Outline

Day 1: Introduction to Python

Lessons:
Introduction to Python and setting up the development environment.
Basic syntax, variables, and data types.
Basic input/output operations.
Basic arithmetic operations.

Day 2: Functions and Modules

Lessons:
Defining and calling functions.
Function arguments and return values.
Introduction to modules and importing them.
Using built-in modules (like math and random).
Project: Basic Calculator with Functions
Refactor the simple calculator project to use functions for each operation.

Day 3: Control Structures

Lessons:
Conditional statements (if, elif, else).
Loops (for, while).
Basic list operations (creating, accessing, modifying lists).
Project: Number Guessing Game
Create a number guessing game where the user has to guess a randomly generated number within a certain range.

Project: Simple Calculator
Create a simple calculator that can perform basic arithmetic operations (addition, subtraction, multiplication, and division) based on user input.

Day 4: Data Structures and File Handling

Lessons:
Advanced data structures: dictionaries, sets, tuples.

File handling: reading from and writing to files.
Project: Contact Book
Create a simple contact book that allows users to add, view, and search contacts.

Day 5: Introduction to Object-Oriented Programming (OOP)

Lessons:
Classes and objects.
Attributes and methods.
Inheritance and polymorphism.
Project: Simple Bank Account System
Create a simple bank account system that allows users to create accounts, deposit, and withdraw money.

Day 1: Introduction to Python

Introduction

How the Internet Works

The internet is a global network of interconnected computers that communicate through a standardized set of protocols. When you visit a website, several steps happen behind the scenes to deliver the web page to your browser.

1. **Your Computer (Client)**: You open a web browser and type a URL (e.g., www.example.com) into the address bar.
2. **DNS Lookup**: The URL is translated into an IP address by the Domain Name System (DNS), which is like the internet's phone book.
3. **Request Sent**: Your browser sends a request to the server hosting the website using the IP address obtained from the DNS.
4. **Server Response**: The server processes the request and sends back the requested web page.
5. **Rendering the Page**: Your browser receives the data and renders the web page on your screen.

Diagram

Here's a simplified diagram to illustrate this process:

lua
Copy code
```
+---------------+          +-------------+          +------------+
|               |          |             |          |            |
| Your Browser  |  <------> | DNS Server  |  <------> | Web Server |
|               |          |             |          |            |
+---------------+          +-------------+          +------------+
```

1. **Your Browser**: Sends a request for a web page.
2. **DNS Server**: Translates the URL to an IP address.
3. **Web Server**: Hosts the website and responds to the request with the web page data.

Why Python?

Python is a popular programming language for several reasons:

- **Easy to Learn**: Python's syntax is clear and resembles English, making it an excellent choice for beginners.
- **Versatile**: Python is used in various fields, including web development, data science, artificial intelligence, machine learning, automation, and more.
- **Large Community**: Python has a vast and active community, providing plenty of resources, libraries, and frameworks to help you in your projects.
- **High Demand**: Python developers are in high demand, with many job opportunities in different industries.

Average Earnings of Python Programmers

Understanding the financial benefits of learning Python can be motivating. Here's a breakdown of average earnings based on experience:

- **Entry-level Python Developers**: $60,000 - $80,000 per year
- **Mid-level Python Developers**: $80,000 - $120,000 per year
- **Senior-level Python Developers**: $120,000+ per year

These figures can vary based on factors like location, industry, and specific skills.

Setting Up the Development Environment

Step 1: Installing Python

1. Visit the official Python website.
2. Download the latest version of Python.
3. Run the installer and follow the instructions. Make sure to check the box that says "Add Python to PATH".

Step 2: Setting Up an Integrated Development Environment (IDE)

We recommend using Visual Studio Code (VS Code).

1. Download and install VS Code.
2. Open VS Code and install the Python extension from the Extensions marketplace.

Writing Your First Python Program

Let's start with a simple program to ensure everything is set up correctly.

1. Open VS Code.
2. Create a new file and save it with a `.py` extension, for example, `hello.py`.

Write the following code:
python
Copy code
```
print("Hello, world!")
```

3.
4. Save the file and run it. In VS Code, you can run the file by right-clicking in the editor and selecting "Run Python File in Terminal".

You should see `Hello, world!` printed in the terminal.

Basic Syntax and Variables

Variables

Variables are used to store data. Python is dynamically typed, which means you don't need to declare the type of a variable.

python
Copy code
```
# Variables
name = "Alice"
age = 25
height = 5.5

print(name)  # Output: Alice
print(age)   # Output: 25
print(height) # Output: 5.5
```

Data Types

Python has several built-in data types:

1. **Numeric Types**: `int`, `float`
2. **String Type**: `str`

3. **Boolean Type**: `bool`

python
Copy code
```python
# Numeric Types
x = 10       # int
y = 3.14     # float

# String Type
greeting = "Hello, world!"

# Boolean Type
is_python_fun = True

print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'float'>
print(type(greeting))  # Output: <class 'str'>
print(type(is_python_fun))  # Output: <class 'bool'>
```

Basic Input/Output Operations

Output

The `print()` function is used to display output.

python
Copy code
```python
# Output
print("Hello, world!")  # Output: Hello, world!
print("My name is", name)  # Output: My name is Alice
print("I am", age, "years old")  # Output: I am 25 years old
```

Input

The `input()` function is used to take user input. It returns the input as a string.

python
Copy code

```
# Input
user_name = input("Enter your name: ")
print("Hello,", user_name)
```

Basic Arithmetic Operations

Python supports basic arithmetic operations:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`

python
Copy code
```
# Arithmetic Operations
a = 10
b = 3

sum = a + b
difference = a - b
product = a * b
quotient = a / b

print("Sum:", sum)  # Output: Sum: 13
print("Difference:", difference)  # Output: Difference: 7
print("Product:", product)  # Output: Product: 30
print("Quotient:", quotient)  # Output: Quotient: 3.3333333333333335
```

Project: Simple Calculator

Project Description

We'll create a simple calculator that can perform basic arithmetic operations (addition, subtraction, multiplication, and division) based on user input.

