to organize your code using objects, which combine data (attributes) and functionality (methods). This approach makes your code more modular, reusable, and easier to maintain.

1. Classes and Objects

a. What Are Classes and Objects?

- **Class**: A class is a blueprint for creating objects. It defines a structure that contains attributes (data) and methods (functions) that operate on that data.
- **Object**: An object is an instance of a class. When you create an object from a class, you instantiate that class.

Analogy:

- Think of a class as a blueprint for a house, and the house you build using that blueprint is an object. Each house (object) built from the blueprint (class) may have different characteristics (attributes) like color, size, etc.

Syntax:

```
class ClassName:
    # Class attributes and methods go here
    pass
```

Example:

```
# Open IDLE and type the following code

# Define a class called 'Car'
class Car:
    # The 'pass' statement is a placeholder for an empty block of code
    pass

# Create an object (instance) of the Car class
my_car = Car()

# Print the object
print(my_car)
```

b. The `__init__` Method (Constructor)

- **Definition**: The `__init__` method is a special method (constructor) that gets called when you create an object from a class. It is used to initialize the object's attributes.
- **Syntax**: `def __init__(self, parameters):`

Example:

```
# Open IDLE and type the following code

# Define a class called 'Car'
class Car:
    # Constructor method
    def __init__(self, make, model, year):
        # Initialize attributes
        self.make = make
        self.model = model
        self.year = year

# Create an object of the Car class
my_car = Car("Toyota", "Corolla", 2020)

# Access the object's attributes
print(f"My car is a {my_car.year} {my_car.make} {my_car.model}.")
```

Explanation:

- The `__init__` method initializes the attributes `make`, `model`, and `year` for each object of the `Car` class.
- `self` refers to the current instance of the class and is used to access the attributes and methods.

c. Exercise:

1. Define a class called `Student` with attributes `name`, `age`, and `grade`. Create an object of this class and print the student's information.
2. Modify the `Car` class to include an attribute for `color` and print the car's color along with its other attributes.

---

2. Attributes and Methods

a. What Are Attributes?

- **Attributes**: Attributes are variables that belong to a class or an object. They represent the data or state of the object.

Example:

```
# Open IDLE and type the following code

# Define a class called 'Dog'
class Dog:
    def __init__(self, name, breed):
        # Attributes
        self.name = name
        self.breed = breed

# Create an object of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")

# Access the object's attributes
print(f"My dog's name is {my_dog.name} and it is a {my_dog.breed}.")
```

b. What Are Methods?

- **Methods**: Methods are functions defined within a class that operate on the attributes of the object. They define the behavior of the object.

Example:

```
# Open IDLE and type the following code

# Define a class called 'Dog'
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    # Define a method
    def bark(self):
        print(f"{self.name} says Woof!")

# Create an object of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")
```

```
# Call the object's method
my_dog.bark()
```

Explanation:

- The method `bark` is defined inside the `Dog` class. It prints a message when called.
- You can call methods using the object (e.g., `my_dog.bark()`).

c. Modifying Attributes via Methods

You can define methods that modify the attributes of an object.

Example:

```
# Open IDLE and type the following code

# Define a class called 'Dog'
class Dog:
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    # Method to increase the dog's age
    def birthday(self):
        self.age += 1

    # Method to display the dog's information
    def display_info(self):
        print(f"{self.name} is a {self.age}-year-old {self.breed}.")

# Create an object of the Dog class
my_dog = Dog("Buddy", "Golden Retriever", 5)

# Call the object's methods
my_dog.display_info()

# Celebrate the dog's birthday
my_dog.birthday()

# Display the updated information
my_dog.display_info()
```

Explanation:

- The `birthday` method increases the dog's age by 1.
- The `display_info` method prints the dog's current information.

d. Exercise:

1. Define a class called `BankAccount` with attributes `owner` and `balance`. Implement methods to `deposit` money, `withdraw` money, and `check_balance`.
2. Modify the `Student` class to include a method that updates the student's grade and another method that displays the student's information.

---

3. Putting It All Together: A Complete Example

Let's create a more comprehensive example that brings together everything you've learned so far.