Step-by-Step Instructions

1. **Prompt the user** for two numbers and an operation.
2. **Perform the operation** based on user input.
3. **Display the result**.

Code Example

python
Copy code

```
# Simple Calculator

# Getting user input
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

# Getting the operation choice from the user
print("Select operation: \n1. Add\n2. Subtract\n3. Multiply\n4.
Divide")
choice = input("Enter choice(1/2/3/4): ")

# Performing the operation
if choice == '1':
    result = num1 + num2
    print(f"The result of {num1} + {num2} is {result}")
elif choice == '2':
    result = num1 - num2
    print(f"The result of {num1} - {num2} is {result}")
elif choice == '3':
    result = num1 * num2
    print(f"The result of {num1} * {num2} is {result}")
elif choice == '4':
    if num2 != 0:
        result = num1 / num2
        print(f"The result of {num1} / {num2} is {result}")
    else:
        print("Error! Division by zero.")
else:
    print("Invalid input")
```

Explanation

- We use `input()` to get user input and `float()` to convert it to a floating-point number.
- We print a menu for the user to select an operation.
- We use conditional statements (`if`, `elif`, `else`) to perform the chosen operation and display the result.

Summary

Today, we've covered:

- How the internet works.
- Why Python is a great choice for programming.
- Average earnings of Python programmers to motivate you.
- Setting up Python and an IDE.
- Writing a simple "Hello, world!" program.
- Understanding variables and basic data types.
- Basic input/output operations.
- Basic arithmetic operations.
- Creating a simple calculator.

Tomorrow, we'll delve deeper into control structures like conditional statements and loops. We'll also build a fun number guessing game. Make sure to practice today's concepts to reinforce your learning.

Feel free to ask any questions you might have or discuss any issues you faced today. See you tomorrow!

Day 2: Control Structures

Introduction

Control structures are essential in programming as they control the flow of execution based on certain conditions or repeatedly execute a block of code. Today, we'll cover conditional statements and loops in Python.

Outline

1. Conditional Statements
2. Loops
3. Project: Number Guessing Game

Conditional Statements

Conditional statements allow you to execute specific code blocks based on certain conditions. This is essential for decision-making in your programs. In Python, the main conditional statements are `if`, `elif`, and `else`.

If Statement

The `if` statement allows you to execute a block of code if a condition is true.

Example: Deciding What to Wear Based on Weather

Imagine you are deciding what to wear based on the temperature:

python
Copy code
```
temperature = 25

if temperature > 20:
    print("Wear shorts!")  # Output: Wear shorts!
```

Explanation: If the temperature is greater than 20 degrees, the program prints "Wear shorts!". If the condition is not true, it does nothing.

Elif Statement

The `elif` (short for else if) statement allows you to check multiple conditions. It runs if the previous conditions were not true.

Example: Assigning Grades Based on Score

Think of grading students based on their scores:

python
Copy code
```
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")  # Output: Grade: B
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

Explanation: This example evaluates the score and assigns a grade. If the score is 85, it falls in the range for a Grade B. If the score doesn't meet the condition of the `if` statement, it checks the `elif` statements in order.

Else Statement

The `else` statement runs if none of the previous conditions were true.

Example: Checking If a Number is Positive, Negative, or Zero

python
Copy code
```
number = -5

if number > 0:
    print("The number is positive.")
elif number == 0:
    print("The number is zero.")
else:
    print("The number is negative.")  # Output: The number is
negative.
```

Explanation: This example checks if a number is positive, zero, or negative. If the number is not greater than 0 and not equal to 0, it must be negative.

Nested If Statements

You can nest `if` statements within each other to create more complex conditions.

Example: Age and Driving Eligibility

Imagine checking if someone is eligible for a driving license:

python
Copy code
```
age = 19

if age >= 18:
    print("You are eligible for a driving license.")
    if age >= 21:
        print("You are also eligible to rent a car.")
    else:
        print("But you are not eligible to rent a car.")  # Output:
You are eligible for a driving license. \n But you are not eligible
to rent a car.
else:
    print("You are not eligible for a driving license.")
```

Explanation: This example checks if the person is old enough to get a driving license. If they are, it further checks if they are old enough to rent a car.

Loops

Loops allow you to execute a block of code repeatedly as long as a condition is true. In Python, the main loops are `while`and `for`.

While Loop

The `while` loop executes as long as a condition is true.

Example: Counting Sheep

Imagine you are counting sheep to fall asleep:

python
Copy code
```
count = 1

while count <= 5:
    print(f"Sheep {count}...")  # Output: Sheep 1... \n Sheep 2... \n 
Sheep 3... \n Sheep 4... \n Sheep 5...
    count += 1  # Increment count by 1
```

Explanation: This example uses a `while` loop to count sheep. The loop continues as long as the count is less than or equal to 5. Inside the loop, the count is incremented by 1 each time it runs.

For Loop

The `for` loop iterates over a sequence (such as a list, tuple, or string) and executes the block of code for each item in the sequence.

Example: Listing Favorite Fruits

Imagine you are listing your favorite fruits:

python
Copy code
```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(f"I love {fruit}!")  # Output: I love apple! \n I love 
banana! \n I love cherry!
```

Explanation: This example uses a `for` loop to iterate over a list of fruits. The loop runs once for each fruit in the list, printing a statement each time.

Range Function

The `range()` function generates a sequence of numbers, which is useful for loops.

Example: Printing Numbers in a Range

python
Copy code
```
# Using range in a for loop
for i in range(5):
    print(i)  # Output: 0 \n 1 \n 2 \n 3 \n 4

# Specifying start, stop, and step
for i in range(1, 10, 2):
    print(i)  # Output: 1 \n 3 \n 5 \n 7 \n 9
```

Explanation:

- The first loop uses `range(5)`, which generates numbers from 0 to 4.
- The second loop uses `range(1, 10, 2)`, which generates numbers from 1 to 9, incrementing by 2 each time (i.e., 1, 3, 5, 7, 9).

Break and Continue Statements

The `break` statement exits the loop prematurely, while the `continue` statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

Break Example: Stopping at a Certain Number

python
Copy code
```
for i in range(10):
    if i == 5:
        break
    print(i)  # Output: 0 \n 1 \n 2 \n 3 \n 4
```

Explanation: This example uses a `for` loop to print numbers. The loop stops when `i` equals 5 because of the `break`statement.

Continue Example: Skipping Even Numbers

python
Copy code
```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)  # Output: 1 \n 3 \n 5 \n 7 \n 9
```

Explanation: This example uses a `for` loop to print numbers from 0 to 9. The `continue` statement skips the print statement for even numbers, so only odd numbers are printed.

Project: Number Guessing Game

Project Description

We'll create a number guessing game where the user has to guess a randomly generated number within a certain range. The game will provide feedback if the guess is too low, too high, or correct.

Step-by-Step Instructions

1. **Generate a random number** between 1 and 100.
2. **Prompt the user** to guess the number.
3. **Provide feedback** if the guess is too low, too high, or correct.
4. **Repeat** until the user guesses the correct number.

Code Example

python
Copy code
```
import random

# Number Guessing Game

# Generate a random number between 1 and 100
number_to_guess = random.randint(1, 100)
attempts = 0

print("Welcome to the Number Guessing Game!")
print("Guess a number between 1 and 100.")

# Game loop
while True:
    user_guess = int(input("Enter your guess: "))
    attempts += 1

    if user_guess < number_to_guess:
        print("Too low! Try again.")
    elif user_guess > number_to_guess:
        print("Too high! Try again.")
    else:
        print(f"Congratulations! You've guessed the number in
{attempts} attempts.")
        break
```

Explanation

- We use the `random` module to generate a random number between 1 and 100.
- The game loop continues to prompt the user for a guess until the correct number is guessed.
- Each guess is compared to the randomly generated number, and feedback is provided to guide the user.

Summary

Today, we've covered:

- **Conditional Statements**: `if`, `elif`, and `else` to control the flow of execution based on conditions.
- **Loops**: `while` and `for` loops to execute code repeatedly.
- **Range Function**: To generate sequences of numbers.
- **Break and Continue Statements**: To control loop execution.
- **Project**: Created a number guessing game to apply these concepts.

Tomorrow, we'll dive into functions and modules, learning how to organize and reuse code efficiently. We'll also refactor our simple calculator project to use functions.

Feel free to ask any questions you might have or discuss any issues you faced today. See you tomorrow!

USES:

Conditional Statements

Conditional statements are used in various real-life situations where decisions need to be made based on certain conditions.

Example 1: ATM Withdrawal

Situation: A user wants to withdraw money from an ATM.

python
Copy code
```
balance = 500  # User's account balance
withdraw_amount = 100  # Amount user wants to withdraw

if withdraw_amount <= balance:
    balance -= withdraw_amount
    print("Withdrawal successful!")
    print(f"Remaining balance: ${balance}")
else:
    print("Insufficient funds!")
```

Explanation: This code checks if the withdrawal amount is less than or equal to the balance. If true, it deducts the amount from the balance and confirms the transaction. Otherwise, it notifies the user of insufficient funds.

Example 2: Online Shopping Cart

Situation: Calculating the total price of items in an online shopping cart and applying a discount if the total exceeds a certain amount.

python
Copy code

```
total_price = 120  # Total price of items in the cart

if total_price > 100:
    discount = total_price * 0.1  # 10% discount
    total_price -= discount
    print(f"Discount applied! Your new total is: ${total_price}")
else:
    print(f"Your total is: ${total_price}")
```

Explanation: This code checks if the total price is greater than $100. If true, it applies a 10% discount. Otherwise, it prints the total price without any discount.

Example 3: Voting Eligibility

Situation: Checking if a person is eligible to vote based on their age.

python
Copy code
```
age = 20  # Person's age

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Explanation: This code checks if the person's age is 18 or above. If true, it confirms their eligibility to vote. Otherwise, it informs them that they are not eligible.

Loops

Loops are used in scenarios where repetitive tasks need to be performed. They help automate processes that would otherwise be tedious and error-prone.