Problem: Create a class `Book` that has the following attributes:

- **title**: The title of the book
- **author**: The author of the book
- **pages**: The number of pages in the book
- **current_page**: The current page the reader is on (default is 1)

Implement the following methods:

- **turn_page()**: Increment the `current_page` by 1.
- **bookmark()**: Print the current page number and a message saying that the page is bookmarked.
- **details()**: Print the title, author, and total number of pages.

Solution:

```
# Open IDLE and type the following code

# Define the Book class
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
        self.current_page = 1

    # Method to turn the page
```

```
    def turn_page(self):
        if self.current_page < self.pages:
            self.current_page += 1
        else:
            print("You're at the end of the book!")

    # Method to bookmark the current page
    def bookmark(self):
        print(f"Bookmarked page {self.current_page} in
'{self.title}'.")

    # Method to display book details
    def details(self):
        print(f"Title: {self.title}, Author: {self.author}, Pages:
{self.pages}")

# Create an object of the Book class
my_book = Book("The Great Gatsby", "F. Scott Fitzgerald", 180)

# Display the book's details
my_book.details()

# Turn the page and bookmark the current page
my_book.turn_page()
my_book.bookmark()
```

Explanation:

- The `Book` class has four attributes: `title`, `author`, `pages`, and `current_page`.
- The `turn_page` method increments the `current_page`.
- The `bookmark` method prints the current page number and bookmarks it.
- The `details` method prints the book's details.

Object-Oriented Programming (OOP) (Part 2) (Using IDLE)

In this part of the module, we will expand on your knowledge of Object-Oriented Programming (OOP) by diving into four key concepts: inheritance, polymorphism, encapsulation, and abstraction. These principles help in writing more modular, reusable, and maintainable code.

1. Inheritance and Polymorphism

a. Inheritance

- **Definition**: Inheritance allows a new class (child class) to inherit attributes and methods from an existing class (parent class). This promotes code reuse and organization by allowing you to build upon existing functionality.
- **Syntax**:

```
class ParentClass:
    # Parent class attributes and methods
    pass

class ChildClass(ParentClass):
    # Child class attributes and methods
    pass
```

Example:

```
# Open IDLE and type the following code

# Define a parent class called 'Animal'
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

# Define a child class called 'Dog' that inherits from 'Animal'
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

# Define another child class called 'Cat' that inherits from 'Animal'
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows.")
```

```
# Create objects of the child classes
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Call the speak method on both objects
dog.speak()
cat.speak()
```

Explanation:

- The `Animal` class is the parent class with a `name` attribute and a `speak` method.
- The `Dog` and `Cat` classes inherit from `Animal` and override the `speak` method with behavior specific to dogs and cats, respectively.

b. Polymorphism

- **Definition**: Polymorphism allows objects of different classes to be treated as objects of a common parent class. This means you can call the same method on different objects, and each object can behave differently based on its class.

Example:

```
# Open IDLE and type the following code

# Define a function that accepts any animal and calls its speak method
def animal_speak(animal):
    animal.speak()

# Use the function with different objects
animal_speak(dog)  # Output: Buddy barks.
animal_speak(cat)  # Output: Whiskers meows.
```

Explanation:

- The `animal_speak` function can accept any object of a class that inherits from `Animal`. This is polymorphism in action, as the `speak` method behaves differently based on the object's class.

c. Exercise:

1. Create a `Vehicle` class with a method `move()`. Then, create two child classes: `Car` and `Bicycle`, each with its own implementation of the `move()` method. Create objects of both classes and demonstrate polymorphism.

2. Create a base class `Shape` with a method `area()`. Create two subclasses, `Rectangle` and `Circle`, each implementing the `area()` method differently. Write a function that calculates the area of any shape.

---

2. Encapsulation and Abstraction

a. Encapsulation

- **Definition**: Encapsulation is the bundling of data (attributes) and methods (functions) that operate on that data within a single class. It also restricts access to certain components of an object to prevent unauthorized access and modification.
- **Private and Public Attributes**: In Python, attributes are public by default. To make an attribute private, prefix its name with double underscores __.