Example 1: Sending Emails to Multiple Recipients

Situation: Sending promotional emails to a list of recipients.

python
Copy code
```
recipients = ["alice@example.com", "bob@example.com",
"charlie@example.com"]

for recipient in recipients:
    print(f"Sending email to {recipient}...")
    # Code to send email
    print(f"Email sent to {recipient}")
```

Explanation: This code iterates over a list of email recipients and simulates sending an email to each one.

Example 2: Processing Sensor Data

Situation: Reading temperature data from sensors and triggering alerts if the temperature exceeds a certain threshold.

python
Copy code
```
sensor_data = [22, 24, 19, 30, 28, 35, 18]

for temperature in sensor_data:
    print(f"Current temperature: {temperature}°C")
    if temperature > 30:
        print("Alert! Temperature too high!")
```

Explanation: This code iterates over a list of temperature readings and prints each one. If a reading exceeds 30°C, it triggers an alert.

Example 3: Calculating Average Exam Score

Situation: Calculating the average score of students in an exam.

python
Copy code
```
scores = [88, 76, 92, 85, 69]

total_score = 0
```

```
for score in scores:
    total_score += score

average_score = total_score / len(scores)
print(f"The average score is: {average_score}")
```

Explanation: This code iterates over a list of exam scores, sums them up, and calculates the average score.

Project: To-Do List Application

Situation: Managing tasks in a to-do list application.

Adding Tasks: Using a loop to repeatedly prompt the user to add tasks until they decide to stop.

python
Copy code
```
tasks = []

while True:
    task = input("Enter a new task (or type 'done' to finish): ")
    if task.lower() == 'done':
        break
    tasks.append(task)

print("Your tasks are:")
for task in tasks:
    print(f"- {task}")
```

Explanation: This code uses a `while` loop to continuously prompt the user to enter tasks. The loop breaks when the user types 'done'. It then prints all the tasks using a `for` loop.

Marking Tasks as Done: Using conditional statements to mark tasks as done.

python
Copy code

```
completed_tasks = []

for task in tasks:
    mark_done = input(f"Did you complete the task '{task}'? (yes/no): ")
    if mark_done.lower() == 'yes':
        completed_tasks.append(task)

print("Completed tasks:")
for task in completed_tasks:
    print(f"- {task}")
```

Explanation: This code iterates over the list of tasks and asks the user if they have completed each task. If the user answers 'yes', the task is added to the completed tasks list.

Summary

Real-life situations and projects where conditional statements and loops are used include:

- **Conditional Statements**:
    - ATM withdrawal process
    - Online shopping cart discount application
    - Voting eligibility check
- **Loops**:
    - Sending emails to multiple recipients
    - Processing sensor data for alerts
    - Calculating average exam scores
- **Project Example**: To-Do List Application for managing tasks and marking them as done.

By understanding these examples, you can see how conditional statements and loops are essential tools in automating decisions and repetitive tasks in various real-life applications. Feel free to ask any questions or discuss any issues you faced today. See you tomorrow for our session on functions and modules!

Python Projects Using Conditional Statements and Loops

Project 1: Rock, Paper, Scissors Game

Description: Create a simple Rock, Paper, Scissors game where the user plays against the computer.

python
Copy code
```
import random

choices = ["rock", "paper", "scissors"]

while True:
    user_choice = input("Enter rock, paper, or scissors (or 'quit' to
stop): ").lower()
    if user_choice == 'quit':
        break

    if user_choice not in choices:
        print("Invalid choice. Please try again.")
        continue

    computer_choice = random.choice(choices)
    print(f"Computer chose: {computer_choice}")

    if user_choice == computer_choice:
        print("It's a tie!")
    elif (user_choice == "rock" and computer_choice == "scissors") or
\
         (user_choice == "paper" and computer_choice == "rock") or \
         (user_choice == "scissors" and computer_choice == "paper"):
        print("You win!")
    else:
        print("You lose!")
```

Explanation: This project uses a `while` loop to keep the game running until the user types 'quit'. It also uses `if`, `elif`, and `else` statements to determine the outcome of each round.

Project 2: Simple Calculator with Functions

Description: Create a calculator that performs basic arithmetic operations using functions and loops.

python
Copy code

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y != 0:
        return x / y
    else:
        return "Error! Division by zero."

while True:
    print("\nSelect operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Quit")

    choice = input("Enter choice(1/2/3/4/5): ")

    if choice == '5':
        break
    if choice in ['1', '2', '3', '4']:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))

        if choice == '1':
            print(f"The result is: {add(num1, num2)}")
        elif choice == '2':
            print(f"The result is: {subtract(num1, num2)}")
        elif choice == '3':
            print(f"The result is: {multiply(num1, num2)}")
        elif choice == '4':
            print(f"The result is: {divide(num1, num2)}")
    else:
        print("Invalid input")
```

Explanation: This project uses functions to perform arithmetic operations and a `while` loop to keep the calculator running until the user decides to quit. Conditional statements determine which operation to perform based on user input.

Project 3: Contact Book

Description: Create a simple contact book that allows users to add, view, and search contacts.

python
Copy code

```
contacts = {}

def add_contact(name, phone):
    contacts[name] = phone

def view_contacts():
    if contacts:
        for name, phone in contacts.items():
            print(f"Name: {name}, Phone: {phone}")
    else:
        print("No contacts found.")
```

```
def search_contact(name):
    if name in contacts:
        print(f"Found: {name} - {contacts[name]}")
    else:
        print(f"Contact {name} not found.")

while True:
    print("\nContact Book Menu")
    print("1. Add Contact")
    print("2. View Contacts")
    print("3. Search Contact")
    print("4. Quit")
    choice = input("Choose an option: ")

    if choice == '4':
        break
    elif choice == '1':
        name = input("Enter name: ")
        phone = input("Enter phone number: ")
        add_contact(name, phone)
        print("Contact added successfully.")
    elif choice == '2':
        view_contacts()
    elif choice == '3':
        name = input("Enter name to search: ")
        search_contact(name)
    else:
        print("Invalid choice. Please try again.")
```

Explanation: This project uses a dictionary to store contacts and provides a menu to add, view, and search for contacts. A `while` loop keeps the program running until the user chooses to quit, and conditional statements control the menu options.

Project 4: Student Grade Management System

Description: Create a system to manage student grades, calculate averages, and determine pass/fail status.

python
Copy code
```
students = {}

def add_student(name, grades):
    students[name] = grades

def view_students():
    if students:
        for name, grades in students.items():
            average = sum(grades) / len(grades)
            status = "Pass" if average >= 60 else "Fail"
            print(f"Name: {name}, Grades: {grades}, Average:
{average:.2f}, Status: {status}")
    else:
        print("No students found.")

while True:
    print("\nStudent Grade Management Menu")
    print("1. Add Student")
    print("2. View Students")
    print("3. Quit")
    choice = input("Choose an option: ")

    if choice == '3':
        break
    elif choice == '1':
        name = input("Enter student name: ")
        grades = list(map(int, input("Enter grades separated by
spaces: ").split()))
        add_student(name, grades)
        print("Student added successfully.")
    elif choice == '2':
        view_students()
    else:
        print("Invalid choice. Please try again.")
```

Explanation: This project uses a dictionary to store student names and their grades. It provides options to add students and view their details, including their average grades and pass/fail status. A `while` loop and conditional statements manage the menu options and logic.

Project 5: Inventory Management System

Description: Create a system to manage product inventory, including adding, viewing, and updating stock levels.

python
Copy code
```python
inventory = {}

def add_product(name, quantity):
    if name in inventory:
        inventory[name] += quantity
    else:
        inventory[name] = quantity

def view_inventory():
    if inventory:
        for name, quantity in inventory.items():
            print(f"Product: {name}, Quantity: {quantity}")
    else:
        print("No products in inventory.")

def update_product(name, quantity):
    if name in inventory:
        inventory[name] = quantity
        print(f"Updated {name} to {quantity} units.")
    else:
        print(f"Product {name} not found in inventory.")

while True:
    print("\nInventory Management Menu")
    print("1. Add Product")
    print("2. View Inventory")
    print("3. Update Product Quantity")
    print("4. Quit")
    choice = input("Choose an option: ")

    if choice == '4':
        break
    elif choice == '1':
        name = input("Enter product name: ")
        quantity = int(input("Enter quantity: "))
        add_product(name, quantity)
        print("Product added successfully.")
    elif choice == '2':
        view_inventory()
    elif choice == '3':
        name = input("Enter product name: ")
        quantity = int(input("Enter new quantity: "))
        update_product(name, quantity)
    else:
        print("Invalid choice. Please try again.")
```

Explanation: This project manages an inventory of products, allowing users to add new products, view the inventory, and update product quantities. A `while` loop keeps the program running, and conditional statements handle the menu choices and logic.

Summary

These projects demonstrate practical uses of conditional statements and loops in Python:

1. **Rock, Paper, Scissors Game**: Uses conditional statements to determine the outcome of each round.
2. **Simple Calculator with Functions**: Uses a loop to keep the calculator running and conditional statements to perform different operations.
3. **Contact Book**: Uses a loop to manage the contact book and conditional statements to handle different menu options.
4. **Student Grade Management System**: Uses a loop to manage student grades and conditional statements to calculate averages and determine pass/fail status.

5. **Inventory Management System**: Uses a loop to manage inventory and conditional statements to add, view, and update product quantities.

By working on these projects, you can see how powerful and versatile conditional statements and loops are in solving real-life problems. Feel free to ask any questions or discuss any issues you faced today. See you tomorrow for our session on functions and modules!

Day 3: Functions and Modules

Introduction

Functions and modules are essential concepts in Python that help you organize and reuse code. Functions allow you to encapsulate code into reusable blocks, while modules help you organize related functions, classes, and variables into separate files.

Outline

1. Functions
   - Defining and calling functions
   - Function arguments and return values
2. Modules
   - Creating and importing modules
   - Using built-in modules
3. Project: Refactor Calculator with Functions

Functions

Defining and Calling Functions

A function is a block of code that performs a specific task. You define a function using the `def` keyword, followed by the function name and parentheses. You call a function by using its name followed by parentheses.

Example: Greeting Function

python
Copy code
```python
def greet():
    print("Hello, world!")

greet()  # Output: Hello, world!
```

Explanation: This example defines a function `greet` that prints a greeting message. The function is then called, which executes the code inside the function.

Function Arguments

Functions can take arguments (also known as parameters) to provide input values. These arguments are specified inside the parentheses during the function definition.

Example: Personalized Greeting

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

greet("Alice")  # Output: Hello, Alice!
greet("Bob")    # Output: Hello, Bob!
```

Explanation: This example defines a function `greet` that takes one argument, `name`, and prints a personalized greeting. The function is called twice with different names.

Default Arguments

You can provide default values for function arguments. If the caller does not provide a value for the argument, the default value is used.