Example:

```
# Open IDLE and type the following code

# Define a class called 'BankAccount'
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner          # Public attribute
        self.__balance = balance    # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance

# Create an object of the BankAccount class
account = BankAccount("Alice", 1000)

# Access public and private attributes
print(account.owner)       # Output: Alice
print(account.get_balance())  # Output: 1000
```

```
# Trying to access the private attribute directly will raise an
AttributeError
# print(account.__balance)  # Uncommenting this line will raise an
error

# Deposit and withdraw money
account.deposit(500)
print(account.get_balance())  # Output: 1500

account.withdraw(200)
print(account.get_balance())  # Output: 1300
```

Explanation:

- The `__balance` attribute is private and cannot be accessed directly from outside the class. Methods like `deposit`, `withdraw`, and `get_balance` provide controlled access to the private attribute.

b. Abstraction

- **Definition**: Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object. This is achieved using classes and interfaces that define the behavior of an object without revealing its internal workings.

Example:

```
# Open IDLE and type the following code

from abc import ABC, abstractmethod

# Define an abstract class called 'Payment'
class Payment(ABC):
    @abstractmethod
    def make_payment(self, amount):
        pass

# Define a subclass called 'CreditCardPayment' that implements the
abstract method
class CreditCardPayment(Payment):
    def make_payment(self, amount):
        print(f"Processing credit card payment of {amount}.")

# Define another subclass called 'PayPalPayment' that implements the
abstract method
```

```
class PayPalPayment(Payment):
    def make_payment(self, amount):
        print(f"Processing PayPal payment of {amount}.")

# Create objects of the subclasses and make payments
credit_card = CreditCardPayment()
paypal = PayPalPayment()

credit_card.make_payment(100)  # Output: Processing credit card
payment of 100.
paypal.make_payment(150)       # Output: Processing PayPal payment of
150.
```

Explanation:

- The `Payment` class is an abstract class that defines an abstract method `make_payment`. Subclasses like `CreditCardPayment` and `PayPalPayment` provide their own implementations of this method, hiding the complex details from the user.

c. Exercise:

1. Define an abstract class `Employee` with an abstract method `calculate_salary()`. Then, create two subclasses: `FullTimeEmployee` and `PartTimeEmployee`, each implementing the `calculate_salary()` method differently. Demonstrate abstraction by calculating the salary of both types of employees.
2. Modify the `BankAccount` class to include encapsulation for both the `owner` and `balance` attributes. Provide methods to set and get the owner name while ensuring valid data.

Project: Library Management System (Using IDLE)

In this project, you'll build a **Library Management System** using Object-Oriented Programming (OOP) principles in Python. The system will allow users to check out, return, and search for books. We will model the system using classes to represent books, users, and the library itself.

Project Structure

- **Classes**:
    1. **Book**: Represents a book with attributes like title, author, and availability status.

2. **User**: Represents a user with attributes like name and a list of borrowed books.
3. **Library**: Manages the collection of books and handles check-out, return, and search operations.

1. Class Definitions

a. Book Class

The `Book` class models a book in the library.

Attributes:

- `title`: The title of the book.
- `author`: The author of the book.
- `is_checked_out`: A boolean indicating whether the book is currently checked out.

Methods:

- `__init__`: Initializes the book with a title, author, and availability status.
- `__str__`: Returns a string representation of the book's title and author.
- `check_out`: Marks the book as checked out.
- `return_book`: Marks the book as available.

```
# Open IDLE and type the following code

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.is_checked_out = False

    def __str__(self):
        return f"'{self.title}' by {self.author}"

    def check_out(self):
        if not self.is_checked_out:
            self.is_checked_out = True
            print(f"{self.title} has been checked out.")
        else:
            print(f"{self.title} is already checked out.")

    def return_book(self):
        if self.is_checked_out:
            self.is_checked_out = False
```

```
            print(f"{self.title} has been returned.")
        else:
            print(f"{self.title} was not checked out.")
```

b. User Class

The `User` class models a library user.

Attributes:

- `name`: The name of the user.
- `borrowed_books`: A list of books that the user has currently checked out.