Example: Greeting with Default Name

python
Copy code
```python
def greet(name="world"):
    print(f"Hello, {name}!")

greet()          # Output: Hello, world!
greet("Alice")   # Output: Hello, Alice!
```

Explanation: This example defines a function `greet` with a default argument `name` set to "world". If no argument is provided, the function uses the default value.

Return Values

Functions can return values using the `return` statement. The returned value can be stored in a variable or used directly.

Example: Adding Two Numbers

python
Copy code
```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # Output: 8
```

Explanation: This example defines a function `add` that takes two arguments, `a` and `b`, and returns their sum. The returned value is stored in the variable `result` and printed.

Multiple Return Values

Functions can return multiple values by using tuples.

Example: Basic Arithmetic Operations

python
Copy code
```
def arithmetic_operations(a, b):
    return a + b, a - b, a * b, a / b

add, subtract, multiply, divide = arithmetic_operations(10, 2)
print(f"Add: {add}, Subtract: {subtract}, Multiply: {multiply}, 
Divide: {divide}")
# Output: Add: 12, Subtract: 8, Multiply: 20, Divide: 5.0
```

Explanation: This example defines a function `arithmetic_operations` that performs basic arithmetic operations and returns the results. The returned values are unpacked into separate variables and printed.

Modules

Modules are files containing Python code that can define functions, classes, and variables. You can import modules to reuse their code in other programs.

Creating and Importing Modules

To create a module, save your Python code in a file with a `.py` extension. You can then import this module into other files using the `import` statement.

Example: Creating a Module

Create a file named `my_module.py` with the following content:
python
Copy code
```
def greet(name):
    print(f"Hello, {name}!")

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

1.

Create another file named `main.py` and import `my_module`:
python
Copy code
```
import my_module

my_module.greet("Alice")  # Output: Hello, Alice!
result = my_module.add(3, 5)
print(result)  # Output: 8
```

2.

Explanation: This example creates a module `my_module` with two functions, `greet` and `add`. The module is then imported in `main.py`, and its functions are called.

Using Built-in Modules

Python comes with a standard library of built-in modules that provide various functionalities. You can import and use these modules in your programs.

Example: Using the `math` Module

python
Copy code
```
import math
```

```
number = 16
square_root = math.sqrt(number)
print(f"The square root of {number} is {square_root}")  # Output: The
square root of 16 is 4.0
```

Explanation: This example imports the built-in `math` module and uses its `sqrt` function to calculate the square root of a number.

Example: Using the `random` Module

python
Copy code
```
import random

random_number = random.randint(1, 100)
print(f"Random number between 1 and 100: {random_number}")
```

Explanation: This example imports the built-in `random` module and uses its `randint` function to generate a random number between 1 and 100.

Project: Refactor Calculator with Functions

Project Description

We'll refactor the simple calculator project from Day 1 to use functions for each arithmetic operation. This will make the code more organized and reusable.

Step-by-Step Instructions

1. **Define functions** for addition, subtraction, multiplication, and division.
2. **Modify the main code** to use these functions.

Code Example

python
Copy code
```
# Define functions for arithmetic operations
```

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y != 0:
        return x / y
    else:
        return "Error! Division by zero."

# Main code

while True:
    print("\nSelect operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Quit")

    choice = input("Enter choice(1/2/3/4/5): ")

    if choice == '5':
        break

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

        if choice == '1':
            print(f"The result is: {add(num1, num2)}")
        elif choice == '2':
            print(f"The result is: {subtract(num1, num2)}")
        elif choice == '3':
            print(f"The result is: {multiply(num1, num2)}")
        elif choice == '4':
            print(f"The result is: {divide(num1, num2)}")
    else:
        print("Invalid input")
```

Explanation: This code refactors the simple calculator project by defining separate functions for each arithmetic operation. The main code uses these functions to perform calculations based on user input.

Summary

Today, we've covered:

- **Functions**: How to define and call functions, use arguments and return values, and handle multiple return values.
- **Modules**: How to create and import modules, and use built-in modules for additional functionality.
- **Project**: Refactored the simple calculator project to use functions, making the code more organized and reusable.

Tomorrow, we'll explore data structures and file handling. We'll learn how to work with lists, dictionaries, sets, and tuples, and how to read from and write to files. We'll also build a contact book project to apply these concepts.

Feel free to ask any questions you might have or discuss any issues you faced today. See you tomorrow!

[python]: Absolutely! Let's move on to Day 4, where we'll explore data structures and file handling. We'll explain each item extensively with relatable examples and build a contact book project to apply these concepts.

# Day 4: Data Structures and File Handling

## Introduction
Data structures allow you to store and organize data efficiently, while file handling lets you read from and write to files. Today, we'll cover lists, dictionaries, sets, tuples, and basic file operations.

### Outline
1. Data Structures
   - Lists
   - Dictionaries
   - Sets
   - Tuples
2. File Handling
   - Reading from files
   - Writing to files
3. Project: Contact Book

## Data Structures

### Lists

A list is a collection of items that are ordered and changeable. Lists are defined using square brackets.

Example: Grocery List

```python
# Define a list
grocery_list = ["milk", "eggs", "bread"]

# Accessing items
print(grocery_list[0])  # Output: milk
```

# Modifying items
grocery_list[1] = "cheese"
print(grocery_list) # Output: ['milk', 'cheese', 'bread']

# Adding items
grocery_list.append("butter")
print(grocery_list) # Output: ['milk', 'cheese', 'bread', 'butter']

# Removing items
grocery_list.remove("bread")
print(grocery_list) # Output: ['milk', 'cheese', 'butter']

### Dictionaries

A dictionary is a collection of key-value pairs. Dictionaries are unordered, changeable, and indexed.

Example: Student Grades

```python
# Define a dictionary
grades = {"Alice": 90, "Bob": 85, "Charlie": 92}

# Accessing items
print(grades["Alice"])  # Output: 90

# Modifying items
grades["Bob"] = 88
print(grades)  # Output: {'Alice': 90, 'Bob': 88, 'Charlie': 92}

# Adding items
grades["Dave"] = 95
print(grades)  # Output: {'Alice': 90, 'Bob': 88, 'Charlie': 92, 'Dave': 95}

# Removing items
del grades["Charlie"]
print(grades)  # Output: {'Alice': 90, 'Bob': 88, 'Dave': 95}
```

### Sets

A set is a collection of unique items that are unordered and unindexed. Sets are defined using curly braces.

Example: Unique Colors

```python
# Define a set
colors = {"red", "green", "blue"}

# Adding items
colors.add("yellow")
print(colors)  # Output: {'red', 'green', 'blue', 'yellow'}

# Removing items
colors.remove("green")
print(colors)  # Output: {'red', 'blue', 'yellow'}
```

### Tuples

A tuple is a collection of items that are ordered and unchangeable. Tuples are defined using parentheses.

Example: Coordinates

```python
# Define a tuple
coordinates = (10, 20)

# Accessing items
print(coordinates[0])  # Output: 10

# Tuples are immutable, so you cannot modify items directly
# However, you can create a new tuple with modified values
new_coordinates = (coordinates[0], 30)
print(new_coordinates)  # Output: (10, 30)
```

## File Handling

File handling allows you to read from and write to files. Python provides built-in functions to handle file operations.

### Reading from Files

To read from a file, you open the file in read mode and use the `read()` method.

Example: Reading a File

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

Explanation: This example opens a file named `example.txt` in read mode and prints its content.

### Writing to Files

To write to a file, you open the file in write mode and use the `write()` method.

Example: Writing to a File

```python
# Open a file in write mode
with open("example.txt", "w") as file:
    file.write("Hello, world!")

# Open the file again to check its content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # Output: Hello, world!
```

Explanation: This example opens a file named `example.txt` in write mode and writes "Hello, world!" to it. It then opens the file again in read mode to print its content.

### Appending to Files

To append to a file, you open the file in append mode and use the `write()` method.

Example: Appending to a File

```python
# Open a file in append mode
with open("example.txt", "a") as file:
    file.write("\nAppend this line.")

# Open the file again to check its content
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
    # Output: Hello, world!
    #         Append this line.
```

Explanation: This example opens a file named `example.txt` in append mode and writes a new line to it. It then opens the file again in read mode to print its updated content.

## Project: Contact Book

### Project Description

We'll create a simple contact book that allows users to add, view, and search contacts. The contacts will be stored in a file, so they persist between program runs.

### Step-by-Step Instructions

1. **Create functions** to add, view, and search contacts.
2. **Use file handling** to store and retrieve contacts.

### Code Example

```python
import json

# Define file path for storing contacts
file_path = "contacts.json"

# Function to load contacts from file
```

```
def load_contacts():
    try:
        with open(file_path, "r") as file:
            return json.load(file)
    except FileNotFoundError:
        return {}

# Function to save contacts to file
def save_contacts(contacts):
    with open(file_path, "w") as file:
        json.dump(contacts, file)

# Function to add a contact
def add_contact(name, phone):
    contacts = load_contacts()
    contacts[name] = phone
    save_contacts(contacts)
    print(f"Contact {name} added successfully.")

# Function to view contacts
def view_contacts():
    contacts = load_contacts()
    if contacts:
        for name, phone in contacts.items():
            print(f"Name: {name}, Phone: {phone}")
    else:
        print("No contacts found.")

# Function to search for a contact
def search_contact(name):
    contacts = load_contacts()
    if name in contacts:
        print(f"Found: {name} - {contacts[name]}")
    else:
        print(f"Contact {name} not found.")

# Main code
while True:
    print("\nContact Book Menu")
    print("1. Add Contact")
    print("2. View Contacts")
    print("3. Search Contact")
    print("4. Quit")
    choice = input("Choose an option: ")

    if choice == '4':
        break
    elif choice == '1':
        name = input("Enter name: ")
        phone = input("Enter phone number: ")
        add_contact(name, phone)
    elif choice == '2':
        view_contacts()
    elif choice == '3':
        name = input("Enter name to search: ")
        search_contact(name)
    else:
        print("Invalid choice. Please try again.")
```

### Explanation

1. **File Handling with JSON**: We use the `json` module to store and retrieve contacts in a file. JSON is a lightweight data-interchange format that is easy to read and write.
2. **Functions**:
    - `load_contacts()`: Loads contacts from the file.
    - `save_contacts(contacts)`: Saves contacts to the file.
    - `add_contact(name, phone)`: Adds a new contact and saves it.
    - `view_contacts()`: Displays all contacts.
    - `search_contact(name)`: Searches for a contact by name.
3. **Main Code**: The main code provides a menu for the user to interact with the contact book. It uses a `while` loop to keep the program running until the user chooses to quit.

## Summary

Today, we've covered:
- **Data Structures**: Lists, dictionaries, sets, and tuples, and how to use them to store and organize data.
- **File Handling**: Reading from and writing to files, and how to use JSON for data storage.
- **Project**: Created a contact book application to manage contacts using functions and file handling.