Methods:

- `__init__`: Initializes the user with a name and an empty list of borrowed books.
- `borrow_book`: Adds a book to the user's borrowed_books list and marks the book as checked out.
- `return_book`: Removes a book from the user's borrowed_books list and marks the book as returned.
- `list_borrowed_books`: Displays the list of borrowed books.

```
# Open IDLE and type the following code

class User:
    def __init__(self, name):
        self.name = name
        self.borrowed_books = []

    def borrow_book(self, book):
        if len(self.borrowed_books) < 3:  # Example limit: 3 books per user
            if not book.is_checked_out:
                self.borrowed_books.append(book)
                book.check_out()
            else:
                print(f"{book.title} is already checked out.")
        else:
            print("You have reached the limit of borrowed books.")

    def return_book(self, book):
        if book in self.borrowed_books:
            self.borrowed_books.remove(book)
            book.return_book()
        else:
            print(f"{book.title} is not in your borrowed books list.")

    def list_borrowed_books(self):
        if self.borrowed_books:
```

```
            print(f"{self.name} has borrowed the following books:")
            for book in self.borrowed_books:
                print(f" - {book}")
        else:
            print(f"{self.name} has not borrowed any books.")
```

c. Library Class

The `Library` class manages the collection of books and users.

Attributes:

- `books`: A list of all books in the library.
- `users`: A list of all users registered in the library.

Methods:

- `__init__`: Initializes the library with an empty list of books and users.
- `add_book`: Adds a new book to the library's collection.
- `add_user`: Registers a new user in the library.
- `find_book`: Searches for a book by title.
- `list_books`: Lists all books in the library, including their availability status.

```
# Open IDLE and type the following code

class Library:
    def __init__(self):
        self.books = []
        self.users = []

    def add_book(self, book):
        self.books.append(book)
        print(f"Added {book} to the library.")

    def add_user(self, user):
        self.users.append(user)
        print(f"Added user {user.name} to the library.")

    def find_book(self, title):
        for book in self.books:
            if book.title.lower() == title.lower():
                return book
        print(f"No book found with title '{title}'.")
```

```
        return None

    def list_books(self):
        if self.books:
            print("Books in the library:")
            for book in self.books:
                status = "Available" if not book.is_checked_out else "Checked out"
                print(f" - {book} [{status}]")
        else:
            print("The library has no books yet.")
```

2. Putting It All Together

Now that we have defined the classes, let's put everything together and test the system.

```
# Open IDLE and type the following code

# Initialize the library
library = Library()

# Add some books to the library
book1 = Book("1984", "George Orwell")
book2 = Book("To Kill a Mockingbird", "Harper Lee")
book3 = Book("The Great Gatsby", "F. Scott Fitzgerald")

library.add_book(book1)
library.add_book(book2)
library.add_book(book3)

# Add a user
user1 = User("Alice")
library.add_user(user1)

# User borrows a book
user1.borrow_book(book1)

# List all books in the library
library.list_books()

# User returns a book
user1.return_book(book1)

# List all books in the library after return
library.list_books()
```

```
# Search for a book by title
searched_book = library.find_book("1984")
if searched_book:
    print(f"Found book: {searched_book}")
```

3. Key Features

- **Adding Books**: Books can be added to the library.
- **User Registration**: Users can be registered in the library system.
- **Borrowing Books**: Users can borrow books, and the system checks if the book is available.
- **Returning Books**: Users can return books, making them available for others.
- **Searching for Books**: Users can search for books by title.
- **Listing Books**: The library can display all books with their availability status.

4. Possible Enhancements

To extend the functionality of this project, you could add:

- **User Limits**: Restrict the number of books a user can borrow at once.
- **Due Dates**: Implement due dates for borrowed books and a system for tracking overdue books.
- **User Authentication**: Add login and authentication features for users.
- **Book Reservations**: Allow users to reserve books that are currently checked out.

Summary

In this project, you've built a simple **Library Management System** using classes to represent books, users, and the library. This project helps solidify your understanding of OOP concepts like inheritance, encapsulation, and polymorphism, and gives you hands-on experience with real-world problem-solving in Python.