Tomorrow, we'll explore object-oriented programming (OOP) concepts such as classes, objects, inheritance, and polymorphism. We'll also build a simple bank account system to apply these concepts.

Feel free to ask any questions you might have or discuss any issues you faced today. See you tomorrow!

[python]: Great! Let's move on to Day 5, where we'll explore Object-Oriented Programming (OOP) concepts. We'll explain each item extensively with relatable examples and build a simple bank account system to apply these concepts.

# Day 5: Introduction to Object-Oriented Programming (OOP)

## Introduction
Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to organize and manage code. OOP concepts help you structure your code in a more modular, reusable, and scalable way.

### Outline
1. Classes and Objects
2. Attributes and Methods
3. Inheritance and Polymorphism
4. Project: Simple Bank Account System

## Classes and Objects

### Classes

A class is a blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class can have.

Example: Defining a Class

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

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Explanation: This example defines a `Dog` class with a class attribute `species`, instance attributes `name` and `age`, and instance methods `description` and `speak`.

### Objects

An object is an instance of a class. You create an object by calling the class.

Example: Creating Objects

```python
# Create a new Dog object
my_dog = Dog("Buddy", 3)

# Accessing instance attributes
print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 3

# Calling instance methods
print(my_dog.description())  # Output: Buddy is 3 years old
print(my_dog.speak("Woof"))  # Output: Buddy says Woof
```

Explanation: This example creates an object `my_dog` of the `Dog` class and accesses its attributes and methods.

## Attributes and Methods

### Attributes

Attributes are variables that belong to a class or an object. There are two types of attributes: class attributes and instance attributes.

Class Attributes

Class attributes are shared by all instances of the class.

Example: Class Attributes

```python
class Car:
    # Class attribute
    wheels = 4
```

Instance Attributes

Instance attributes are unique to each instance.

Example: Instance Attributes

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

### Methods

Methods are functions that belong to a class. There are three types of methods: instance methods, class methods, and static methods.

Instance Methods

Instance methods operate on an instance of the class.

Example: Instance Methods

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

    def description(self):
        return f"The car is a {self.color} {self.model}"
```

Class Methods

Class methods operate on the class itself rather than on instances.

Example: Class Methods

```python
class Car:
    wheels = 4

    @classmethod
    def is_motor_vehicle(cls):
        return True
```

Static Methods

Static methods do not operate on an instance or the class and are used to group related functions within a class.

Example: Static Methods

```python
class Car:
    @staticmethod
    def honk():
        return "Honk! Honk!"
```

## Inheritance and Polymorphism

### Inheritance

Inheritance allows you to define a new class that inherits attributes and methods from an existing class. This helps in reusing code and creating a hierarchical relationship between classes.

Example: Inheritance

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow"
```

Explanation: This example defines a base class `Animal` with a method `speak`. The `Dog` and `Cat` classes inherit from `Animal` and implement the `speak` method.

### Polymorphism

Polymorphism allows you to use a unified interface for different data types. It enables objects of different classes to be treated as objects of a common superclass.

Example: Polymorphism

```python
animals = [Dog("Buddy"), Cat("Whiskers")]

for animal in animals:
    print(animal.speak())
```

Explanation: This example creates a list of `Dog` and `Cat` objects and calls the `speak` method on each, demonstrating polymorphism.

## Project: Simple Bank Account System

### Project Description

We'll create a simple bank account system that allows users to create accounts, deposit, and withdraw money. We'll use classes to represent the bank account and its operations.

### Step-by-Step Instructions

1. **Define a `BankAccount` class** with attributes for the owner's name and balance.
2. **Implement methods** for depositing and withdrawing money.
3. **Create instances** of `BankAccount` and perform operations.

### Code Example

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. New balance is {self.balance}")

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"{amount} withdrawn. New balance is {self.balance}")
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.balance

# Creating an account
account = BankAccount("John Doe")

# Making transactions
account.deposit(1000)
account.withdraw(500)
account.withdraw(600)
```

### Explanation

1. **Class Definition**: The `BankAccount` class has an initializer method to set the owner's name and balance.
2. **Methods**:
    - `deposit(amount)`: Adds the amount to the balance and prints the new balance.
    - `withdraw(amount)`: Subtracts the amount from the balance if there are sufficient funds and prints the new balance; otherwise, it prints an error message.
    - `get_balance()`: Returns the current balance.
3. **Creating an Account**: An instance of `BankAccount` is created for "John Doe" with an initial balance of 0.
4. **Making Transactions**: The `deposit` and `withdraw` methods are called to perform transactions on the account.

## Summary

Today, we've covered:
- **Classes and Objects**: How to define and use classes and objects in Python.
- **Attributes and Methods**: Understanding class and instance attributes, and different types of methods.
- **Inheritance and Polymorphism**: Using inheritance to reuse code and polymorphism to create a unified interface.
- **Project**: Created a simple bank account system to apply OOP concepts.

This concludes our 5-day Python workshop! Over the past few days, we've covered the fundamentals of Python, including:
1. Introduction to Python
2. Control Structures
3. Functions and Modules
4. Data Structures and File Handling
5. Object-Oriented Programming

By now, you should have a solid understanding of Python and be ready to tackle more advanced projects. Remember to practice regularly and explore additional resources to continue improving your skills.

Feel free to ask any questions you might have or discuss any issues you faced during the workshop. Happy coding!
